



# Understanding Computer Graphics Student Problem-Solving through Source-Code Analysis

Maximilian Rudolf Albrecht Wittmann

A thesis submitted in fulfillment

of the requirements for the degree of

Doctor of Philosophy

at the

Department of Computing

Faculty of Science

Macquarie University

Principal Supervisor: A/Prof Manolya Kavakli

Associate Supervisor: Dr Matt Bower

2013

Copyright by  
Maximilian Wittmann

2013

## I Statement of Candidate

I certify that the work in this thesis entitled “Understanding Computer Graphics Student Problem-Solving through Source-Code Analysis” has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree to any other university or institution other than Macquarie University.

I also certify that the thesis is an original piece of research and it has been written by me. Any help and assistance that I have received in my research work and the preparation of the thesis itself have been appropriately acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

The research presented in this thesis was approved by Macquarie University Ethics Review Committee, reference number: HE27FEB2009-D06330 on 27.02.2009

Signed,

---

Maximilian Wittmann, 40107884  
/ / 2013

## II Abstract

The first aim of this developmental study was to provide insight into the types of problems faced by Computer Graphics students through the analysis of students' programming. The second aim, supporting the first, was to develop analytic approaches to help educators analyse the student programming process in detail. An analysis method based on Grounded Theory (Change-Coding) coded changes in students' computer programs in terms of 'action', 'error' and 'problem'. This was supported by the development of an analysis and data-gathering software program called SCORE. Amongst other findings, quantitative evaluation of the data showed that 44% of changes in the first and 27% of changes in the second assignment were related to 'General Programming' tasks rather than to Computer Graphics programming tasks.

Limitations of the Change-coding results led to the development of a coding approach (Segment-Coding) which focused on coding of sets of related versions of a program (Segments). Detailed qualitative analysis of Segments led to the identification of several issues related to student problem-solving in Computer Graphics programming. These issues include 'Conceptual' issues related to misunderstanding of concepts, 'Cognitive Difficulty of Spatial Programming' issues relating to students' spatial visualization ability, and 'Interplay of different problems' issues which involve students being overwhelmed by having to solve multiple problems at once. These issues were found to affect different parts of the problem-solving process, leading to the development of a four-stage process model of student programming problem-solving consisting of the 'Identify', 'Understand', 'Apply' and 'Perfect' phases. The analysis also revealed that three-dimensional spatial programming is a challenging topic, with students' initial implementation of compound rotations being incorrect 94% of the time.

An automatic approach for the machine-identification of Segments contained in Project Histories was developed to support educators and researchers in identifying significant parts of the programming process for detailed analysis. The Machine-Segmenting algorithm produces sets of related versions that are statistically similar to those produced by a human researcher. Thus the machine-supported Segment-Coding method provides a more time-efficient approach to analysing Computer Science student programs compared to a completely manual analysis.



### III Dedication and Acknowledgements

I would like to dedicate this doctoral dissertation to my partner, Celine Sun, and to my parents, Hedwig and Wolfgang Wittmann. Without their assistance and emotional support, there is no way I could have stuck it out over the years it took to complete this work. This dissertation is as much theirs as it is mine. Now that it's all over, we finally get our weekends back!

I would also like to acknowledge my supervisors, Dr. Matt Bower and Dr. Manolya Kavakli, both of whom provided me tremendous assistance, both in practical terms and as emotional support, through this long journey. Manolya has been at my side since those early honours years, and I'll never forget the days I spent with Matt, combing through every page of my hundred-thousand-word thesis.

I would also like to thank Dr. Steve Cassidy for those early experiences with research which inspired me to pursue my honours degree, and Dr. Scott McCallum for being supportive of my research during the time I spent teaching Comp330.

I am also grateful to Dr. Greg McLean for his support throughout the years of my study, and for helping me through some of the darker days of my doctoral work.

And last but certainly not least, thanks to all the wonderful students who let me probe their programming. Thanks for making my teaching fun, and my doctorate possible!

## IV Foreword

This thesis tries to be many things. Too many perhaps, it has been suggested, for its own good. There is however reason to this apparent madness. In attempting to uncover students' problems with Computer Graphics programming, the lack of an existing framework of analysis methods for the analysis of source code became apparent and took the project into a new and rather unexpected direction.

As a result, this thesis dedicates significant real estate to the development and testing of data-collection / analysis methods for the analysis of versions of source code. Indeed, I see the methods produced as the primary contribution produced by my work. However, the proof is in the pudding, and the fact is that I developed these methods for a reason. Thus my analysis of CG student problems serves to validate the developed methods while at the same time forming its own quite separate contribution.

I have tried my best to highlight the way in which my work fulfils both these aims (production of a set of data-gathering and analysis methods as well as the analysis of student problems during CG programming) but this was a challenging task since the two topics interleave so thoroughly. As a result the thesis is sometimes more repetitive than I would wish, but this was done as an honest attempt to make each chapter and section self-sufficient and understandable in its own right.

The thesis tries to be many things, but at the same time it is limited in its scope and depth. The research presented is in no way a thorough and complete analysis of all types of student errors in CG programming. These results and the conclusions based on them should be seen as part of an exploratory study which aims to provide some early insights. Even more importantly they should provide impetus for further formal study of the field of CG programming, whether with the methods proposed in this thesis or with other qualitative or quantitative methods.

The work carried out as part of the thesis project should also not be seen as 'best practice' for the application of the developed methods. Early versions of the developed method were flawed. Later additions to the developed methods were under-utilised despite their potential because they were developed towards the end of the project. So just as I hope that my foray into the analysis of CG programming might inspire other CG educators to develop a deeper and wider analysis of the topic, I hope that the methods described in this thesis may be useful to other researchers and perhaps brought to greater maturity, whether through the further development of my own software toolkit or through the development of a new analysis environment which builds on some of the ideas laid out in this thesis.

## V Table of Contents

I Statement of Candidate .....	3
II Abstract .....	4
III Dedication and Acknowledgements .....	5
IV Foreword.....	6
V Table of Contents.....	7
VI List of Figures .....	16
VII List of Tables .....	34
1 Introduction .....	40
1.1 Research Context .....	40
1.2 Research Questions .....	44
1.3 Contributions of the thesis .....	45
1.4 Structure of the thesis .....	46
2 Literature Review .....	49
2.1 Literature Review Introduction.....	49
2.2 Computer Graphics Education .....	50
2.2.1 The OpenGL API .....	50
2.2.2 The Computer Graphics curriculum in Computer Science .....	51
2.2.3 Visual Analysis, Perception, Communication and Debugging.....	52
2.2.4 Computer Graphics Teaching Tools .....	53
2.3 Spatial Ability .....	55
2.3.1 Definition of Spatial Ability .....	55
2.3.2 Structure of Spatial Ability .....	55
2.3.3 Spatial Ability and Performance.....	56
2.3.4 Improving Spatial Ability .....	57
2.4 Computer Science Education .....	59
2.4.1 Introduction .....	59
2.4.2 Research objectives in the analysis of student programming .....	60

2.4.3	Research methods in student programming analysis.....	61
2.4.4	Granularity of Source Code Analysis.....	62
2.4.5	Level of Abstraction of Project History Analysis .....	68
2.4.6	Error / Problem Classification Schemes .....	69
2.5	Software Engineering.....	73
2.5.1	Introduction .....	73
2.5.2	Project History Analysis and Co-Change .....	73
2.5.3	Approaches to applying Co-Change .....	77
2.6	Literature Review Summary.....	78
3	Methodology.....	85
3.1	Introduction .....	85
3.2	Terminology .....	86
3.3	Description of the data to be analysed.....	87
3.4	Data-Analysis Methods for analysis of source code .....	88
3.4.1	Evaluation of different research approaches.....	88
3.4.2	Grounded Theory: An overview .....	89
3.4.3	Developed research methods .....	92
3.4.4	Adherence to and deviation from Grounded Theory .....	99
3.5	Software Engineering methods to support the analysis of source code .....	101
3.5.1	Software Engineering Methods to assist Segment-Coding.....	101
3.5.2	Evaluation of developed software engineering methods.....	103
3.6	Data.....	104
3.6.1	Description of Data-Gathering Approach .....	104
3.6.2	Data-Gathering Iterations .....	106
3.6.3	Data Selected for Analysis.....	110
3.6.4	Analysis of Student Perceptions .....	119
3.7	Methodology Summary .....	122
4	The SCORE Toolkit.....	126

4.1	SCORE Toolkit Introduction.....	126
4.2	Rationale for the development of a Qualitative Data Analysis tool for the analysis of source code versions .....	128
4.3	The SCORE Data-Gathering Plug-In .....	128
4.4	Core Functionality and Views .....	130
4.4.1	Diff View .....	131
4.4.2	Note-Taking.....	132
4.4.3	Execution-Based.....	134
4.4.4	Core Functions Walkthrough .....	137
4.5	SCORE Toolkit Future Work .....	137
4.5.1	Supplementing with Subjective Data – Student Comment Extension.....	137
4.5.2	Modification-level Note-Taking .....	139
4.5.3	Dual offline/online plug-in Project History storage .....	139
4.6	SCORE Toolkit Conclusion .....	139
5	Change-Coding .....	142
5.1	Change-Coding Introduction.....	142
5.2	Development of Categorisation Scheme .....	143
5.2.1	Development of Dimensions and Categories.....	144
5.2.2	Detailed Description of Classification Categories .....	147
5.2.3	Change-Coding Rules / Application of Classification Scheme.....	152
5.3	Change-Coding with SCORE .....	154
5.3.1	Setting up Change-Coding using the SCORE Analyser .....	155
5.3.2	Change-Coding Panel .....	155
5.3.3	Graphing Change-Coding Results.....	156
5.3.4	Change-Coding Metrics.....	157
5.3.5	Performing the Change-Coding Process using the SCORE Analyser .....	158
5.4	Change-Coding Results .....	160
5.4.1	Action / Error / Problem Data .....	160

5.4.2	Relationship between Action/Error/Problem Dimensions .....	164
5.4.3	Discussion of Change-Coding data.....	166
5.5	Limitations of the Change-Coding method.....	167
5.6	Change-Coding Conclusion .....	170
6	Segment-Coding.....	172
6.1	Segment-Coding Method Introduction.....	172
6.2	Enabling Detailed Analysis with the SCORE Analyser .....	173
6.2.1	Line Histories.....	174
6.2.2	SCORE Detailed Analysis Features .....	180
6.2.3	Performing the Segment-Coding Process using the SCORE Analyser .....	186
6.3	Segment-Coding Classification Method.....	187
6.3.1	Application of the Change-Coding Classification Scheme to Segment-Coding.....	187
6.3.2	Approach to Categorisation and Segmenting .....	189
6.3.3	Approach to the Analysis of Segments .....	190
6.3.4	Sub-categorisation of Segment classification categories.....	192
6.4	Segment-Coding with SCORE .....	198
6.4.1	SCORE Analyser Segment-Coding Facilities .....	198
6.4.2	Segment-Coding using the SCORE Analyser.....	202
6.5	Segmenting Results.....	204
6.5.1	Measuring Segment Significance .....	204
6.5.2	Method of Segment Referencing.....	206
6.5.3	Identified Segments .....	207
6.5.4	Discussion of Segmenting Results.....	218
6.6	Analysis of Segment Contents .....	219
6.6.1	Method of Segment Content Analysis .....	219
6.6.2	Presentation of Project History data .....	223
6.6.3	Example of in-depth Qualitative Analysis of Segment Contents .....	225
6.6.4	Summary of Qualitative Analysis of Segment Contents .....	241

6.6.5	Quantitative Analysis of Spatial Programming in Animation Segments .....	249
6.6.6	Identified Issues .....	252
6.6.7	A Model of Student Problem-Solving during Computer Graphics Programming .....	266
6.7	Analysis of Segment Features .....	275
6.8	Updating the Segment-Coding method for future research .....	279
6.8.1	Updating the coding method .....	279
6.8.2	Evaluating other metrics for measuring Segment significance.....	281
6.9	Limitations of the Segment-Coding method.....	283
6.9.1	Measurement of Time Taken per Change .....	283
6.9.2	Time-intensive nature of analysis .....	283
6.10	Segment-Coding Conclusion .....	284
7	Line History Generation and Machine-Segmenting.....	288
7.1	Line History Generation and Machine-Segmenting Introduction.....	288
7.2	Implementation of Line History Generation .....	290
7.2.1	Generation Algorithm Description.....	291
7.2.2	Generation Algorithm Example Run .....	295
7.3	Evaluation of the Line History Generation Algorithm.....	298
7.4	Method for the Evaluation of Machine Segmenting and Change-Identification Algorithms 301	
7.4.1	Precision, Recall and A/E Ratio .....	302
7.4.2	Calculating P-Values.....	304
7.4.3	Fit, Spread and Average Segment Size.....	306
7.4.4	Evaluating algorithms using Precision/Recall, Fit/Spread and P-Values.....	309
7.4.5	Presentation of Results .....	311
7.5	SCORE Machine Segmenting Generation and Evaluation Facilities.....	312
7.5.1	Generating Machine Segments.....	312
7.5.2	SCORE Analyser Segment Output .....	313
7.5.3	Machine-Segmenting Walkthrough .....	318

7.6	Description of Machine-Segmenting and Change-Identification Algorithms .....	320
7.6.1	Identification of interesting Changes by Random Selection .....	320
7.6.2	Identification of interesting Changes by Measuring Recent Modification of Files (File Metrics Method) .....	320
7.6.3	Identification of interesting Changes by Measuring Recent Modification of Lines (Line Metrics Method) .....	322
7.6.4	Generation of Segments using Line Histories (LH Method) .....	323
7.6.5	Generation of Segments using Line History Graphs (LH-Graph Method) .....	325
7.7	Evaluation of Machine-Segmenting and Change-Identification Algorithms .....	329
7.8	Description of LH-Graph Extension Algorithms .....	330
7.8.1	Compile Filter Algorithm .....	331
7.8.2	Small-Segment Filter algorithm .....	331
7.8.3	Short-Lifespan Inclusion Algorithm .....	332
7.8.4	Line-History Friend Algorithm .....	332
7.8.5	Line Proximity Algorithm .....	334
7.8.6	Code Parsing Algorithm .....	335
7.8.7	Text Similarity Algorithm .....	335
7.8.8	SimProx Algorithm .....	336
7.8.9	Visit Expressions .....	336
7.8.10	SimProx and Visit Expressions Combined .....	338
7.9	Evaluation of LH-Graph Extension Algorithms .....	338
7.10	Evaluation of Machine Segmentation, Change-Identification and Extension Algorithms on an independent data set .....	340
7.11	Performance of LH-Graph for different types of Problems .....	343
7.11.1	Well-identified Changes (true positives) .....	343
7.11.2	Not identified (false negative) .....	347
7.11.3	Wrongly identified (false positives) .....	350
7.12	Limitations of the evaluation method .....	352



7.12.1	Subjectivity and errors in the manual segmenting of Project Histories .....	352
7.12.2	Non-generality of analysed Project Histories .....	353
7.13	Proposed Extensions.....	354
7.13.1	Reconsidering Small Segments .....	354
7.13.2	Segment Line History Overlap.....	354
7.13.3	Dynamic analysis .....	355
7.13.4	Student Annotation / Feedback.....	355
7.14	Line History Generation and Machine-Segmenting Conclusion .....	355
8	Conclusion.....	358
8.1	Foreword.....	358
8.2	Findings and Outcomes.....	359
8.2.1	A Grounded-Theory based method of source-code level Project History analysis ....	359
8.2.2	Software-Engineering-based approaches to Project History analysis .....	360
8.2.3	Issues in student Computer Graphics programming .....	361
8.2.4	A model of Student Problem-Solving during Computer Graphics Programming .....	366
8.2.5	Spatial Programming is difficult .....	368
8.2.6	Students use different solution approaches for different three-dimensional spatial programming tasks .....	369
8.2.7	Development of a source code data collection and analysis toolkit .....	370
8.3	Significant Contributions.....	370
8.3.1	A new method of source-code level Project History analysis for Computer Science Education research .....	370
8.3.2	A better understanding of student Computer Graphics programming and the role of spatial programming.....	371
8.4	Future Directions .....	371
8.4.1	Proposal for a new topic of Computer Graphics Education: Visio-Spatial Programming and Debugging .....	372
8.4.2	Future Research and Development .....	377
8.4.3	Broader applications of the SCORE analysis toolset .....	378

8.5	Concluding Remarks.....	380
References .....		383
9	Appendix .....	398
9.1	Glossary of Terms.....	398
9.2	Final Ethics Report .....	401
9.3	Methodology.....	405
9.3.1	Qualitative and Quantitative Research Approaches.....	405
9.3.2	Comparison of Qualitative Research Methods .....	406
9.3.3	Defining Grounded Theory .....	407
9.3.4	Analysis of Student Perceptions .....	415
9.3.5	Reflection Questions .....	425
9.3.6	Description of Assignment Specifications .....	430
9.4	Literature Review .....	435
9.4.1	Factors and Facets of Spatial ability.....	435
9.4.2	Spatial Ability and Performance.....	439
9.4.3	Improving Spatial Ability .....	440
9.4.4	Snapshot Approaches .....	442
9.4.5	Review of Computer Graphics Literature .....	443
9.5	The SCORE Toolkit.....	458
9.5.1	Preparation of SCORE Data .....	458
9.5.2	SCORE Implementation Overview.....	459
9.5.3	A Short SCORE Manual.....	467
9.6	Change-Coding Method .....	518
9.6.1	Description of the Initial Classification Scheme.....	518
9.6.2	Raw Coding Data .....	520
9.7	Segment-Coding Method.....	522
9.7.1	Generation Algorithm Pseudo-Code .....	522

9.7.2	Performance of Line History Generation algorithm using different settings with LH-Graph generation.....	524
9.7.3	Generation Algorithm Full Evaluation Data .....	526
9.7.4	Example of Image/PDF Change browser output.....	547
9.7.5	Measuring Segment Significance: Time Taken versus Number of Changes .....	554
9.7.6	Segment Reports / Memos .....	560
9.7.7	Analysis of Segment Contents.....	627
9.7.8	Analysis of Segment Features .....	802
9.8	Machine Segment Generation .....	855
9.8.1	Description of data used in Evaluation .....	855
9.8.2	Deconstructive Approach to discovering good algorithm settings.....	858
9.8.3	Ensuring Consistent Results .....	859
9.8.4	Machine-Generation Algorithm Pseudo-Code.....	860
9.8.5	Format of Machine-Generation Method Evaluation Data.....	868
9.8.6	LH-Graph Distance/Proximity and Modification Settings .....	872
9.8.7	Machine-Segmenting Algorithm Results.....	877
9.8.8	LH-Graph Extension Algorithm Pseudo-Code .....	884
9.8.9	Evaluation of LH-Graph Extension Algorithms.....	895
9.9	SGL Parser .....	908
9.9.1	SGL Parser Manual .....	908
9.9.2	Practical 8 Worksheet – Introduction to the SGLParser .....	912
9.9.3	Practical 9 Worksheet – Transformations with the SGLParser .....	916
9.9.4	Practical 10 Worksheet – Projections with the SGLParser .....	921

## VI List of Figures

Figure 1: Computer graphics then (Spacewar, 1962) and now (Star Wars Champions of the Force) ..	41
Figure 2: A student working hard on his Computer Graphics assignment .....	42
Figure 3: 3DGT - A Self-Training Tool for Learning 3D Geometrical Transformations, part of CGGEMS .....	54
Figure 4: A Project History consists of N-1 Changes, each of which consists of all the Modifications from one Version to the next.....	86
Figure 5: Relationship between Grounded Theory entities; several codes form a category and analysis of categories leads to the discovery of themes underlying the data (adapted from Saldaña (2009)).....	90
Figure 6: Total Time / Total # of Changes Boxplots for Assignment 1 .....	109
Figure 7: Total Time / Total # of Changes Boxplots for Assignment 3 .....	110
Figure 8: The main SCORE interface .....	126
Figure 9: Relationship between SCORE, the student and the researcher .....	129
Figure 10: The SCORE Diff View on the left, a close-up of the connector pane on the right .....	131
Figure 11: The bottom row shows the Arbitrary Diff View's controls .....	132
Figure 12: The Category View, the Note View and the Note Summary View.....	133
Figure 13: The SCORE Editor .....	135
Figure 14: The screen capture window is shown in the top-right corner; the window is movable ...	136
Figure 15: The Screen-Capture View at four different Changes .....	136
Figure 16: Mock-up of the SCORE Plug-in Commenting Extension; A close-up on the left, as part of an Eclipse distribution on the right.....	138
Figure 17: Change-Coding View (left) and Quick-Coding View (right) .....	156
Figure 18: A graph mapping rolling averages of the number of Changes belonging to different Coding classification categories .....	157
Figure 19: Coding Metrics View for a single dimension.....	158
Figure 20: Coding using the Quick Coding and Coding Views.....	159
Figure 21: Example of modifications affecting the line marked 'L', including addition, deletion and modification to the line text marked as L-prime and L-prime-prime .....	174
Figure 22: Example of the Line History table data structure associated with the line shown in Figure 21 .....	175
Figure 23: A line mutation changes the line's text, as marked in yellow.....	176
Figure 24: A Line History composed of several mutations, modifying the axis of a transformation .	177

Figure 25: Line History from John A1; without moved-line detection, the Line History loses four of its six entries .....	178
Figure 26: A program-flow problem is debugged through moving of the line "menu.setLocation(-1,-1)" .....	179
Figure 27: Example from Michael A1. Without ghost detection, the line history would be split into four separate Line Histories .....	180
Figure 28: Change Browser and Details Window.....	181
Figure 29: Two examples of Details Window Views; moved lines are indicated via grey text and arrows, mutated lines using green text, deleted lines using red text and added lines using blue text .....	182
Figure 30: The Change browser in Details display mode (the right-hand side panel is colored pink due to the Changes having been 'selected' by being surrounded with brackets in the input window) ...	183
Figure 31: The Line History View with the Line History's states (1) the categorisation panel (2) and the summary and additional controls panel (3).....	184
Figure 32: Line History in detail .....	185
Figure 33: Mutant Filter with some items removed for clarity .....	186
Figure 34: Line History implementing an upper arm transform for the star jump animation .....	192
Figure 35: Implementation of the Star Jump animation at Changes 456-459.....	192
Figure 36: The Segment-Coding View, showing the database summary (top-left), the Segment summary (right) and the currently selected Segment's details (bottom-left).....	199
Figure 37: Segment Time Generation settings panel.....	201
Figure 38: Details of Segment JOHN_A3.SP.2 as found in the appendix.....	207
Figure 39: Example table summarising rotations produced during implementation of an animation .....	222
Figure 40: An example Line History .....	224
Figure 41: An example sequence of Changes showing modifications to source code for those Changes .....	225
Figure 42: The student's initial naïve construction at (182) .....	226
Figure 43: Line History showing the upper arm's assembly modifications from (28-182).....	226
Figure 44: The incorrect assembly becomes apparent at (310-317) after the student implements keys to rotate the head.....	227
Figure 45: First attempt at rotation about a joint at (321).....	227
Figure 46: Rotation of the head about the misplaced joint at (321) .....	228
Figure 47: Rotation about the origin at (334) .....	228

Figure 48: The corrected assembly, with the pre and post-translate adding up to the head's original translate (first commented-out translate).....	229
Figure 49: Effect of the moving of translation statements at (380 & 384, 385, 386, 387, 389).....	230
Figure 50: Experimentation with addition and removal of pre and post-translates in the placing of the left palm at (380 & 384, 385, 386, 387, 389).....	231
Figure 51: Adding a y and z dimension to the post-translate at (388) .....	232
Figure 52: Foot assembly at (404).....	232
Figure 53: Rotation of the foot caused by different assemblies at (404, 405, 407 and 411) .....	233
Figure 54: Enabling avatar movement at (434); move variables highlighted in yellow .....	234
Figure 55: An approach to proper whole-body positioning and orientation .....	234
Figure 56: The if block enables body-orientation, the else block enables orientation of the head only at (444).....	235
Figure 57: Effect of the body orientation on the head's rotation at (444) and (446).....	235
Figure 58: Individual rotation and translation of limbs to achieve rotation of the whole avatar .....	236
Figure 59: Rotation of the body by individual rotation and translation of limbs at (487).....	236
Figure 60: Construction method at (506) using a universal 'body' rotate value for whole-avatar rotation .....	237
Figure 61: Rotation of the head or body increments both the head's and body's rotate value at (506) .....	237
Figure 62: As the head rotates about its global 'joint' with the body, the body rotates away from the head at (505).....	238
Figure 63: Assembly at (522).....	239
Figure 64: Final construction at (532) .....	240
Figure 65: Rotation can be applied separately to the extremities and the body at (532).....	240
Figure 66: Animation attempt involving the implementation of an entirely new avatar assembly at (589) .....	241
Figure 67: The four phases of the Computer Graphics student programming problem-solving process .....	267
Figure 68: The 'Identify' phase of problem-solving .....	267
Figure 69: The 'Understand' phase of problem-solving.....	267
Figure 70: The 'Apply' phase of problem-solving.....	268
Figure 71: The 'Perfect' phase of problem-solving .....	268

Figure 72: Limbs break apart during the "Initial Anim Experiment" Segment due to a non-hierarchical assembly, leading the student to pinpoint the problem as relating to the assembly transformation calls in the 'Identify' phase .....	271
Figure 73: During the 'Understand' phase, the student trials different assembly approaches from (309-330) lead to different incorrect assemblies as shown, with the lower leg finally rotating correctly about the upper leg in the final Change at (331) .....	273
Figure 74: During the 'Apply phase' the approach developed while assembling the leg is applied to the arm, again requiring some experimentation in the 'Apply' phase .....	273
Figure 75: In an example of a 'Perfect' phase the arm is moved into place at Change (358), resulting in a correct avatar assembly as shown on the right .....	274
Figure 76: The currently implemented Change-Segment data model .....	279
Figure 77: A new, more fine-grained Modification-Change-Subsegment-Segment data model .....	280
Figure 78: Sequence of modifications to a line (the line in question is the 'circleCount' line) .....	295
Figure 79: Line History showing modifications to a line .....	295
Figure 80: Matching of MAINTAINED lines (step 1).....	296
Figure 81: Matching of Mutated Lines (Step 2) .....	296
Figure 82: Matching of <i>Ghost</i> lines (step 3).....	297
Figure 83: Generation of Added and Deleted lines (step 4) .....	298
Figure 84: The Machine Segmentation Settings Window.....	313
Figure 85: SCORE Analyser Machine-Segment Generation Views; Control Panel (upper-left), Debug View (upper-right), Segment list (lower-left), Navigation View (lower-right) .....	315
Figure 86: SCORE Analyser Machine-Segment Generation Debug View.....	316
Figure 87: SCORE Analyser Machine-Segment Generation Segment Navigation View.....	316
Figure 88: Excerpt from the Segment Timeline View .....	317
Figure 89: Representation of a Change in the Timeline .....	318
Figure 90: Diff algorithm comparing two Changes. ....	321
Figure 91: Graph showing the rolling average of modifications (deletions and additions) in the recent past.....	322
Figure 92: Line History 3 is used as a basis to generate the Segment 1,2,4,6 consisting of the Changes connected to that Line History .....	324
Figure 93: Changes are connected by edges for which Line Histories form junctions .....	326
Figure 94: Line Histories navigated as a graph, with navigation commencing from Change 2 .....	328
Figure 95: Changes surrounding the Segment's changes (marked 1) are marked by the algorithm .	333

Figure 96: Lines surrounding a line modified in the Segment document are searched for modification in the adjacent document.....	334
Figure 97: The different parts of adjacent-Change detection at which expressions take effect.....	337
Figure 98: Summary of Machine-Segment /Human-Segment overlap.....	344
Figure 99: Excerpt from Ida.GL.1 implementing Line Stipple .....	344
Figure 100: Excerpt 2 from Ida.GL.1 implementing Line Stipple .....	345
Figure 101: Summary of Machine-Segment / Human-Segment Overlap for Thomas.Sp.1.....	346
Figure 102: Excerpt from Thomas.Sp.1.....	346
Figure 103: Summary of Machine-Segment / Human-Segment Overlap for Christopher.GP.8.....	348
Figure 104: Excerpt from Christopher.GP.8. Changes belonging to the same machine-generated Segment are marked in the same colour.....	349
Figure 105: A false positive Machine Segment's Changes from Christopher A1.....	351
Figure 106: The four phases of the student Computer Graphics programming problem-solving process .....	367
Figure 107: The SGLParser main interface.....	373
Figure 108: SGLParser visualization of a transformation, showing the local coordinate systems associated with the individual transformations .....	374
Figure 109: The SGLParser with the projectionview window, showing the view volume.....	375
Figure 110: Example of a view volume shown in the projection view .....	375
Figure 111: Continual analysis of student programming, development and evaluation of teaching techniques.....	381
Figure 112: Relationship between Grounded Theory entities; several codes form a category and analysis of categories leads to the discovery of themes underlying the data (adapted from Saldaña (2009)).....	411
Figure 113: Graph of the Results .....	418
Figure 114: Graph of the categorisation of open-ended questions .....	419
Figure 115: On the left, a graph showing the evaluation method used in Computer Graphics Education papers; on the right, the categories into which reviewed papers fall.....	445
Figure 116: The eNVyMyCar Graphics Engine (vers. 15766) .....	451
Figure 117: SCORE Architecture / Component Overview .....	460
Figure 118: SCORE Data Model.....	462
Figure 119: SCORE User Interface Model .....	463
Figure 120: SCORE Event-Dispatching and Handling Mechanism .....	464
Figure 121: Versioning and History Data Model.....	467



Figure 122: Relationship between SCORE, the student and the researcher .....	469
Figure 123: The Main View includes the central Diff View and Navigation Bars and Info Panels as well as the "Additional Components View" which includes additional views and components .....	471
Figure 124: The Diff Panes and the Connector Pane which shows which lines from the old version lines from the new version map to.....	472
Figure 125: The Modification Summary panel displays all mutation, addition, deletion and ghost modifications occurring between the two versions .....	473
Figure 126: The info panels which describe the current and previous version shown in the Main Diff view .....	473
Figure 127: The scrollbars are used to select the version to be displayed in the main Diff view .....	474
Figure 128: Deleting a group.....	474
Figure 129: Creating a mutant connection .....	475
Figure 130: Creation of a group (Left); the resulting group (Right).....	475
Figure 131 The "Additional Components" pane .....	476
Figure 132: The Timeline View.....	477
Figure 133: Representation of a Change in the Timeline .....	477
Figure 134: The Timeline Details view .....	478
Figure 135: The Notes pane displaying notes for the version and file as well as the overall note (Left); the summary view displaying all of the version notes (Right). .....	479
Figure 136: The Change Browser displays notes on the right-hand side.....	480
Figure 137: The Bookmark View .....	480
Figure 138: The Categorising View.....	481
Figure 139: The "Compile" tab in the "Additional Components" pane contains all of the compilation, editing and screen capturing functionality .....	483
Figure 140: The Screen Capture window as part of the application (Left) and a close-up (Right).....	484
Figure 141: The Code Editor.....	485
Figure 142: The Line History view .....	486
Figure 143: A close-up of the modifications of a Line History as shown in the Line History view .....	487
Figure 144: Close-up of a Line History's time data, shown in a text field at the top of each Line History's description .....	487
Figure 145: The Filter panel (on the right of the Line History View) allows filtering and reordering of Line Histories.....	488
Figure 146: The Change Browser view.....	489
Figure 147: A close-up of modifications as summarised in the Line History view.....	489

Figure 148: The hotbuttons panel .....	490
Figure 149: On top modifications as displayed in the main Change Browser view, on the bottom the same modifications as displayed using the Details view .....	491
Figure 150: The Change Browser set to 'Graphic' display; it displays the 'Detail View' for each Change .....	492
Figure 151: The generated PDF document containing the Details view for every Change .....	493
Figure 152: The Change-Coding and Quick-Coding panels .....	496
Figure 153: The set-up panel for the graph .....	497
Figure 154: A graph mapping rolling averages of the number of Changes belonging to different Change-Coding classification categories.....	498
Figure 155: Change-Coding Metrics View for a single dimension .....	499
Figure 156: The Segment-Coding View .....	499
Figure 157: The Segment Version view.....	500
Figure 158: A close-up of the Segment Tree view .....	501
Figure 159: A close-up of the Segment List view .....	502
Figure 160: A close-up of the Segment Details view .....	502
Figure 161: An extract of a generated text file containing Segment descriptions .....	503
Figure 162: The Time Data Generation Settings panel .....	504
Figure 163: The Machine-Segmenting Settings panel .....	506
Figure 164: the Segmentation Controls panel .....	507
Figure 165: SCORE Analyser Machine-Segment Generation Views; Control Panel (upper-left), Debug Console (upper-right), Output console (lower-left), Links View (lower-right).....	509
Figure 166: Machine-Segment Generation Debug Console .....	510
Figure 167: Machine-Generation Links View .....	511
Figure 168: The Timeline view; the top row is shows human-identified Segments, the bottom two rows show the Segmenting created by different runs of a machine algorithm .....	511
Figure 169: Evaluation data in HTML format .....	513
Figure 170: Close-up of machine-generated segment descriptions from the HTML Evaluation data document.....	514
Figure 171: Human-Identified Segment to Machine-Identified Segments.....	514
Figure 172: Percentage of Selected Changes to Recall and Precision (RecPrecSoFar) .....	515
Figure 173: Recall to Precision (RecToPrec).....	516
Figure 174: Recall and Precision by % of total Changes selected (MachineOccsToRecPrec) .....	517
Figure 175: Evaluation Summary Statistics.....	517

Figure 176: Project Summary Evaluation Statistics .....	518
Figure 177: Boxplot of Average Time by Problem Difficulty.....	556
Figure 178: Boxplot for Size ~ Problem Difficulty .....	557
Figure 179: Initial approach to rotating objects around their parent at (1141).....	628
Figure 180: Incorrect rotation of an object based on rotating its lower and upper points separately .....	629
Figure 181: New approach to rotating objects at (1146) .....	630
Figure 182: Rotation of a child object at (1172) .....	630
Figure 183: State of the source code implementing the 'temp' variable at (1174).....	632
Figure 184: Initial child-parent rotate implementation at (658) .....	633
Figure 185: Line History for the calculation of the object centre; at (673) the student changes to utilising the lower-left corner instead .....	633
Figure 186: Storage of the new position causes loss of precision in (673).....	634
Figure 187: The calculated distance is not stored, meaning it is recalculated every time at (675) ...	635
Figure 188: The correction at (686) only calculates the distance when the object is moved in relation to the parent.....	636
Figure 189: Line History showing the development of the angle calculation .....	637
Figure 190: Initial child-parent rotation implementation at 834.....	637
Figure 191: At (845) The student attempts to store the angle between parent and child when the child is first added to the parent.....	638
Figure 192: Student attempts to calculate the parent-child angle inside the display function at (870) .....	639
Figure 193: At (876) the child's coordinates are changed as it is rotated, meaning the angle is correctly calculated.....	639
Figure 194: At (931) the parent's centre is incorrectly set to equal the child's position.....	641
Figure 195: The student's final solution at (943) .....	641
Figure 196: The student stores parent and child centres separately .....	642
Figure 197: The Scene Tree implementation, based on lecture notes, supports hierarchical transformations .....	643
Figure 198: Change 223, Incorrect reversal of y-translate sign (y -> -y).....	644
Figure 199: Excerpt of the Initial naive assembly at (225).....	644
Figure 200: Line History for modifications to the right upper leg's translate during naive assembly	644
Figure 201: Naive assembly becomes apparent when the upper leg rotation leaves the lower leg unaffected at (229-232) .....	645

Figure 202: Excerpt from Christopher's hierarchical assembly at (255).....	645
Figure 203: The incorrect assembly is apparent at (704), the re-orientation of limbs to hide the broken assembly at (705).....	646
Figure 204: Rotation around own centre (118), around origin at (119).....	647
Figure 205: Experimentation with translate statements for positioning the head joint from (120-123) .....	648
Figure 206: Effect of different attempts at positioning the head joint at (120-123).....	648
Figure 207: Incorrect assembly at (126) .....	649
Figure 208: Experimentation with the head's transform from 119-122 .....	650
Figure 209: The student discovers the correct assembly method.....	651
Figure 210: Assembly implementation at (134).....	652
Figure 211: Error in (134) caused by pre and post-translates cancelling each other out.....	652
Figure 212: Timeline for Changes implementing the proper assembly method .....	653
Figure 213: At Change 10, the student creates a hierarchical assembly but incorrectly translates the arm back to the origin ( $z = -1.5$ ) and up ( $y = 1.5$ ).....	654
Figure 214: Line History showing the modification to the lower arm's translate statement from 13-18 .....	654
Figure 215: Effect of experimentation with the lower-arm translate statement.....	654
Figure 216: At (20), the student incorrectly copies the upper arm's (0, 1.0, 1.5) translate in front of the lower arm. ....	655
Figure 217: Effect of incorrect translations at (24,25,26) and correction at (27).....	655
Figure 218: Line History containing the right arm's translate statement.....	655
Figure 219: A rotation used to transform the left arm's assembly transformations to produce the right arm .....	656
Figure 220: Creating the arm by rotating it rotates the arm's local axes which will cause transformations based on global axes to not have the desired effect.....	656
Figure 221: Line History containing the leg translate statement .....	657
Figure 222: Incorrect translations during assembly from (35-44), with the final correct translation at (45) .....	658
Figure 223: A rotate command added in front of the leg's assembly block to enable leg rotation ...	659
Figure 224: A view of the incorrect rotation of the leg about the origin at (53) .....	659
Figure 225: The two assemblies (first introduced at (310) and (311)) the student switches between from (310-319).....	660
Figure 226: The steps of the student's construction of the left leg from (309-331) .....	661

Figure 227: At (327) the student adds a second translate call, mirroring the first .....	662
Figure 228: Proper assembly of the upper leg at (330) .....	662
Figure 229: Screen captures of the resulting incorrect limb rotation at (336).....	663
Figure 230: The almost-correct arm construction at (335), foiled by the upper arm's additional (180, 1,0,0) rotate call.....	663
Figure 231: The use of +z instead of -z values leads to the affected right arm's limbs (lower arm and palm) being translated onto their parent. Screen captures of the construction at (340, 342).....	664
Figure 232: Changes (345), (347), (348), (350), and (351) (which is identical to 348).....	665
Figure 233: The student's initial assembly is already properly hierarchical.....	666
Figure 234: The student incorrectly adds an x-translate to the assembly .....	666
Figure 235: At (83), the incorrect centres of rotation (incorrect joints) become apparent .....	667
Figure 236: Screen capture of work on arm rotation at (84-88, 90-91, 95-103) .....	667
Figure 237: The upper arm's pre/post translates added to the upper arm rotation assembly (86) ..	668
Figure 238: Assembly at (91) after removal of the y-translate.....	669
Figure 239: Proper arm construction at (98) .....	669
Figure 240: Leg assembly at (121) and (122) .....	670
Figure 241: Head assembly at (126) and (127) .....	670
Figure 242: Timeline for Ida's proper assembly of avatar .....	670
Figure 243: The student's initial naïve construction at (182) .....	671
Figure 244: The upper arm's assembly modifications from (28-182).....	671
Figure 245: The incorrect assembly becomes apparent at (310-317) after the student implements keys to rotate the head.....	672
Figure 246: First attempt at rotation about a joint at (321) .....	672
Figure 247: Rotation of the head about the misplaced joint at (321) .....	673
Figure 248: Rotation about the origin at (334) .....	673
Figure 249: The corrected assembly, with the pre and post-translate adding up to the head's original translate (first commented-out translate).....	674
Figure 250: Experimentation with addition and removal of pre and post-translates in the placing of the left palm at (380 & 384, 385, 386, 387, 389).....	675
Figure 251: Effect of the moving of translation statements at (380 & 384, 385, 386, 387, 389) .....	676
Figure 252: Adding a y and z dimension to the post-translate at (388) .....	677
Figure 253: Foot assembly at (404).....	677
Figure 254: Rotation of the foot caused by different assemblies at (404, 405, 407 and 411) .....	678
Figure 255: Enabling avatar movement at (434); move variables highlighted in yellow .....	679

Figure 256: An approach to proper whole-body positioning and orientation .....	679
Figure 257: The if block enables body-orientation, the else block enables orientation of the head only at (444).....	680
Figure 258: Effect of the body orientation on the head's rotation at (444) and (446).....	680
Figure 259: Individual rotation and translation of limbs to achieve rotation of the whole avatar ....	681
Figure 260: Rotation of the body by individual rotation and translation of limbs at (487).....	681
Figure 261: Construction method at (506) using a universal 'body' rotate value for whole-avatar rotation .....	682
Figure 262: Rotation of the head or body increments both the head's and body's rotate value at (506).....	682
Figure 263: As the head rotates about its global 'joint' with the body, the body rotates away from the head at (505).....	683
Figure 264: Assembly at (522) .....	684
Figure 265: Final construction at (532).....	685
Figure 266: Rotation can be applied separately to the extremities and the body at (532).....	685
Figure 267: Animation attempt involving the implementation of an entirely new avatar assembly at (589).....	686
Figure 268: Initial Attempt at Animation Algorithm at Change 465 .....	687
Figure 269: The first working Animation Algorithm .....	688
Figure 270: Initial animation approach at (165) .....	689
Figure 271: Addition of iterating loop in animation function.....	690
Figure 272: Addition of a busy loop to 'time' the animation at (172) .....	691
Figure 273: Implementation of a 'timer' using the C++ Sleep function at (188) .....	691
Figure 274: The Sleep function is placed into the animation loop at (190).....	692
Figure 275: Correct animation algorithm implemented at (197), with the function drawing a single frame each time it is called by glutIdleFunc .....	692
Figure 276: Initially (153) the animation is triggered inside the function drawing the body .....	693
Figure 277: The first animation algorithm produces a single static rotation at (166).....	693
Figure 278: The first proper attempt at implementing an animation at (178).....	694
Figure 279: The function which increments the animation variable.....	694
Figure 280: An animation stored using a three-dimensional array.....	695
Figure 281: Animation algorithm in (81).....	696
Figure 282: At (81), the 'q' key is set to bind the animation function to the glutIdleFunc.....	696

Figure 283: Two of the animation's 'frames' as implemented in (538), consisting of a moving rectangle .....	697
Figure 284: The frame-drawing mechanism, implemented inside the display function at (534) .....	697
Figure 285: The 'timer' function moves the animation to the next frame after 'frame_delay' milliseconds.....	697
Figure 286: The student replaces rectangles with avatar assembly inside the frame functions at (549) .....	698
Figure 287: The final 'animation' algorithm consists of a single transformation applied to the avatar, active while a key is pressed .....	699
Figure 288: Gimbal Lock, produced by a 90-degree rotation about the local y-axis in the second step, leads to the local x-axis and z-axis rotating about the same global axis .....	701
Figure 289: Implementation of the Walking animation's arm movement at (210), (211) (two images) and (215).....	702
Figure 290: Development of the pickup animation's arm animation, bringing together the arms to pick up an object at with incorrect attempts at (370, 371 and 373) and the correct solution at (374) .....	704
Figure 291: Implementation of avatar arm movement for the Star Jump animation at (456-459)...	705
Figure 292: Modification of the lower arm's rotation from z (391) to x (404) to y (405) with a direction reversal at (406).....	707
Figure 293: Implementation of the full-body rotation of the avatar for the Waving animation; the initial attempt is a y-rotation at (702) modified to a z-rotation at (703) and finally to an x-rotation at (704).....	709
Figure 294: Implementation of the leg's movement for the walking animation, using an x-rotation at (541), modifying that to a y-rotation at (571) and finally correcting it to a z-rotation at (576).....	711
Figure 295 : Implementation of the rotation of the upper arm at (938, 971, 1194, 1195, and 1197) .....	712
Figure 296: Line History for the development of the upper arm transform .....	712
Figure 297: First attempt at arm movement for pickup animation at (202, 204, 205, 206, 207, and 209) .....	714
Figure 298: Changes 266, 268 and 269 illustrate the student's difficulty in correctly understanding the composition of rotations. ....	715
Figure 299: The sequence of rotations used to orient limbs, which causes gimbal lock when the Y-rotation value is 90 or 270 (or -90/-270) degrees. ....	715
Figure 300: First Line History showing experimentation with the lower arm rotation .....	716

Figure 301: Second Line History showing experimentation with the lower arm rotation .....	717
Figure 302: Separate grid lines show the local coordinate system for the lower and upper arms....	717
Figure 303: Changes addressing the gimbal lock problem by not making the first frame change the y-rotation to a value close to 90 degrees .....	718
Figure 304: A line history showing changes to a single transformation in the segment 336-351, involving 3 axis changes and 3 direction changes. ....	719
Figure 305: The torso rotation Line History .....	719
Figure 306: Development of leg movement from (320-323).....	720
Figure 307: Line History for the lower arm rotation.....	721
Figure 308: Some different stages of the swim animation's implementation at (463, 473, 476, 490, 491, 506, 507, 509, and 570) .....	722
Figure 309: Changes (454-456) involve one axis change and one sign change for a simple initial orientation .....	722
Figure 310: Line History for one of the swim animation's upper arm rotations .....	723
Figure 311: The removed arm transform. ....	724
Figure 312: Line History showing John's implementation of the first-person View; the three phases of development are highlighted in red (fixed), blue (linear movement) and green (spherical movement) .....	726
Figure 313: Line History for the gluLookAt call containing John's implementation of the first-person View .....	728
Figure 314: The initial attempt at (227) using fixed points, and the incorrect lookAt using spherical coordinates at (614), the correct lookAt with spherical coordinates removed from the lookAt at (615) .....	729
Figure 315: Line History showing a rotation applied in an attempt to 'orbit' the View .....	729
Figure 316: Line History showing the gluLookAt call for John's implementation of the minimap view .....	730
Figure 317: Line History containing the projection call for John's implementation of the minimap View .....	730
Figure 318: Source code for drawing the minimap background with a separate projection at (69) .	731
Figure 319: Line History containing Ida's gluLookAt implementation for the first-person View .....	733
Figure 320: Line History for the lookAt call implementing Ida's top-down View .....	733
Figure 321: Line History implementing the projection for Ida's minimap View.....	734
Figure 322: Line History showing the implementation of the lookAt call producing the third-person View .....	735



Figure 323: Line History showing the implementation of the Origin View .....	736
Figure 324: Line History for the lookAt function implementing Thomas's third-person View.....	738
Figure 325: Timeline showing Changes for the implementation of the third-person View .....	739
Figure 326: Line History showing the implementation of the lookAt call for the first-person View..	741
Figure 327: The first-person View at (533, 534-536, 537, 538, 539) .....	741
Figure 328: The top-down View at (550), (554), (556) and the final View at (560) after modification of the rotation call.....	742
Figure 329: Line History showing the rotate call designed to produce a top-down View by rotating the scene.....	743
Figure 330: Line History showing the implementation of the top-down View's projection .....	743
Figure 331: Line History showing development of the Third-person view.....	744
Figure 332: The Third-Person View in (193) and (248) .....	745
Figure 333: Line History showing development of the First-person view .....	745
Figure 334: Line History showing the development of a View that is manually moveable via the keyboard .....	746
Figure 335: Line History for the implementation of the First-Person View.....	748
Figure 336: Implementation of 'from' variables to store the avatar's (x,y,z) head position at (612).	749
Figure 337: The first-person View showing only the inside of the avatar's chest at (692, 712).....	749
Figure 338: Line History implementing the Third-Person View.....	751
Figure 339: The application's output at (688) with the minimap shown in the top-right corner .....	752
Figure 340: The door icon shape.....	754
Figure 341: Three steps of the implementation of the door shape from 342-347 .....	754
Figure 342: The initial incorrect code at 1034 and the almost-correct code at 1041 (The error lies in the first line).....	754
Figure 343: Development of an 'arrow' shape at Changes 1034, 1035, 1037, 1039, 1043 .....	755
Figure 344: Implementation of an arrow at 396, 397,399,403,408,415. Two completely incorrect changes create no arrow at 396 and 397. Notice start of an arrow at 399) .....	755
Figure 345 : Error-prone development of the delete icon (2103-2110, 2115, 2119) .....	755
Figure 346: Implementation of a simple two-dimensional grid, Changes Student grid 133-150, 164, and 227 .....	756
Figure 347: Changes to the hit condition (266,267,268,269,271, 275) .....	756
Figure 348: Positioning of text in the status indicator.....	759
Figure 349: Tweaking the text for one of the actions (1243, 1248-1250) .....	759
Figure 350: Tweaks to the glViewport size for 11, 401, 422-426, 428-439. ....	760

Figure 351: Tweaking of the lower arm's x-rotation in the swim animation .....	761
Figure 352: Positioning of buttons based on modulus mathematics at 137, 138, 139, 141, 142, 144, 146, and 158 .....	763
Figure 353 : Code for spatial buttons at 137, 139, 146 and correct code at 158 .....	763
Figure 354: Ida's approach to rotating child objects about their parent at (1146) .....	764
Figure 355: History Line shownign Ida's implementation of calculation of the angle between parent and child .....	765
Figure 356: John's implementation of parent-child angle calculation .....	765
Figure 357: Thomas's implementation of parent-child angle calculation .....	766
Figure 358: The final (incorrect) solution at (1126) which is part of the student's final submission .	767
Figure 359: History Line for the development of the conditional for testing the x-distance from avatar to object (a separate conditional exists for testing of z-distance).....	768
Figure 360: Line History showing the debugging using cout statements of the conditional statement testing distance between avatar and pickup object .....	769
Figure 361: Line History showing Thomas's implementation of the conditional for testing the distance between the avatar and the pickup object .....	769
Figure 362: Line History showing Thomas's implementation of the conditional for testing the angle between the avatar and the pickup object.....	771
Figure 363: Final attempt at using the distance and angle to object to determine whether to pick up an object at (817); removed in the next Change .....	771
Figure 364: A semicolon after the DELETE_MODE macro breaks the switch statement in (371) .....	776
Figure 365: The student code for creating parent-child relationships and the fix that would have resolved the bug in the student's code.....	777
Figure 366: Source code implementing the parent-child functionality at (166); last panel shows the fix at (325) .....	779
Figure 367: Incorrectly returning -1 inside the loop body .....	782
Figure 368: Implementation of a Scene Graph traversal function .....	784
Figure 369: The inefficient algorithm for switching between limbs .....	785
Figure 370: The 'if' clause is always triggered, making deselection of an object impossible. ....	789
Figure 371: The correct solution always executes the selection algorithm .....	790
Figure 372: Student attempt to add objects by initialising and rendering them in the mouse handler .....	792
Figure 373: Attempting to draw a wall using a hanging glBegin statement (1903) .....	793

Figure 374: The student's first attempt at drawing lines; different approaches at 74, 77, 81, and 83 .....	796
Figure 375: The source code of the initial lighting implementation.....	800
Figure 376: Lighting with large attenuation at (112) and (111) top-left and top-right, negative attenuation at (221) bottom-left and proper attenuation at (220) bottom-right.....	801
Figure 377: BoxPlot for Segment Log-Average Time per Change broken down by Category .....	805
Figure 378: LogAvgTime ~ Category Means Plot .....	805
Figure 379: Histogram and BoxPlot for Time Spent per Change .....	816
Figure 380: Histogram and BoxPlot for the Log of Time Spent per Change .....	817
Figure 381: Box plots for Average Time per Student for A1, A2.....	819
Figure 382: Box plots for Average Time per Student for A1 and A2 combined.....	819
Figure 383: Scatterplot showing the relationship between Average Time and Total time spent for individual students.....	820
Figure 384: Histogram and BoxPlot for Segment Average Time per Change .....	821
Figure 385: Histogram and BoxPlot for Segment Log-Average Time per Change .....	822
Figure 386: Histogram and BoxPlot for Segment Logarithm-transformed Time per Change .....	823
Figure 387: Scatterplot with Linear Regression Line .....	824
Figure 388: Line History for the line modified in 978 .....	825
Figure 389: List of Short View Changes.....	826
Figure 390: Christopher segment timeline for implementation of a first-person View .....	827
Figure 391: List of Long View Changes.....	828
Figure 392: Long Changes for the implementation of the pickup animation.....	829
Figure 393: Short Changes for the implementation of the pickup animation.....	830
Figure 394: Segment timeline for the implementation of John's first-person View .....	830
Figure 395: Short Changes for the implementation of John's first-person View .....	831
Figure 396: Line History for the line modified in (211).....	832
Figure 397: Long Changes for the implementation of John's first-person View .....	832
Figure 398: Segment timeline for the implementation of the Pickup animation.....	833
Figure 399: Long Changes occurring during the implementation of the Pickup animation .....	836
Figure 400: Short Changes occurring during the implementation of the Pickup animation .....	840
Figure 401: Segment Timeline for the implementation of Views.....	841
Figure 402: Long Changes occurring during the implementation of Views.....	842
Figure 403: Segment timeline for the implementation of avatar assembly.....	843
Figure 404: Long Changes occurring during implementation of avatar assembly .....	845

Figure 405: Short Changes occurring during implementation of avatar assembly Short Changes ....	846
Figure 406: Segment timeline for the implementation of the third-person View .....	847
Figure 407: Long Changes occurring in the implementation of a third-person View.....	848
Figure 408: Short Changes occurring in the implementation of a third-person ViewShort Changes	848
Figure 409: Segment timeline for the implementation of a walk animation .....	849
Figure 410: An example of a Long Change (remaining Changes are not shown as the source code is too verbose).....	850
Figure 411: Segment timeline for the implementation of the first-person View.....	850
Figure 412: Long Changes occurring during the implementation of the first-person View .....	852
Figure 413: Short Changes occurring during the implementation of the first-person View .....	852
Figure 414: Segment timeline for the implementation of the walk animation.....	853
Figure 415: Long Changes in the implementation of the walk animation.....	854
Figure 416: Short Changes in the implementation of the walk animation.....	855
Figure 417: Machine-Segmenting result.....	860
Figure 418: Main SGL Parser Interface .....	908
Figure 419: SGL Parser control panel.....	909
Figure 420: Marking lines; unmarked on the left, marked with a yellow marker on the right .....	910
Figure 421: The SGL Parser showing two transformation states of a cube .....	911
Figure 422: The workspace selection dialog.....	913
Figure 423: The main window for the SCPP application.....	914
Figure 424: The SGL Parser running a simple program.....	915
Figure 425: Successfully opened workspace directory .....	917
Figure 426: Hand Angles .....	917
Figure 427: A transformation coordinate line marker once it's been placed.....	918
Figure 428: Toggling coordinate lines for an OpenGL command .....	918
Figure 429: The state of the local coordinate system at the last transformation command for the lower hand. Notice the coordinate system has been transformed to the left of the origin. ....	918
Figure 430: The marker from the lower hand has been removed, and a marker has been added to the last rotate command of the upper hand. ....	919
Figure 431: The local coordinate system axes for the upper hand's glRotatef command. Notice it has been rotated about the x axis.....	920
Figure 432: The completed hand from the front.....	921
Figure 433: How your final hand should look (side view).....	921
Figure 434: The completed hand from behind. ....	921

Figure 435: Unmaximize .....	921
Figure 436: The SGL Parser interface.....	922
Figure 437: Zooming in and out using the w-s keys. The viewing volume is a green pyramid from the eye to the near plane (items in this part of the viewing volume would NOT be visible) and a blue pyramid from the near to the far plane.....	923
Figure 438: Rotating using the a-d keys.....	923

## VII List of Tables

Table 1: Descriptive Statistics for time spent and number of Changes in Assignment 1 .....	108
Table 2: Total Number of Changes and Time spent across students in Assignment 1 .....	109
Table 3: Descriptive Statistics for time spent and number of Changes in Assignment 3 .....	110
Table 4: Total Number of Changes and Time spent across students in Assignment 3 .....	110
Table 5: Descriptive data for Assignment 1 Project Histories.....	113
Table 6: Student completion of tasks for Assignment 1 .....	113
Table 7: Descriptive data for Assignment 3 Project Histories.....	115
Table 8: Student completion of tasks for Assignment 1 .....	116
Table 9: Implementation Statistics for the two projects which comprise the SCORE Analyser .....	127
Table 10: Example of Change-Coding data .....	159
Table 11: Percentage of Changes falling into the different Coding classification categories for the Action, Error and Problem dimension in Assignment 1; the three top categories (including the most Changes) are marked in blue .....	161
Table 12: Percentage of Changes falling into the different Coding classification categories for the Action, Error and Problem dimension in Assignment 3 .....	163
Table 13: Time/Changes spent on Error Changes divided by Time/Changes spent on Action Changes .....	164
Table 14: Time/Changes spent on Error Changes divided by Time/Changes spent on Problem Changes.....	165
Table 15: Example of calculated Segment Time and Modification Data .....	202
Table 16: Table showing all Segments (for all students/assignments) belonging to a category .....	208
Table 17: Table showing the size of the largest Segment per category for each student.....	211
Table 18: Table showing the "Initially Incorrect / Total" and "Rotation/Mod" metrics derived from the evaluation of rotation transformations.....	223
Table 19: Summary statistics for all implemented rotations for each animation .....	250
Table 20: Summary statistics for composite transformation rotations only for each animation; animations marked as N/A did not contain compound transformations.....	251
Table 21: Descriptive Statistics for Segment Average Time per Change broken down by Category .	276
Table 22: ANOVA Residuals.....	277
Table 23: ANOVA t and p-values .....	277
Table 24: ANOVA r-squared and p-value .....	277
Table 25: Tukey's Range Test (aka Tukey's Honest Significance Difference).....	278
Table 26: Precision / Recall for line history mutant generation .....	300

Table 27: Precision for line history ghost generation .....	300
Table 28: Ratio, spread/fit and p-value values averaged over all students and both assignments ...	312
Table 29: Best run of FileMetrics, LineMetrics, LH and LH-Graph algorithms.....	329
Table 30: Evaluation results with and without the compile filter .....	339
Table 31: Evaluation results of application of extension algorithms. Algorithms in red produce no improvement. ....	340
Table 32: Evaluation of algorithms on the independent data set .....	341
Table 33: Evaluation of the LH-Graph algorithm with and without compile filter .....	342
Table 34: Evaluation of LH-Graph extension algorithms on the independent data set (including CompileFilter in all runs).....	342
Table 35: Issues related to student problem-solving in Computer Graphics .....	362
Table 36: Student Concepts and Misconceptions in Computer Graphics .....	363
Table 37: Potential solution approaches to addressing student problem-solving issues .....	364
Table 38: Description of the model's phases of Computer Graphics student problem-solving.....	368
Table 39: Result of Likert questions; +++ agree strongly, ++ agree, + somewhat agree, / neutral, - somewhat disagree, -- disagree, --- strongly disagree.....	417
Table 40: Summary of results for the rank order question .....	422
Table 41: Implementation Statistics .....	459
Table 42: Example of calculated Segment Time and Modification Data .....	505
Table 43: Assignment 1 Overall Raw Coding Data.....	520
Table 44: Assignment 3 Overall Raw Coding Data.....	521
Table 45: Data for LH-Graph Generation with different line history generation settings.....	525
Table 46: Data for LH-Graph Generation with different line history generation settings.....	526
Table 47: Generation results for Assignment 1 using the settings mutant=0.1 ghost=X .....	526
Table 48: Generation results for Assignment 3 using the settings mutant=0.1 ghost=X .....	527
Table 49: Generation results using the settings mutant=0.1 ghost=X averaged across Assignment 1 and Assignment 3.....	528
Table 50: Generation results for Assignment 1 using the settings mutant=0.25 ghost=X .....	528
Table 51: Generation results for Assignment 3 using the settings mutant=0.25 ghost=X using the settings mutant=0.25 ghost=X.....	529
Table 52: Generation results using the settings mutant=0.25 ghost=X averaged across Assignment 1 and Assignment 3.....	530
Table 53: Generation results for Assignment 1 using the settings mutant=0.5 ghost=X .....	531
Table 54: Generation results for Assignment 3 using the settings mutant=0.5 ghost=X .....	532

Table 55: Generation results using the settings mutant=0.5 ghost=X averaged across Assignment 1 and Assignment 3.....	533
Table 56: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.1 dist=10 ...	533
Table 57: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.1 dist=10 ...	534
Table 58: Generation results using the settings mutant=0.5 ghost=0.1 dist=10 averaged across Assignment 1 and Assignment 3 .....	535
Table 59: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.25 dist=10 .	536
Table 60: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.25 dist=10 .	536
Table 61: Generation results using the settings mutant=0.5 ghost=0.25 dist=10 averaged across Assignment 1 and Assignment 3 .....	537
Table 62: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.5 dist=10 ...	538
Table 63: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.5 dist=10 ...	539
Table 64: Generation results using the settings mutant=0.5 ghost=0.25 dist=10 averaged across Assignment 1 and Assignment 3 .....	540
Table 65: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.1 dist=5 .....	540
Table 66: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.1 dist=5 .....	541
Table 67: Generation results using the settings mutant=0.5 ghost=0.1 dist=5 averaged across Assignment 1 and Assignment 3 .....	542
Table 68: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.25 dist=5 ...	543
Table 69: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.25 dist=5 ...	543
Table 70: Generation results using the settings mutant=0.5 ghost=0.25 dist=5 averaged across Assignment 1 and Assignment 3 .....	544
Table 71: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.5 dist=5 .....	545
Table 72: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.5 dist=5 .....	546
Table 73: Generation results using the settings mutant=0.5 ghost=0.5 dist=5 averaged across Assignment 1 and Assignment 3 .....	547
Table 74: Descriptive Statistics for Average Time by Problem Difficulty.....	556
Table 75: Average Time ~ Problem Difficulty ANOVA results.....	556
Table 76: Descriptive Statistics for Size ~ Problem Difficulty.....	557
Table 77: ANOVA for Size ~ Problem Difficulty.....	558
Table 78: Pair-wise t-test pooled standard deviation of Size ~ Problem Difficulty .....	558
Table 79: ANOVA of Total Time ~ Problem Difficulty .....	559
Table 80: List of Rotations implemented as part of work on the Walking .....	702
Table 81: List of Rotations implemented as part of work on the Pickup animation .....	704



Table 82: List of Rotations implemented as part of work on the Start Jump animation.....	706
Table 83: List of Rotations implemented as part of work on the Walking animation .....	707
Table 84: List of Rotations implemented as part of work on the Pickup animation .....	708
Table 85: List of Rotations implemented as part of work on the Wave animation.....	709
Table 86: List of Rotations implemented as part of work on the Walking animation .....	711
Table 87: List of Rotations implemented as part of work on the Pickup animation .....	713
Table 88: List of Rotations implemented as part of work on the Pickup animation .....	719
Table 89: List of Rotations implemented as part of work on the Walk animation.....	721
Table 90: List of Rotations implemented as part of work on the Swim animation .....	724
Table 91: Descriptive Statistics for Segment Average Time per Change broken down by Category .	804
Table 92: Descriptive Statistics for Segment Log-Average Time per Change broken down by Category .....	806
Table 93: ANOVA Residuals.....	806
Table 94: ANOVA t and p-values .....	807
Table 95: ANOVA r-squared and p-value .....	807
Table 96: Tukey's Range Test (aka Tukey's Honest Significance Difference).....	808
Table 97: Quartiles for Assignment 3.....	809
Table 98: Percentage of Short and Long Changes for the analysed Animation (left) and View (right) Segments.....	810
Table 99: Ratio of Short and Long Changes of Anim / View .....	810
Table 100: Descriptive Statistics for Time Spent per Change .....	816
Table 101: Descriptive Statistics for Log Time Spent per Change.....	817
Table 102: Total Time spent on A1 and A3 .....	818
Table 103: Quantiles for Average Time per student for A1, A3 and both assignments combined ....	818
Table 104: Descriptive Statistics for Segment Average Time per Change .....	821
Table 105: Descriptive Statistics for Segment Log-Average Time per Change .....	822
Table 106: Descriptive Statistics for Segment Size .....	823
Table 107: Residuals for Linear Regression Avg ~ size.....	823
Table 108: Coefficients for Linear Regression Avg ~ size .....	824
Table 109: Result of Linear Regression Analysis of AvgTime ~ Size.....	824
Table 110: Count of Long and Short Changes.....	825
Table 111: Segment timeline for the implementation of the pickup animation.....	828
Table 112: Number of Long and Short Changes for the implementation of the pickup animation ...	828

Table 113: Number of Long and Short Changes for the implementation of John's first-person View .....	831
Table 114: Number of Long and Short Changes occurring during the implementation of the Pickup animation .....	840
Table 115: Number of Long and Short Changes in the implementation of Views .....	842
Table 116: Number of Long and Short Changes occurring during implementation of avatar assembly .....	846
Table 117: Number of Long and Short Changes in the implementation of a third-person View .....	848
Table 118: Long and Short Changes occurring during the implementation of a walk animation .....	850
Table 119: Number of Long and Short Changes for the implementation of the first-person View ...	851
Table 120: Number of Long and Short Changes in the implementation of the walk animation .....	853
Table 121: Description of Project History Change data from Assignment 1 .....	856
Table 122: Description of Project History Change data from Assignment 3 .....	856
Table 123: Example Data for the first phase of the deconstructive approach .....	858
Table 124: Example Data for the second phase of the deconstructive approach .....	859
Table 125: Example Data for the third and final phase of the deconstructive approach .....	859
Table 126: Example of Evaluation Data for a run of the Machine-Segmenting algorithm for A1 .....	869
Table 127: Example of Evaluation Data for a run of the Machine-Segmenting algorithm for A3 .....	869
Table 128: Summary data from an example Segment generation run for Assignment 1 .....	870
Table 129: Summary data from an example sement generation run for Assignment 3 .....	871
Table 130: Ratio, spread/fit and p-value values averaged over all students and both assignments .	871
Table 131: Average Segment Size of segments making up the top % of Changes for A1 .....	872
Table 132: Average Segment Size of segments making up the top % of Changes for A3 .....	873
Table 133: A/E ratios for different distance and depth settings for A1 and A3 .....	874
Table 134: A/E ratios for different modification settings for A1 and A3. ....	875
Table 135: Summary Table for runs of the Random algorithm .....	877
Table 136: Summary Table for runs of the FM algorithm .....	877
Table 137: Summary Table for runs of the FM algorithm .....	879
Table 138: Summary Table for runs of the LH algorithm .....	880
Table 139: Summary Table for the evaluation of the LH-Graph algorithm using different distance settings .....	881
Table 140: Summary Table for the first phase of the evaluation of the LH-Graph algorithm using different modification settings .....	883

Table 141: Summary Table for the second phase of the evaluation of the LH-Graph algorithm using different modification settings .....	883
Table 142: Overall Summary of Compile-filter algorithm runs.....	896
Table 143: Overall Summary of Small-Segment filter algorithm runs .....	897
Table 144: Overall Summary of Short-Lifespan inclusion algorithm runs .....	898
Table 145: Overall Summary of Friend algorithm runs.....	898
Table 146: Overall Summary of Line Proximity algorithm runs .....	899
Table 147: Overall Summary of Code Parsing algorithm runs .....	900
Table 148: Overall Summary of Text Similarity algorithm runs .....	901
Table 149: Overall Summary of SimProx algorithm runs not using keepAlive .....	902
Table 150: Overall Summary of SimProx algorithm runs using keepAlive.....	903
Table 151: Overall Summary of SimProx algorithm runs using keepAlive and including proximity search for included Changes using the setting (Lev=0.75, Prox=10, Dist=1) .....	904
Table 152: Overall Summary of Expression extension runs.....	905
Table 153: Overall Summary of the combination of SimProx and Expression algorithms .....	906

# 1 Introduction

---

## 1.1 Research Context

Computer Graphics is a field of growing importance. Recent advances of Computer Graphics programming techniques and hardware have had a significant impact on the way in which we interact with computers. The internet will soon move to three dimensions, with major browser software developers having announced their intent to allow their browsers to deliver 3D content (Boutin, 2009) and companies such as Apple are already planning their next generation of three-dimensional user interfaces (PatentlyApple Staff, 2012). Advances in stereoscopic displays will cause this trend to accelerate, necessitating the move to user interfaces that blend seamlessly with the three-dimensional content and allow for the intuitive manipulation of three-dimensional objects. Three-dimensional (3D) technologies are also increasingly utilised to produce virtual representations of the world, such as Google's recent move to three-dimensional maps (*Google Maps 3d*, 2012; Hachman, 2012). Professionals too benefit from the advances in Computer Graphics. Pilot training has been shown to be more effective when incorporating Computer-Graphics driven simulation exercises (Hays, Jacobs, Prince, & Salas, 1992). This is also true of the training of other specialised life-saving professions such as surgery (e.g. Seymour et al., 2002) or fire-fighting (e.g. Stone, 2001). Modern engineering relies heavily on tools utilising Computer Graphics such as Computer-Assisted Design<sup>1</sup> (CAD) tools which streamline the design process, and virtual walk-throughs are replacing physical prototypes in submarine design, reducing costs and enabling faster development (Design World Staff, 2007).

According to experts such as Industrial Light & Magic's<sup>2</sup> Kim Libreri "another ten years from now, it's (real-time rendered graphics) just going to be indistinguishable from reality" (Hillier, 2012). The production of these Computer-Graphics-based systems will require software engineers who can apply a mix of Computer Graphics skills and more traditional Computer Science programming techniques. The training of such specialists in the higher-education context will fall to Computer-Graphics educators.

Computer Graphics programming is a young field even when compared to Computer Science as a whole. It entered its early period from 1970-1990 with revolutionary two-dimensional applications

---

<sup>1</sup> Computer Assisted Design (CAD) tool: A software application for computer-aided design and drafting.

<sup>2</sup> Industrial Light and Magic: A highly regarded motion picture visual effects company

such as SKETCHPAD (Sutherland, 1964) and Spacewar! (Graetz, 1981) moving beyond the text-only console paradigm (see Figure 1). Since computers were expensive and equipped with limited graphics facilities by today's standards, it was difficult to achieve three-dimensional graphics. Also, few software packages existed for creating graphics applications, and most of these were commercial and expensive (Hitchner & Sowizral, 2000). As a result, most educators created their own limited libraries for their students to use. This in turn meant that students had to start at the very bottom, learning primitive operations and algorithms. Computer Graphics was taught almost exclusively in two dimensions (2D), as a synthesis of image generation and image manipulation. The modern period (1990-today) is marked by a steep fall in the price of computers and a very rapid increase in the power of dedicated graphics hardware exceeding Moore's law, as well as the availability of free, open-source 3D Computer Graphics API's such as OpenGL<sup>3</sup>. Modern Computer Graphics courses can have students work in three dimensions, utilising shaders to achieve advanced texturing and lighting of objects. These are just some of the new topics made accessible through advances in Computer Graphics technology.



**Figure 1: Computer graphics then (Spacewar, 1962) and now (Star Wars Champions of the Force)**

This means that educators are faced with the challenge of producing new material for a rapidly changing field, with technological advances driving radical shifts in the types of Computer Graphics programming which are relevant in real-world projects and which can feasibly be taught to students. The advances have rendered old approaches dealing only with two-dimensional content incomplete. Computer Graphics education is still a young field in terms of education. This means that Computer Graphics educators lack the experience that comes with the large number of students who have passed through Computer Science labs for general CS courses. Also, while there is a considerable body of research examining first-year Computer Science Education, there is little comparable

---

<sup>3</sup> OpenGL: A standard specification defining a cross-language, multi-platform API for producing 2D and 3D computer graphics (<http://www.opengl.org/>).

research analysing Computer Graphics students. As a result, less is known about how to teach Computer Graphics programming, or what types of problems face when developing their first applications. Indeed, a literature review by the author (Wittmann, Kavakli-Thorne, & Bower, 2009) (see Appendix Section 9.4.5) showed that little of the research conducted in Computer Graphics Education applies a rigorous methodology. There is little to no formal work identifying problems in Computer Graphics Education or theory describing Computer Graphics programming.

This means that key questions regarding the student's Computer Graphics programming are left unanswered. What is the student in Figure 2 doing? What parts of the syllabus does he<sup>4</sup> struggle with? Which concepts does he need support with? Is he being taught Computer Graphics in a way which will set him on his way to become a successful Computer Graphics programmer? If these questions can be answered, this will allow for the development of a syllabus which addresses issues of particular challenge via well-crafted learning materials and scaffolding.



**Figure 2: A student working hard on his Computer Graphics assignment**

Many of the answers lie in the programming he is currently engaging in. His final submission will not contain any hint of many of the problems the student faces. Most of these problems will have been resolved prior to submission. Some of these problems will be minor and resolved with hardly an error while others will present significant stumbling blocks overcome only through hours of hard labour. Even those errors still present in the final submissions are mere snapshots of a long process of failed problem-solving. The final submission will contain only the result of a long programming process spanning tens of hours. Enough to assign a mark, but what remains in that final submission is only the tip of an iceberg. Below the waterline marked by the submission, in all those versions

---

<sup>4</sup> To avoid gender bias an effort has been made to alternate between the use of gender-specific pronouns except in cases where the pronoun refers to an actual person

implemented, revised, tested, revised again and sometimes deleted, lies the real heart of the student's problem-solving process.

Many different approaches could be applied in an attempt to uncover student problems. Subjective perceptions could be elicited in an interview. One challenging aspect of this approach is that it requires volunteer students willing to participate in the research. More importantly, would the participant student remember and be able to give a precise account of her problems? More recent problems would be remembered better than problems which occurred during the early parts of the assignment, and many other cognitive effects would serve to distort her recollection. Even assuming good recall, how much light could the student shed on her problem-solving process? Would she remember details such as the different approaches attempted in solving a problem or what the error underlying his attempt turned out to be? If the student never correctly resolved a problem, then she would be unable to do so as she would not know what the problem was. Even if she had resolved the problem she might misunderstand or misidentify the associated error.

A phenomenological<sup>5</sup> approach might be utilised in order to allow the researcher to operate in closer proximity to the student's problem-solving. The student could be asked to think aloud while completing a task, with both her actions and thoughts meticulously recorded using cameras, microphones and screen-capture software. Such an approach would require the researcher to be present for the many hours required for the completion of the task for each student, and the presence of the researcher and applied think-aloud protocols may well serve to change the student's work habits and problem-solving approaches from those the student would usually apply. Given the time-constraints which affect not only the researcher but also volunteer participants, such a study can only include a relatively small number of students and must utilise simplified task designs lacking external validity<sup>6</sup>, since having a student develop an entire assignment in this fashion is not feasible.

While both of these approaches can produce valuable data, they are both limited in their ability to access the student's actual real-world work on assessment tasks. An alternative approach is to capture student problem-solving data by producing a history of all of the student's programming actions. This approach is similar to that utilised by versioning systems<sup>7</sup> but instead of only versioning source code at explicit commits every single modification is stored, which will make the final Project History contain a complete transcript of the student's programming available to the researcher.

---

<sup>5</sup> Phenomenology: An approach used in the social sciences to eliciting, exploring and describing subjective experiences; in this case, used to elicit students' subjective experiences of learning programming.

<sup>6</sup> External validity: the degree to which it is valid to generalise the analysis results of an experiment or study

<sup>7</sup> Versioning System (also version control system): A system for the management of changes to documents, in this case source code. Most versioning systems maintain a history of these changes, storing a new version each time a change is processed.

Such an approach produces a large volume of data consisting of hundreds or thousands of versions, each associated with a source code documents comprising tens or hundreds of lines. Approaches to analyse such data in the literature includes the use of machine-approaches to mine compilation logs for errors (e.g. Jadud & Henriksen, 2009), providing insight into student syntax errors produced during the coding of the assignment. Another approach would be for the researcher to focus on abstract code quality metrics such as cyclomatic complexity<sup>8</sup>.

While this approach is effective in some domains, Computer Graphics programming does not only involve simple syntax/semantics errors. This means that compilation errors do not shed much light on the nature of student problems. The analysis of abstract metrics is an even coarser tool, providing less detail on the precise nature of student problems than an analysis of the final assignment would. Both can be valuable tools in understanding different aspects of student programming, but neither is appropriate for a detailed problem-level understanding of Computer Graphics programming.

## 1.2 Research Questions

A critical question is whether it is possible to develop a feasible method for analysis of source-code as is stored in a version control system. Such a method would allow for the forensic analysis of student programming and could uncover the problems faced by students during their Computer Graphics programming which in turn would help to provide guidance for Computer Graphics education and scaffolding for students, leading to better learning outcomes. This thesis has two central aims. The first aim of this thesis is to develop a method for the analysis of source code as is as is stored in a version control system. The second aim is to apply this method to the analysis of student problems in Computer Graphics programming.

The analysis of student problems in Computer Graphics programming in this research project places special emphasis on what shall be referred to as ‘spatial programming’. Spatial programming has different meanings depending on the research context, so it is important to clarify how it is used in this thesis. Spatial programming is used to refer to programming actions which involve two or three-dimensional space. An example would be drawing a rectangle. Another example would be applying a three-dimensional rotation to an object.

Addressing each of these aims is a substantial task in its own right, and both are inexorably connected. The developed method enables the analysis of Computer Graphics programming, whereas the analysis drives the development and also validates the method by demonstrating its

---

<sup>8</sup> Cyclomatic Complexity: A software metric measuring the complexity of a software program in terms of the number of linearly independent paths through the program’s source code ([http://en.wikipedia.org/wiki/Cyclomatic\\_complexity](http://en.wikipedia.org/wiki/Cyclomatic_complexity)).



effectiveness (or ineffectiveness) in providing insight into student programming and its problems. For this reason, the development of the method and the application of the method to the investigation of Computer Graphics programming will be discussed hand-in-hand. Based on these aims, the following research questions are proposed (In the remainder of this thesis, research questions and hypotheses will be referenced by their number; for example, a reference to the first research question will read 'RQ1':

**RQ1) How can the analysis of students' coding as captured in a version control system be used to analyse student programming problem-solving?**

**RQ2) What kinds of problems do students learning Computer Graphics Programming experience?**

**RQ3) Is 'spatial programming' a difficult area/topic of Computer Graphics programming for students?**

### **1.3 Contributions of the thesis**

This research project intends to make two significant contributions to the literature. The first is to provide a better understanding of the problems Computer Graphics students face, including identifying the types of problems faced by students as well as the role played by spatial programming. As of yet there is little work on identifying such problems in Computer Graphics Education (Computer Graphics Education) research. Such understanding would be significant as it would expand the state of knowledge in Computer Graphics Education providing a stronger theoretical basis for future research. A better understanding of student problems may also help provide better scaffolding to enhance student learning and to design more effective teaching materials.

The second contribution this thesis aims to make is the development of a method for the analysis of source code as stored in a version control system, including both software to capture and enable analysis of such data as well as a method of processing resulting data and developing theory from it. In the current literature, the most closely related approaches of Project History analysis are reliant on machine- processable output data (usually compilation error data) which is not available when the area of investigation falls outside the simple syntax and semantic errors of interest in the analysis of introductory Computer Science coursework. The novel method proposed in this thesis is applicable to arbitrary programming. It will provide a new tool for the exploration of student programming in any Computer Science domain and for projects of any shape or size.

## 1.4 Structure of the thesis

This chapter presents the research problem which is addressed in this thesis which is then distilled into the three research questions. The chapter also lists the contributions to the fields of Computer Graphics Education and Computer Science Education this thesis intends to make. The chapter also provides an overview of the structure of the thesis.

The next chapter (Chapter 2, Literature Review) will present literature relating to Computer Graphics Education and Spatial Ability, exposing the current state of Computer Graphics Education research as well as the challenges relating to spatial programming. A review of methods of investigation of student programming problems offers a background to the method developed in this research project, including details on its novel features. A review of software-engineering literature is provided to ground the methods developed for detailed investigation and Machine-Segmenting of Project Histories. Chapter 3 provides a brief review of the Grounded Theory methodology which underlies the analysis methods developed. This is followed by a description of the rationale that drove the development of the two developed analysis methods and the methodology that underlies the development of these methods, as well as a brief overview of the methods themselves. The proposed methods consist of both a theoretical description describing how to apply the method as well as a software toolset which is used to capture data (Project Histories) and which enables the application of the methods through method-specific functionality. The data gathered as part of this research process as well as the method by which a subset of this data was selected for analysis is described in Section 3.6.

This toolset's Project History capturing functionality and its basic features is described in Chapter 4 (The SCORE Toolkit). The recording of Project History data is achieved via the Eclipse SCORE plug-in, whereas the SCORE Analyser facilitates analysis and the application of developed methods.

The first developed method (Change<sup>9</sup>-Coding) utilising the SCORE Analyser is described in Chapter 5 (Change-Coding). The SCORE Analyser functionality for the application of Change-Coding is described, followed by an overview of the coding scheme and its development. Results from the application of the Change-Coding method are analysed quantitatively, providing some insight into the distribution of student efforts during their work on assignments. However, the method was found to produce results mainly reflective of the assignment specification rather than of student

---

<sup>9</sup> In the body of this thesis the term 'version' and 'Change' are often used interchangeably; the difference is that whereas the term 'version' identifies a particular version of source code, the term 'Change' refers to the set of modifications which transform one version (the old version) into a new version. The first developed method is called the 'Change-Coding' method since individual Changes are coded during analysis.

problem-solving; the quantitative analysis failed to provide insight into the way in which students problem-solve.

Limitations of the Change-Coding method led to the development of a new method (Segment-Coding) which is described in Chapter 6 (Segment-Coding). The primary difference between the Segment-Coding method and the Change-Coding method is that versions contained in Project Histories are first sorted into 'Segments' of related versions. These Segments are then coded and analysed. The way in which the SCORE Analyser functionality enables a more in-depth analysis of source code is presented, followed by a description of the theoretical application of the Segment-Coding method including the coding scheme used and the way in which data is subdivided for analysis. This is coupled with a description of how The SCORE Analyser functionality enables the application of the method. A list of coded Segments produced through the analysis of Project Histories is presented, after which the analysis of Segments focusing first on Segment contents and then on Segment features is discussed. The *Analysis of Segment Contents* focuses on a mainly qualitative analysis of problem-solving through examination of modifications contained in Segment versions intended to reveal the nature of student problems; the analysis also includes a quantitative analysis of three-dimensional transformations to investigate the difficulty or lack thereof with which students tackle spatial programming. This analysis leads to the identification of student issues with Computer Graphics programming. A phase model of student problem-solving based on these issues is proposed and grounded through an example showcasing a student moving through the phases of the model. The analysis of Segment contents is followed by a largely quantitative analysis of Segment features, focusing on the time taken per implementation of version for versions belonging to Segments. This analysis focuses on the difference in problem-solving approach taken by students when working on problems falling into different classification categories.

Chapter 7 (Line History Generation and Machine-Segmenting) describes the implementation and presents evaluation of an algorithm for the generation of histories of lines (Line Histories) on which much of the functionality provided by the SCORE Analyser is based. The chapter then describes several methods of Machine-Segmenting based on the concept of Line Histories, as well as a method of evaluation of such methods developed as part of this research project. These algorithms group related versions into Segments without human intervention, thereby enabling less time-intensive application of the Segment-Coding method in future work which addresses the Segment-Coding method's primary limitation.

As such, research questions relating to the analysis of Computer Graphics Education problems are addressed in the Change-Coding method (Chapter 5) and Segment-Coding (Chapter 6) chapters. The

most significant contribution is made by the Segment-Coding chapter. The research question relating to the development of the analysis method is addressed in the chapters on the software developed as part of this research project (Chapter 4), Change-Coding (Chapter 5), Segment-Coding (Chapter 6) and Line History Generation and Machine-Segmenting (Chapter 7).

The Conclusion (Chapter 8) presents the findings from preceding chapters, detailing how these findings serve to answer the research questions posed at the outset of this thesis as well as highlighting the significant contributions to the body of research made by this thesis. A new topic of Computer Graphics Education called 'Visio-Spatial Programming and Debugging' based on the findings is proposed; this new topic is intended to provide support for student weaknesses and problems identified during analysis. Other potential applications of the SCORE software toolset and analysis method are discussed. Finally future work on the different strands of the research presented in this thesis is discussed.

A glossary of the most important terms introduced in the body of this thesis is presented in Appendix Section 9.1.

In the body of this thesis, capitalization and quotation marks are sometimes used in unorthodox ways. For example, in later chapters the names of categories are capitalized. When category names consist of more than two words, all words are capitalized. Sometimes quotation marks are used to further emphasize a category name or other term. This was done in order to help the reader better understand the structure of sentences which refer to categories or other codes.

For the sake of completeness this thesis contains many forward references. The reader is advised not to follow these forward references on the first reading, as some of the finer details regarding the forward-referenced section will become more readily apparent only after reading subsequent chapters.

# 2 Literature Review

---

## 2.1 Literature Review Introduction

This literature review seeks to provide background on the problem domain of Computer Graphics Education (Computer Graphics Education) and spatial programming, as well as on methods of analysis in both Computer Science Education (Computer Science Education) and software engineering which share commonalities with the analysis method developed in this research project.

In order to provide background for **RQ1** regarding the problems faced by novice Computer Graphics students, the Computer Graphics syllabus and the OpenGL API (the environment in which most students learn Computer Graphics programming) are discussed in Sections 2.2.1 and 2.2.2. In addition, proposals by Computer Graphics educators to extend the syllabus to better support student learning are examined in Section 2.2.3. Some existing learning tools are also examined in Section 2.2.4 to demonstrate that Computer Graphics educators intuitively seek to scaffold spatial programming, indicating that they identify this topic as particularly difficult.

Section 2.3 presents literature regarding spatial ability to provide background on spatial programming as relates to **RQ2**. Descriptions of spatial ability in the psychology literature presented in Section 2.3.2 serve to demonstrate that spatial ability underlies spatial programming. The role of spatial ability in learning performance is examined in Section 2.3.3, whereas the literature presented in Section 2.3.4 will examine literature on approaches for improvement of spatial ability. This is relevant since if this research project identifies problem areas relating to spatial programming, this means that spatial ability interventions may be beneficial for Computer Graphics students.

Having discussed the theoretical background motivating this research project in the aforementioned sections, the review will present literature on Computer Science Education methods for analysing source code (Section 2.4) since this research project focuses on the analysis of student source code to address the underlying research questions. Analysis methods for the identification of student problems in general Computer Science Education will be presented in Section 2.4.3, with Sections 2.4.4 and 2.4.5 focusing in on those methods that rely on the analysis of source code and in particular those approaches which have emerged in recent years for the analysis of Project Histories. Different error/problem classification schemes utilised in the analysis of programming will be discussed in Section 2.4.6. Finally, Software Engineering methods of source code analysis focusing on

the analysis of Project Histories (as stored in versioning systems) will be presented in Section 2.5. Software engineering approaches are utilised for two purposes in this research project. The first application is to enable detailed analysis through the production of so-called ‘Line Histories’ (the implementation is presented in Section 6.2.1.2). The second application is to implement a method of Machine-Segmenting Project Histories for further manual analysis (see Chapter 7). These applications enable the use of the method of analysis proposed in this thesis. The review serves to position the presented work in the literature.

## 2.2 Computer Graphics Education

Computer Graphics Education research (Computer Graphics Education) is a sub-field of Computer Science Education (Computer Science Education) which focuses on the teaching of Computer Graphics programming in a Computer Science context. While sharing many similarities it should be differentiated from the teaching of Computer Graphics in engineering, since engineering Computer Graphics education focuses on the use of CAD tools for modelling.

As preparation for this research project a comprehensive review of Computer Graphics Education literature (Wittmann et al., 2009) was conducted. Since the literature reviewed was not directly relevant to this project the literature review can be found in the appendix, Section 9.4.5. Very little of the existing literature was identified as relevant to this project as most of it consisted of the presentation of teaching tools and methods, most of which were not thoroughly evaluated. The review did not identify any research focusing on an in-depth investigation of spatial programming or the role of spatial ability in Computer Graphics Education.

### 2.2.1 The OpenGL API

The OpenGL API<sup>10</sup> is the most widely used graphics API and hence probably the most influential teaching tool in Computer Graphics education, even though it was not designed with pedagogy in mind. The OpenGL API and implementation have shaped syllabi ever since OpenGL was widely embraced as the API of choice for Computer Graphics education and academic research (Angel, 1997; Cunningham, 2000). The result was a shift from 2D to 3D programming in Computer Graphics education, and from bottom-up approaches requiring the development of basic functionality from scratch to top-down approaches involving learning how to use powerful graphics libraries.

---

<sup>10</sup> API: An *Application Programming Interface* is a specification which describes the methods / functions / entities to be provided by implementing software. It allows developers utilising implementing software to rely on learning the programming interface rather than the underlying implementation.

OpenGL has some shortcomings as a pedagogical tool for teaching 3D Computer Graphics. One of these is that it is dependent on the operating system<sup>11</sup> (OS) for windowing functionality, and every OS has a different implementation with which OpenGL needs to interact. This problem can be alleviated with the widely-ported GLUT library which hides OS-specific windowing details underneath a simple, abstracted interface. Unfortunately, neither OpenGL nor GLUT provides much additional functionality; there are no GUI<sup>12</sup> widgets and only a very simple implementation of on-screen text. Interactivity would allow the student to explore her own application and dynamically making changes may aid the student in gaining a better understanding of complex transformations, for example.

### 2.2.2 The Computer Graphics curriculum in Computer Science

The latest curriculum recommended by a SIGGRAPH<sup>13</sup> working group in 2004 (Cunningham, Hansmann, Laxer, & Shi, 2004) recommends students studying Computer Graphics units have knowledge of the following concept areas:

- Transformations
- Modelling: primitives, surfaces, and scene graphs
- Viewing and projection
- Perception and colour models
- Lighting and shading
- Interaction, both event-driven and using selection
- Animation and time-dependent behaviour
- Texture mapping

Since this research project examines the role of spatial ability in Computer Graphics Education through the investigation of spatial programming, the study focuses on tasks involving transformations, viewing and projection and animation. Since modelling and user interaction are both connected to spatial programming, they are also part of the assessment tasks investigated in this research project.

---

<sup>11</sup> OS: An *operating system* is a software system which manages a computer's hardware and software resources; Microsoft Windows and MacOS are examples of operating systems.

<sup>12</sup> GUI: A *graphical user interface* is the point of contact between a user and a software application; it consists of control and display elements such as text boxes and buttons and allows the user to interact with the application.

<sup>13</sup> ACM SIGGRAPH: The "**S**pecial Interest **G**roup on **GRAPH**ics and Interactive Techniques" is a group focused on computer graphics and interactive techniques and is also involved in Computer Graphics Education.

### 2.2.3 Visual Analysis, Perception, Communication and Debugging

Several Computer Graphics educators have called for the introduction of a new teaching topic or technique that goes beyond the traditional boundaries of the Computer Science Education focus on programming, mathematics, algorithms and graphics library APIs. For instance, the technique of 'Visual Analysis' was designed to support students' 'visual understanding' of computer graphics lighting and shading algorithms (Wolfe, 2002). It is proposed that this in turn would allow students to visually identify algorithms underlying an implementation, as well as allow them to recognise errors in implemented algorithms. Using this technique, students could then effectively debug their implementation of such algorithms. Without an understanding of the desired visual effect of an algorithm students would not be able to recognise and correct errors.

Visual analysis (Wolfe, 2002) would consist of teaching students to recognise visual cues such as the sharpness of shadows, presence and positioning of highlights, presence or absence of reflections and so on to identify underlying algorithms or errors in the implementation of an algorithm. The technique consists of a presentation structure to be followed during the semester (Sears & Wolfe, 1995) and a learning tool (Wolfe & Sears, 1996). The technique is conjectured to allow students to develop visual analysis skills through analysing examples of the application of different lighting and shading algorithms. In a similar vein the topic 'Visual Perception' (Chalmers & Dalton, 2002) in was proposed in order to give students a better understanding of modelling, thereby allowing them to develop richer three-dimensional models and scenes. The topic involving teaching students the basics of drawing and sketching.

In 'Graphical Problem Solving and Visual Communication in the Beginning Computer Graphics Course' (Cunningham, 2002) the author proposes that students be taught visual problem-solving and communication techniques not commonly taught in Computer Science courses. The author suggests that students' lack of means to communicate and express themselves visually hinders them when trying to develop solutions to graphical problems. The author also proposes that students' inability to express themselves visually prevents students from taking full advantage of the opportunities Computer Graphics applications provide for visualization of problems. Such visual problem-solving and communication skills are to be taught through the presentation of good examples, as well as critiquing and re-working of submitted student work.

In the work presented above, the focus was on visual features of Computer Graphics programming output, such as shading, lighting or rendering techniques affects a scene. In the context of this research thesis, the emphasis is on spatial aspects of Computer Graphics programming which is a somewhat different because it relates to positioning and orientation of objects rather than their

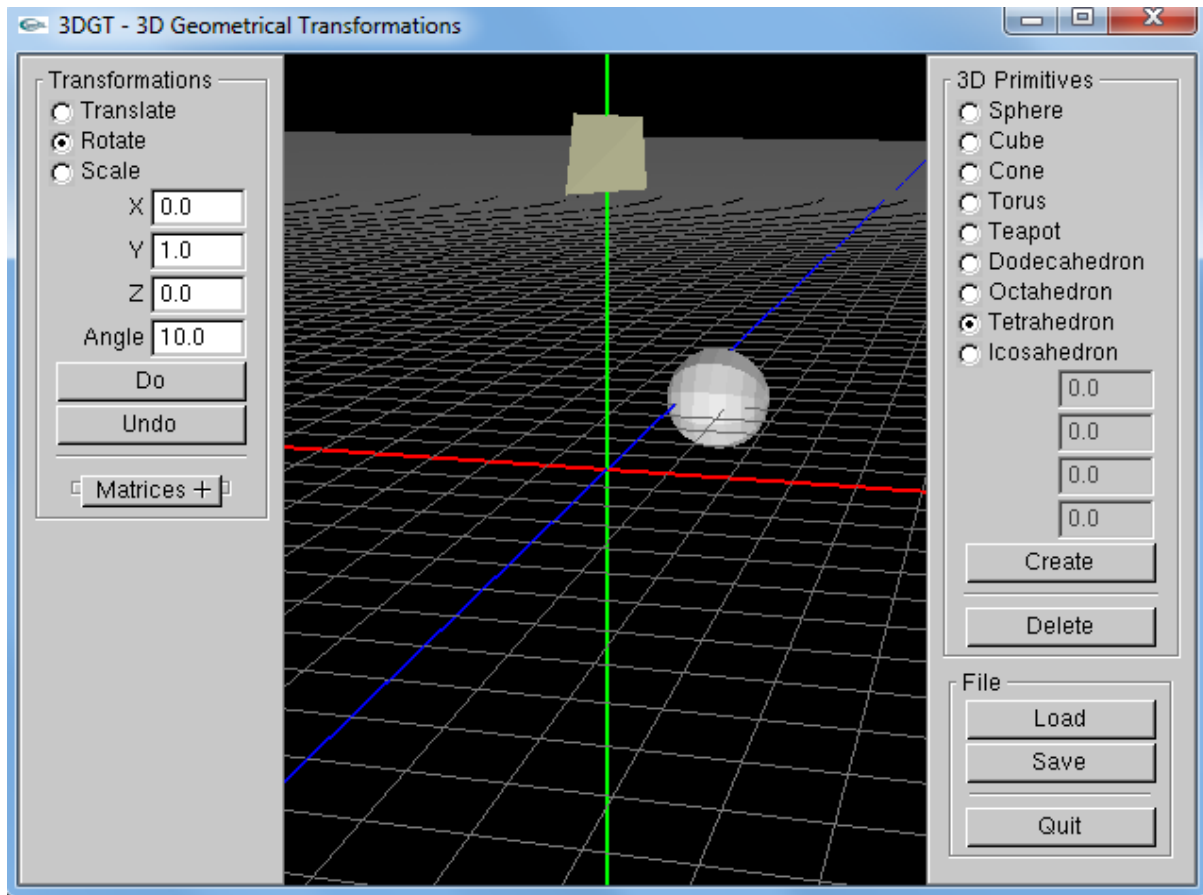


appearance. However, the same intuition underlies the aforementioned propositions as well as the motivation for this thesis: certain features of Computer Graphics programming fall outside the teaching content of general Computer Science Education. These features are not merely the sum of their parts; teaching the programming and mathematics underlying certain concepts is insufficient to effectively support students in developing Computer Graphics programs. As Wolfe points out (Wolfe, 2002) with regard to visual debugging, without methods of debugging applicable to the task domain students will be unable to properly identify and address errors in their solution attempts. While none of the discussed proposals appears to have led to significant redevelopment of the Computer Graphics syllabus, this thesis aims to make a case for the benefits of the introduction of a 'Visio-spatial Analysis and Debugging' component. Such a component would provide the tools to allow students to effectively debug spatial programming relating to transformations and viewing tasks involving three-dimensional transformations.

#### **2.2.4 Computer Graphics Teaching Tools**

Several tools have been designed or proposed to allow students to visualize different concepts involved in Computer Graphics programming. These tools range from visualizing two-dimensional drawing and pixel manipulation to shading and lighting as well as visualization of transformations and views. Two tools allowing the visualization of transformations and/or views will be discussed. These provide representative examples of the functionality and features of many of the tools designed for the visualization of views and transformations.

The CGGEMS repository is maintained by SIGGRAPH, though it does not seem to have been active in the last two years. Its intent to serve as a repository for Computer Graphics learning aids (Figueiredo, Eber, & Jorge, 2003). One of the visualization tools included in the repository is 3DGT, described as "a self-training tool for learning 3D Geometrical Transformations", shown in Figure 3. It allows for the creation of objects to which transformations can then be applied.



**Figure 3: 3DGT - A Self-Training Tool for Learning 3D Geometrical Transformations, part of CGGEMS**

Another example is the 'Programmable Tutor for OpenGL Transformations' (Andújar & Vázquez, 2006). It is a visualization tool which provides a view which allows the user to see the View frustum<sup>14</sup> created by the use of projection calls<sup>15</sup> and lookAt calls<sup>16</sup> in global space. It allows the user to simultaneously observe the effect of these calls in a separate window which shows the scene using the actual view(s). It also allows the addition of objects to the scene, as well as the transformation of these objects. It allows the user to add, remove or reorder transformation calls from individual objects. The user can also examine the state of the modelview and projection matrices<sup>17</sup>. The literature review of Computer Graphics education (see Appendix Section 9.4.5) refers to several more Computer Graphics Education learning tools.

<sup>14</sup> View frustum: the view frustum is the region of space in the modelled world that may appear on the screen ([http://en.wikipedia.org/wiki/Viewing\\_frustum](http://en.wikipedia.org/wiki/Viewing_frustum))

<sup>15</sup> Projection call: The function call that sets up a projection from model space to display space in OpenGL

<sup>16</sup> View call: The function call that sets up an OpenGL view; objects falling into the View frustum are visible on the screen

<sup>17</sup> ModelView and Projection matrices: Matrices by which points in space are multiplied to produce the point's final position on-screen

The fact that educators have developed such visualization tools shows that Computer Graphics educators intuitively understand that students have difficulty with the visualization and understanding of Views and Transformations. One aim of this research project is to go beyond the anecdotal accounts and personal experiences of Computer Graphics instructors to explore whether there is a factual basis for the intuition that spatial programming in the form of View and Transformation implementation are challenging to students, what the nature of this challenge is and how it expresses itself. If this can be shown, then it will provide impetus for the inclusion of techniques and tools to scaffold student learning of three-dimensional transformation/view concepts and skills into the core syllabus of Computer Graphics courses.

Also, while visualization methods found in literature take the form of stand-alone visualization tools it may be fruitful to teach students methods of ‘spatial debugging’ of views and transformations. Students could then apply these techniques during their work on assessment tasks. This would enable them to effectively recognise and debug errors as part of their standard curriculum.

## **2.3 Spatial Ability**

### **2.3.1 Definition of Spatial Ability**

Spatial ability was delineated as a separate and unique factor of human cognitive ability when it was identified as part of ‘mechanical ability’ and measured using tests such as the ‘Minnesota mechanical abilities tests’ (Paterson, Elliott, Anderson, Toops, & Heidbreder, 1930). In the modern context, Visio-Spatial ability (‘visio-spatial ability’, ‘spatial ability’, ‘spatial visualization ability’ or ‘visual-spatial ability’ will be used interchangeably to refer to the same concept) is part of most models of human cognition. It is included in the influential Working Memory model as proposed by Baddeley & Hitch (1974), where it is modelled with its own dedicated component, the visiospatial sketchpad. In this model, the visiospatial sketchpad performs spatial manipulation and imagery tasks.

### **2.3.2 Structure of Spatial Ability**

Structural descriptions of spatial ability range from monolithic to structures comprised of hierarchies of sub-factors. For an overview of the development of structural explanations, see Mohler (2009). The following is a summarised examination of several frequently cited theoretical models of spatial ability; a more detailed description can be found in the appendix, Section 9.4.1.

Research utilising a factor analysis<sup>18</sup> approach has shown that spatial ability can be broken down into two sub-abilities. *Spatial Visualization* is the ability to rotate or otherwise manipulate (three-dimensional) objects in place. *Spatial Orientation* is the ability to be able to compare spatial patterns with one another, to understand spatial relationships between objects and to be able to mentally take different perspectives and orientation (Kozhevnikov & Hegarty, 2001; McGee, 1979).

A different model based on factor analysis breaks down spatial ability differently; instead of a distinction between orientation and visualization, it focuses on the speededness of spatial tasks (Pellegrino, Alderton, & Shute, 1984). *Spatial Relations*, the ability to recognise a relatively simple object after rotation/transformation (to be applied in tasks that involve pairs of stimuli to be compared); high ability in this factor is characterised by fast performance where the stimulus presented is comparatively simple. *Spatial Visualization* involves complex manipulation of complex objects such as folding of objects which involves the movement of internal parts of the stimulus presented; high ability in this factor is characterised by the ability to produce the correct answer for this more complex stimulus. An extension to the model also proposes further factors related to the speed with which patterns can be identified (flexibility of closure, closure speed, perceptual speed) (Carroll, 1993). This model is probably the most cited in the spatial ability literature.

A third model of spatial is based on research utilising a facet theory<sup>19</sup> analysis (Guttman & Greenbaum, 1998). It identifies different facets of spatial ability relating to whether a task involves the rotation of objects or not (Guttman, Epstein, Amir, & Guttman, 1990) and whether the task is two or three-dimensional (Stumpf & Eliot, 1999).

### 2.3.3 Spatial Ability and Performance

Since its early use in evaluating 'mechanical ability' (Paterson et al., 1930), the impact of spatial ability on different academic pursuits has been tested. The following examples illustrate the different contexts in which spatial ability has been implicated as a causal factor for performance. The design of the studies below follows a similar pattern. Students enrolled in a university course are asked to take a spatial ability test at or before the beginning of the semester (pre-test), and then their assessment results are correlated to their spatial ability as measured via the test. Some studies also utilise a post-test at the end of the semester to measure any change in spatial ability caused by their exposure to course materials.

---

<sup>18</sup> Factor Analysis: Factor analysis is a statistical method used to describe variability among observed, correlated variables in terms of a potentially lower number of unobserved variables called factors. ([http://en.wikipedia.org/wiki/Factor\\_analysis](http://en.wikipedia.org/wiki/Factor_analysis))

<sup>19</sup> Facet Theory: Facet theory (FT) is a methodology for designing and analysing observations in which a mapping sentence consisting of different facets forms the hypothesis against which data is tested.

Spatial ability has been found to impact achievement in physics assessment tasks (Pallrand & Seeber, 1984), programming (Jones & Burnett, 2008) and source-code navigation (Jones & Burnett, 2007), engineering education (Hsi, Linn, & Bell, 1997) and chemistry education (Carter, LaRussa, & Bodner, 1987; Pribyl & Bodner, 1987). Spatial ability has been found to affect performance most strongly for problems which require problem-solving skills, whereas it was much less significant for rote problems or problems that required the application of a known algorithm for their solution (Carter et al., 1987). A more detailed review can be found in the appendix, Section 9.4.2.

Spatial ability does not only affect outcomes because of its impact on student problem solving, but it can also affect learning when that learning is dependent on materials that are of a highly spatial nature. Spatial ability has a significant impact on how students learn best using visualizations (Höffler, 2010). Students with weak spatial ability benefit more from dynamic visualizations (e.g. animations) whereas students with strong spatial ability benefitted more from static visualizations (e.g. diagrams). It is hypothesized that this is because students with strong spatial abilities benefit most from building their own mental spatial representations from static visualizations. Students with weaker spatial skills are unable to build their own mental spatial representations and hence will benefit more from dynamic visualisations. Such dynamic visualizations do not require students to build a comprehensive spatial representation, with the visualization itself providing an external representation of the spatial concept. This finding also has implications for Computer Graphics Education. Since the visual output of Computer Graphics programming can be seen as a form of visualisation, it would be beneficial to provide students with weak spatial skills with tasks and/or tools which allow them to dynamically manipulate space during their spatial programming, whereas not providing stronger students with such scaffolding may improve their learning. This leads to the question of how to best approach assessment tasks such as assignments for which in the interest of fairness the same tasks and materials must be provided to all students

The reviewed literature presents strong evidence that spatial ability is connected to performance and problem-solving in a wide range of academic fields and that this connection is strongest with problems that are of a complex rather than rote nature.

#### 2.3.4 Improving Spatial Ability

The significant impact of spatial ability on learning raises a further question: is spatial ability fixed, or can it be improved through intervention? While this research project does not itself propose any methods for improving spatial skills, if the third research question **RQ2** regarding spatial programming being challenging is answered in the affirmative the question of how to proceed arises. If spatial ability is non-malleable then little can be done aside from dissuading students with low

spatial ability from pursuing Computer Graphics courses. However, if spatial ability is malleable, then identification of student problems relating to spatial programming problems could be addressed through proactive interventions. For example, Computer Graphics courses could include instruction specifically targeted at improving student spatial abilities. This section will present research on the effect of experience and training on spatial ability.

Spatial ability training has been shown to permanently improve spatial ability (Baenninger & Newcombe, 1989), with long-term training providing a higher level of improvement than short-term training and task-specific training providing better improvement than more general spatial ability training.

However, not all attempts at improving spatial ability to improve learning outcomes have been successful. A study exploring spatial training for mathematics students (Ferrini-Mundy, 1987) provided general (non-mathematics specific) spatial training to a treatment group enrolled in a calculus course. The study found that the spatial training had no effect mathematics ability as measured via course performance. The spatial training did have a significant effect on one measure of spatial ability designed for the study (and hence probably to some extent reflecting the material of the training course) while not having a significant effect on the other spatial ability test. While the spatial ability training's results on spatial ability were mixed, participation in the mathematics course (with practice effects controlled for) did improve spatial ability on both measures. This suggests that spatial ability training must be well-designed to be effective, and may also indicate a need for task-oriented spatial ability training.

In the context of this research project the most significant research on spatial ability training is that conducted in engineering. Engineering involves manipulation of three-dimensional objects in a similar manner than that carried out programmatically in Computer Science Computer Graphics. This is likely also the reason that much of the literature on spatial ability training is engineering education research. Many skills essential to engineers such as the use of CAD tools<sup>20</sup> to produce three-dimensional models or the visualization of three-dimensional objects based on a set of side or top-down orthogonal views are very likely to be linked to spatial ability. The *spatial visualization* factor as described earlier seems especially relevant. Studies as early as 1955 (Blade & Watson, 1955) showed that the study of engineering improves spatial ability.

---

<sup>20</sup> CAD tool: Computer-Assisted Design tools assist in assist in the creation, modification, analysis, or optimization of a design ([http://en.wikipedia.org/wiki/Computer-aided\\_design](http://en.wikipedia.org/wiki/Computer-aided_design)). Modern CAD tools allow the user to produce three-dimensional models.

A study of engineering students (Hsi et al., 1997) showed a significant correlation between spatial ability (as measured via administered spatial ability tests) and course performance. That study also examined the effect of a remedial course targeted at improving the spatial ability of students that performed poorly on a spatial test at the beginning of the semester. While there was no evaluation of general performance gain, the intervention did close the spatial ability gap between males and females (females having scored significantly poorer on the pre-test). The intervention also significantly reduced drop-out rates for the students that had participated in the remedial course. A remedial course for engineering students using Google SketchUp (Martín-Dorta, Saorin, & Contero, 2008) produced a statistically significant improvement in spatial ability and reduced the gender gap which was no longer statistically significant after the intervention. A study evaluating an intervention involving pencil-and-paper sketching as well as physical artefacts (Alias, Black, & Gray, 2002) found that the intervention produced significant improvement only for an engineering drawing task and not for other spatial ability tests involving cube construction and mental rotation, suggesting the need for spatial instruction tailored towards the particular area of study rather than generalised spatial ability improvement courses.

As an international study (Leopold, Gorska, & Sorby, 2001) showed, it is not just targeted intervention that improves spatial ability; engineering education itself can lead to significant increases in spatial ability. The study examined whether engineering courses offered across three different institutions increased spatial ability and whether spatial ability was a predictor for success in the final examination. Results showed that spatial ability was significant predictor of success in the final assignment. Also, those engineering courses which had the most significant spatial components produced the largest improvement to spatial ability. This suggests both that spatial ability is a significant factor to consider in students' engineering education and that carrying out tasks requiring spatial reasoning leads to an improvement in spatial ability.

The Appendix (Section 9.4.3) contains a review of literature showing the positive impact of spatial ability training in other domains such as architecture and mathematics.

## **2.4 Computer Science Education**

### **2.4.1 Introduction**

The previous section on Spatial Ability provided background on the motivation for investigating spatial programming. However, the methods of data-gathering and analysis used in psychology are not necessarily the most appropriate when analysing computer programming. This section presents a survey of research methods utilised in the Computer Science Education literature, focusing most on those methods which involve analysis of Project Histories. This provides background for the

Change-Coding and Segment-Coding methods of project-history analysis proposed in Chapters 5 and 6 respectively.

The term *(Student) Programming Process* will be used to describe the programming in which a student or developer engages in working on a project. The *Programming Process* for a particular project or piece of software commences with the creation of the first version. It completes with the creation of the final version of the program. The *Programming Process* can be captured / recorded via storage of versions during the program's development. For example, Project Histories as stored in versioning systems are a recording of the *Programming Process*. However, if versions are not submitted at every modification then they are *coarse-grained* recordings that do not capture the *Programming Process* in its entirety.

Even *fine-grained* storage of versions after every modification does not capture every feature of the *Programming Process*. While the source code associated with the *Programming Process* is captured, the *Programming Process* also involves cognitive processes that are not accessible via version histories. It may also include other artefacts such as UML diagrams or sketches, as well as discussions with fellow students or educators. Approaches to capture other features of the *Programming Process* exist. For example, *think-aloud* protocols (Van Someren, Barnard, & Sandberg, 1994) provide a means to elicit some of the cognitive activity occurring during the *Programming Process*. It is also possible to evaluate collected artefacts such as diagrams associated with the *Programming Process*. However, some of these techniques require a significant additional overhead and cannot be automated in the way that a source-code based recording of the *Programming Process* via version snapshots can be.

#### **2.4.2 Research objectives in the analysis of student programming**

In the Computer Science Education literature, many different motivations for the analysis of the student programming process exist. It is often used for the analysis of common student errors or problems (Bryce, Cooley, Hansen, & Hayrapetyan, 2010; Vee, Meyer, & Mannock, 2005), the analysis of student debugging practices (Fitzgerald et al., 2008) or the evaluation of submission or intelligent tutoring systems or the analysis of student group work (Y. Liu, Stroulia, Wong, & German, 2004). This list is not exhaustive; investigation of the student programming process lies at the heart of Computer Science Education research. Identification of student problems and misunderstandings can then be utilised to develop new teaching tools and strategies.



### 2.4.3 Research methods in student programming analysis

Several different approaches utilised in the analysis of the student programming process will be outlined in the following sections.

#### 2.4.3.1 Controlled Experiment

One approach is to conduct formal experiments, modifying the independent variable(s) to observe change to the dependent variable. Such an approach was taken in a study which aimed to discover whether student programming errors occurred due to misconceptions (Anderson & Jeffries, 1985). In this study, the experimenters varied the complexity of the problem stimulus (individual lines of source code, which students were to mentally evaluate and produce the output of) without varying the type of problem. Students made fewer errors when the problem was less complex. This suggests that it was not a misconception which caused students to give incorrect answers (as they showed their understanding of the concept on simpler problems) but rather some other cognitive effect linked to complexity. The authors proposed that errors began to increase due to a lack of student working memory<sup>21</sup>. The experimental approach benefits from high internal validity since it occurs under controlled conditions, but care must be taken to design experiments which match conditions of real-world application to ensure external validity. Also, to create a robust formal experiment the researchers need to have a clear understanding of the effects they are attempting to discover in order to elicit those effects successfully and reliably. Finally, the effort required in setting-up of a formal experiment and recruiting of participants can often prove to be significant roadblocks, especially since experiments may need to be repeated due to flaws in the original experimental design.

#### 2.4.3.2 Subjective Description

Researchers also frequently utilise subjective descriptions of the programming process through observational studies with think-aloud protocols, interviews and instruments such as questionnaires. One example is the validation of a model of programming errors (Ko & Myers, 2003) involving students developing programs using the Alice 3D Programming Environment<sup>22</sup> while thinking aloud about their actions and strategies. The programming process was recorded using video cameras, and the resulting transcripts were manually evaluated by the researchers. A similar approach was used to identify novice programmer learning challenges and how feedback affects the challenge level of different concepts (Butler & Morgan, 2007). Such subjective can provide rich insight into the

---

<sup>21</sup> Working Memory: The memory system which actively holds information in the mind to do verbal and nonverbal tasks such as reasoning and comprehension, and to make it available for further information processing ([http://en.wikipedia.org/wiki/Working\\_memory](http://en.wikipedia.org/wiki/Working_memory)).

<sup>22</sup> Alice: A freeware object-oriented educational programming language with an integrated development environment (<http://www.alice.org/>)

cognitive processes underlying student programming that are not accessible to direct, objective techniques.

However, gathering and evaluation of materials such as audio/video transcripts is extremely time-consuming and hence only feasible for a small number of subjects. In addition, the use of subjective descriptions of internal processes leads to risks to internal validity that must be carefully considered and pre-empted through the use of skilful interviewing techniques. A task conducted under such conditions also suffers from a potential lack of external validity, as it must be relatively quick to complete (or else evaluation becomes infeasible) and the think-aloud protocols and other forms of monitoring may lead to subject behaviours and performance that deviate from 'normal' homework or assignment programming conditions. Studies that rely primarily on other methods of investigation can still profit from the insight into cognitive processes by including questionnaires or interviews in their research protocol.

#### **2.4.3.3 Source Code Analysis (Snapshot or Project History)**

Finally, one can analyse the programming process by analysing the source code that is produced as output. This approach has the advantage of high external validity since it can utilise code produced as part of regular student coursework. It is also generally the simplest approach to utilise since data-gathering can be set up in a transparent way. This allows the gathering of data from large numbers of students with little to no extra effort once the data-gathering mechanism is in place. What follows is a more detailed look at different approaches to source code analysis.

#### **2.4.4 Granularity of Source Code Analysis**

##### **2.4.4.1 Static Snapshot Analysis**

The first method of analysing source code is to focus on a single version of code and to analyse this version for quality or errors. In many cases, this single version is a student's final assignment submission. However, since none of the development of the source code that occurred during the programming process is available, such an analysis will not be able to detect those problems that students deal with during development but fix before the final submission.

A 1971 analysis of student and professional programs (Knuth, 1971) utilised a combination of static and dynamic analysis approaches as well as manual analysis of loop constructs to examine how real-world programs were implemented to provide guidance for compiler design. The static analysis software package *Verilog Logiscope* was used to perform static analysis of student programs (Mengel & Ulans, 1998; Mengel & Yerramilli, 1999). The analysis focused on code metrics relating to source code complexity and formatting as well as on call graphs, which were analysed qualitatively.

Another approach involves the design of a fill-in-the-gap exercise online submission framework (Truong, Roe, & Bancroft, 2004) in which submitted solutions are compared to a model solution and evaluated using source code quality metrics to identify problems such as excessive complexity.

Each of these approaches analyses only a single version of student source code without examining the greater context in which the version was produced. Snapshot analysis methods are generally used in conjunction with static analysis methods. This may be to gain an understanding of code quality, or to gain understanding of commonalities in structure of programs in general as in Knuth's study (1971). Coarse-Grained Project History Analysis

The approach of 'snapshot analysis' of a single version of a program can be used to investigate the properties of the finished product. The approach is less useful for the analysis of errors or problems that occur during the programming process since most of these problems will have been fixed in the final submitted version of the program. This is why most research in Computer Graphics Education which analyses source code utilises methods to produce transcripts of student programming. Such transcripts are created by capturing and storing different versions of source code during the program's development. Approaches to capturing and storing versions of source code can be classified as either coarse-grained or fine-grained. A coarse-grained approach captures a snapshot of source code (a version) whenever a student performs an explicit 'commit' action. This approach mirrors and is often implemented via a version control system. The following paragraphs review examples of the application of coarse-grained approaches to Project History analysis.

A study on student work habits and their impact (Y. Liu et al., 2004) analysed CVS data to study the software development process of student teams working on group assignments. Metrics such as time spent and lines modified (at the team and individual member level) as well as metrics on a file level (number of revisions, modified lines of code) were analysed. A pair of studies (Allevato, Thornton, Edwards, & Perez-Quinones, 2008; Edwards et al., 2009) analysed data mined from the WebCat Online Submission and Automatic Grading System (Edwards & Perez-Quinones, 2008). Both code metrics and time metrics relating to the times at which students submitted their assignments to WebCat were analysed. Results showed that the total number of submissions and early submission were both positively correlated with performance.

A thorough examination of software engineering metrics as predictors for success in student assignments was undertaken by Mierle et al (2005). The study examined whether any of 166 metrics extracted from coarse-grained Project Histories correlated with student performance. The only metrics that were found to be related to student performance were one measuring student effort

(lines of code written) and one measuring the quality of formatting which was taken to also be an indirect measurement of effort. The results of this study show that while Project Histories are a rich data source, the data itself is of such complexity that the application of high-level, abstract metrics does not provide a detailed understanding of the student programming. The study by Mierle et al. (2005) may also have been hampered by the coarse-grained approach to data-gathering (with students manually submitting their assignments at non-specific intervals) since this means that some features may not accurately measure the underlying property since there is no guarantee as to how early or how often in the development process students will submit their assignment.

The coarse-grained studies presented here examined several versions of source code, thereby establishing a developmental context in which individual versions are produced. However as the approaches are coarse-grained this context is incomplete, with most of the modifications made by students not accessible since they were not explicitly submitted to the Project History. Thus, these studies utilised abstract metrics such as complexity or lines of code created, rather than concrete data on the programming process such as output, errors or an analysis of the source code itself, much like the 'snapshot' approaches did. However, the coarse-grained approach adds a temporal dimension to the analysis, allowing for the evaluation of additional factors such as the timeframe in which the assignment was completed. The coarse-grained method provide insight into the stage of development a student's project was in at different points during the programming process through analysis of the amount of change (in terms of lines of code) occurring between commits/submissions. This insight is not provided by 'snapshot' approaches to program analysis.

However, the coarse-grained method is not suited for a concrete source-code level analysis.

The limitation of the coarse-grained versioning approach is that it suffers from the same problem as the analysis of only a single snapshot *between* commits, in that data on student problem-solving occurring between commits is lost to the researcher. If a programmer commits only the final version of a program, then the approach produces the same data as a snapshot-only approach. Since student commit behaviour is inconsistent most studies of this type analyse abstract code metrics or student work patterns instead of individual student errors or problems.

#### **2.4.4.2 Fine-Grained Project History Analysis**

A fine-grained approach captures a snapshot of source code every time a student executes a compilation and/or save action. Researchers also often capture additional data such as error messages. This produces a full transcript of the student programming process compile-to-compile. It then becomes possible to identify errors in individual programming actions. This provides the

potential to gain a precise picture of student problems occurring during the programming process; however, achieving this requires the development of methodologies to analyse these rich data. Compared to snapshot-only or coarse-grained approaches, fine-grained approaches are likely to produce many versions that need to be analysed, from dozens for small laboratory exercises to hundreds or thousands of versions per assignment per student (in the case of this research project) for larger projects, and manual analysis without assistance becomes difficult.

The utilisation of fine-grained approaches seems to be a recent development in Computer Science Education. An early application of the approach is the development of an Eclipse plug-in (McKeogh & Exton, 2004) which captures various Eclipse IDE messages and events, including mouse and keyboard events. This approach operates at a very low level (keystrokes), and the plug-in does not store source code changes as such. It is proposed as a means to gathering data on the student programming process. There appears to be no follow-up research using the plug-in or any research utilising a similarly low-level approach. This may be because of the difficulty of evaluating large quantities of such low-level data.

The earliest comprehensive application of the fine-grained analysis approach discovered in the reviewing of literature was in the implementation of Marmoset, a “course project snapshot and submission system” similar to WebCat. Its design and implementation is presented by Spacco, Hovenmeyer & Pugh (Spacco, Hovemeyer, & Pugh, 2004) with evaluation of the system’s submission and testing features presented by Spacco et al. (Spacco et al., 2006). Of interest to this research project is the evaluation of Marmoset as a Project History analysis tool. Marmoset was used to produce fine-grained Project Histories. Versions of student code were then unit tested and run through a static analysis bug finder (FindBugs) to evaluate the performance of the bug finder tool. As the paper acknowledges, the analysis is limited by the fact that Marmoset only allows data to be gathered from individual snapshots (such as lines changed from the last snapshot); it does not produce any data on the relationship between modifications beyond that gained by direct comparison of two subsequent versions. For example, Marmoset is unable to analyse whether a modified line is modified again later.

An extension to Marmoset to create a simple ‘line history’ by calculating and matching change deltas (as calculated by diff) from snapshot to snapshot. Such deltas would be grouped under various ‘equivalence classes. For example, deltas that are equivalent when whitespace is removed, or deltas which are equivalent except for a ‘small’ change (Spacco, Strecker, Hovemeyer, & Pugh, 2005). However, the proposed approach would be able to only deal with such modifications if the lines

maintained their order as a standard diff algorithm<sup>23</sup> would otherwise recognise the swapped lines as deltas. The order in which they would be compared would lead to the 'swapped' line being matched with the wrong line in the other document. This means that a moved line's history would be broken. There seem to be no studies discussing the implementation or application of the extension proposed in Spacco et al. (2005).

A set of tools which shall be called the *Jadud toolset* consists of a plug-in to the BlueJ IDE which captures data on the student side, and a server with which the plug-in communicates and where gathered data are stored in a database (Jadud & Henriksen, 2009). Of all the approaches to the fine-grained analysis of Project Histories, the Jadud toolset was used in the most rigorous analysis and produced the most significant results. At each compile event, the plug-in stores the source code text of all files involved in the compilation, time spent on the compilation, whether the compilation was successful or not (compile status) and any error messages produced by the compilation attempt. BlueJ also allows invocation of methods directly, and data on these invocations (method name, method class, parameter types and values, return value and exception thrown by the method if any) are also stored. The storing of source code differentiates this approach from most other fine-grained systems; while Marmoset also captures source code, neither ClockIt nor Retina (two other systems reviewed) store these data. Access to source code associated with the programming process and hence with student errors and problems makes a more in-depth analysis of those problems on a code level possible.

Initial research using the Jadud toolset (Jadud, 2005) examined the different types of compile errors and their relative frequency for novice Computer Science students, as well as time data relating to the time between compilations. The results show which types of errors occur most for students. The results also show that students tend to spend less time between compiles after producing compile errors. The analysis focused solely on errors detected via compiler error messages (syntax and some semantic errors); logic errors or runtime errors are not captured by the system. The toolset also includes a 'code browser'. It is described as allowing researchers to easily analyse successive versions of a student's program (Rodrigo, Tabanao, Lahoz, & Jadud, 2009). It also provides a visualization which shows each version/change, the associated error (if any), time spent since the last change, the number of characters modified and a visual representation of where in the code the error occurred. It allows the researcher to identify groups of versions in which the same error occurs, and/or errors occur in proximity. This will then enable detailed qualitative analysis utilising the code browser.

---

<sup>23</sup> Diff is a file comparison utility that outputs the differences between two files based on calculation of the longest common subsequence between the two files (<http://en.wikipedia.org/wiki/Diff>).

The Jadud toolset is used in a follow-on study (Jadud, 2006a) which includes some source-code level analysis of student code in the form of a vignette<sup>24</sup> which includes code excerpts from the programming of a weak student. The qualitative findings indicated that students facing an error frequently move on to implement additional code. Strong students later return to correct the error while weak students introduce additional errors and thus worsening their predicament. The qualitative analysis appears to be relatively informal. It focuses on a small number of demonstrative examples and no formal mechanisms of applying source-code level analysis to larger data sets are presented.

The Jadud toolset (Jadud & Henriksen, 2009) has been shown to be an effective tool for the analysis of the student programming process judging by the significant contributions made using the analysis approach. However, some limitations of this approach should be noted. The approach focuses on compile-time errors and hinges on the analysis of compiler error messages. This limits its effectiveness to the detection of simple syntax (and some semantic) errors. While extensions to capture run-time errors (crashes) would enable the detection of some logic errors, any errors that do not produce an unexpected termination of the program would still be undetectable. This means that the approach is unsuited to analysis of the programming process in a context where the focus lies on non-syntax/semantic errors such as Computer Graphics Education. Another limitation relates to the low-level analysis of source code. While the 'code browser' provided as part of the Jadud toolset (Jadud, 2006a) is designed to ease the analysis process, it provides few features to aid in the analysis, and the comparison of two versions at a time through its web interface is bound to be tedious. In addition, the tool appears to provide no features for note-taking or other forms of assistance for qualitative analysis. This may be the reason no findings based on a thorough and rigorous analysis of source code using the Jadud toolset are presented, only a small number of short illustrative examples.

ClockIt (Norris, Barry, Fenwick Jr, Reid, & Rountree, 2008) is a plug-in to the BlueJ IDE which stores fine-grained compilation error data on the student programming process. The aim of the tool is to analyse student work and error patterns (Fenwick Jr et al., 2009) continuing from work presented by Jadud (2005). Analysis focuses on the evaluation of compilation error data and data related to the time and number of student compilations.

Another tool which captures fine-grained Project History data is Retina. Retina c is a tool to analyse student programming, similar in both purpose and method to Jadud's work and ClockIt. It is

---

<sup>24</sup> Vignette: A technique, used in structured and depth interviews as well as focus groups, providing sketches of fictional (or fictionalized) scenarios (<http://www.srmo.sagepub.com/view/keywords-in-qualitative-methods/n60.xml>).

implemented as a plug-in to Eclipse and BlueJ and as a modification to the javac compiler. It stores compile event occurrences, associated error messages (including runtime error messages) as well as file name and line number at which the error occurred. Data on student compile errors is used to provide assistance to students. Retina provides students and instructors access to summaries of the error types encountered by students and can provide real-time monitoring of students, providing hints to students based on the types of compile-time errors they make.

This section presented several fine-grained approaches for the recording and analysis of the student programming process. With the exception of the Jadud toolset (2005), none of the tools described in the reviewed work seems to have been used in formal analysis of student programming, instead seeming to function mainly as a proof of concept.

It may be that the underlying reason is that many of these tools were developed quite recently. Perhaps more research utilising them will be published in the future. Another explanation for the bottleneck in generating analysis from the data produced by these tools may be that while they implement data collection methods that allow for the collection of (large amounts of) student Project History data, they do not provide tools or a method to cope with complex and verbose source code data.

The data that was analysed was almost exclusively time / work habit data and compilation error message data. This relatively abstract data may not be providing an in-depth understanding of the student programming process. It does not describe how problems or errors come about or how they are addressed. An approach to address this limitation might be based on a formal analysis of all source code modifications occurring in a Project History; such an approach would require a tool which can deal with the complex and verbose source code data.

A method for dealing with such data is proposed in this thesis. Used with the supporting software developed as part of this research project, it enables in-depth analysis of Project Histories at the source code level. This stands in contrast to the approaches discussed in this section which operate at an output or metric level.

#### **2.4.5 Level of Abstraction of Project History Analysis**

In addition to the granularity dimension which specifies how thoroughly the analysed data captures the programming process, methods of Project History analysis can also be classified according to the *Level of Abstraction* at which Project History data is analysed.



The highest level of abstraction is the *metric* level; at the metric level the focus is on high-level, abstract metrics or properties such as code quality metrics or time data relating to student work habits. Most of the work using coarse-grained analysis approaches described in Section 0 relies on *metric*-level output, such as the study by Mierle et al. (2005) in which many software engineering metrics are evaluate to determine whether any are suitable in predicting student performance. The failure of the study to find any metrics not directly based on the amount of work performed by students shows the limitation of this approach. The high level of abstraction allows for only cursory insight into the student programming process.

A lower level of abstraction is the *output* level. At this level, some part of the program's output is analysed. This may be its actual output which is compared to use cases or otherwise processed, or it may be error output such as compiler or runtime error messages. Most fine-grained approaches such as those presented in Section 2.4.4.2 utilised this level of abstraction (Jadud, 2005; Murphy, Kaiser, Loveland, & Hasan, 2009; Norris et al., 2008), machine-analysing compilation logs to discover compilation errors.

The lowest level is the *source code* level. At this level, the actual text of the source code is analysed. This level of analysis will be more time-consuming and more difficult to automate due to the need for human understanding; rather than complete automation, analysis approaches at this level will probably focus on augmenting the researcher's abilities by focusing her on relevant parts of the data and providing summaries. While work using Jadud's toolset (Jadud & Henriksen, 2009) included some illustrative source-code level examples in the form of vignettes, the main analysis was conducted through analysis of error messages at an *output* level. None of the reviewed approaches uses a true *source-code* level analysis.

The method of Project History analysis developed in this research project is *fine grained* and operates mainly at a *source code* level of abstraction in order to get as precise a view of the student programming process as is possible based on an analysis of source code, making it a novel approach to Project History analysis.

#### **2.4.6 Error / Problem Classification Schemes**

Having reviewed source code analysis approaches for the analysis of the student programming process, the following is a review of some different approaches to classifying errors (discovered utilising source code analysis or by other means) used in the literature. This review was utilised to support the development of the error classification scheme used in this research project.

#### **2.4.6.1 Concrete Classification of Individual Errors**

One approach to the classification of errors is to map individual error types directly to classification categories as used by Jadud (2005) where compilation errors were classified into 48 different categories according to the compilation error message produced by the error. The result is a classification scheme which concretely describes errors at a low level of abstraction. For example, student difficulty with a while-loop does not have a single error category but may involve several different kinds of error categories relating to the conditional statement, brackets and so on. This approach is only appropriate when the total number of different error types is manageable as is the case with compilation errors as there is a finite and known set of possible error messages based on the compiler implementation. It cannot be applied to error types that are not part of such a limited set (e.g. logic errors) since there are infinitely many different permutations of any given error type, making a concrete mapping impossible.

#### **2.4.6.2 Classification and Categorisation according to similarity of Individual Errors**

Another approach is to collect similar errors in classification categories, meaning these categories can be more abstract than those used in a direct mapping of errors to categories. A study analysing laboratory assistant error reports (Bryce et al., 2010) involved classification of reported errors into categories. Some categories group a number of concrete errors. An example is the if-statement error category. It includes using incorrect logic for a loop conditional as well as syntax or semantic problems relating to the implementation of a loop construct. Other error categories are far more abstract, such as 'search/sort' errors involving errors in a sorting or searching algorithm, or 'problem solving' errors in which the student is unable to understand a problem, or is unable to formulate a solution to a problem. These are cognitive error types that may be used to code a wide range of actual errors.

A similar approach was taken in the analysis of errors based on a compilation-to-compilation comparison of source code (Vee et al., 2005). The error classification scheme used in that project includes relatively low-level errors. An example of this are the syntactical / syntax or type error categories (in which the wrong type is used to store a variable). The scheme also more abstract cognitive errors such as 'Not following hints' when students did not heed instructions. Errors such as 'Not following hints' are also more context-dependent and unique to the particular task/exercise at hand when compared to the concrete error categories used in the concrete one-to-one mapping used by Jadud (2005).

Another study of Computer Science student errors as reported by educators (Hristova, Misra, Rutter, & Mercuri, 2003) utilised a classification scheme similar to those presented earlier. Unlike the other studies error types were assigned to one of three overarching categories: syntax errors, semantic errors and logic errors. Syntax errors are low-level errors involving mistakes such as the confusion of the assignment operator with the comparison operator or mismatching brackets. The only semantic error identified is the invocation of a class method on an object. Logic errors form the third category and include errors such as improper casting or non-void functions not returning values. None of the errors in this classification scheme are very abstract. This stands in contrast to the 'Problem Solving' error or 'Not following hints' error categories (as examples) used in a study by Bryce et al. (Bryce et al., 2010) and a study by Vee, Meyer and Mannock (Vee et al., 2005) respectively.

The Bug Catalogue I (Johnson & Science, 1983) is perhaps one of the most comprehensive research projects ever completed on student programming errors, and its classification scheme has served as a basis for more modern research such as was conducted by Fitzgerald et al. (2008). The classification scheme proposed by Johnson & Science (1983) is based on the concept of a 'programming plan' which is in turn based on the cognitive science concept of a 'schema'. Programming plans contain the necessary information for addressing a particular programming problem. The classification scheme consists of two dimensions. The first is the *Type of Plan* dimension, which specifies *how* a plan is wrong. A plan may be missing necessary actions (*Missing*), may include superfluous actions (*Spurious*), may have actions in the wrong place (*Misplaced*) or have actions that contain errors (*Malformed*). The second dimension *Plan Component* specifies of what type the problematic action is. This second dimension mirrors other classification schemes presented above in that it gives details on the concrete type of error. For example, it may be an input/output error or an error relating to the updating of a variable. This approach to error classification provides the most thorough description of errors of any of the classification schemes reviewed so far. However, it too focuses on individual error instances as would be found in a particular version of a student's project. In contrast, errors could be seen as occurring in a larger context, spanning several versions, in which they are resolved.

#### **2.4.6.3 Classification and Categorisation of periods of problem-solving**

A different approach to error classification was utilised in the analysis of anecdotes from professional software developers related to difficult-to-debug problems (Eisenstadt, 1993). The classification scheme focuses on the complex problems contained in the anecdotes, any of which would span many versions of a program rather than being located in a single version of the program as were the errors classified by the preceding approaches. Anecdotes are classified along three

dimensions. The first dimension is the '*what caused*' dimension which pinpoints the root of the problem; it can be a *memory* problem, an *end-user behaviour* problem or a *lexical/syntax* problem, for example. This dimension is roughly analogous to the single dimension in most other error classification schemes, though it is more abstract (for example, all syntax and semantic errors are collapsed into a single category). The second dimension is the '*why difficult*' dimension, which specifies why the problem was difficult to solve. It may have suffered from a *cause/effect chasm* in which the effect of an error occurs at an unexpected place, a *difficult debug* problem in which normal debugging practices are difficult or infeasible to apply, or a *faulty assumption/model* problem in which the programmer has an incorrect understanding of a concept (for example, thinking a queue works like a stack). The final dimension is the *how found* dimension which reflects how a solution to the problem was discovered. The first item in this dimension is '*gather data*' which means that the error was found via cout statements or breakpoints. The second item is '*interspeculation*' which involves reasoning about the problem off-line (possibly utilising pen and paper). The third item is *expert recognised cliché*, which indicates that the problem was handed over and discovered by another programmer. The final item is *controlled experiment* in which the programmer sets up an experiment to uncover the root of the error. The *controlled experiment* item suggests a top-down approach, compared to the bottom-up approach indicated by a '*gather data*' approach.

A process-oriented approach such as is applied by Eisenstadt (1993) is more appropriate for a source-code level analysis of the student programming process as captured in a fine-grained Project History. At this level of resolution it becomes possible to look past individual errors to larger problems. A focus on such problems requires a classification scheme which can capture and code these problems. If only the individual errors were coded instead, then the context in which these errors formed larger problems would be inaccessible to subsequent analysis. Which approach is most appropriate depends on the context. When analysing students who are just facing their first programming challenges, it is mainly simple syntax errors and misunderstandings which lie at the root of most problems. However, as students master simple syntax and semantics and tasks move to more complex algorithmic problems, a look at those errors in isolation does not provide sufficient insight into the nature of these problems. A more process-oriented approach spanning larger segments of the programming process becomes necessary.

## 2.5 Software Engineering

### 2.5.1 Introduction

Having presented research on methods of source code analysis as used in Computer Science Education, this section will now discuss methods of source code analysis used in Software Engineering. Such techniques are useful in supporting program comprehension (Zimmermann, Zeller, Weissgerber, & Diehl, 2005) or identification or prediction of potential bugs (Williams & Hollingsworth, 2005). These software engineering approaches are reviewed as background to the development of a machine approach to Project History analysis which will be presented in Chapter 7. The purpose of this machine approach is to facilitate a key step of the analysis process and to thereby speed up analysis considerably.

The methods discussed here can be broken down along the same *granularity* dimension as the Computer Science Education work discussed earlier (see Section 2.4.4). *Snapshot* approaches utilise only one version of a program. *Project-history* approaches utilise versions as committed to a versioning system by developers. An overview of *snapshot* analysis approaches can be found in the appendix, Section 9.4.4. The review will focus on *project-history* analysis approaches since this research project involves the analysis of Project Histories rather than single versions of source code.

### 2.5.2 Project History Analysis and Co-Change

Snapshot approaches analysing a single version of source code can be used to discover features such as which parts of source code are coupled with which other parts, or which set of functions or methods are called as a result of some initial method being called. However, snapshot approaches are not useful in gaining insight into any features of the source code related to the code's development over time since they access only a single static snapshot of the development process. Since the study of the programming process lies at the heart of this research project, the review will move on to approaches analysing Project Histories which can provide assistance in understanding not only the finished product but also the development of source code over the life of a project. In software engineering research these approaches are applied to mine historical information of the development of a program. This is then used to predict the occurrence of bugs or to discover parts of the code that will likely be involved in future modifications. Most of the approaches utilised in Software Engineering research are *coarse-grained* since they utilise standard versioning systems. These involve manual submission mechanisms. This meaning that only those versions the programmer chooses to commit will be available for analysis. Different software-engineering approaches to Project History analysis will be reviewed next.

### 2.5.2.1 Analysis of Individual Versions

A Project History can be utilised as a corpus<sup>25</sup> of versions with individual versions undergoing analysis independent of the context in which they occur. The methods of Project History analysis from Computer Science Education analysed in Section 2.4.4 fall into this category. The approach utilised in these methods is to store details of the program's execution (such as error messages) at each point, hence taking no regard of the wider context in which the version occurs.

An example utilising analysis of individual versions is found in work by Williams & Hollingsworth (Williams & Hollingsworth, 2005). A project's history is mined for instances in which a null value check is added to an existing function call (that was not before preceded by a null value check). For functions for which null value checks have frequently been retroactively added in the past, the tool will warn the programmer if such checks are omitted.

A context-free approach was also utilised in the analysis of Project Histories to measure the effect of software tools on development effort by detecting whether a tool had been used on the source code for a particular commit and then calculating the time taken by the programmer to produce that commit (Atkins, Ball, Graves, & Mockus, 1999).

### 2.5.2.2 Detection of Related Entities through Co-Change

While the analysis of individual versions in isolation is appropriate if the context is irrelevant to the analysis, some analysis involves or requires the detection of related entities. An entity could be any unit of programming, such as a file, a class or a method.

Most approaches to detect related programming entities that were found in the literature rely on *Co-Change*. The term appears to originate from work by Hassan & Holt (2004), but is related to the earlier concept of a 'ripple effect' introduced by Haney (1972). *Co-Change* involves the simultaneous modification (addition, deletion or change) of multiple entities in the same version. For example, two methods whose method bodies have undergone changes in the same version have experienced *co-change*, suggesting the methods may be related. The addition of two method calls *fooStart()* and *fooStop()* is also a form of co-change, indicating the methods may be related.

Co-Change was first used to develop a heuristic for the prediction of change propagation (Hassan & Holt, 2004). Change Propagation in this context means that the modification of one entity will necessitate the modification of other entities. For example, the modification of function A may necessitate the modification of function B, which in turn may necessitate the modification of functions C and D; this means that when queried for candidates for Change Propagation for A, the

---

<sup>25</sup> Corpus: A large and structured set of texts

heuristic should ideally suggest the modification of B, C and D. Hassan & Holt (2004) trialled three different heuristic approaches: The *Code Structure* approach is a static analysis approach, utilising code dependencies such as one method calling another or one method initializing an object of the other entity's type. The *Code Layout* approach utilises the location of entities to identify related entities; for example, it is suspected that entities belonging to the same class are related. The *Developer Data* approach assumes that entities modified by the same developer are related. Finally, the *Co-Change* approach assumes that entities modified together in the project's history are related. It is the last approach that is of interest. The results (Hassan & Holt, 2004) show that the 'Developer Data' heuristic and 'Code Structure' heuristic perform poorly. This indicates that static relationships between entities, as well as relation of entities to developers, are poor predictors of the development relatedness. This means they are unsuited to detecting change propagation tendency from one entity to another. On the other hand, both the 'Code Layout' approach and the 'Co-Change' approach are shown to be effective at proposing correct Change Propagation suggestions, achieving significant accuracy and precision values.

The approach utilised by Hassan and Holt (2004) was adopted and extended in what shall be called the *Zimmermann approach* proposed by Zimmermann et al. (2005); this approach was then applied to the discovery of error patterns (Livshits & Zimmermann, 2005) and Aspect Mining (Breu & Zimmermann, 2006). While Hassan & Holt (2004) proposed a method for identifying a set of entities which is often co-modified with a single input entity, work presented by Zimmermann et al. (2005) extended this approach to detecting the set of entities that are co-changed with an input set of entities rather than a single input entity. Rules suggesting entities to be modified based on a set of input entities are probabilistic. The rules are ranked according to the rule's support count which is the number of transactions from which the rule was derived. This is the number of transactions in which the entities in the input set as well as the entities of the to-be-suggested set are present. Rule confidence is calculated based on the number of transactions from which the rule was derived divided by the total number of transactions in which the set of input entities is involved. Evaluation is performed by utilising an early part of the Project History to build prediction rules. These are then utilised to predict which entities are co-modified in the later part of the Project History which was not used to build prediction rules. The method (Zimmermann et al., 2005) performs well, with 57% of predictions of co-change being confirmed to be correct predictions, demonstrating the effectiveness of the co-change approach to discovering implementation relationships between entities.

Zimmermann's approach was also applied to the identification of *Cross-Cutting Concerns*<sup>26</sup> (Breu & Zimmermann, 2006) to develop 'History-Based Aspect Mining' (HAM) technique. The approach utilised in the paper is largely the same as the core Zimmermann approach used by Zimmermann et al. (2005) but adds the concept of reinforcement, in which sets of related entities are merged if they are within a certain time threshold of each other (*temporal locality*) or if the transaction they were mined from comes from the same developer (*possessional locality*) if and only if the sets of methods modified contain the same methods. These sets are called aspect candidates. So for example, if two aspect candidates are generated based on transactions which are in close temporal proximity and they share the same set of methods M, then they are merged to a single candidate. The core approach (Breu & Zimmermann, 2006) without reinforcement was found to produce high precision and possessional locality reinforcement improved on the core result, but temporal locality reinforcement was found to be ineffective.

### 2.5.2.3 Structuring of Project Histories through Co-Change

The previously introduced HAM technique (Breu & Zimmermann, 2006) served as the basis for the development of the COMMIT method of crosscutting concern identification (Adams, Jiang, & Hassan, 2010). Instead of mining concerns from individual transactions as they are visited, a graph is produced with each entity being represented by a node. Each entity node (e.g. the node describing a method when using method-level granularity) is connected by an edge to the node of any entity with which it is co-changed during any of the visited transactions. Edges are weighted by the mutual information that exists between connected entities based on how many entities co-add or co-remove dependency on entity A and entity B alone, and how many co-add or co-remove dependency on entity A and B together. Edges with a low weight are then pruned, creating multiple disconnected graphs. The nodes (entities) in each graph are used as concern candidates. Evaluation of the technique showed that COMMIT (Adams et al., 2010) outperformed HAM (Breu & Zimmermann, 2006), finding more cross-cutting concerns and also producing cross-cutting concerns involving more entities than HAM. The COMMIT technique is different to techniques presented so far because it does not limit itself to analysing relationships between entities. Instead, it utilises these discovered relationships to form a structure of relationships spanning the entire project's history; this structure is then navigated to discover additional relationships. This type of approach

---

<sup>26</sup> In computer systems, a *concern* is a set of behaviours which form a unit of functionality. Usually these concerns should be modularised and decoupled in order to improve the quality of the code, since increased coupling will increase complexity and make maintenance or development more difficult. Cross-cutting concerns are concerns which affect other concerns, leading to undesirable coupling or code duplication. Such concerns can prove problematic during the maintenance of software systems.



structures the Project History and then navigates this structure. It is similar to the approach used in the Machine-Segmenting method of Project History analysis proposed and evaluated in Chapter 7.

### **2.5.3 Approaches to applying Co-Change**

Different types of entity can be used in detecting co-change. An approach may seek to identify files or functions that are co-modified in the same version or it may examine individual lines of co-modified source code. This section will discuss work utilising co-change at an abstract level (file/class/method) and an approach utilising co-change at the line level.

#### **2.5.3.1 File, Class or Method Co-Change**

All of the work reviewed so far (Breu & Zimmermann, 2006; Livshits & Zimmermann, 2005; Zimmermann & Weissgerber, 2004; Zimmermann et al., 2005) utilises method-level co-change. This means that these approaches attempt to find related methods.

In these approaches source code is first parsed to discover entities (methods, classes or fields). A diff algorithm is then used to detect changes within entities or the addition or deletion of existing entities from one version/transaction to the next.

While such approaches are appropriate for the software engineering context in which they were applied, method-level or file-level co-change is less useful in an analysis of student programming. This is because student projects often do not include many methods and files. Students may even resort to monolithic functions (even if instructed not to do so) which would render co-change on a method or file level meaningless.

#### **2.5.3.2 Line Co-Change**

An approach based on utilising individual lines to structure a project's history is presented by Canfora and colleagues (Canfora, Cerulo, & Di Penta, 2006, 2007; Canfora & Cerulo, 2006). In this approach, instead of focusing on the modification of methods the focus is put on modification of individual lines. A mapping is generated between each pair of versions, mapping each line in the old version to the corresponding line in the new version where possible, including lines that have undergone minor modification. Initial work taking this approach (Canfora & Cerulo, 2006) utilised a simple diff algorithm, meaning that the approach was unable to deal with moved lines; however, an approach to detect moved lines was implemented by Canfora et al. (2007). Through the application of this mapping for every pair of versions, a 'line history table' can be generated. This line history table can be used to retrieve the state of each unique line created during the implementation of a program, and to retrieve the locations at which the line was added, modified or deleted. Evaluation

of the approach by Canfora et al. (2007) shows it produces mappings with error rates below 5%, well below the margin required for statistical significance.

A line-based approach provides the fine level of granularity required for the low-level, detailed analysis conducted as part of this research project and will work even for projects in which students work mainly in a very small number of methods or files where method or file co-change would not produce useful results. An approach to generating line histories (which was developed without knowledge of the existing approach described previously and provides additional features) will be presented and evaluated as part of this thesis in Section 7.2 and Section 7.3.

### 2.5.3.3 Line Co-Change with mapping to higher-level entities

In the future, it may be useful to develop a hybrid approach which provides line-level resolution while also mapping individual lines to higher-level entities such as methods. However, such a mapping is not straightforward, as lines may be moved to different entities. This would enable an understanding of overarching program structures represented by program entities while at the same time maintaining the line-level resolution required for the kind of analysis performed as part of this research project.

## 2.6 Literature Review Summary

This literature review aims to provide an overview of the literature related to the research objective of discovering student programming problems and the research objective of developing a fine-grained source-code level method of Project History analysis, both to demonstrate the validity of the research questions and to ground the proposed method of analysis.

The first part of the literature review is focused on giving background on Computer Graphics Education and spatial ability which, based on the descriptions of spatial ability which have been reviewed, plays a significant role in Computer Graphics Education. Section 2.2 examines Computer Graphics literature to provide background on the field of Computer-Science Computer Graphics education as well as the state of research in that field. As was discussed, there is little if any literature providing a formal analysis of student problems during Computer Graphics programming (Wittmann et al., 2009). Answering the second research question **RQ1** would help address this gap in the research.

The review of Computer Graphics literature also presents the topics recommended for inclusion in Computer Graphics Education courses by SIGGRAPH (Cunningham et al., 2004) on which the homework assignment projects examined in this research project are based (see Section 2.2.2). The review of Computer Graphics education further examines proposals for new topics or teaching

techniques in Computer Graphics Education (Chalmers & Dalton, 2002; Cunningham, 2002; Wolfe & Sears, 1996; Wolfe, 2002) (see Section 2.2.3). These involve training students' visual perception abilities in order to improve their understanding of lighting and shading or three-dimensional modelling. Such proposals are examples of Computer Graphics educators intuiting that teaching based only on the mathematical and programming foundations of Computer Graphics is insufficient. Such instruction fails to develop the abilities required for the effective implementation and debugging of Computer Graphics content. Should the research presented in this thesis confirm this intuition then this would suggest that the development of new methods of spatial Computer Graphics teaching may improve student learning. A topic incorporating this idea called 'Visio-spatial Programming and Debugging' will be proposed in the Conclusion (Section 8.4) to this thesis.

Some examples of tools for the visualization of transformations and views are presented in Section 2.2.4 (Andújar & Vázquez, 2006; Figueiredo et al., 2003); the development of such teaching tools confirms that other Computer Graphics educators share the intuition that spatial programming (specifically spatial programming relating to three-dimensional transformations and views) is a challenging topic for which students require support, an intuition which will be tested through the third research question **RQ2**.

The review of spatial ability presented in Section 2.3 is intended to provide background on the factors underlying spatial programming which is foundational to Computer Graphics Education. Descriptions of spatial ability based on psychology research (Section 2.3.2) indicate that spatial programming, being based on three-dimensional transformations and perspective-taking as described in models of spatial ability in a range of literature (Carroll, 1993; Guttman et al., 1990; Kozhevnikov & Hegarty, 2001; McGee, 1979; Pellegrino et al., 1984; Stumpf & Eliot, 1999), is cognitively rooted in spatial ability.

As discussed in Section 2.3.3 several studies across different disciplines have shown that spatial ability affects performance (Carter et al., 1987; Jones & Burnett, 2007; Pallrand & Seeber, 1984; Pribyl & Bodner, 1987); one of the disciplines is engineering (Hsi et al., 1997) which is especially pertinent as engineering education includes tasks similar to those in Computer Graphics education, indicating that spatial ability may be a factor for Computer Graphics students. Research (Höffler, 2010) has also shown that students with poor spatial ability benefit most from dynamic visualizations. On the other hand, students with high spatial ability benefit more from static visualizations. This finding has potential application in the teaching of Computer Graphics education since a graphics program can serve either as a static or a dynamic visualization of a Computer Graphics concept such as a three-dimensional transformation.

Research regarding improvement of spatial ability is discussed in Section 2.3.4. There is a particular focus on engineering, since engineering tasks involving three-dimensional objects are similar to those required in Computer Science Computer Graphics Education. Several studies have produced statistically significant improvements in spatial ability (Alias et al., 2002; Baenninger & Newcombe, 1989; Hsi et al., 1997; Leopold et al., 2001; Martín-Dorta et al., 2008). This provides the impetus for exploring the role of spatial programming in Computer Graphics Education. The difficulty of spatial reasoning can cause learning difficulties in other contexts, and there is evidence that such problems can be overcome given the proper training. Thus if **RQ2** is answered in the affirmative then it may be worth investigating such training in a Computer Graphics context to support student learning. However, given that existing approaches are designed for other domains and that previous research has shown it is important to develop spatial ability training that reflects actual tasks from the domain rather than aiming at a general improvement to spatial ability (Alias et al., 2002) if spatial programming can be shown to be a significant problem then it may be helpful to students to develop Computer Graphics-programming specific spatial ability improvement materials to be integrated into the Computer Graphics Education syllabus.

The second part of the literature review is focused on literature relating to **RQ3**, the development of a method of analysis to provide insight into student problems and errors. Section 2.4 reviews Computer Science Education literature approaches for the analysis of student problems. This provides background for the development of the Change-Coding and Segment-Coding methods presented in Chapters 5 and 6. Different methods of analysis of student programming are discussed in Section 2.4.3. The *Controlled Experiment* method involves setting up an experiment in which a specific facet of programming behaviour can be examined (e.g. Anderson & Jeffries, 1985). The *Subjective Description* method involves students thinking aloud about their cognitive processes occurring during programming (e.g. Butler & Morgan, 2007; Ko & Myers, 2003). The *Source Code Analysis* method involves analysis of source code produced by students and is the method chosen for this research project. Compared to the other two methods, its application is non-intrusive and requires less effort. It also does not require interviewing skills and experience as would be required for think-aloud protocols. Furthermore, basing the analysis on source code produced under real-world learning conditions ensures high external validity.

In examining different approaches to analysing source code in the literature, three different approaches were identified. The *static snapshot* method bases analysis on a single version of source code, usually the final version submitted by students. Examples of the application of this approach are found in a range of reviewed work (Knuth, 1971; Mengel & Ulans, 1998; Mengel & Yerramilli,

1999; Truong et al., 2004). The *coarse-grained Project History* analysis method analyses Project Histories produced through the periodic manual submission of source code either through a versioning system or an online submission system (Allevato et al., 2008; Edwards et al., 2009; Y. Liu et al., 2004; Mierle et al., 2005). Both the *snapshot* and *coarse-grained* approaches are limited by the fact that the data captured does not fully describe the problem-solving process, meaning that studies using these examples largely rely on evaluation of abstract metrics such as code quality metrics and/or analyse student submission behaviour rather than developing a deep understanding of student problem-solving. The *fine-grained* analysis of Project Histories does not rely on students manually submitting their assignments periodically, but instead uses approaches to store a version of student source code every time a modification is made. Examples of the fine-grained approach to analysis can be found in several reviewed papers (Jadud & Henriksen, 2009; Jadud, 2005; Murphy et al., 2009; Norris et al., 2008; Rodrigo et al., 2009; Spacco et al., 2006, 2004). Such approaches can produce the most detailed data consisting of a full transcript of all student programming actions. This data can then be used for detailed analysis of students' problem-solving. However, as will be discussed in the next paragraph most researchers did not apply the fine-grained analysis method to the analysis of students' problem-solving, focusing instead on individual student errors as captured in compilation error messages.

Methods of fine-grained Project History analysis can also be classified according to the type of data that is analysed as described in Section 2.4.5. *Metric-level* approaches analyse abstract metrics such as software engineering metrics measuring code quality or student work habits via comparison of submission time stamps, such as was used in several of the studies reviewed (Allevato et al., 2008; Edwards et al., 2009; Y. Liu et al., 2004; Mierle et al., 2005).

*Output-level* approaches measure the output of different versions of a program as stored in the Project History. The most commonly examined type of output are compilation error messages. These are gathered through machine-processing of compilation error logs, or via capture of compilation errors at the compiler level (Jadud & Henriksen, 2009; Jadud, 2005, 2006a; Murphy et al., 2009; Norris et al., 2008; Rodrigo et al., 2009). Most of the fine-grained Project History analysis methods reviewed in this chapter utilise output-level approaches. However, output-level approaches do not provide insight into the problem-solving process since such insight requires an understanding of student programming actions, not merely the errors which are produced as the result of these programming actions. In addition, analysis of compilation error messages is unsuited for domains such as Computer Graphics Education. In Computer Graphics programming many errors are logic errors. These errors do not produce any compilation error messages.

The least abstract method is the *source-code* level of analysis, at which the source code itself is analysed and evaluated. While several examples of metric and output-level analysis were presented, none utilised a rigorous source-code level of analysis. Jadud's work (Jadud, 2005, 2006a, 2006b) does provide a glimpse into the potential of this approach through the presentation of a small number of demonstrative vignettes, but these vignettes are used for illustrative purposes rather than as the primary method of analysis (which is compilation error analysis) and hence do not provide a complete picture of student problem-solving. It may be that the tools and methods described by Jadud (Jadud, 2006a) are not powerful enough for a comprehensive formal source-code level analysis. The work presented also does not include any formal methods of source-code level analysis. One of the aims of this thesis is to describe a method of rigorous source-code-level analysis. This method is composed of two parts, formal theoretical underpinnings and analysis software which enables thorough source-code level analysis for large Project Histories.

Section 2.4.6 presents an examination of classification schemes used to classify programming errors or problems. The review showed that most categories used in such classification schemes are quite concrete. They are tied to a particular type of concrete syntax or semantic error (Bryce et al., 2010; Hristova et al., 2003; Jadud, 2005; Johnson & Science, 1983; Vee et al., 2005). The approach that will be developed in this research project is more like the approach taken in work by Eisenstadt (1993). That work utilises a problem-oriented classification scheme, classifying stretches of programming involving many versions instead of individual errors occurring in single versions and is hence well-suited to describing a student's problem-solving process holistically rather than superficially as a count of different concrete error types.

The aim of the method analysis developed as part of this research project is to gain a depth of insight into student problems not available to other methods, because these use less direct approaches to observation of student programming actions. In order to fulfil this goal, the method will be fine-grained, recording every source code modification made by students. This approach is not novel, having been used in previous work (Jadud & Henriksen, 2009; Jadud, 2005; Murphy et al., 2009; Norris et al., 2008; Rodrigo et al., 2009; Spacco et al., 2006, 2004). However, it will also be a source-code level method of analysis, compared to previous work which focused either on abstract code metrics and time data (Allevato et al., 2008; Edwards et al., 2009; Y. Liu et al., 2004; Mierle et al., 2005) or output data (Jadud & Henriksen, 2009; Jadud, 2005, 2006a; Murphy et al., 2009; Norris et al., 2008; Rodrigo et al., 2009). The method developed in this thesis will provide a novel means of analysis for Computer Science Education researchers.

After having examined methods of Project History analysis in the Computer Science Education context, Section 2.5.2 presents literature on software-engineering methods of Project History analysis with a focus on methods utilising co-change of entities. This review serves as background for a method of line history generation as well as a machine-approach to Project History analysis, both of which will be described in Chapter 7.

Three different types of Project History analysis method were identified and are presented in Section 2.4.4. The first type of method (see Section 2.5.2.1) treats a Project History as a corpus of versions. Each version is analysed in isolation. Two examples utilising this analysis approach were discussed (Atkins et al., 1999; Williams & Hollingsworth, 2005). Since the focus of this research project is on problem-solving spanning multiple source code versions this type of method is not appropriate. The second method, presented in Section 2.5.2.2 utilises '*Co-Change*', the simultaneous change of entities in the same version, to identify related entities. This approach appears to have been recently developed by Hassan & Holt (2004) and has been utilised in several projects over the last ten years (Adams et al., 2010; Breu & Zimmermann, 2006; Hassan & Holt, 2004; Livshits & Zimmermann, 2005; Zimmermann et al., 2005). The third approach presented in Section 2.5.2.3 goes a step further, traversing discovered relationships to produce a graph representing the structure underlying the Project History and then traversing this graph to answer questions about these relationships. Only a single example of the application of this method was found in the literature (Adams et al., 2010). At present, most approaches found in the literature fall into the first or second category, with only a single approach involving the building-up of a structure. This may be because the research direction is relatively recent with most of the research utilising co-change having been conducted in the last five years. In addition, software engineering researchers are interested in understanding the relationship between different entities whereas in the context of this research project the aim is to gain a complete understanding of all programming actions in the Project History. This method is augmented with a machine method for the detection of related source-code modifications (Machine-Segmenting) described in Chapter 7. Machine-Segmenting facilitates analysis by reducing the amount of manual work required to structure a Project History.

Co-Change can be applied at different levels. For example, co-change can identify related methods, related files or even individual related lines of source code. Section 2.5.3 examines literature using co-change at the file/method and at the atomic line level. The bulk of the software-engineering work reviewed as part of this review utilises method-level co-change. An approach to generating line histories storing all modifications for a given line of source code was reviewed (Canfora et al., 2007). This approach can be used to detect line-level co-change. In this research project, a line-level

approach to the detection of co-change is used. The reason is twofold. First, the best resolution for discovery of structure underlying a Project History is provided by utilising a line-level approach rather than a coarser method-level approach. Second, student programming habits are poor. Students often do not follow good practice and create monolithic methods, meaning that a method-based co-change analysis would detect little of interest. In addition, programs developed for assignments will not involve that many methods even for students who use good object-oriented programming practices. The Machine-Segmenting approach presented in Chapter 7 utilises line-level co-change to discover the structure underlying a Project History. It identifies related versions and groups them into ‘Segments’. These Segments can then be analysed manually.

With this chapter having provided a background for the development of the methods of Project History analysis presented in this thesis as well as on Computer Graphics as the domain of investigation, the next chapter will discuss the methodology on which this research project is based.



# 3 Methodology

---

## 3.1 Introduction

This thesis involves solving three separate but related tasks. The first task is the development of a method of source-code analysis. This task relates to both research aims: firstly to develop a method for the analysis of source code as stored in version control systems and secondly to apply this method to the analysis of student difficulties in learning Computer Graphics programming. Since the task relates to both research aims, it also relates to all three research questions (RQ1: How can the analysis of students' coding as captured in a version control system be used to analyse student programming problem-solving? RQ2: What kinds of problems do students learning Computer Graphics Programming experience? RQ3: Is 'spatial programming' a difficult area/topic of Computer Graphics programming for students?).

The second task is the application of this method to Computer Graphics student source code to gain a better understanding of student problems. This problem relates to the research aim relating to understanding CG student problem solving, and hence to research questions **RQ2** and **RQ3**.

The third task involves the development of a software support package which facilitates the developed analysis method. This problem relates to the research aim relating to the development of a method of source-code analysis and hence addresses **RQ1**.

This research project involves two different methodologies to address these different tasks. One is aimed at the development of a data-analysis method for the analysis of source-code data, based on existing research methods. This methodology is described in Section 3.4. The other is aimed at developing software-engineering based machine methods to support and facilitate the developed data-analysis method. This methodology is described in Section 3.5.

Section 3.2 presents an overview of terminology developed to discuss the analysis of source code, whereas Section 3.3 provides a description of the source code data analysed as part of this research project.

Since the developed methods are complex and the thesis presents a large amount of material, this chapter provides summary descriptions and rationale for the methods, not detailed description of the methods. Precise descriptions are positioned just before the discussion of the data analysis and

evaluation for which they were utilised in order to save the reader from having to go back and forth in the thesis frequently.

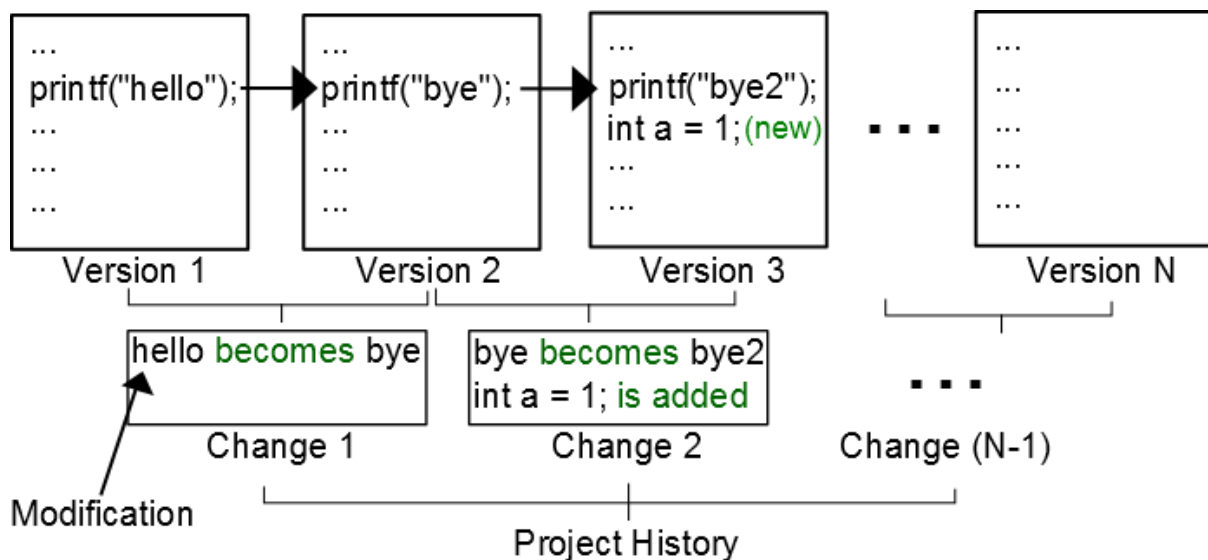
## 3.2 Terminology

This section will introduce several terms used throughout this dissertation which relate to the capture and analysis of source code modifications. These and other terms are also described in the glossary in the appendix, Section 9.1. It may be beneficial to keep a print-out of the glossary at hand during the reading of this thesis.

A **Version** in the context of this thesis refers to a version of a source code file. Each time a source code file is changed, a new version of that file is created.

Whereas a **Version** is a single source code file, a **Change** is encapsulates all the edits from one version (see the term version) to the next. Figure 4 shows the relationship between Versions (on top) and Changes (below). These edits are referred to as **Modifications**. A Modification is a change to a line of source code. The different types of modifications are discussed in Section 6.2.1.2.

A **Project History** is a view of a student's project as stored in a version control system. It consists of all the Changes the student has produced while working on a project. Figure 4 shows the relationship between Versions, Changes and the Project History.



**Figure 4: A Project History consists of N-1 Changes, each of which consists of all the Modifications from one Version to the next**

A **Line History** for a given line of source code consists of all the Modifications which have been applied to that line in the Project History. **Line History Generation** is the process of generating a Line

History for each unique line of source code in the Project History. This process involves creating new Line Histories whenever a new line is discovered (Added), and adding modifications to existing Line Histories whenever a Change contains a modification to a line which had been added previously.

A **Segment** consists of a set of Changes. In the context of this thesis, Segments are used to identify sets of Changes which deal with the same underlying programming problem. For example, if a student is working on implementing an animation during a set of Changes, these Changes will be grouped as a Segment. Likewise, a set of Changes implementing a data structure would be grouped as a Segment. Such Segments can then be analysed and coded. The previous two examples would likely have different codes applied to them because of the underlying programming task (animation in the first example, general CS in the second example). The process of mapping Changes to Segments (a many-to-one mapping) is called **Segmenting**. It can either be carried out by the researcher or by so-called **Machine-Segmenting** algorithms which attempt to identify related Changes automatically.

The **SCORE Analyser** is a software application developed as part of this research project. It provides features designed to facilitate **Project History** analysis. These features are discussed in more detail in Chapter 4.

### 3.3 Description of the data to be analysed

The primary data source for this research project was source code as would be stored in a version control system. This data consists of all versions of source code produced by students during the programming of their assignments, including every modification / programming action. The sequence of these Changes will be referred to as a Project History. Figure 4 shows how the different versions of source code relate to form a Project History.

Data was gathered from students participating in an introductory Computer Graphics course at a tertiary institution. Two assignments were captured during two different iterations of the course. The data is described in more detail later in this chapter in Section 3.6.

As discussed in the literature review (Section 2.4.4.2) such data has only rarely been analysed in the CS context, and never to the level of detail needed to answer the research questions posed in this dissertation. For this reason, this research project included a significant amount of effort expended in developing data analysis methods with which to analyse these project histories.

### **3.4 Data-Analysis Methods for analysis of source code**

One of the aims of this research project is to develop a method for the analysis of source code as stored in version control systems. Another is to gain insight into student problems that arise during their production of Computer Graphics programs. The research method(s) chosen for this research project need to be suited to these aims.

#### **3.4.1 Evaluation of different research approaches**

Research methods fall into two broad categories. The first is the quantitative research approach. When using a quantitative approach the researcher states hypotheses based on postpositive claims (Creswell, 2012). The second is the qualitative research approach. Instead of relying on postpositive claims, the researcher bases knowledge claims on constructivist perspectives. Hypotheses are not validated in the same definitive involving statistical analysis of measurements. Rather, qualitative research is generally open-ended (Creswell, 2013). Since qualitative analysis does not depend on the application of statistical methods like quantitative analysis does, the scale of data-gathering can be much smaller in qualitative studies. A more in-depth review of the differences of these approaches is given in the appendix, Section 9.3.3.1.

In this study, both methods were utilised as will be outlined in Section 3.4.3. However, because of the exploratory nature of this study and the verbosity of the data to be analysed which limited sample size, qualitative approaches produced the most interesting insights relating to student programming CG problem-solving.

There are many different qualitative research methods. Five such methods are reviewed in the appendix, Section 9.3.2. The research method which was found to be best suited to this research project is the Grounded Theory method. The Grounded Theory (GT) method's focus is on "Developing a theory grounded in data from the field" (Creswell, 2012). Its unit of analysis is at the level of studying a process, an action or an interaction involving many individuals.

Its unit of analysis is ideally suited to this research project as the aim is to analyse student programming over the course of an assignment, which is indeed a process. This process can be seen to be composed of many programming actions. GT can be applied at both of these levels.

Furthermore, the focus on developing theory matches with one of the aims of this thesis, which is to develop a theoretical understanding of student problems during Computer Graphics programming. For these reasons, Grounded Theory was selected as the method underlying the Segment-Coding method, which is the primary data analysis method developed as part of this research project. The next section will provide a brief overview of GT.

### 3.4.2 Grounded Theory: An overview

This section gives a brief overview of Grounded Theory and its rationale as well as its key aims and processes. A more detailed description is provided in the appendix, Section 9.3.3.

There are three types of Grounded Theory analysis (see Charmaz, 2006; Corbin & Strauss, 1990; Glaser & Strauss, 1967; Glaser, 1992), some of which include additional analysis steps. A discussion of the three types is provided in the appendix, Section 9.3.3.1. However, the central processes and methods are common to all these types of GT. It is these central processes that were built into the GT-inspired method (Segment-Coding) developed as part of this research project, rather than the details pertaining to particular GT types. These particulars are not relevant in the context of this study because the data analysed (source code) is so fundamentally different from that analysed with GT in other contexts (spoken or written words). A more detailed explanation of how this study differs to the contexts in which GT is usually applied, as well as how the Segment-Coding method does and does not adhere to GT is provided in Section 3.4.4.

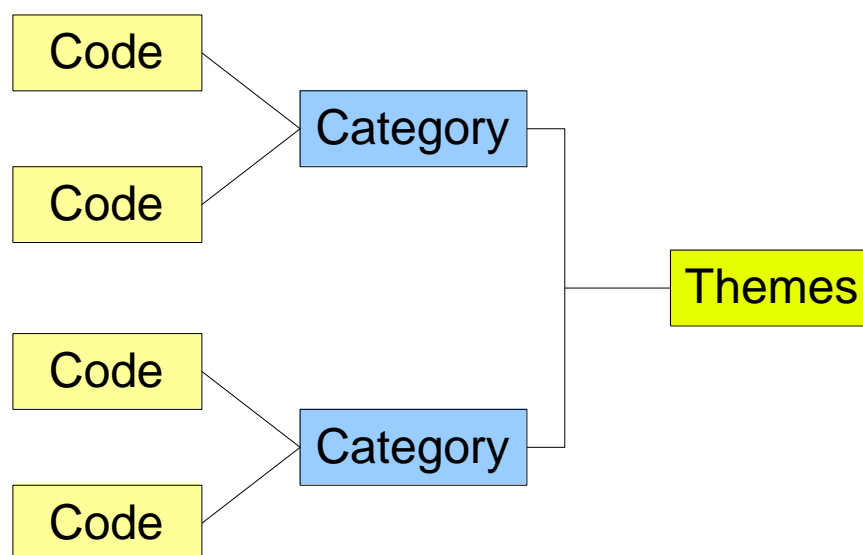
The aim of GT is to produce theory grounded in data. According to Strauss and Corbin (1998, p. 25) “Theorizing is the act of constructing (we emphasize this verb as well) from data an explanatory scheme that systematically integrates various concepts through statements of relationship. A theory does more than provide understanding or paint a vivid picture. It enables users to explain and predict events, thereby providing guides to action.” This coincides with the aim of developing an understanding of Computer Graphics student problems. The theoretical basis of such an understanding will involve explaining student programming actions and predicting the difficulties faced by Computer Graphics students in general.

Grounded theory can be seen to be composed of four central processes / methods. These are coding of data (an approach it shares with many other research methods), the Constant Comparative Method (CCM) of analysis, the process of memoing and theoretical sampling and saturation.

The Constant Comparative Method (CCM) is the backbone process of Grounded Theory. Its application ensures that theory developed via application of GT methods remains ‘grounded’ in the data. It involves coding gathered data and then refining these codes in four stages (Glaser & Strauss, 1967, p. 105) until theory is developed. These stages are comparing incidents applicable to each category, integrating categories and their properties, delimiting the theory, and finally writing the theory. During the comparative step, incidents coded (coding will be described in more detail later in this section) in the same category are compared with each other. Commonalities are identified; this leads to the development of properties of the categories. In addition, differences may lead to the

dividing or sub-dividing of categories. During the integration step, the relationships between different categories and their properties are examined. The delimiting step involves moving from a (large) set of concrete categories to a (smaller) set of more abstract categories, which reduces the scope of analysis and allows the researcher to focus in on the key concepts and relationships in the data. Finally, during the theory-writing step, theoretical memos describing the connections of categories as well as the properties of categories are combined to develop the major themes of the theory (Glaser & Strauss, 1967, p. 113). It should be noted that CCM is not a linear approach moving from one phase to the next. Even as a researcher enters a later phase, she will still be periodically be working on the preceding phases, refining categories, properties and relationships even as the final theory takes shape. A more in-depth description of the CCM can be found in the appendix, Section 9.3.3.3.

General coding of qualitative data is described Saldaña (2009). The relationship between codes, categories and themes is shown in Figure 5. At the lowest level of the coding process codes are applied to data. A code is a short phrase which captures the essential attributes of the underlying data, and this coding scheme is produced dynamically as data is analysed, with codes being added, merged or replaced as analysis proceeds. The coding and re-coding of data proceeds until a functional coding scheme which properly captures the underlying data is developed.



**Figure 5: Relationship between Grounded Theory entities; several codes form a category and analysis of categories leads to the discovery of themes underlying the data (adapted from Saldaña (2009))**

Once the codes reach a level of maturity they are collected in categories. The properties of codes and categories are compared, and relationships and attributes are elicited. Based on this, themes

underlying the data are developed. These themes crystallize the theoretical content of codes and categories and the collection of themes is used to build an overarching theory.

In GT, coding is broken down into three main phases. An overview of these phases is provided in the following paragraphs. A more detailed description which includes the additional coding steps introduced by Glaser & Strauss (Axial coding) and Glaser (Theoretical coding) can be found in the appendix, Section 9.3.3.5. The first is open coding, which is “the analytic process through which concepts are identified and their properties and dimensions are discovered in data” (Strauss & Corbin, 1998, p. 101). During open coding, data is broken down, examined and constantly compared for similarities and differences. Conceptually similar actions are grouped as categories (Strauss & Corbin, 1998, pp. 102–103). These categories function as abstractions of individual actions or incidents, leading to the discovery of the most important common properties of different incidents, as well as providing an understanding of the relationship between incidents which fall into different categories.

The next phase is called ‘selective’ coding. During selective coding major categories are integrated to form a larger theoretical scheme. The outcome of this process is the developed GT theory (Strauss & Corbin, 1998, p. 143).

The final phase is called integration. During integration, the researcher integrates findings from memos and diagrams, recognizing relationships between concepts and categories (Strauss & Corbin, 1998, p. 144). Those concepts which have been elevated to categories at this stage are abstractions which represent the experiences of many persons or groups. However, because of the constant comparison which occurs during the application of GT, these categories still have relevance to all cases / instances in the study (Strauss & Corbin, 1998, p. 145).

During coding (especially during selective coding) and integration, the researcher also produces ‘memos’. Memos are “very specialized types of written records—those that contain the products of analysis or directions for the analyst. They are meant to be analytical and conceptual rather than descriptive” (Strauss & Corbin, 1998, p. 217). These memos may describe categories and their properties, relationships between different categories, or they may record a researcher’s intuitions about future research directions. The integration of these memories leads to the development of theory. A more detailed description of memoing can be found in the appendix, Section 9.3.3.7.

The final process which directs and eventually terminates the research process is theoretical sampling which eventually leads to theoretical saturation. Theoretical sampling is different to sampling as it is commonly used because the aim is not to ensure the generalizability of results.

Instead, the aim is to “maximize opportunities to compare events, incidents, or happenings to determine how a category varies in terms of its properties and dimensions” (Strauss & Corbin, 1998, p. 202). The researcher chooses which data to analyse based on questions raised during earlier analysis. This process continues until all questions are answered and “(a) no new or relevant data seem to emerge regarding a category, (b) the category is well developed in terms of its properties and dimensions demonstrating variation, and (c) the relationships among categories are well established and validated” (Strauss & Corbin, 1998, p. 212). The point at which these conditions are fulfilled is called theoretical saturation. Once theoretical saturation is achieved, the integration of categories and memos will produce a precise and well-developed theory which provides broad coverage of the area being analysed. A more detailed description of theoretical sampling and saturation can be found in the appendix, Section 9.3.3.6.

### **3.4.3 Developed research methods**

This section gives a description of the developed data analysis research methods (Change-Coding, Segment-Coding, Segment Feature Analysis, Animation Analysis and Free-Answer Questionnaires), as well as providing some insight into the rationale behind the development of the different methods. The following Section (Section 3.4.4) will describe in more detail how the primary data analysis method developed (Segment-Coding) is related to the Grounded Theory method described in Section 3.4.2.

#### **3.4.3.1 Change-Coding**

The *Change-Coding* method was the first data analysis method developed as part of this research project. Development of the method started with the initial examination of Project History data gathered in the 2009 iteration of an introductory Computer Graphics unit described in more detail in Section 3.6.2.1.

This initial analysis focused on one student in particular. This student had produced the best assignment and had most deeply engaged with the assignment’s core tasks. Informal analysis based on note-taking discovered several problems underlying the student’s development of his assignment, especially regarding spatial programming. Informal analysis of other assignments revealed other students also struggled with spatial concepts, but also revealed that students tended to avoid working on tasks related to spatial concepts whenever they could, which was possible due to the rather open nature of the assignment specification in 2009.

The GT process of theoretical sampling aims at a selection of data suited to the analysis and investigation of tentative concepts or intuitions developed during earlier coding. For this reason, the



2010 iteration of Comp 330 included more narrowly specified assessment tasks, especially for the third assignment involving three-dimensional spatial programming. These tasks focused students' development efforts in ways which would elicit Project Histories containing programming involving Computer Graphics concepts, with a special emphasis on spatial programming. Spatial programming had been identified as key both in the Computer Graphics literature (see Section 2.2 and appendix Section 9.4.5) as well as through analysis of 2009 iteration data. These changes to the assignment specification were possible because they also brought the assignment specification more in line with the learning outcomes for the unit as a whole.

Theoretical sampling also provides the rationale behind the assignments chosen for analysis which are presented in Section 3.6.3. Analysis of 2009 iteration assignments had shown that poor assignments do not provide a good basis for the analysis of Computer Graphics concepts and problems since poorly performing students often do not even attempt to implement many of the core tasks. For this reason, Project Histories in which most of the core tasks had at least been attempted were chosen. Two of the assignments belonged to two of the highest-performing students since analysis of 2009 data had shown that the highest-performing students explored concepts very deeply. This in turn meant their Project Histories contain very rich data relating to core concepts. The other three assignments chosen belonged to students who had at least attempted all the core tasks. This ensured that there was enough interesting data to analyse while also providing some insight into the nature of difficulties for a broader spectrum of students.

Selection of data was followed by coding, the driving process of the GT methodology. Since GT is a social science methodology this coding is usually applied to the analysis of interview transcripts, field notes or similar natural-language data. To apply it to students' Project Histories requires these Project Histories to be segmented into units which can then be coded. This means that part of the development of a method of analysis involves the development of a method of segmenting Project Histories as well as the development of tools which can store segmenting and coding results for analysis. Once the data is segmented appropriately, a set of codes that will interchangeably be termed the *coding scheme* or *classification scheme* is developed and used to code the data.

The method which developed from this initial analysis segments Project Histories at the Change level. Each Change is treated as an individual Segment, and the Change is coded according to the modifications which it contains.

The initial coding scheme is presented in Chapter 5. It included three dimensions of coding; these were the error dimension specifying any error present in a Change, the action dimension describing

the task on which the student was working on in that Change and the output dimension which described any problems with the output, such as the program crashing or a spatial element being misplaced. The application of three separate coding dimensions was based on the observation that a single dimension failed to capture a Change's context sufficiently. In retrospect, it revealed a problem central to the segmenting approach based on Segments comprised of individual Changes. When individual Changes are categorised, it is not possible to take into account the context in which they occur, leading to very low-level and fragmented categories.

During its application the classification scheme was expanded and reworked to better represent the underlying data. This process is explained in more detail in Section 5.2. This scheme was then used to code ten selected Project Histories.

In practice, it was found that the first iteration of the classification scheme did not work well. It was too difficult to apply as some of the codes overlapped, and many of the codes also did not have any easily interpretable meaning. This was especially true of codes of the 'output' dimension which focused on a superficial 'what-is-observed' view of the output. The categorisation scheme is laid out in detail in Section 5.2.

A new version of the classification scheme was developed to address these limitations. The shallow and unproductive 'output' dimension was replaced with a 'problem' dimension that coded the underlying problem rather than the observed output. More importantly the action codes were collapsed; where multiple codes described similar Changes without adding useful information these codes were merged. The resulting set of codes was used for all three dimensions (action/error/problem). The resulting codes ended up describing the major types of task involved in the assignments which in turn was a reflection of key concepts from the Computer Graphics syllabus described in Section 2.2.2, involving categories such as 'Spatial' for the coding of spatial programming actions and 'Pipeline' for the coding of actions relating to the OpenGL pipeline. The development of the classification scheme is described in detail in Section 5.2.1, whereas the categories of the final classification scheme are described in Section 5.2.2. The reworked classification scheme was then used to re-code all ten Project Histories, an action which required substantial time and effort. Results from the analysis of these coding results are presented in Section 5.3.

As described in more detail in Section 5.5 the application of the improved classification scheme with the Change-Coding method still faced limitations. The codes themselves were not expressive enough to capture the properties of the underlying data fully because the analysis of individual Changes lost

the context in which these Changes occurred. The result was a set of low-level codes from which few features could be abstracted, which meant that it was not possible to connect these codes to form themes or concepts of student programming processes. In addition, since the codes did not capture many of the features of the problems underlying student programming, when data was re-analysed this required the researcher to re-establish an understanding of Changes. This is time-intensive since it requires the comprehension and debugging of poorly documented source code.

Analysis of codes when using the Change-Coding method was quantitative rather than qualitative. Rather than comparing and contrasting Changes, relative frequency of Changes along the different dimensions was analysed. This makes Change-Coding a mixed data analysis method rather than an analysis method based on GT. This was beneficial in that it provided a quick overview of the data which helped guide further analysis. However, by reducing the richness of the data, much of its deeper meaning was lost.

While the Change-Coding method did provide a high-level macro view of student activity and the distribution of student effort in terms of tasks worked on during their assignment programming, part of the Grounded Theory approach involves going back to the data after coding to refine codes and eventually to develop themes. The Change-Coding method's limitations meant that the Grounded Theory methodology could not be applied at a level of depth at which it could lead to the production of well-developed themes.

However, the development of codes, later to become categories, did proceed along GT lines. Initial codes were revised, added to, removed from and updated when a new type of Change made this necessary. Thus, while Change-Coding was a mixed method rather than a GT method, the GT aspects of the method laid the foundation for the GT-based Segment-Coding method which is discussed next.

#### **3.4.3.2 Segment-Coding**

When analysing the results of the application of the Change-Coding method it became apparent that its chief limitation was not due to the classification scheme but rather due to the approach chosen to segment the Project Histories. The choice of individual Changes as Segments (the unit of analysis) caused the loss of context in which the Changes occurred. This makes meaningful coding difficult, and also makes it difficult to reconstruct the context and hence the meaning underlying a Change from its codes. In terms of the context of interview transcript analysis in which GT is usually applied this would be equivalent to choosing to code individual words instead of sentences or paragraphs. The result is a shallow coding which produces what Glaser (2008) is termed "Conceptual Floppery", a

multitude of concepts which cannot be related to one another because they contain too little meaning.

In addition, evaluation of Changes in a purely quantitative way (by comparing relative frequency of codes) reduced the potency of the Change-Coding method by eliminating much of the depth present in the underlying data.

To address this problem, a new method of segmenting was utilised. The resulting data analysis method is called the Segment-Coding. Segment-Coding requires the researcher to first segment the Project History to be analysed by identifying and grouping related Changes into Segments. Changes are related when they address the same underlying task or problem. After having produced these Segments, the researcher then codes them in much the same way as was used to code individual Changes.

The Change-Coding classification scheme was applied to this new type of Segment, but the multi-dimensional approach to coding (which had been introduced due to the perceived issues with coding individual Changes) was dropped. Instead, a single coding dimension classifying the problem/task underlying the Segment was used. However, when compared to coding using the Change-Coding method the coding of Segments allows for the coding of entire problems potentially involving different attempted problem-solving strategies, misunderstandings and error types.

Instead of simply counting the frequency of Segments falling into different categories, a very detailed GT memoing process (much of the outcome of which is captured in the Appendix, Section 9.7.6) produced detailed insight into the nature of student problem-solving rather than just a high-level overview. This new Segment-Coding method is presented in Chapter 6.

Segment-Coding also required the development of new analysis functionality in the SCORE Analyser (which will be described in Section 6.2) to enable the researcher to examine not just an individual Change but also the context in which it occurs. This functionality is based on the concept of *Line Histories*. An algorithm for the generation of Line Histories was developed as part of this research project. The method of evaluation is described in Section 3.5.

The Segment-Coding method was applied to the same ten Project Histories that were used for the Change-Coding method. Results of the segmentation of Project Histories are presented in Section 6.5. After segmentation each student's Project History had been divided into Segments. These Segments were then coded and their contents described in more detail in associated Segment memos. Detailed analysis and comparison of Segments (as per the constant comparative method outlined in Glaser & Strauss, 1967) with one another showed that Segments assigned to the same

classification categories were significantly different to one another which led to the development of a set of sub-codes presented in Section 6.3.4. For example, General Programming was broken down further into categories related to algorithmic problem-solving (Computer Science Concepts), object-oriented problem-solving (C++ Object-Oriented Programming) and basic syntax and semantic errors (Syntax/Semantics). In terms of the terminology of the coding framework presented by Saldaña (2009) the classification categories described in Section 5.2.2 serve as 'categories', whereas the sub-codes serve as 'codes'.

The process of Segment analysis involved continual comparison of Segments coded in the same category as well as between Segments of different categories, leading to understanding of student problem-solving improving incrementally and being stored in Segment memos, which over time led to a deep and thorough understanding of student problem-solving. The comparison of coded Segments led to the discovery of commonalities in student problem-solving relating to problems. The central focus of this analysis was the failure of student problem-solving in different contexts. Commonalities in Segments associated with breakdowns of student problem-solving led to the identification of issues which affected different phases of the student problem-solving process described in Section 6.6.6. The phases of the problem-solving process identified during the development of the different issues are the basis of a model of student programming problem-solving which is presented in Section 6.6.7.

Line by line coding (Charmaz, 2006, p. 50) is analogous to analysing what happens from Change to Change. As Charmaz suggests, this allows the researcher to remain open to data in order to see nuances in it while building up understanding. It enables the researcher to gain close look at what participants say (in our case do) and struggle with. This coding helped to develop categories during Change-Coding (Charmaz, 2006, p. 51).

Incident to Incident coding is analogous to coding Segments and describing them (as in the Segment-Coding method). This allows identification of properties of emerging concepts (Charmaz, 2006, p. 53). As Charmaz suggests this is important because the data to be analysed is observational, and hence the Change-by-Change coding does not give a sense of its context (Charmaz, 2006, p. 53).

It should be noted that even the Segment-Coding method includes a measure of quantitative analysis. As is discussed in Section 6.5.1, the dimension of Segment difficulty is initially measured primarily by analysing the number of Changes in a Segment, which is a quantitative measure. However, this measure serves only as a rough guideline for the following research, mainly to exclude

small and hence insignificant Segments. The main process of analysis was qualitative and involved a detailed memoing process (results are presented in Appendix Section 9.7.6).

The following three data analysis methods should be seen as complementary to the Segment-Coding method. The Segment-Coding method was the primary data-analysis method for the analysis of student source code developed during this research project.

### **3.4.3.3 Segment Feature Analysis**

In addition to the GT-based qualitative analysis of Segment contents discussed in the previous sections, analysis of features of Segments discovered during analysis of Segment contents was conducted to discover whether Segments falling into different classification categories show any qualitative differences. Details of this analysis are presented in Section 6.7.

This analysis focused on the average time spent per Change for Segments of different classification categories. Statistical analysis was carried out to determine whether average time spent per Change differed based on Segment classification category since such differences could indicate a difference in student approach to the solution of Segments falling into different categories. In performing this analysis the techniques of linear modelling, analysis of variance and Tukey's honest significance test were used. These were carried out using the R statistical computing software environment<sup>27</sup>.

Evaluation of Segment feature data using statistical methods raised a new question regarding the difference in average time between different types of Segment. To answer this question a more detailed analysis of selected Segments including the examination of individual Change times was carried out. This analysis involves examining two different types of Segment related to spatial programming and comparing the ratio of Changes falling into the shortest 25% of Changes to those falling into the longest 25% of Changes, as well as a comparative qualitative analysis of the contents of the two different types of Segment. Due to the relatively small sample size of Segments examined this analysis was placed in the appendix, Section 9.7.8.4.

Since Segment Feature analysis consists of quantitatively analysing Segments which are the result of the application of a qualitative data analysis method (Segment-Coding), Segment Feature analysis is a mixed method.

### **3.4.3.4 Animation Analysis**

In addition to the qualitative analysis of different types of Segments, one type of Segment was isolated for further quantitative analysis. The implementation of a certain spatial task (the

---

<sup>27</sup> R: An open source programming language and software environment for statistical computing and graphics (<http://www.r-project.org/>).

implementation of avatar animations (see Appendix Section 9.3.6.2 for a description of the avatar animation task) was found to cause significant difficulty to all students during Segment-Coding analysis. For this reason, the ratio of correct to total programming actions in this type of Segment was calculated for each of these Segments and compared to a value based on how well students would do if applying these programming actions by chance rather than through the application of spatial problem-solving abilities. The method itself is discussed in more detail in Section 6.6.1.2 while results from the examination of animations are discussed in Section 6.6.5.

### **3.4.3.5 Free-Answer Questionnaires**

In addition to the analysis of source code, students were also asked to fill out questionnaires as part of their course work. These questionnaires included a quantitative part (a seven-point Likert scale) as well as a qualitative part (open-ended questions). The analysis of these questionnaires is presented (briefly) in Section 3.6.4.

### **3.4.4 Adherence to and deviation from Grounded Theory**

This section describes how the primary data analysis method developed (Segment-Coding) adheres to and differs from GT as it is usually applied. The Segment-Coding (and to some extent the Change-Coding) methods adhered to GT in several key ways. The constant comparative method of analysis (Glaser & Strauss, 1967, Chapter 5) was applied throughout the Change-Coding and Segment-Coding process, with incidents being compared to one another continuously until the completion of the project, and with incidents being compared to the properties of their categories and later their sub-categories.

The research project involved both a version of ‘line-by-line coding’ (Charmaz, 2006, p. 51) in the form of Change-Coding and analysis of individual Changes, as well as ‘incidence-by-incidence’ coding (Charmaz, 2006, p. 53) via the coding and analysis of Segments. The process was captured via the development of memos, many of which can be found in the Appendix, Section 9.7.6.

Open coding (see Section 9.3.3.5 for an explanation of open and selective coding) and the development of core categories occurred mainly during the application of the Change-Coding method. This involved the development of categories, which included the deletion, addition and modification of categories throughout the early stages of the research project, as well as changes in how these categories were applied. This process is described in more detail in Section 5.2.

Selective coding focusing on the completion of category definitions and the development of subcategories occurred during Segment-Coding, and the process is captured in the memos relating

to categories presented in Sections 5.2.2 and 6.3.4 as well as in the Segment memos presented in Appendix Section 9.7.6.

This research project also involved extensive theoretical sampling. When questions arose during the analysis of a Change or a Segment, further analysis focused on finding and examining or re-examining similar Changes or Segments. When this led to the properties of categories being modified or broken down, analysis involved returning to previously coded incidents to tease out additional details, leading to the development of further properties or theoretical nuances.

The research project also made heavy use of memoing. Some memos relate to single Changes (equivalent to a line). More commonly, memos relate to a Segment (equivalent to an incident). While some of the earlier memos are no longer available, many of these memos can be found in the Appendix, section 9.7.6. Furthermore, the descriptions of categories found in Sections 5.2.2, 6.3.4 and 6.5.3 are also (developed) memos which break the categories and sub-categories down into their relevant properties. Finally, the overarching themes presented in Section 6.6.6 can be understood as memos and / or as theoretical codes.

While axial coding was not formally applied, as Charmaz suggests (2006, p. 61) the research project did involve extensive comparison of categories, which led to the development the aforementioned themes.

Finally, these themes were drawn together to form a framework theory during the final integration phase, presented in Section 6.6.7, as suggested by Glaser & Strauss (1967).

However, the data analysis method presented here also differs from GT in some ways. One reason for this is the nature of the data. It contains none of the implicit social meaning that natural language contains, but it is context-dependent in a different way. Each programming action can only be understood in the broader context of the other related programming actions. However, when seen in context, such actions can be part of a continuous 'narrative' of great length. The way in which GT is applied as part of the Segment-Coding method it focuses very strongly on process and hence there is far less emphasis on the concept of dimensionality. Only two core dimensions are applied. These are difficulty (measured mainly quantitatively) and the issue with which a category is associated.

Furthermore, the use of the 'qualitative' metric of the number of Changes in segments for difficulty could be seen to lend a 'mixed' aspect to Segment-Coding. However, this was only used in a screening fashion and was complimented by thorough by quantitative analysis and memoing.



Perhaps most importantly, it is difficult to determine whether analysis reached theoretical saturation, which is generally considered essential for the development of a full-fledged theory (Strauss & Corbin, 1998). Further research efforts are needed to evaluate whether identified categories are theoretically saturated. In addition, some CG topics are not addressed in this dissertation and hence there remain some categories yet to be discovered. However, the categories and themes developed as a result of this research project provide a solid basis for future research.

In conclusion, the proposed Segment-Coding data analysis method should be seen as based on the principles of GT rather than adhering strictly to the GT method. The primary reason for this is the different nature of the data analysed in this research project. The analysis presented in this thesis demonstrates its efficacy as a method of source code analysis.

### **3.5 Software Engineering methods to support the analysis of source code**

The Segment-Coding data analysis method's main limitation is that it is difficult and time-consuming to apply.

One reason for this is that it is difficult to remember the context in which a modification occurs. To ameliorate this issue, a mechanism to generate so-called *line histories* was implemented. This allows the researcher to examine the context in which individual modifications occur.

Another reason is that the process of identifying sets of related Changes (related in resolving a particular problem) is a very time-consuming process, especially for assignments which consist of thousands of versions. For this reason, a mechanism called machine-segmenting was developed. This mechanism aims to automate the discovery of sets of related Changes (Segments).

Both these methods are based on software engineering methods. An overview of these methods is presented in Section 3.5.1. An overview of the methods designed to evaluate these software engineering methods is presented in Section 3.5.2.

Again, the detailed description of these methods is left to later chapters, closer to their application to data.

#### **3.5.1 Software Engineering Methods to assist Segment-Coding**

##### **3.5.1.1 Line History Generation**

The Segment-Coding method described in Section 3.4.3.2 requires the development of a method of detailed analysis of source code. The new functionality which enables detailed analysis (described in Section 6.2) is based on the concept of *Line Histories*. Line Histories store all modifications occurring

to a line in the Project History. The implementation and evaluation of the Line History generation algorithm is described in detail in Section 6.2.1.2. In addition to enabling detailed analysis, Line Histories were also utilised in Machine-Segmenting, described in the next section.

Since the generation of Line Histories was central to other parts of the project, it was important to demonstrate that the Line History generation algorithm is effective at producing accurate line histories. Evaluation of Line Histories involved a manual analysis of generation results, identifying lines which were incorrectly detected as the same line (false positives) and lines that were incorrectly not detected as being the same line (false negatives), as well as the total number of lines that were correctly identified as being the same line from one document to the next (true positives). Based on this data precision and recall for the Line History generation algorithm is reported in Section 7.3. The accuracy of the proposed method exceeded 95%, which shows that the proposed Line History Generation algorithm is effective at discovering Line Histories.

### 3.5.1.2 Machine-Segmenting

Machine-Segmenting was developed as a method of supporting the efficient segmenting of Project Histories for the application of an analysis method such as GT since manual segmenting is time-consuming. In line with GT requirements, the Machine-Segmenting methods developed as part of this research project are content-agnostic, based only on the relationship between lines of source code using the concept of ‘co-change’ as described in Section 2.5.2.2.

The development of Machine-Segmenting is aimed at fulfilling the research goal of providing a method for source-code-level analysis of Project Histories.

The approach used in the evaluation of Machine-Segmenting algorithms was developed by the researcher and is based on the comparison of the results produced by a random segmenting algorithm to the results produced by the machine algorithm, both of which are compared against a ‘correct’ segmenting produced by the researcher. The developed Machine-Segmenting algorithms produced statistically significant results for all ten project histories to which they were applied. This shows that they are similar to the Segments produced by a human analyst (the author) to a statistically significant extent. This means that Machine Segmenting is a promising method for the identification of Segments. In future research projects it will facilitate the application of the Segment-Coding data analysis method. Two metrics called *spread* and *fit* which measure the quality of Segments was also produced.

There will frequently be references to ‘interesting’ Changes in the context of Machine-Segmenting. As will be discussed in Section 6.5.1, Segment size is used as one of the main metrics for determining

whether a Segment's Changes are dealing with a significant or a trivial problem. For this reason, Changes are considered 'interesting' if they are part of large Segments.

Since the Machine-Segmenting algorithms were developed based on experiences with the Segment-Coding method they were available only during the later stages of the research project and hence did not contribute directly to the analysis of Computer Graphics programming.

The proposed evaluation method and evaluation framework is presented in Section 7.4 while Sections 7.7, 7.9 and 7.10 present results for the evaluation of machine generation algorithm implementations.

### **3.5.2 Evaluation of developed software engineering methods**

The evaluation of the developed software engineering methods is based on quantitative analysis of data produced by these methods. Four evaluation metrics are described in this section. One relates to the evaluation of both line history generation and machine-segmenting (Precision and recall) while three relate to the evaluation of machine-segmenting only (Actual / Expected ratio, Calculating P-Values using Simulation, Spread and Fit).

#### **3.5.2.1 Precision and Recall**

Both the generation of line histories and identification of interesting changes are in some sense Information Retrieval<sup>28</sup> tasks. The most commonly used measures of performance of Information Retrieval algorithms are recall and precision (Manning, Raghavan, & Schütze, 2008, Chapter 8). Therefore, precision and recall were utilised to measure the performance of these algorithms; definitions of precision and recall and additional details are provided in Section 7.4.1.1.

#### **3.5.2.2 Actual and Expected Ratio**

For the analysis of Line-History generation, precision and recall are not sufficient because without any further context it is unclear what constitutes a 'good' precision or recall value as this depends on the data in question. For example, if an assignment consists entirely of 'interesting' Changes, then simply selecting all Changes would produce perfect accuracy and precision. If 'interesting' Changes are rare, it becomes more difficult to determine interesting changes correctly.

For this reason, a new metric called the Actual / Expected ratio (A/E ratio) was introduced. This metric compares the actual recall/precision to the expected recall/precision for an algorithm which 'selects' all Changes. This measure is explained in more detail in Section 7.4.1.2.

---

<sup>28</sup> Information Retrieval: An area of study involving the searching and retrieval of documents or the content of documents.

### 3.5.2.3 Calculating P-Values using Simulation

The Actual/Expected ratio gives an indication of how well algorithm is performing. However, it does not show whether the algorithm is performing well to a statistically significant degree.

One way to measure the likelihood of identifying given number of Changes as interesting correctly could be to use hypergeometric binomial test as described in Section 7.4.2.2. However, for reasons described in Section 7.4.2.3 this is inadequate; it produces ‘optimistic’ (low) p-values. For this reason, a simulation approach was utilised. In this approach, a random algorithm randomly selected interesting ‘Segments’ a large number of times, and the number of interesting Changes correctly identified was compared to that of the actual algorithm. The details of this approach are described in Section 7.4.2.4.

### 3.5.2.4 Spread and Fit

While Simulation enables calculation of p-values, these p-values do not indicate how well generated Segments fit real Segments, only how well they do at finding ‘interesting’ Changes.

To discover how well generated Segments fit real Segments, two metrics called ‘spread’ and ‘fit’ are introduced.

In brief, ‘spread’ measures how spread out Changes belonging to the same ‘real’ Segment are across machine-generated Segments (ideally they would not be spread out at all, that is, all contained in the same machine-generated segment). A simplistic way to achieve perfect spread would be to select all Changes into a single Segment.

To counter-balance this, ‘Fit’ measures how well machine-Segments ‘fit’ real segments. It measures how many ‘real’ segments a machine-generated segment contains Changes. A perfect algorithm would assign each Change to the correct Segment, thereby achieving perfect fit. The naïve algorithm mentioned earlier (which assigns each Change to the same Segment) would achieve poor ‘fit’ since this Segment would contain Changes from many different ‘real’ Segments.

Both measures are defined and explained in more detail in Section 7.4.3.

## 3.6 Data

### 3.6.1 Description of Data-Gathering Approach

All data gathered were sourced from the elective unit *Comp330 “Computer Graphics”* which is a third-year course offered once a year (in the first semester) at Macquarie University. The course coordinator for both semesters during which data gathering was carried out was the primary researcher’s supervisor. The primary researcher was also involved in the teaching of the unit,

serving as instructor for Tutorials and Practicals and as lecturer for a third of the course. The text used was “Computer Graphics: Using OpenGL” (Hill & Kelley, 2001), a widely used text for teaching introductory Computer Graphics. The content of the book largely mirrors the topics proposed by the SIGCSE SIGGRAPH working group (Cunningham et al., 2004).

Data on the student programming process was gathered via the SCORE Eclipse plug-in developed as part of this research project which was provided to students as part of an Eclipse distribution to be used for all Comp330 programming (details on the SCORE plug-in can be found in Section 4.3). The SCORE plug-in automatically captured and stored all modifications made by students while they were working on their assignments and this data was then submitted as part of students’ final project submission. The analysis of this data forms the core of this research project. Two assignments, Assignment 1 and Assignment 3, were captured using the SCORE plug-in. Assignment 2 was a non-programming pencil-and-paper assignment which was not analysed.

In addition to the data gathered by the SCORE plug-in, students were also asked to complete reflection questions relating to their work on Assignment 3 which were submitted along with that assignment. The analysis of results is presented in Section 3.6.4.

Students were also asked to sit a 15 minute spatial ability test (the Purdue Visualization of Rotation test (Bodner & Guay, 1997). The results are not discussed in this thesis as the sample size was deemed too small to produce statistically significant results since data could not be obtained during the 2011 semester due to Comp330 not being offered in 2011 as explained in Section 3.6.2.

Interviews with volunteer students were also planned. Three such interviews were conducted, but because the tools required for proper analysis of student Project Histories were not ready at the time of the interviews the interviews were not deemed to have produced useful data and are therefore excluded from this thesis.

Ethics approval from the Macquarie University Human Ethics Committee<sup>29</sup> was obtained for the gathering and use of this data. While data was gathered from all students, students could choose to participate or not participated in this research project. Data from non-participating students would then be excluded from the data to be analysed. All data was analysed (and participation forms examined) only after the end of the semester to prevent any unintentional bias based on student participation in the research project.

---

<sup>29</sup> [http://www.research.mq.edu.au/for/researchers/how\\_to\\_obtain\\_ethics\\_approval](http://www.research.mq.edu.au/for/researchers/how_to_obtain_ethics_approval)

This data-gathering approach was used in two iterations of the unit in 2009 and 2010. Details on these iterations are discussed in the next section.

## 3.6.2 Data-Gathering Iterations

### 3.6.2.1 Semester 1, 2009 (Pilot iteration)

Data gathering was first carried out in 2009 as a means of developing and testing methodological approaches. The first assignment involved producing a simple graphics painting application which allowed the user to choose between different ‘brushes’, colours, stipple patterns and so on. The assignment was designed to give students an introduction to the OpenGL API, as well as to teach them basic 2D spatial programming and event-driven<sup>30</sup> programming. The third assignment required students to implement a three-dimensional game or simulation to develop their understanding of three-dimensional transformations and views. It was relatively unstructured, designed to allow students to engage creatively with the learning content. The second assignment for this and the following iteration was a pen-and-paper assignment which did not involve programming and was therefore excluded from the analysis.

Data gathering for the semester led to the identification and correction of bugs in the SCORE data-gathering plug-in mostly related to the handling of paths in non-Windows operating systems, which led to the SCORE plug-in not producing Project Histories for students using Linux or MacOS that semester.

More significantly the open-ended third assignment turned out to be too lax in its structure, with many students focusing much of their effort on non-core tasks not related to viewing or transformations. This led to the development of a stricter, more structured assignment specification for the 2010 iteration of Comp330. Due to this laxness, assignments were difficult to compare to one another which made them hard to analyse, and most importantly many students focused on non-Computer Graphics tasks and engaged in very little spatial programming which meant that much of the analysis effort would have been spent on non-Computer Graphics topics.

Data from the 2009 iteration of Comp330 was used in the development of the initial version of the SCORE Analyser (the functionality is presented in Chapter 4) as well as the first method of project-history analysis developed, the Change-Coding method presented in Chapter 5. However due to the limitations associated with the data it was not analysed in-depth, with analysis moving on to data

---

<sup>30</sup> Event-driven programming: A programming paradigm in which program flow is driven by events; such events can be produced by interaction with the user interface (such as the user pressing a button) or by threads or any other arbitrary event source. In the context of this research project, events are usually generated by the user interface.

from the 2010 iteration once it became available. Data for the 2009 iteration may be analysed at some point but this analysis has not been conducted to date and no formal analysis of the data is presented in this thesis.

### **3.6.2.2 Semester 1, 2010 (Main data source)**

The second iteration of data gathering was carried out in the first semester of 2010. Of 26 students initially enrolled in the unit, 12 withdrew with 16 students remaining enrolled in the unit. Of these, 15 agreed to participate in the research project. The age range was 19-28 years, with a median age of 21 years.

Seventeen students completed the first assignment. Plugin problems prevented the use of two assignments and one was excluded due to the student deciding not to participate, which left 14 students' assignments for analysis. Fifteen students completed the second assignment, with one assignment's data not being gathered properly by the SCORE plug-in and one student deciding not to participate, which left 13 students' assignments for analysis. Eleven students produced analysable assignments for both Assignment 1 and Assignment 3.

Compared to the 2009 iteration, the specifications for Assignment 3 in the 2010 iteration produced far more structured assignments and ensured that students worked on three-dimensional transformation and viewing problems. Analysis of assignments revealed flaws in the specification, but as explained in the next section the Comp330 course was not offered in 2011 and hence there was no chance to improve on the 2010 assignment specification.

All analysis presented in this thesis is based on data from the 2010 iteration. The Project History data can be found in the electronic appendix and can be viewed using the SCORE Analyser which is also included in the electronic appendix.

The timeline for this research project included a third iteration of data gathering to be conducted during the Comp 330 unit in 2011. Unfortunately Comp 330 was not offered in 2011 due to low enrolment numbers which removed this opportunity for a refinement of the data gathering process and assignment specifications. As a result extensions of the SCORE plug-in based on 2010 experiences could not be implemented as part of this research project. Comp 330 is being offered in the first semester of 2012 and data are being gathered, but this data will not be evaluated as part of this research project since it is too close to completion. The chief researcher is not participating in the teaching of Comp 330 in 2012.

A detailed description of the specifications for Assignment 1 and Assignment 3 is provided in the appendix, Section 9.3.4.

#### **3.6.2.2.1.1 Assignment 1**

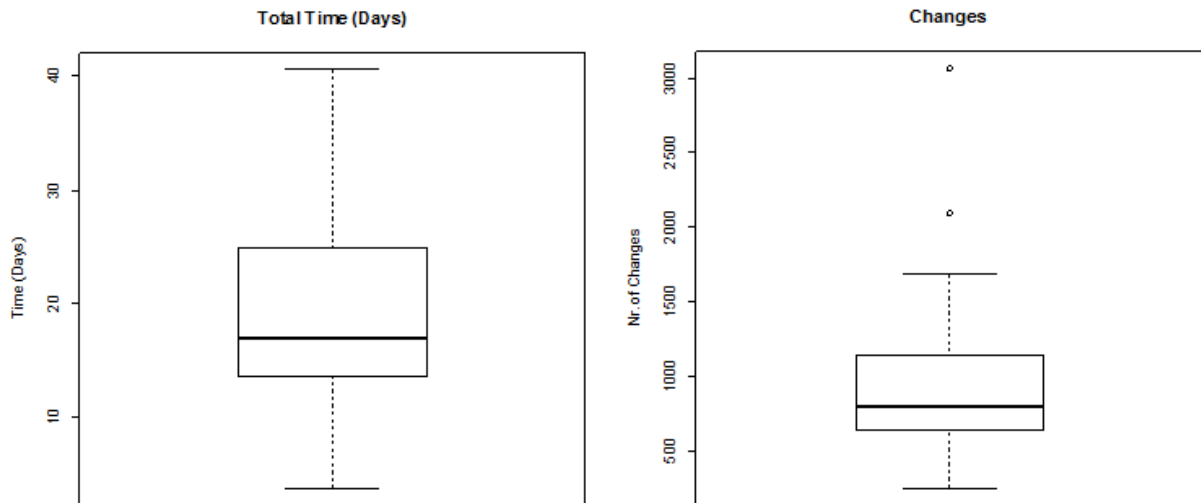
For the 2010 iteration of Comp330, Assignment 1 involved the implementation of a UML diagram creation tool. It was designed to familiarise students with the OpenGL API and to teach them two-dimensional spatial programming as well as introducing them to event-driven programming through the implementation of a graphical user interface. The assignment is described in more detail in Appendix Section 9.3.6.1.

Table 1 shows the average number of Changes, Average and Median time per Change as well as the average Total Time spent on the assignment for the 14 useable student submissions. On average, a student assignment consisted of 1062 individual Changes, with a median time of 17 hours spent on completing the assignment. Figure 6 shows a boxplot for the distribution of the number of Changes as well as the Total Time in hours for all fourteen assignments. The method used to calculate the total time spent by students is based on heuristics described in more details in Section 9.7.8.2. This heuristic may over or under-estimate time spent on any given individual Change; whether the effect on the sum of all times is an over or under-estimation is unclear, but the numbers appear to be reasonable and should at least allow for a rough estimation of time spent as well as for a comparison between different students (this applies to all time totals presented in the following sections).

**Table 1: Descriptive Statistics for time spent and number of Changes in Assignment 1**

	Mean	Sd	median
Number of Changes	1062.07	753.44	802.5
Avg Time per Change	74.23	23.44	76.28
Median Time per Change	43.82	17.2	44.5
Total Time (s)	70310.73	34935.55	61179.4
Total Time (hrs)	19.53	9.7	16.99





**Figure 6: Total Time / Total # of Changes Boxplots for Assignment 1**

As shown in Figure 6 all 14 assignments together involve a total of 14869 individual Changes and a total time of 273.43 hours spent on the assignment by the 14 students.

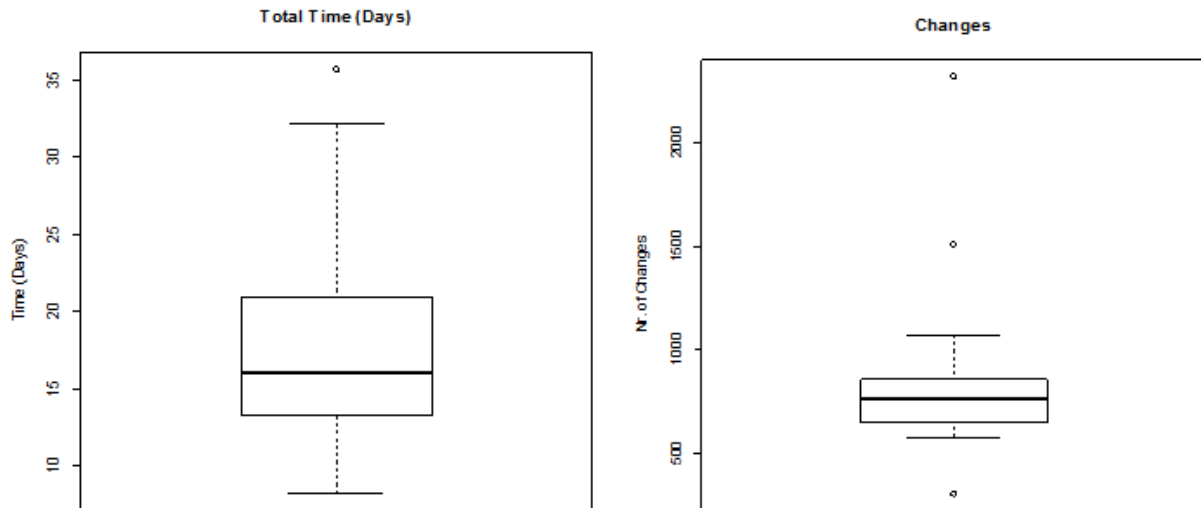
**Table 2: Total Number of Changes and Time spent across students in Assignment 1**

	Changes	Total (s)	Time (h)
Total	14869	984350.19	273.43

### 3.6.2.2.1.2 Assignment 3

Assignment 3 involved the assembly of an avatar using three-dimensional transformations, as well as the implementation of time-driven animations for the avatar and the implementation of different Views. It was intended to ensure students developed an understanding of three-dimensional transformations and the compositing of transformations, the implementation of Views and the use of time-driven programming models. The assignment is described in more detail in Appendix Section 9.3.6.2.

Data for averages of total number of Changes and Time per Change for the 13 useable Assignment 3 submissions are presented in Table 3. Boxplots for time spent and the number of Changes created by students is shown in Figure 7. On average students produced 891 Changes for Assignment 3, which is slightly less than the number of Changes produced for Assignment 1. Students spent an average of 16 hours on their assignment, which is again slightly lower compared to the 17 hours students spent on Assignment 1 on average. As shown in together the 13 assignments comprise 11589 Changes and an estimated 242 hours of programming.



**Figure 7: Total Time / Total # of Changes Boxplots for Assignment 3**

**Table 3: Descriptive Statistics for time spent and number of Changes in Assignment 3**

	Mean	Sd	Median
Change Total	891.46	513.58	764.00
Avg Time	81.64	25.18	75.78
Median Time	49.46	17.60	45.00
TotalTime (s)	67077.35	28875.17	57895.92
TotalTime (hrs)	18.63	8.02	16.08

**Table 4: Total Number of Changes and Time spent across students in Assignment 3**

	Changes	Total (s)	Time (h)
Total	11589	872005.54	242.22

The submission assignment also included a set of reflection questions intended to provide insight into student perceptions on Computer Graphics programming, to be submitted along with the project. Analysis of these questions is presented in Section 3.6.4.

### 3.6.3 Data Selected for Analysis

#### 3.6.3.1 Selection Criteria & Selected Assignments

As was shown in Section 3.6.3.3, assignment Project Histories included a large amount of data. For the 2010 iteration, Assignment 1 Project Histories totalled 14869 Changes and Assignment 3 Project Histories totalled 11589 Changes, for a total of 26458 Changes. At a conservative 30 seconds of analysis time per Change, this would work out to 220 hours of analysis for all assignments, or nearly

a month of analysis time at eight work hours a day seven days a week. This estimate only includes the coding, not the analysis of coding results and the comparison of coded items which would have required a considerable amount of additional time.

Furthermore, part of this research project included the development of the analysis method used to analyse the data. Iterative modifications and improvements of the methods meant that data had to be reanalysed several times, which at 27 straight days of analysis work would have been excessive.

As the magnitude of work involved in analysis became apparent, the project's scope changed to include the development of methods that could simplify the analysis process and reduce the amount of data to be analysed by focusing on the most important data in order to make a fine-grained analysis approach feasible. However, the development of these methods required evaluation of different approaches based on a comparison with a 'correct' solution derived via manual analysis.

To make analysis feasible a subset of Project Histories was analysed. Five students' first and third assignments were selected for manual analysis. One criterion for the inclusion of a student's assignments was that that student had at least attempted most tasks for both assignments. Some students attempted very few tasks and analysis of their assignments would not have yielded good data for analysis purposes. Two of the chosen students had been at the top of the class, one had been a good performer, one had performed adequately and one student had performed relatively poorly in both assignments despite attempting most tasks and investing a good amount of time, especially for the first assignment. It was hoped that this cross-section of performance levels would provide the best level of coverage of the types of problems faced by different students given the small sample size.

Ages of the five selected students ranged from 19-25 years, with a median age of 21 years, matching that of the population from which they were selected.

The five students' ten assignments totalled 9784 Changes, which using the same 30-second per Change formula yields an analysis time of around 81 hours or around ten work days. Given the different iterations of the analysis method, the real time spent on analysis was a multiple of the ten days predicted for a single analysis run. Since the application of methods was found to be very time-consuming part of the aim of this research project was to develop machine methods to speed up the process by automatically segmenting Project Histories. This led to the development of the Machine-Segmenting approaches described in Chapter 7.

These ten Project Histories from the 2010 iteration were used both for the analysis of student problems while learning Computer Graphics, presented in Chapters 5 and 6, as well as the

development of the SCORE Analyser and Machine Segment identification methods presented in Chapters 4 and 7.

While it was hoped that the remaining seventeen assignments could be analysed using the developed machine-approaches to identify ‘interesting’ areas of programming and cut down analysis time, the machine approach was still undergoing significant modification and extension in the latter stages of the research project and a lack of time ultimately prevented the inclusion of machine analysis of the remaining assignments as part of this research project. The analysis will be conducted after the completion of this research project.

### **3.6.3.2 Topics Examined**

Eight topics were mentioned in the suggested Computer Graphics curriculum reviewed in Section 2.2.2. They are listed here:

- Transformations
- Modelling: primitives, surfaces, and scene graphs
- Viewing and projection
- Perception and colour models
- Lighting and shading
- Interaction, both event-driven and using selection
- Animation and time-dependent behaviour
- Texture mapping

This research project focuses on students’ first experiences with Computer Graphics programming, and data-gathering was conducted as part of a one-semester third-year unit which is designed to provide a basic understanding of Computer-Graphics programming. Because of the limited amount of material that can be conveyed in a single semester, some of the core topics could not be treated in detail. This study focuses on transformations, modelling, viewing and projection, interaction and animation. These topics were seen as the hard core of the topic and were hence emphasized. Lighting is treated only very briefly, and shading and texture mapping tasks are entirely absent from the analysed assignments. Hence, this research project does not provide insight into student issues or problems in the areas of shading or texture mapping.

### **3.6.3.3 Data on Selected Student Assignments**

This section will present descriptive data on the Project Histories chosen for detailed analysis. The assignment specifications are described in more detail in Appendix Section 9.3.6.

#### **3.6.3.3.1 Assignment 1**

**Table 5: Descriptive data for Assignment 1 Project Histories**

Name	Nr. Changes	Avg. Time	Median Time	TotalTime in Seconds	TotalTime in Hours
Ida	1084	88.64	54	96085.76	26.69
John	717	104.22	60	74725.74	20.76
Michael	2090	60.08	34	125567.2	34.88
Thomas	1137	51.76	29	58851.12	16.35
Christopher	937	45.57	27	42699.09	11.86
Total	5965	-	-	397928.9	110.54

As Table 5 shows, the five Assignment 1 submissions analysed as part of this research project involved 5965 Changes and a total estimated student work time of 110 hours. The data also show that students spent differing amounts of time per Change on average, with Ida and John spending around a minute per Change whereas Michael, Thomas and Christopher only spent around half that time per Change.

**Table 6: Student completion of tasks for Assignment 1**

	John	Ida	Thomas	Christopher	Michael
Buttons	✓	✓	✓	✓	✓
Status Window	✓	✓	✓	✓	✓
Drop-down Menu	✓	✓	✓	Bug	✓
Line Style	✓	✓	✓	✓	No
User Interface Clipping	✓	✓	✓	✓	✓
Grid	✓	✓	✓	✓	✓
Object Persistence	✓	✓	✓	✓	Bug
Add Furniture	✓	✓	✓	✓	✓
Add Room	✓	✓	Bug	✓	✓
Change Object Colours	✓	✓	✓	✓	✓
Object Selections	✓	✓	✓	✓	✓

Move Object	✓	✓	✓	✓	✓
Parent Object	✓	✓	✓	✓	No
Parent-Child Move	✓	✓	✓	✓	No
Parent-Child Rotate	✓	Partial	✓	No	No
Assignment Tasks Completed	All	Almost All	Almost All	Most	Partial

Table 6 summarises how well the students whose assignments were analysed performed on Assignment 1. In-depth analysis of student problems is presented in Section 6.6.

Student John completed all assignment tasks correctly and submitted a very well-polished assignment.

Student Ida completed all tasks except the implementation of the Child-Parent rotate functionality. While the student did attempt the task she was not successful in resolving errors in the implementation.

Thomas completed all tasks. The only problem in Thomas's final submission involved an event-driven programming error in the creation of rooms.

Christopher's implementation utilising the GLUT drop-down menu has a bug in the event-driven code, which causes selection of the section option to activate the third menu option. The student did not attempt the Child-Parent rotation task.

Despite apparently spending a significant amount of time on the assignment (the longest of all five students), Michael is not successful in completing many of the assignment tasks. The student does not implement functionality allowing setting of line stipple or width. The student also utilises a poor, extremely inefficient algorithm for storing and drawing items. The student does not attempt to implement object parenting and hence also does not implement parent-child movement or rotation, and thus ends up not implementing a substantial amount of functionality.

### 3.6.3.3.2 Assignment 3

**Table 7: Descriptive data for Assignment 3 Project Histories**

Name	Nr. Changes	Avg. Time	Median Time	TotalTime in Seconds	TotalTime in Hours
John	707	106.7	68	75436.9	20.95
Michael	574	51.24	30	29411.76	8.17
Ida	669	101.01	60	67575.69	18.77
Christopher	1075	52.27	33	56190.25	15.61
Thomas	794	60.31	36	47886.14	13.30
Total	3819	-	-	276500.7	76.81

Table 7 presents descriptive data on the five Assignment 1 submissions chosen for analysis. The total number of Changes is 3819 and the five students spent an estimated 77 hours on completing the assignment. This is much lower than the 110 hours spent on Assignment 1; 26 hours of the difference are due to Michael who put in the most work (34 hours) in for the first assignment, but the least for the second assignment (8 hours). Time spent per Change follows a similar pattern to Assignment 1, with John and Ida spending almost twice as long per Change as Michael, Christopher and Thomas.

**Table 8: Student completion of tasks for Assignment 1**

	John	Ida	Thomas	Christopher	Michael
Avatar Assembly	✓	✓	✓	Partial	Partial
Avatar Movement	✓	✓	✓	✓	Partial
Avatar Animations	✓	✓	✓	Partial	No
Camera Views	✓	✓	✓	Partial	Partial
User Interface	✓	✓	✓	✓	No
Lighting	✓	✓	✓	✓	✓
Interactivity	✓	✓	✓	✓	No
Assignment Tasks Completed	All	All	All	Partial	Few

**Assignment outcomes are summarised in**



Table 8. These problems are investigated in detail in Chapter 6, Section 6.6.

John and Ida completed all tasks for Assignment 3. Both produced very polished assignments, receiving the highest marks for the unit.

Thomas also completed all tasks but produced very simple animations with too little limb movement, receiving less of the 'Quality' mark than John or Ida.

Christopher was not successful in implementing the avatar's assembly properly as he did not develop one of the key spatial concepts. The student also implemented only one simple animation rather than the required three. The student implemented all required views, but did not implement the required orbit/zoom functionality. The student's implementation also suffers from a programming bug relating to his implementation of a Scene Graph algorithm. The final result partially meets the assignment's specifications.

Michael also did not implement the avatar's assembly properly. Michael did not implement any real animations because he did not implement a proper time-based animation algorithm. Michael also did not implement any proper views, with the only implemented view having to be moved manually via key strokes. Michael did not implement functionality to allow the avatar to move in the avatar's facing direction, instead simply moving the avatar along the x and z axes. The submitted assignment did not meet most of the assignment specification's requirements.

### **3.6.3.4 Evaluation of Data**

#### **3.6.3.4.1 Initial Examination of Data**

Initial work in this research project utilised data from the 2009 iteration of Comp 330. Source code was analysed manually, with different version files being examined with text editors. Based on the early experiences with the Project History data, the first version of the SCORE Analyser was implemented. The first version of the SCORE Analyser was in essence a diff viewer which allowed the viewing of the difference between subsequent versions of a Project History's files. This allowed for an informal exploration of student actions and the nature of modifications performed on source code by students, as well as some idea of the enormity of the task of analysing such a large number of Changes.

#### **3.6.3.4.2 Coding Analysis of Data; First Classification Scheme**

The initial examination of 2009 data led to the development of the Change-Coding method of analysis (described in Chapter 5) and the implementation of Note-taking functionality in the SCORE Analyser (described in Section 4.4.2). These functionalities could be utilised to store structured

(Coding) and unstructured (note-taking) data relating to student modifications occurring in Changes. This in turn produced a better understanding of the nature of student modifications and the uncovering of structure in individual Project Histories relating to different tasks being worked on. It also became clear that a mere reading of source code often did not make clear the effect of student programming actions and so the SCORE Analyser was improved to allow the researcher to compile and execute versions of a student's code, and to modify and debug the source code using a built-in editor. This functionality is described in Section 4.4.

#### ***3.6.3.4.3 Coding Analysis of Data; Second Classification Scheme***

The initial coding classification scheme was found to produce hard-to-analyse data leading to unsatisfactory results. A new Coding scheme was developed (see Section 5.2 for details on the development of the coding scheme); this required the re-coding of all Project Histories. The new Change-Coding classification scheme was better suited to coding the data than the original scheme but still lacked descriptive power (as will be discussed in detail in Section 5.5). It produced only a surface-level description of analysed Project Histories and did not provide insight into the nature of student problems.

#### ***3.6.3.4.4 Segmentation Analysis of Data; Different Segmenting and Sub-Classification***

The limitations of the Change-Coding method in terms of both the significant time requirements and the shallow insight provided by the results led to the development of a new method of Project History analysis called Segment-Coding presented in Chapter 6. Segment-Coding Analysis is based on treating the Project History as composed of different Segments of related Changes, rather than as individual Changes. The development of a new approach again required the re-analysis of all Project Histories, but this time in two stages. The first involved the detection of Segments, the second the classification of these Segments. Application of the Segment-Coding method led to the development of classification sub-categories described in Section 6.3.4.

Segmenting Project Histories and understanding sets of related Changes proved to be very time-intensive and difficult when based only on a Change-by-Change browsing of the Project History. This led to the development of views summarising modification information either on a line-level (Line History View, Section 6.2.2.2) or a Change level (Change browser, Section 6.2.2.1). This functionality was implemented based on the development of an algorithm to generate Line Histories (described in Section 6.2.1.2) summarising all modifications made to a line in the project's history.

The application of the Segment-Coding method was found to provide more detailed insight into student problem-solving. Analysis provided not just a high-level overview of student work patterns but an understanding of student issues and misconceptions during problem-solving which the Change-Coding method's application did not. Results based on Segment analysis are presented in Sections 6.6 and 6.7.

#### ***3.6.3.4.5 Development of Machine-Segmenting***

The manual segmentation analysis provided a method of evaluating machine approaches for segmenting of Project Histories. The development of such an approach was considered crucial since manual Segment analysis, while producing more useful results than Change analysis, was still extremely time-consuming.

Machine approaches to identification of 'interesting' Changes or Segments were developed and evaluated against manually identified segments from the ten analysed assignments. Results from this work are presented in Chapter 7. The successful development of a machine approach presented in that chapter will make the Segment-Coding method of Project History analysis (and detailed Project History analysis in general) much faster to carry out and hence feasible for use in other research projects.

#### ***3.6.3.4.6 Overview of the phases of data analysis***

Data from the ten assignments was analysed and re-analysed three times, with each analysis involving substantial coding and re-coding as well as in-depth investigation of source code. Analysis of the data served to shed light on the nature of student problems as will be discussed in Chapters 5 and 6.

Analysis of the data also served to drive the development of the analysis method developed in this thesis, a method which enables detailed analysis of Project Histories. This method includes analysis software which enables the researcher to analyse Project Histories (the SCORE Analyser, Chapter 4) as well as the classification approach described in Chapter 6.

In addition the data was used to develop a method of Machine-Segmenting (Chapter 7) which will enable the efficient application of the proposed Segment-Coding analysis method in future research projects. This involved the implementation of an evaluation framework based on the comparison of results produced during Segment-Coding analysis with those produced by the different proposed machine methods.

### 3.6.4 Analysis of Student Perceptions

This section will provide a summary and analysis of student perceptions of Computer Graphics programming. Student perceptions regarding Computer Graphics programming were investigated using a set of reflection questions, to be submitted as part of their final third assignment submission. This data was gathered as part of the 2010 iteration of the introductory Computer Graphics programming course as described in Section 3.6.2. Fifteen students submitted reflection questions.

As the approach utilised in data-gathering and analysis was less robust than the research presented in this thesis the full analysis of reflection questions has been placed in the appendix, Section 9.3.4 rather than having a full chapter dedicated to the analysis.

Since student perceptions did address two of the research questions (**RQ1** and **RQ2**) and helped shed light on perceived student problems from another angle a short summary of the results is presented in this section. Student perceptions also helped guide further analysis with the analysis method developed as part of this thesis being used to probe these perceived problems in Chapter 6, specifically in the qualitative analysis of Segment contents presented in Section 6.6.

Student perceptions were probed utilising a questionnaire. The questionnaire consisted of six questions to be answered using a seven-point Likert scale, three open-ended questions which allowed students to describe their experiences with Computer Graphics programming and one rank order question in which students were asked to rate six Computer Graphics programming problem types in order of difficulty.

#### 3.6.4.1 The Role of Spatial Programming

All question types exposed student problems related to spatial programming. In the Likert questions most students agreed with the question “I was not always able to completely understand all problems spatially” and almost all students agreed with the question “Working on OpenGL problems during practicals and assignments often required me to think spatially”. Students also ranked spatial transformations and viewing tasks as the most difficult and second-most difficult tasks respectively in the rank order question.

Most students (11 of 15 in answers A1-A17) expressed some difficulty with spatial reasoning. The chief reason given was difficulty relating to spatial visualization (A4, A5, A7, A10, A12, A13, A14, A15, and A16). Ida (A4) wrote “Spatial thinking was the most challenging. I personally believe I am capable in terms of visualising, but it was still quite demanding of the imagination to create scenes without trial and error”. Student 14 (A7) indicates his difficulty with “Transformations and viewing mainly because I was unable to visualise what the commands would achieve in my head”. Student 7

(A14) suggested that his difficulty in visualization led to him utilising trial and error to implement transformations (A15): “I also still cannot visualise transformations well and often resort to trial (sic) and error”.

Several students also indicated difficulty with developing a working model of hierarchical assembly utilising composite transformations (A6, A9, and A17). Student 15 (A9) writes: “I was struggling with manipulating the matrix stack to exactly how I wanted. To be honest the theory is explained in a straight forward manner and backed up by good examples in the OpenGL programming guide. However applying it proved problematic”. This suggests that while material explaining these concepts was available, he was unable to develop these into a correct spatial understanding of the concepts being explained.

Another point brought up by several students (A1, A2, A8) related to the difficulty of understanding spatial actions in terms of the mathematical constructs implementing them. Student 8 (A1) wrote: “Thinking in 3D space, especially trying to calculate a point in 3D space is very hard”. Two students point out the difficulty of understanding OpenGL space in terms of local and global coordinate systems (A3, A11). Student 9 (A3) said that: “Rotation and movement was also difficult due to the fact there were a number of things needed considering (moving the camera around GLGuy, moving the world, moving GLGuy)”.

It should be noted that the four students who produced the best assignment submissions all noted difficulty with spatial thinking (Student9 - A3, Ida - A4, Thomas - A10, John - A12) suggesting that problems related to spatial programming are not limited to students who performed poorly in the unit. In fact, the ‘best’ students produced the clearest and most verbose descriptions of problems relating to spatial thinking.

### 3.6.4.2 Mathematics

Mathematics was also perceived as a difficult topic by students. Many students (6/15) also indicated struggling with the mathematics involved in completing the assignment (A18-A25). Student 16 (A25) states that “Mathematics is the most challenging area” because he “calculated each viewing angle based on trigonometry equations”. Christopher also indirectly expressed difficulty with trigonometric calculations (A23) as he mentioned polar coordinates (based on trigonometric calculation) as “slightly challenging”. Student 8 (A20) brought up his difficulty with debugging problems with his mathematical model for camera movement: “Not sure why the maths didn't work out, it's almost there but fails to perfectly sync up the camera with the head, unfortunately I couldn't figure out what exactly was causing the weird rotation when turning with the head and/or torso

tilted. Wasn't quite sure how to help diagnose the problem, tried heaps of trial and error tests but nothing improved it".

### 3.6.4.3 General Programming

General-Programming<sup>31</sup> related problems were the third most common issue raised by students with 5/15 students expressing problems with non-Computer-Graphics related programming. Several students (A27, A29, and A30) bring up the use of C++ as problematic, with Student 12 (A27) writing that: "I used to program with Java a lot in past one year. So I think C++ is too old and not as effective as using Java". Two students (A26, A33) found OpenGL difficult to learn. Student 7 (A33) writes that "The most difficult aspect is learning OpenGL itself and it took me long time to study it". Student 14 found himself limited by the OpenGL API and state machine model which he found too restrictive (A31) and the Eclipse IDE (A32) which he found did not offer sufficient debugging facilities.

The evaluation of reflection questions students submitted along with their third assignment did help shed some light on student problems. Spatial programming in particular was identified as challenging by many students in all question types.

### 3.6.4.4 Relationship of student perceptions to this study

The results go some way toward answering **RQ2** "Is 'spatial programming' a difficult area/topic of Computer Graphics programming for students?" in the affirmative. The open-ended questions also provided some answers to the research question **RQ1** "What kinds of problems do students learning Computer Graphics Programming experience?", with students identifying problems with spatial programming, mathematics, and with the C++ programming language. The significance of this finding is underlined by the best-performing students all raising this issue in their open-question answers which indicates that this is not an issue caused by insufficient student effort, nor is it one which only affects a subset of students who then tend to perform poorly.

The responses do not provide detailed insight into the nature of student problems. Even though many students gave long answers, they did not describe the precise nature of problems. Also, students' perceptions do not reveal how students addressed these problems during the implementation of their assignments. Furthermore given that students most likely answered the questions after completing their assignments they are unlikely to provide a detailed and accurate description of their problem-solving approach. Instead their recollection is likely to be tainted by various cognitive distortions. For example, they are more likely to be recalling issues faced during the latter phases of the implementation than in the earlier phases.

---

<sup>31</sup> General Programming: Programming related to general Computer Science concepts, the syntax / semantics of the programming language used (in this case C++) or the programming environment (Eclipse IDE)

To better address these questions instead of analysing student recollections about their problem-solving work, this thesis will go to the source and analyse actual student problem-solving by examining all source code modifications carried out by students during their work on assignment assessment tasks using the methods described in Section 3.2.

### 3.7 Methodology Summary

This chapter begins with an introduction to the terminology regarding source code analysis used in this dissertation (Section 3.2), followed by a brief description of the data to be analysed (Section 3.3).

The following two sections break down the methodology used in the development of the data-analysis methods (Section 3.4) and the software engineering methods produced to facilitate data analysis (Section 3.5).

Section 3.4 discusses the different data-analysis methods developed during this research project to analyse Project Histories. Different potential research methods for data analysis are discussed in Section 3.4.1. Because it is well-suited to small sample sizes and exploration of research areas for which there is little pre-existing research to base hypotheses on, Grounded Theory was chosen as the research method which underlies the primary data analysis method developed (Segment-Coding).

The Grounded Theory method is summarized in Section 3.4.2. Applying GT involves applying the Constant Comparative Method. The Constant Comparative Method involves the simultaneous coding, comparing and theorizing about the data being analysed. It involves two coding stages (some forms of GT include additional coding stages) and an integration stage.

The first coding stage is open coding. During open coding instances found in the data are broken down into categories. Instances which fall into the same category are continuously compared to one another. This process leads to the discovery of properties which apply to that category. The second coding stage is selective coding. During selective coding major categories are integrated to form a larger theoretical scheme. During the integration stage, theory formed during the selective coding stage is developed into a full-fledged theory relevant to all instances in the study.

Grounded Theory also involves a process referred to as ‘memoing’, which involves writing up theoretical findings in a narrative fashion. These theoretical memos are combined during the integration stage to form theory.

To advance the analysis, Grounded Theory utilises a process called theoretical sampling which involves finding data which answer open questions in the theory until all questions have been sufficiently answered. The state of all theoretical questions having been addressed is referred to as theoretical saturation.

The developed research methods are summarised in Section 3.4.3. Detailed descriptions are provided in later chapters. The first data analysis method developed is Change-Coding, discussed in Section 3.4.3.1. It is based on the coding of individual Changes. Changes are coded according to the task worked on, as well as the type of error (if any) present in the Change. Coding is followed by quantitative analysis of the coding results. In practice, this approach produced a high-level view of student programming but did not provide insight into the nature of individual student problems or their problem-solving attempts. The Change-Coding method is presented in Chapter 5.

Due to this limitation the Segment-Coding data analysis method was developed. It is presented in Section 3.4.3.2. It utilises the same classification categories as are used in Change-Coding analysis but involves the coding of Segments consisting of related Changes rather than the coding of individual Changes. This method allows for the coding of stretches of programming comprising whole student problems. Application of the Segment-Coding method provided detailed insight into both the nature of these problems as well as into the break-down of student problem-solving which made some problems especially hard to resolve. This analysis method led to the identification of student problem-solving issues described in Section 6.6.6, and to the development of a model of student problem-solving which describes how students encounter these issues during programming, presented in Section 6.6.7. The Segment-Coding method is the primary data analysis method developed and utilised in this research project.

While the core of the Segment-Coding approach involves analysis of Segment contents, this analysis was complemented by an analysis of Segment features. This data analysis method is discussed in Section 3.4.3.3. Segment Feature analysis is based on Segment time data generated by the SCORE Analyser software. Analysis of Segment Features provided insight into the difference between student problem-solving approaches applied to different spatial problems based on the ease with which they could be visualized using OpenGL.

A specialised method of analysis is Animation Analysis, discussed in Section 3.4.3.4. It is a quantitative method which involves calculating the ratio of correct to incorrect modifications to OpenGL transformation calls produced in project histories.



The final data analysis method utilised in this research project is discussed in Section 3.4.3.5. Questionnaires were utilised to discover student perceptions regarding difficulties in their CG programming. The questionnaires included both quantitative (Likert scale) and qualitative (open-answer) questions.

Overall, the Segment Feature analysis, Animation Analysis and Free-Answer Questionnaire Analysis data analysis methods provided a relatively limited contribution to the overall research method. The Change-Coding and the Segment-Coding (especially the Segment-Coding) methods form the backbone of data analysis conducted during this research project. Both these methods are based on the analysis of Changes (either individually or as Segments) produced by students.

Section 3.4.4 discusses how the data analysis methods developed as part of this research project (in particular, the Segment-Coding method) adhere to GT, and how they differ from how GT is applied in other research contexts.

The preceding sections discuss developed data analysis methods. Section 3.5 discusses the software engineering methods developed as well as the ways in which they were evaluated. These methods aim to facilitate the data analysis methods discussed in Section 3.4.3.

Section 3.5.1.1 discusses Line History Generation, which involves discovering all the Modifications of a particular source code line, which together are referred to as Line Histories. Section 3.5.1.2 provides an overview of Machine-Segmenting methods which automatically segment a Project History. Section 3.5.2 discusses various metrics by which the two methods were evaluated.

A description of the data gathered as part of this research project is presented in Section 3.6, focusing on the Project Histories gathered during students' implementation of the first and third assignments (Section 3.6.2.2.1.1 and Section 3.6.2.2.1.2). Students were selected based on whether they had at least attempted most of the major tasks for both assignments and the two best students' assignments were selected because experiences in the pilot iteration (Section 3.6.2.1) had shown that the best students explored concepts deeply. Descriptive data on these Project Histories was provided in Section 3.6.3.3. Selected Project Histories involved a total of nearly ten thousand Changes and roughly 186 hours of student work. The Project History data was analysed and re-analysed three times and was also utilised in the development of Line Histories and the Machine-Segmenting methods.

Questionnaires probing student perceptions of their work on the third assignment were part of the data collected. Analysis of these questionnaires is presented in Section 3.6.4. Analysis of the data shows that students, especially high-achieving students, identified spatial programming as a difficult

issue. Other problems including those related to mathematics, general Computer Science concepts as well as to the use of C++ as a programming language were also identified. While the analysis of student perceptions performed at the outset of this research project provided some insight into the nature of student problems, student description of their problems was not complete enough to provide a good understanding of the nature of student problems or the student problem-solving process. In addition such perceptions may also include inaccuracies since students are required to recollect programming actions which may have occurred days or even weeks earlier (depending on when students filled out the questionnaire). The analysis methods developed and applied as part of this research project are intended to provide more detailed insight and hence a more comprehensive answer to **RQ1** regarding student problems faced during Computer Graphics programming and **RQ2** relating to whether spatial programming is difficult for students.

The next chapter will introduce the analysis software developed as part of this research project which enables all analysis presented in subsequent chapters.

# 4 The SCORE Toolkit

## 4.1 SCORE Toolkit Introduction

The analysis of the student programming process required the development of software to capture and analyse Project Histories. A pair of applications, which together will be referred to as the SCORE toolkit (Wittmann, Bower, & Kavakli-Thorne, 2011), were developed for this purpose. The SCORE toolkit consists of a data-gathering SCORE Plug-in to generate Project Histories and the SCORE Analyser software (shown in Figure 8) to analyse Project Histories, both of which are described in this chapter. Both were developed by the primary researcher as part of this research project.

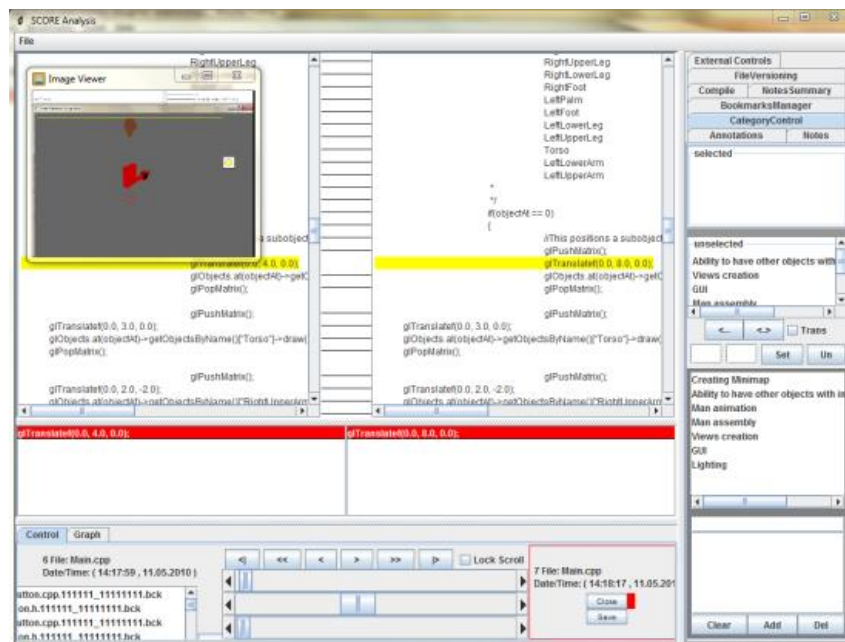


Figure 8: The main SCORE interface

Table 9 lists statistics relating to the implementation of the SCORE Analyser. Together, the analyser and general utility packages contain 118054 lines of code in 312 packages comprising 1058 classes. More details related to the architecture of the SCORE Analyser can be found in Appendix Section 9.5.1.

**Table 9: Implementation Statistics for the two projects which comprise the SCORE Analyser**

	SCORE	GeneralUtility	Total
Number of Methods	5756	848	6604
Number of Classes	1058	217	1275
Number of Packages	312	41	353
Total Lines of Code	118054	15428	133482
Method Lines of Code	78632	9640	88272

This chapter will describe the SCORE Analyser's basic functionality. Following chapters (Chapter 5 "Change-Coding", Chapter 6 "Segment-Coding", Chapter 7 "Line History Generation and Machine-Segmenting") will describe additional functionality implemented to enable different methods of analysis. The SCORE Analyser's functionality will be described in some detail. The description of SCORE Analyser in this and other chapters will be followed by a brief walkthrough of how it is used in practice.

The aim of these descriptive sections is to give the reader an accurate understanding of how analysis using the SCORE Analyser proceeds in practice as all research presented in this thesis is grounded in data through the described analysis methods and a good understanding of the practical application of the method is necessary for an understanding of how results presented in later sections were developed from data. Furthermore since the method itself is one of the main contributions of this research project a detailed description of its application is warranted.

The SCORE plug-in used to generate Project Histories will be described next (Section 4.3), followed by a description of the core functionality and views provided by SCORE in Section 4.4. Section 4.5 proposes some future work to improve the SCORE Analyser and plug-in.

While the body of this thesis provides a brief introduction to the SCORE Analyser, a manual on the use of the SCORE Analyser (Wittmann, 2012a) is located in Appendix Section 9.5.3 and in PDF format as part of the electronic appendix. A video demonstration of the SCORE Analyser's main functionality (Wittmann, 2012b) is also available at <http://www.youtube.com/playlist?list=PL5A1E9556055F67E2&feature=plcp> and is included in the electronic appendix to this thesis.

## 4.2 Rationale for the development of a Qualitative Data Analysis tool for the analysis of source code versions

The SCORE Analyser developed as part of this research project in part functions as a Qualitative Data Analysis (QDA) computer software package. Several other such packages (e.g. NVivo<sup>32</sup>) exist and are used by qualitative researchers. However, their feature sets are not optimally suited for the domain of source-code analysis. While they provide many options for coding data and writing memos (some even provide functionality which allows for the automatic coding of certain types of data) they do not provide any sophisticated methods for source code analysis. The SCORE Analyser provides many such features, making it a better platform for the analysis of source code.

For example, the SCORE Analyser provides line histories which track the development of individual lines of source code over the entire project. This feature is essential to the analysis of source code changes over long periods of programming.

The SCORE Analyser also provides the essential features such as the ability to compile and execute a project at any given Change. In addition, the researcher can also modify and debug source code. Without this ability, it would be very difficult to gain a full understanding of each version of a student's code.

In addition, the SCORE Analyser's extended functionality includes methods for the automatic identification of 'Segments' of related Changes (see Machine-Segmenting, Chapter 7), which has the potential to greatly reduce the amount of work required of the researcher.

The SCORE Analyser can also produce source-code specific metrics from identified Segments (see Section 6.7), such as average time taken per Change, or the number of Changes that compile successfully in a given Segment.

Because other QDA tools do not provide these features, they are unsuitable for the analysis of source code as stored in project histories. The SCORE Analyser provides an alternative custom-tailored for the specific domain of source code analysis.

## 4.3 The SCORE Data-Gathering Plug-In

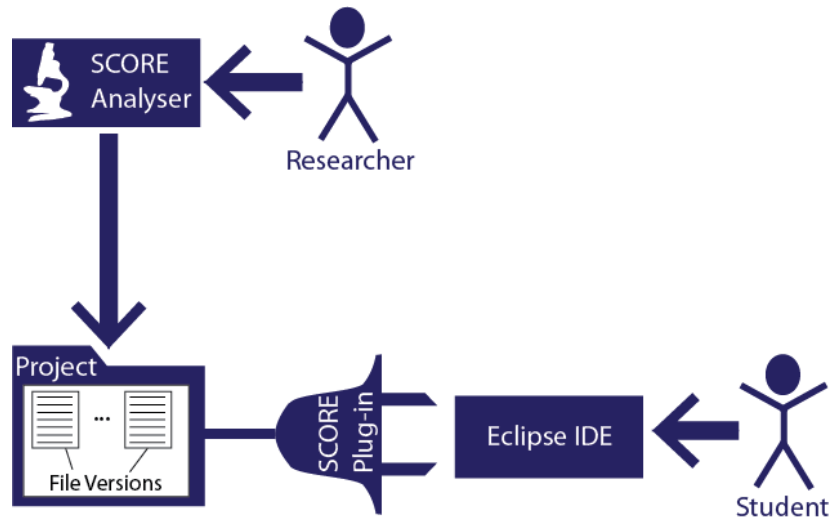
The SCORE Plug-in is used to recording student programming at a fine-grained level. It generates Project Histories containing one Change for every save or compilation action that modifies the text of a file that is part of the project. The SCORE plug-in is a plug-in for the Eclipse IDE. Currently the

---

<sup>32</sup> [http://www.qsrinternational.com/products\\_nvivo.aspx](http://www.qsrinternational.com/products_nvivo.aspx)

SCORE plug-in is configured to capture only C/C++ files, but it could be extended to use with other programming languages available for use with the Eclipse platform.

The SCORE plug-in uses a local storage model instead of communicating with a server, with the versioned code automatically being submitted by the students with their assignments since it is stored in a sub-directory of the main project directory. The way it interacts with Eclipse, the researcher and the student is shown in Figure 9.



**Figure 9: Relationship between SCORE, the student and the researcher**

Installation of the SCORE plug-in can be achieved by copying the plug-in into the Eclipse plug-in directory, or it can be distributed via an Eclipse plug-in site which can be distributed off-line or be put on-line, in which case students connect to the plug-in site via Eclipse to download the plug-in. Alternatively, the SCORE plug-in can be pre-installed into an Eclipse distribution which is then installed on laboratory computers and distributed to students as a zip file. The latter approach was chosen for data collection for this research project since it ensured that the SCORE plug-in was properly installed for all students. When the SCORE plug-in is installed it provides custom branding, including a custom project type from the New Project Wizard with a unique icon. This makes it simple to verify correct plug-in installation.

Once the SCORE plug-in is installed it functions almost transparently. The SCORE plug-in creates a new META-STRUCTURE folder in the project's root directory. When the student saves or compiles files, a version of the file is stored along with a timestamp in the META-STRUCTURE folder and an entry is created in a log file called FileLog.log inside the folder. Other data (which was not analysed as part of this project) including Eclipse user interface events and compilation messages are also stored in log files inside the META-STRUCTURE folder.

The version data is stored as the student develops his assignment. When submitting the assignment, either by zipping the project directory manually or by using the Eclipse project export function, the META-STRUCTURE folder will be zipped along with the student's submission source code. This data is then available to the researcher once the zipped project is extracted. Preparation of the data for analysis using the SCORE Analyser is discussed in Appendix Section 9.5.1.

## 4.4 Core Functionality and Views

Part of the contribution of this research project to the field of Computer Science research is the development of a method for the efficient analysis of Project Histories at a source-code level. Previous approaches analysing Project Histories have focused on metric-level (e.g. Mierle et al., 2005) or output-level (e.g. Jadud, 2006a) analysis. Such analysis can be automated through calculation of abstract metrics or machine-evaluation of compilation errors.

While source-code level analysis has the potential of providing a more in-depth contextual understanding of the student programming process than either metric-level or output-level approaches, such analysis cannot be fully automated. It requires understanding of student actions stored in a Project History, which in turn requires human reasoning. This combined with the large number of Changes that make up a Project History means that source-code level analysis will be more work-intensive. A successful method of source-code level analysis must provide tools to reduce this workload to make the application of the analysis method feasible in an actual research project. The SCORE Analyser developed as part of this project is intended to be such a tool.

This section focuses on tools and views provided by the SCORE Analyser to facilitate the analysis of source code. These tools provide the ability to examine source code text (Diff View, Section 4.4.1), store notes and categorise Changes (Note-Taking, Section 4.4.2) and edit and execute source code and produce screen captures (Execution-Based, Section 4.4.3).

In combination, these views and tools enable the researcher to efficiently browse and debug source code. This allows the researcher to identify whether a given Change contains an. Additional views and functionality complementing those described in this chapter will be introduced in later chapters to support different methods of analysis.

This chapter only describes the SCORE Analyser's core functionality. Functionality implemented as part of the development of the Change-Coding and Segment-Coding methods are described in detail in Sections 5.3 and 6.2.2, whereas Section 7.4.5 describes SCORE Analyser functionality for the Machine-Segmenting of Project Histories. The description of functionality is positioned close to the

analysis performed using that functionality. This is done to maintain a coherent narrative and to illustrate the way in which the functionality was applied in practice.

#### 4.4.1 Diff View

The SCORE Analyser is designed to allow researchers to better understand student programs as a sequence of file changes. At its core lies its ability to visualise these file changes, which in turn enables the researcher to understand, interpret and annotate sequences of changes.

The SCORE Analyser's main interface is a diff-style document version comparison view (see Figure 10). Initially the Diff View utilised a simple diff algorithm to detect lines added and removed between two documents. However the implementation of *Line Histories* described in more detail later in Section 6.2.1.2 enables the Diff view to match not only lines maintained from the previous to the current version, but also moved lines, modified (Mutated) lines as well as lines that are re-introduced from a previous document in which they were deleted (Ghost).

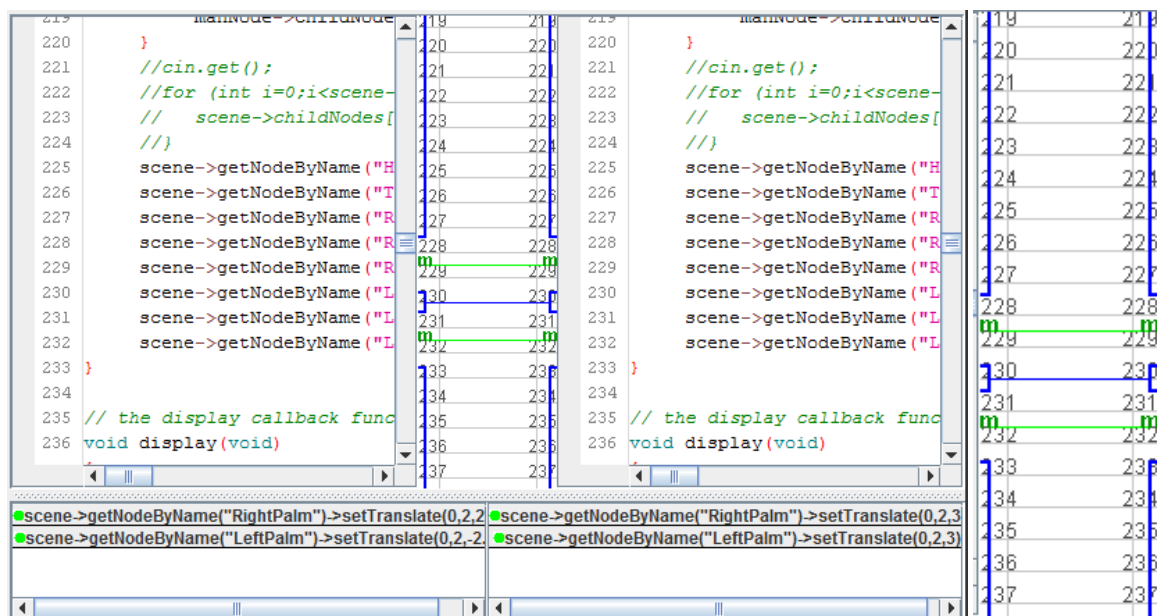


Figure 10: The SCORE Diff View on the left, a close-up of the connector pane on the right

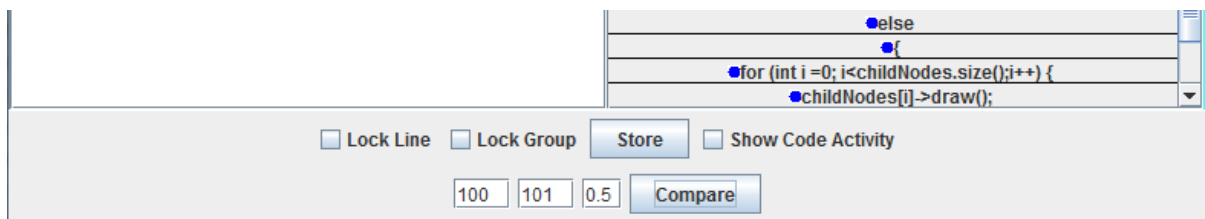
The Diff View displays a Change's previous and current version's source code text in two text areas, and the connector pane (in the centre) displays connectors which link groups of moved, maintained or mutated lines from the left (old) to the right (new) document. Maintained and moved groups of lines are linked by blue bracket connectors. Mutated groups are linked with lines marked with a green 'm' character. Ghost lines are marked with a purple 'g' character. Deleted lines (unmatched lines in the left document) and added lines (unmatched in the right) have no connectors.



Below the difference view are two summary panels. These contain the list of modifications from the previous to the current version. Clicking one of these changes scrolls the diff view to the line involved in that modification. Mutation modifications are marked with a green dot, additions or deletions with a blue dot (deletions appear in the left panel only, additions in the right panel only) and ghost modifications with a purple dot. Moved lines are not currently displayed in the summary view since a 'Move' may involve many lines and would clutter the list. Clicking on a line also sends an event to all of the SCORE Analyser's components, requesting information for that line to be displayed in the component if appropriate. For example, the Line History View (Section 6.2.2.2) will display the Line History for the selected line.

To navigate between a Project History's Changes SCORE provides navigation by overall position of the change or navigation through changes for a selected file via scrollbars.

While the Diff View compares previous and current versions of source code, the *Arbitrary Diff View* can be used to compare any two arbitrary versions of the same file by inputting the version number of each in the control panel shown in Figure 11. This enables the user to quickly produce a summary of all the Changes that occurred between two different versions.



**Figure 11: The bottom row shows the Arbitrary Diff View's controls**

Use of the 'Diff View' is described in more detail in the SCORE Manual, Section 9.5.3.3. A video demonstration of the 'Diff View' is available at <http://www.youtube.com/watch?v=P3IDS1g0bh4&feature=g-upl> and in the electronic appendix.

#### 4.4.2 Note-Taking

Two Views allow the researcher to informally annotate each Change in a student's Project History. The Category View (Figure 12 left) allows the researcher to create categories and then assign Changes to one or more categories which can be defined on the fly. For example, the user might define categories for user-interface implementation, drawing of icons and general programming related to creating data structures.

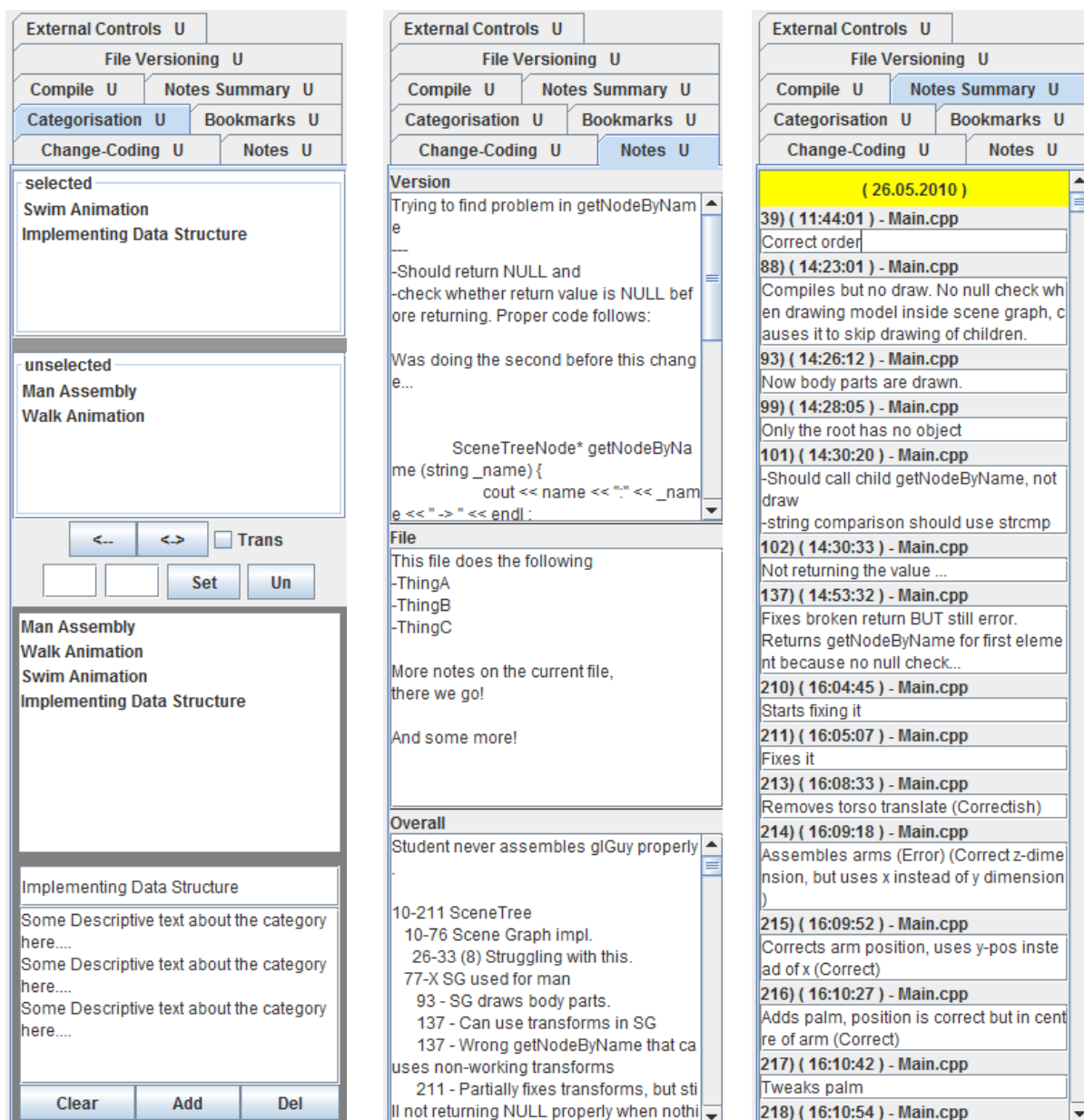


Figure 12: The Category View, the Note View and the Note Summary View

The Note View (Figure 12, middle) is used to store the user's text notes associated with individual Changes, with Versioned Files<sup>33</sup> or with the project as a whole. In Grounded Theory parlance, it enables the memoing process by storing insight gained during analysis. The Note View is updated every time a new Change is displayed in the main Diff View. The Version field accepts user notes regarding the current Change. The File field enables user input of notes containing the File to which the current version belongs. The Overall field can be utilised to store notes for the whole assignment (and is hence not changed when a different Change is viewed in the Diff Viewer).

<sup>33</sup> Versioned File: A file that has been versioned by the SCORE plug-in; a Versioned File consists of one to many Changes, each representing one version of that file as produced during the lifecycle of the project

The Note Summary View (Figure 12, right) presents Version notes for Changes sequentially in a single window. Notes can be written to a summary text file. Notes are also displayed alongside Change data in some other components. Notes are stored as and loaded from text files.

Use of the Note Taking facilities is described in more detail in the SCORE Manual, Section 9.5.3.5. A video demonstration of the Note Taking facilities is available at <http://www.youtube.com/watch?v=P3IDS1g0bh4&feature=g-upl> and in the electronic appendix.

### **4.4.3 Execution-Based**

#### **4.4.3.1 Compilation, Execution and Editor**

Analysing a program by analysing its source code text is difficult. Sometimes the effect of modifications is not apparent. Sometimes the researcher may not have confidence in her understanding of the effects of a particular modification. To resolve such issues it would be beneficial if the researcher could execute project versions. The researcher could then observe the effect of modifications. In cases where the nature of a modification still remained unclear the researcher could then manually debug the source code by making modifications.

The SCORE Analyser can automatically compile each version of the project (producing one executable per project version), allowing the researcher to execute the program at any Change through the Analyser's main interface. When a program version's execution is requested, SCORE will launch the executable if compilation was successful. If compilation was unsuccessful, compile-time error messages are displayed. Program output to standard out is also displayed.

When execution of the program is insufficient to debug and understand a Change, the SCORE Analyser offers a simple code editor (see Figure 13) that allows the researcher to modify and execute a local copy of the project version's code.

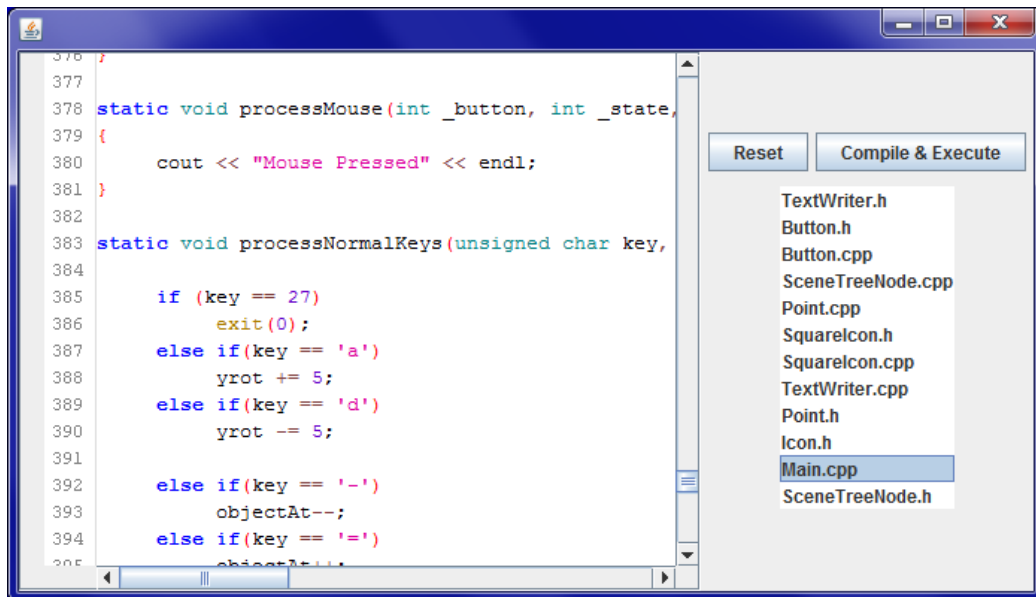


Figure 13: The SCORE Editor

Compilation is currently only enabled for C++ projects using the GNU g++ compiler. Compilation settings and dependencies such as include directories or library paths are configured via xml files. Regular expressions can also be specified in xml files. These regular expressions can be used for the modification of any location-dependent (e.g. absolute path names) source code.

The Execution-based functionality is described in more detail in the SCORE Manual, Section 9.5.3.6. A video demonstration of this Execution-based functionality is available at <http://www.youtube.com/watch?v=xGok5h6xB8s&feature=plcp> and in the electronic appendix.

#### 4.4.3.2 Screen-capture and video-generation

For Computer Graphics programs, the main program output is visual. The SCORE Analyser can use compiled project version executables to automatically generate and store screen captures for every project version. The screen capture for the current Change is then displayed in a separate window (shown as part of the SCORE Analyser in Figure 14; several example screen captures are shown in Figure 15) and updated automatically when the currently selected project version is changed. This view is useful in giving the researcher a quick way to examine the program's output at that stage of the Project History; if this static view does not suffice the researcher can utilise the program execution functionality described earlier.

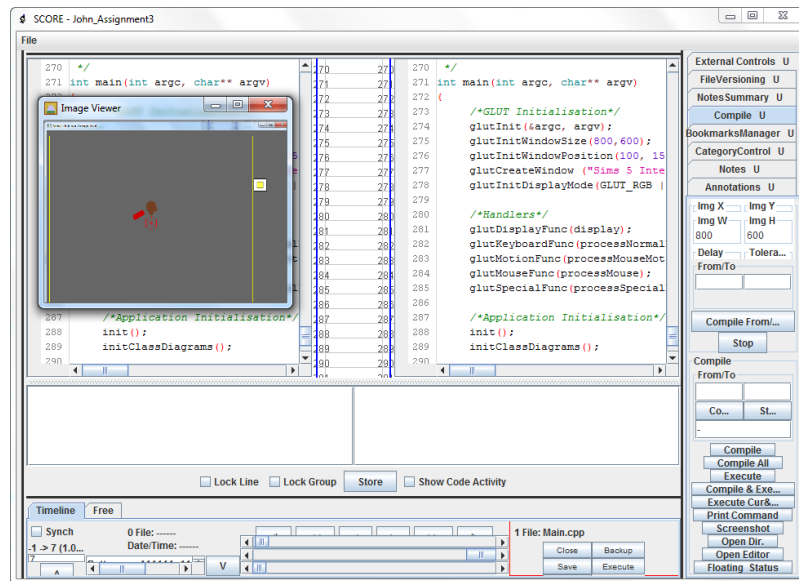


Figure 14: The screen capture window is shown in the top-right corner; the window is movable

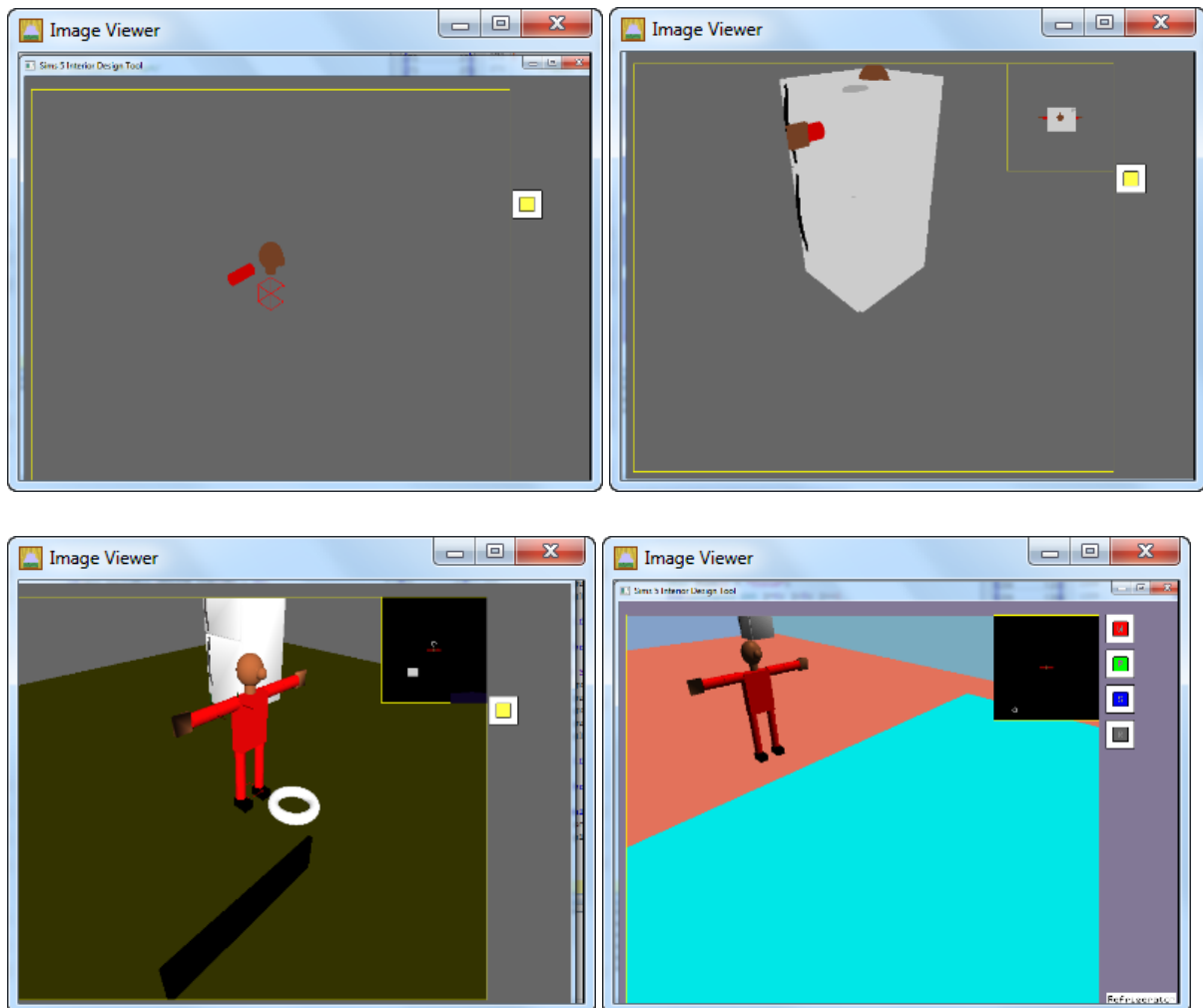


Figure 15: The Screen-Capture View at four different Changes

#### **4.4.4 Core Functions Walkthrough**

When using the SCORE Analyser's core functionality to analyse an assignment, the researcher will navigate between Changes using the main Diff View, examining modifications made from one version to the next.

When the researcher is unsure as to the effect of modifications made to a version, she executes the project at that and the previous Change to observe any differences in output. If this still does not clarify the function of a modification, the researcher uses the built-in editor to debug the modification until she understands the modification's purpose. She thereby develops an understanding of each source-code modification. This allows her to identify student programming errors. As the researcher examines Changes, she takes notes regarding the student's programming actions using the SCORE Analyser's note-taking facilities.

Using this approach, the researcher can build up an understanding of student actions. However, this understanding is only informally structured through notes, and there is no straightforward way of processing data produced in this way for evaluation. Later chapters will introduce methods of structuring and analysing data relating to the student programming process developed during analysis of the Project History.

### **4.5 SCORE Toolkit Future Work**

#### **4.5.1 Supplementing with Subjective Data – Student Comment Extension**

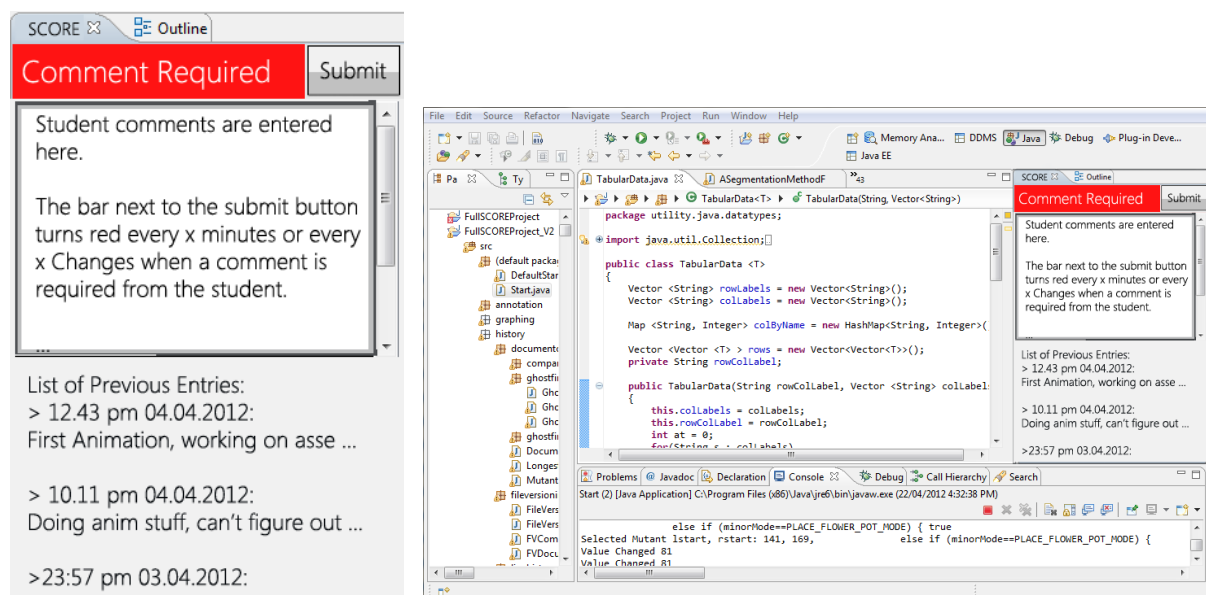
SCORE is intended to provide objective data which can be machine-processed for easier analysis. This avoids some of the issues relating to external validity of subjective methods such as questionnaires or interviews caused by subjective distortions and recall.

SCORE is not a replacement for such approaches. The researcher's interpretation of student actions is required to gain insight into student thinking. Thus, other data-gathering methods which can probe student attitudes and thoughts may prove useful in complementing the methods outlined in this thesis.

An aim for future use of SCORE is to augment gathered data with subjective student commentary. This will allow both a richer understanding of analysed student problems, as well as validation of the researcher's interpretation of student programming efforts. This method will have access to data of a quality and richness unavailable to any approaches focusing only on a single method of data gathering.

Two methods (which could be used in conjunction) could be used to integrate SCORE with subjective student input. The first involves the addition of textual input widgets to the SCORE Eclipse plug-in

described in Section 4.3. These would prompt students to provide feedback on their current actions periodically. They would also encourage students to provide input at any point as they are struggling with a problem, acting much like a journal which could also be used in marking, and could be used to prevent plagiarism as well. A mock-up of this functionality is shown in Figure 16. In addition, journal-keeping can also support student reflection and aid learning (George, 2002). However, in contrast to a stand-alone journal these entries can then automatically be attached to the Change on which the student is currently working, supplementing SCORE data with ‘live’ student commentary and insight into their cognitive processes.



**Figure 16: Mock-up of the SCORE Plug-in Commenting Extension; A close-up on the left, as part of an Eclipse distribution on the right**

The second method would be to perform a quick analysis of ‘interesting’ Segments detected by SCORE. Interviews could then be conducted with volunteer students. Students could walk the researcher through their development of the assignment version by version. The researcher could also ask specific questions to clear up or provide more detail on any poorly understood programming actions.

When used in conjunction with the first method, the researcher’s questions could then act as a second line of data gathering to clear up any questions regarding the student’s comments captured during their programming of their assignment.

This confluence of research approaches may provide researchers with the advantages of both objective and subjective approaches to analysis of student programming, whilst minimising the need to rely only on interpretation of external actions or subjective student data in reaching conclusions.

### 4.5.2 Modification-level Note-Taking

Currently two note-taking mechanisms exist. The first, described earlier, allows taking of notes for individual Changes, files and the Project History as a whole. Another mechanism, described in Section 6.4.1.1, allows the taking of notes for sets of Changes called Segments.

However, it may often be useful to take notes at a more fine-grained level than for whole Changes. A new note-taking mechanism could allow notes for individual modifications, allowing the researcher to store the meaning of these individual lines of source code along with the data, rather than having to store the note in the Change's note and then looking up individual modifications each time.

This would also allow for views which show the most recent note for each individual line in a Change, thereby pulling together notes taken during many different Changes. These notes could then be displayed alongside lines in the main view.

### 4.5.3 Dual offline/online plug-in Project History storage

At the moment the SCORE plug-in stores the Project History locally, capturing it in a folder in the project's root directory. This approach was utilised as it is transparent, requiring no further action on behalf of the student; the student does not have to log in to any server, or even be connected to the internet. The Project History is automatically submitted along with the project when the student compresses the directory and submits the compressed file.

This approach usually works well, but some students (despite instructions to the contrary) take actions which cause the Project History to be incomplete, such as at some point of their development creating a new project and moving their source files but not the directory containing the Project History to this new project. Students also sometimes complete part of the work on their assignment using an Eclipse installation which does not have the SCORE plug-in installed. This results in an incomplete and unanalysable Project History.

The SCORE plug-in could utilise a hybrid approach, storing the Project History locally but also attempting to send the Project History to a remote server whenever an internet connection is detected. This would require the development of a server for the SCORE plug-in to connect to, and would also require students to log in to the server in some fashion.

## 4.6 SCORE Toolkit Conclusion

This chapter introduced the SCORE plug-in which allows the researcher to record student programming in Project Histories while students work on their assignments (Section 4.3), as well as the SCORE Analyser tool which facilitates examination and analysis of Project Histories.



The SCORE Analyser's core functionality was introduced in Section 4.4. The 'Diff View' (Section 4.4.1) allows the user to browse Project Histories with a diff style view, comparing pairs of versions to build an understanding of student programming actions. Note-Taking facilities including both plain-text note-taking and categorising according to user-defined categories are described in Section 4.4.2. These facilities allow the user to store insight gained during analysis and to hence build up this insight over multiple analysis sections. In the context of this research project is based on this enables so-called memoing, essential for the application of the Grounded Theory methodology.

The Execution-based functionality described in Section 4.4.3 enables the user to edit and execute the project at any given version, thereby allowing the user to examine any version's output or to debug the version via the editor. Execution-based functionality also includes the ability to automatically take screen captures of every version of the project which are then displayed in a window when the associated Change is navigated to in the Diff Viewer, allowing the user a quick overview of the progress of the student's work at that point.

Section 4.4.4 presents a brief walkthrough of how this functionality would be used to analyse a student assignment. This involves several steps. One is navigating from Change to Change, analysing modifications from one version to the next. Another is executing and debugging source code versions when the effect or function of a modification is unclear. The last step is the noting down of gained insight via the Note-taking facilities.

Several future extensions to the SCORE toolset are described in Section 4.5. One is supplementing the SCORE plug-in with facilities allowing the student to comment on their programming as they work on their assignments (Section 4.5.1). This would supplement source code data with valuable student perceptions. Another is adding note-taking facilities which allow note-taking for individual modifications (Section 4.5.2) to enable more fine-grained note-taking. A third extension would involve allowing the SCORE plug-in to store data both offline (as is currently the case) and online via communication via a server. This would provide the researcher with more flexibility in data gathering (Section 4.5.3).

The description of the SCORE Analyser in this chapter is brief. A more detailed description of the SCORE Analyser's functionality is given in the SCORE Analyser manual (Wittmann, 2012a) located in the appendix, Section 9.5.3; a short video presentation demonstrating the functionality of the SCORE Analyser is also available (Wittmann, 2012b) online at <http://www.youtube.com/playlist?list=PL5A1E9556055F67E2&feature=plcp>. Also, this chapter only

describes the basic functionality of the SCORE Analyser. Other chapters describe additional functionality implemented as part of the Change-Coding and Segment-Coding analysis methods.

# 5 Change-Coding

---

## 5.1 Change-Coding Introduction

The last chapter presented a plug-in developed to allow Project Histories to be captured during student work on programming tasks. It also presented a software application implemented to allow for the analysis of these Project Histories. This chapter will detail the development of a method of structuring, organising and quantifying Project History data for analysis. It also presents the results of this analysis based on the student assignments from the 2010 data corpus (see Section 3.6.2.2).

The method presented in this chapter is similar to an approach developed for the analysis of Project Histories of students in introductory (first-year) Computer Science courses (Jadud, 2006a) except it uses manual human coding and categorisation instead of machine-evaluation of compilation logs. In Jadud's method (2006a) Project Histories are captured as students work on laboratory exercises. These Project Histories include logs containing all compiler messages, including error messages and warnings. These logs are machine-evaluated to identify errors occurring in different versions of the project. This approach allows the researcher quick and accurate access to the types of errors occurring for students during their work on the laboratory exercise. The analysis of errors under the Jadud method is context-free. Each error is analysed in isolation, not as part of a larger programming plan intended to solve a specific task or problem. It is also output-driven since it analyses program output in the form of error messages, rather than the source code itself.

In the context in which the research presented in this thesis takes place the machine-processing of assignments to detect errors as demonstrated by Jadud (2006a) is not viable for analysing students' Computer Graphics Project Histories. The types of errors and problems arising in Computer Graphics programming and other advanced Computer Science topics are fundamentally different to the syntax and semantic problems encountered in first-year Computer Science courses. Domain-relevant errors and problems rarely cause compilation errors and thus the mining of error messages cannot identify such errors. For example, an incorrectly implemented transformation call will lead to an object being placed incorrectly on screen, but will not produce a compilation error. Furthermore, whether something is 'placed incorrectly' is dependent on the context of the student's work. Thus, there is no straightforward way of machine-detecting the correctness of such a statement.

It may be possible to produce a framework for Computer Graphics assessment tasks that can be partially machine unit-tested (by having very precise assignment tasks and then analysing the state

of the ModelView or Projection matrix state for example). However, this approach would require the creation of very carefully crafted exercises which would also have no room for student creativity. This approach was incompatible with the assignments that were being used in the Comp330 course under investigation.

To enable the analysis of open-ended assignments neither amenable to unit-test or compile error data-mining, an approach to emulate the machine-detection of errors was developed. In this approach, the human researcher manually classifies individual Changes according to a set of classification categories across a set of classification dimensions. This then allows for a comparison of error rates for different types of errors much like was carried out by Jadud (2006a). It will be referred to the *Change-Coding Method* of project-history analysis since it involves the coding of individual Changes according to a classification scheme to produce evaluation data.

This chapter will first describe the development of the coding scheme as well as the scheme's categories utilised in this research project (Section 5.2). This is followed by a description of the SCORE Analyser functionality which enables the coding of Changes (Section 5.3). Results of the application of the Change-Coding method are presented and discussed in Section 5.4. Limitations of the Change-Coding method which led to the development of a second analysis method (Segment-Coding) presented in the following chapter are discussed in Section 5.5.

## 5.2 Development of Categorisation Scheme

The work on introductory student syntax errors on which influenced the development of the Change-Coding method (Jadud, 2006a) could base its classification scheme on the error messages captured in compilation logs because the study was only focusing on students' compilation errors. Each of these error messages could be assigned its own classification category using a one-to-one mapping, and then rates of different errors could be compared.

In contrast, the classification scheme developed for coding Changes in this research project could not be based on a one-to-one mapping of compiler-generated errors to categories because of the nature of Computer Graphics program errors, many of which do not produce associated compile errors. Instead, the classification scheme had to be developed manually through the definition of coding categories that would classify the types of errors occurring during Computer Graphics programming.

This section will discuss the development of the scheme used to classify problems and errors as part of the analysis of student assignments. Section 5.2.1 focuses on the rationale behind the classification scheme, and on the limitations of the initial scheme which led to the development of a

modified version of the scheme. The final version of the classification scheme and its categories is presented in detail in Section 5.2.2. Section 5.2.3 describes the rules for classifying individual Changes.

### 5.2.1 Development of Dimensions and Categories

Development of the classification scheme was iterative, with the limitations of initial approaches being addressed in subsequent iterations. The aim of this process was to develop a classification method which would allow meaningful and useful analysis of student assignments in a general Computer Graphics setting. This approach should not rely on compilation messages or other domain or tool-specific output. For this purpose, the classification scheme needed to be able to describe the data accurately and precisely. An initial and a final version representing the outcome of an iterative development process are presented; each of these versions is itself the outcome of a process of analysis and application of the classification scheme during which additional categories were developed.

The initial attempt at designing a classification scheme involved classifying each Change in three dimensions. The first identified what the student was working on during that Change (Action dimension). The second identified what if any error was contained in the Change's modifications (Error dimension). The third identified and what if anything was wrong with the program's output (Output dimension). The Output and Error dimensions together describe the error in a way similar to that of the compilation error classification scheme described by Jadud (Jadud, 2006b), while the Action dimension went further in describing not only what the student is doing or seeing but also describes the intent behind the modifications. The Action dimension included the following dimensions: Assembly, Placement, Drawing, Animation, Projection, Viewing, Data Structure, Lighting, GUI, Debug, Cleanup, Other. The Error dimension was comprised of the following categories: General Syntax, General Semantics, OpenGL Syntax, OpenGL Semantics, 2D Coordinate, 3D Coordinate, 2D Transform, 3D Transform, Projection, Viewing, Lighting, State Machine or Pipeline, Timing or Iteration, Cleanup, Other, None. The Output dimension was comprised of these dimensions: No Compile, Crash, No Display, Skewed, Missing Items, Misplaced Items, Missized Items, Wrong Animation, Wrong Lighting, Wrong View, Wrong Visual, Other and Correct. The categories are defined in detail in Appendix Section 9.6.1.

This preliminary classification scheme was intended to answer questions such as "What actions are most often associated with errors?". "What errors are most common and what errors are associated with what type of erroneous output?". This classification scheme was used to code all Changes for the ten Project Histories analysed as part of this research project. Because of issues regarding the

analysis of Change-Coding results utilising this classification scheme the scheme was modified. These changes are described in the next section. Due to the change in the classification scheme, all Changes were re-coded. Results of Change-Coding using the preliminary version of the scheme are not presented as they are superseded by results from the application of the final version of the scheme.

The preliminary classification scheme's emphasis on outputs was more in line with Jadud's output-focused approach to coding. This is counter to this study's aims of investigating the student programming process. As such the coding scheme was redesigned to incorporate Action, Error and Problem dimensions. The Action dimension captures what type of task the student is working on in a given Change. The Error dimension captures the type of error (if any) underlies a given Change. The Problem dimension captures the nature of the underlying problem that students were working on in a given Change.

The Action, Error and Problem dimensions are related due to the way in which they describe the student programming process. As the student works on his program, he may work on a task that is spatial in nature. These Changes will be coded as 'Spatial' in the Action dimension. As long as the student does not introduce any errors, the Changes will be coded as 'None' in both the Error and Problem dimensions. As he works on the problem he may make an error with a transformation. The Change in which the error occurs is coded as 'Spatial' in the Action, the Error and the Problem dimensions. This is because the student is working on a spatial task and made a spatial error. This leads to a spatial problem which prevents his program from functioning correctly. In response, the student debugs the program. While the problem is not fixed, Changes will continue to be coded as 'Spatial' in the Action and Problem dimensions. Only those Changes in which the student introduces additional spatial errors will be coded as 'Spatial' in the Error dimension as well as the Action and Problem dimensions, whereas those Changes which partially address the underlying problem but do not completely resolve it will be coded as 'Spatial' in the Problem and Action dimensions only. Thus, mistakes are only coded as Errors for the Change in which they are made, and if no further mistakes are made then the Error dimension will be coded as 'None' for these subsequent Changes. Once the student fixes the problem, if he continues working on 'Spatial' tasks then following Changes will be coded as 'Spatial' in the Action dimension only until a new error is introduced. This means a Problem is almost always rooted in a specific Error. However, in this classification scheme this relationship is not explicitly captured.

Since all dimensions are now related in describing different aspects of the same set of programming tasks types, the items for each Dimension were recast to be the same as one another (and

essentially based upon the Action dimension of the preliminary coding scheme). This allowed the analysis to determine consistencies and disconnects between students' programming processes, their misconceptions, and the problems they are experiencing.

The Problem dimension was a departure from the context-free approach that the preliminary coding scheme was based upon since the identification of a Problem required an understanding of the surrounding context. Contextual features of the Problem, such as where it originated or what other Changes address the same problem, are not formally captured by the Change-Coding method.

The new classification scheme allows for the frequency of Actions to be correlated to the frequency of Errors and Problems. For example, examining the ratio between Actions and Errors for a certain category shows how frequently students make an error when working on a certain type of problem. This allows for comparison between different task types. For example, take a student assignment in which of 100 '*Spatial*' Action Changes 50 are classified as '*Spatial*' Errors. In the same assignment of 100 '*General Programming*' Action Changes only 10 are classified as '*General Programming*' Error Changes. This suggests that the student is five times more likely to make an error when engaging in spatial problem-solving than when tackling '*General Programming*' problems. This would provide data on which kinds of problems students found easy or difficult to solve.

In summary, during the first iteration, dimensions described Changes as follows:

- Action: What task is the student working on during Change?
- Error: What type of instruction causes the error?
- Output: What is visually / perceptually wrong with the program's output?

Each dimension had its own set of categories, leading to a mapping of  $x : y : z$ , where  $x \in$  (Set of Action Categories),  $y \in$  (Set of Error Categories) and  $z \in$  (Set of Action categories).

The second iteration defines categories describing Changes differently:

- Action: What task is the student working on during Change?
- Error: What type of task would be required to fix the error produced in the Change?
- Problem: What type of task would be required to solve the problem in the Change? (Usually based on the type of the error giving rise to the problem)

In the second iteration, all dimensions share a set of categories, leading to a mapping of  $x : y : z$  where  $x \in (\text{Set of Type Category})$  and  $y, z \in (\text{Set of Type Category}) \cup (\text{None Category})$ . The categories are described in more detail in the next section.

This classification scheme was used to re-classify all Changes initially classified according to the first iteration classification scheme. Change-Coding data evaluated in Section 5.3 was classified based on this classification scheme.

### 5.2.2 Detailed Description of Classification Categories

This section contains a detailed description of the categories making up the second version of the classification scheme. These are illustrated with examples of the type of item that would be associated with the Category.

In the descriptions that follow the *Description* item describes the category and what types of items would be coded as belonging to that category. The *Examples* item gives some examples of modifications that would cause a Change to be coded as belonging to that category. The text following the name of the category in brackets is that category's abbreviation.

#### **Math Spatial (SpMa)**

The '*Math Spatial*' category is used to code items which utilise a mathematical function (such as the trigonometric functions cos/sin/tan) or formula or modify such a formula.

Examples:

```
int x = sin(val);          ->   int x = cos(val);
glRotatef(x, 0, 1, 0);    ->   glRotatef(x, 1, 0, 0);
glTranslatef(0, y, 0)     ->   glTranslatef(0, abs(y), 0);
```

#### **Major Spatial (Sp+)**

The '*Major Spatial*' category is used to code items which change a transformation's or primitive draw action's values by at least one order of magnitude, or change the axis of a transformation or primitive draw action either by changing the direction of one of the axis dimensions or by introducing or removing a dimension from the axis (e.g. `glTranslatef(1,1,0)` to `glTranslatef(1,0,0)`).

Examples:

```
glTranslatef(0, 0, 10)  ->   glTranslatef(0, 10, 0)
```



```

glVertex3f(10, 0, 0)    ->    glVertex3f(-10, 0, 0)
glTranslatef(0, 0.1, 0) ->    glTranslatef(0, 10, 0)
glTranslatef(5, 1, 1)   ->    glTranslatef(1, 5, 1)
glTranslatef(5, 0, 0)   ->    glTranslatef(5, 2, 0)
int xpos = y + 5;       ->    int xpos = y * 5;

```

### **Minor Spatial (Sp-)**

The '*Minor Spatial*' category is used for coding items which involve an incremental modification of the coordinates of a primitive draw action or the axis of a transformation. An incremental modification is one that does not change the direction of the spatial action and does not modify the spatial action by an order of magnitude or more. Such actions are usually 'tweaks', utilised by students to achieve pixel-perfect visual results.

Examples:

```

glScalef(0.5, 0.5, 0.5)    ->    glScalef(0.6, 0.6, 0.65)
glVertex3f(1, 0, 0)        ->    glVertex3f(5, 0, 0)

```

### **Combined Spatial (SpComb)**

The '*Combined Spatial*' category is a derived category, including all items that are part of the *Major*, '*Minor Spatial*' and '*Math Spatial*' categories. It is the only derived category. The category involves modifications relating to coordinate changes for creating graphical primitives, axis or angle changes for two or three-dimensional transformations, as well as any other modifications related to spatial programming.

Categorising: See Major/Minor/Math Spatial

Examples: See Major/Minor/Math Spatial

### **General Programming (GP)**

In the context of Change-Coding, the '*General Programming*' category is used to categorise/code items that do not fall into a more precise category. This includes programming relating to syntax and semantics issues as well as perfecting and clean-up actions such as renaming variables or simplifying code. Formatting and commenting actions are not part of the '*General Programming*' category, but instead are assigned to the '*Other*' category.

The category has the same meaning in the context of the Segment-Coding method presented in Chapter 6, except that clean-up actions do not fall into the '*General Programming*' category. Since clean-up actions are not a 'problem', they are not captured via Segments.

Examples:

```
for(int i = 0; i < 10; i++)    ->    for(int i = 0; i < 15; i++)
cout >> "hello" >> endl      ->    cout << "hello" << endl
```

### **Event-Driven (ED)**

The '*Event-Driven*' category codes items that implement or modify code involved in the production of an event-driven programming model. In practice, such coding actions usually involve work on the user interface such as code using an event handler or code that implements UI functionality such as button-hit detection functionality.

Examples:

- A change to the mouse handler or keyboard handler

-Implementation of an ED model involving a mode variable keeping track of the current state of the application; the mode variable is modified by user interface actions such as the pressing of buttons associated with different modes.

### **Viewing or Projection (also referred to simply as 'View') (Vi)**

The '*Viewing or Projection*' category captures items which implement or modify an OpenGL projection or camera view. This includes transformations intended to change the view of the scene rather than positions of objects (changes to the model). Changes to the projection involve calls to `glOrtho`, `glOrtho2D`, `gluPerspective` or `glFrustum`.

Examples:

```
glOrtho2D(-100,100,-100,100)  ->    glOrtho2D(-50,50,-50,50)
glOrtho(-1,1,-1,1,-1,1)      ->    gluPerspective(60, 0.1, 100)
```

The following would be classified as Viewing or Projection only if the `translate` command precedes a viewing command. In other contexts it would count as a Major Spatial or Minor Spatial Change.

```
glTranslatef(1,0,0) ;          ->    glTranslatef(0,1,0) ;
gluLookAt(0,0,0,5,5,5,1,1,1)   gluLookAt(0,0,0,5,5,5,1,1,1)
```

### **Animation (Anim)**

The '*Animation*' category codes items that implement a time-driven mechanism for producing animations (using glut timer or idle functions or busy loops). The use of `glutIdleFunc()`, `glutTimer()` or busy loops to create delay between animation steps is coded as *Animation*.

It is also used to code any other non-spatial components of animations (for example, the reversing of an animation after it runs through using a boolean value).

Examples:

```
glutTimerFunc(0, Timer, 0);      ->    glutTimerFunc(0, Timer, 5);
*none*                          ->    glutIdleFunc(anim);
```

### **General OpenGL (GL)**

The '*General OpenGL*' (or '*OpenGL/GL*') category codes items which involve actions related to OpenGL syntax or semantics, such as a student utilising the incorrect enumerated type with the `glBegin` function for drawing graphical primitives, leading to incorrect output. Another example would be a student utilising a `glVertex3i` call which takes integer input parameters with floating-point values, thereby losing the floating point.

Examples:

```
*none*                          ->    glVertex3i(0.5, 0, 0.5) ;
glBegin(GL_POINTS);             ->    glBegin(GL_LINES);
*none*                          ->    glutIdleFunc(myIdleFunc);
```

### **Pipeline (Pi)**

Changes in the order of OpenGL commands (excepting spatial commands) are classified as '*Pipeline*'. Addition of OpenGL commands to create proper display also fall into this category. This includes adding `glutPostRedisplay()`, `glFlush()` or `glutSwapBuffers()` calls or direct calls to the display function, as well as adding calls to undo the effect of earlier calls (like calling `glLineWidth(1)` in the code after `glLineWidth(3)` was called earlier)

Examples:

```
glutPostRedisplay()             ->    display() (or adding of either call)

glBegin(GL_LINES);              ->    glLineWidth(1);
glVertex2i(0,0);                 ->    glBegin(GL_LINES);
```

<code>glVertex2i(10,0);</code>	<code>glVertex2i(0,0);</code>
<code>glEnd();</code>	<code>glVertex2i(10,0);</code>
<code>glLineWidth(1);</code>	<code>glEnd();</code>

### **Lighting (Li)**

The '*Lighting*' category codes items that involve setting up OpenGL lighting. It does not include keyboard commands or buttons to trigger lighting (these should be categorised as Event-Driven).

Addition or modification of function calls to `glLight`, `glMaterial`, `glColorMaterial`, `glShadeModel`, as well as enabling/disabling lighting-related state-machine states using `glEnable/glDisable (GL_LIGHTING, GL_COLOR_MATERIAL, ...)` and fog-related calls (`glFog`, `glEnable(GL_FOG)`, `glHint(GL_FOG_HINT, ...)`) indicates a modification that should be classed as *Lighting*. Modification of arrays containing the position or color of lighting should also be classified as lighting changes.

Examples:

<code>*none*</code>	<code>-&gt; glEnable(GL_LIGHTING);</code>
<code>lightPos[0] = 5;</code>	<code>-&gt; lightPos[0] = 10;</code>
<code>glLightfv(GL_POSITION, lightPos, 0)</code>	<code>-&gt; glLightfv(GL_POSITION, lightPos, 0)</code>

### **Other (Oth)**

The '*Other*' category codes items that involve commenting modifications and hence do not alter the functionality of the program.

Examples:

<code>//This does this</code>	<code>-&gt;</code>	<code>//Aha! It does that.</code>
-------------------------------	--------------------	-----------------------------------

### **Bad (Bad)**

The '*Bad*' category codes items that involve only formatting modifications, as well as the first Changes in a Project History which are associated with the program skeleton provided to students and do not contain student work. Such modifications do not have any effect on the functioning of the program and hence are excluded from the analysis. The label *Bad* does not indicate that such Changes are bad practice, but rather that they are 'Bad' in the context of analysis and are to be excluded.

### **Unknown (?)**

Items marked for future investigation because the content is currently not well-understood. The researcher should return to such items and attempt to classify them. As few items as possible should remain in this category after analysis is complete. Overall, 11 of 10572 Changes remained in the 'Unknown' category after analysis was complete.

## **5.2.3 Change-Coding Rules / Application of Classification Scheme**

### **5.2.3.1 Rule 1) Cosmetic or rote changes should be classed as General Programming**

Changes which do not modify the logic of the underlying code should be classified as 'General Programming'.

Examples:

- Declaring of array to hold values (changing of values may be classed in a more specific category)
- Changing the name of a variable.
- Correcting simple spelling errors (incorrect calls to GL functions exempted; these should be classed 'General OpenGL')

### **5.2.3.2 Rule 2) Code most significant**

When programming actions that could be assigned to different categories co-occur in the same Change, the Change should be classified according to which modifications produce the most significant change to the functionality of the program.

Example:

- A Change that involves three changes to coordinates and one change to event-driven code should be coded as a '*Spatial*' Change

### **5.2.3.3 Rule 3) Code the most specific**

If a Change involves multiple modifications, then it should be coded according to the most specific modification type, according to this hierarchy:

- Specific: ('Spatial', 'Event-Driven', 'Viewing/Projection', 'Lighting', 'Animation')
- Less Specific: ('General OpenGL', 'Pipeline')
- General: ('General Programming')

- Unspecific: ('Other')

Example:

- A Change involving modifications falling into the *Spatial*, *General OpenGL* and *General Programming* categories should be coded as belonging to the *Spatial* category.

### 5.2.3.4 Rule 4) Errors: Better, worse or the same

If a part of code that is wrong is changed, the change should be classed as an error for coding with the error categorisation dimension if it makes the problem worse or leaves it the unchanged; it should not be classed as an error if it fixes part of the problem even if it doesn't entirely fix the problem.

Example:

If an array is size 10, then:

```
// ERROR (Makes worse)
for(int i = 0; i < 11; i++)
{
    cout << a[i+1] << endl;
}
->
for(int i = 0; i < 15; i++)
{
    cout << a[i+1] << endl;
}
```

```
// ERROR (Unchanged)
for(int i = 0; i < 11; i++)
{
    cout << a[i+1] << endl;
}
->
for(float i = 0; i < 11; i++)
{
    cout << a[i+1] << endl;
}
```

```
// NO ERROR (Makes better, but still wrong [because of the +1 in a[i+1]])
for(int i = 0; i < 11; i++)
{
    cout << a[i+1] << endl;
}
->
for(int i = 0; i < 10; i++)
{
    cout << a[i+1] << endl;
}
```

### 5.2.3.5 Rule 5) Commenting/Uncommenting code

When code is commented as part of a clean-up action (removing dead code), the commenting should be coded as *'Other'*, like normal commenting. If code is commented as part of problem-solving, it should be coded in as the same category as the problem that is being solved. When code is uncommented, that change should be coded in the same category as the problem that is being solved.

Examples:

```
//In this case, should be coded Other since the first glColor line was  
//superfluous (overridden by the second glColor call) and is being removed  
//as part of cleanup
```

```
glColor3f(0.1, 0.1, 0.1);          ->    //glColor3f(0.1, 0.1, 0.1);  
glColor3f(0.5, 0.1, 0.1);          glColor3f(0.5, 0.1, 0.1);
```

```
//This case should be coded as MajorSpatial, since the work being done with  
//vertices is spatial. The uncommenting of the same line should again be  
//coded MajorSpatial.
```

```
glBegin(GL_LINES);                ->    glBegin(GL_LINES);  
glVertex3f(1.0, 1.0, 0.0);         //glVertex3f(1.0, 1.0, 0.0);  
... ETC ...                        ... ETC ...  
glEnd();                           glEnd();
```

## 5.3 Change-Coding with SCORE

The idea of coding assignment Changes is similar to the approach to analysis student Project Histories based on compilation errors proposed by Jadud (2006a). Though in that work, errors were identified by machine analysis of compilation error messages, and the number of different error types was collected.

In fields like Computer Graphics, most errors are logic errors and hence do not generate compilation errors that can be machine-processed. To apply the approach to Computer Graphics programming, a mechanism for manual coding of Changes was implemented. This coding was intended to replace compilation-error analysis by associating Changes with error types manually.

This section will discuss the SCORE Analyser functionality implemented for the application of the Change-Coding method of analysis. After coding of an assignment's Changes the researcher could then look for the occurrence of runs of a particular error or problem type. These runs could then be used to identify parts of the student programming process that involved the addressing of a specific

error. In addition, statistics could be generated on how often each error type occurred, how many times a programming action was associated with an error of a certain type and so on.

### **5.3.1 Setting up Change-Coding using the SCORE Analyser**

The SCORE Analyser allows the researcher to code each Change along an arbitrary number of user-specified dimensions. Such coding dimensions can be binary (true/false) in which case they are represented via checkboxes, string-based and represented via text fields or they can include a set of categories, one of which is applied to any given Change, in which case they are represented via combo boxes. The latter set-of-categories approach is the one utilised for coding in this research project, with each Change being coded in error, problem and action dimensions. All dimensions share the same set of categories which include items such as ‘General Programming’, ‘Spatial-Maths’ and ‘Event-Driven’. This chapter discusses how coding is applied in practice; the development of the classification codes is discussed in Section 5.2.

The user sets up dimensions and their categories via xml files, with the Analyser’s Change-Coding component automatically displaying subcomponents allowing for the coding of Changes in those dimensions, and saving and loading Change-Coding data according to the user-specified format.

### **5.3.2 Change-Coding Panel**

The user codes Changes via the Change-Coding View (shown in Figure 17, left), using categories and labels defined via configuration files (consistent between all SCORE assignments that belong to the same project) which are represented using drop-down combo boxes, text fields and check boxes. Coding a Change involves modifying the components shown in the Change-Coding View. For quick coding for dimensions involving categories, the user can utilise a ‘Quick Coding panel’ (Figure 17, right).



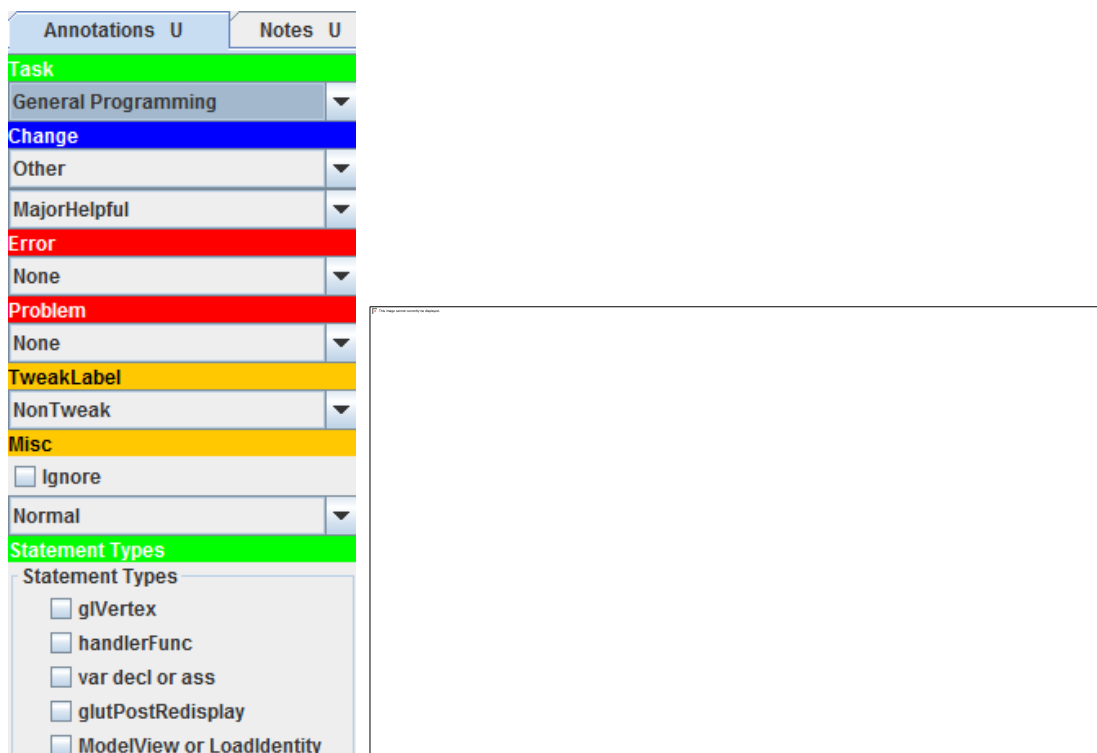


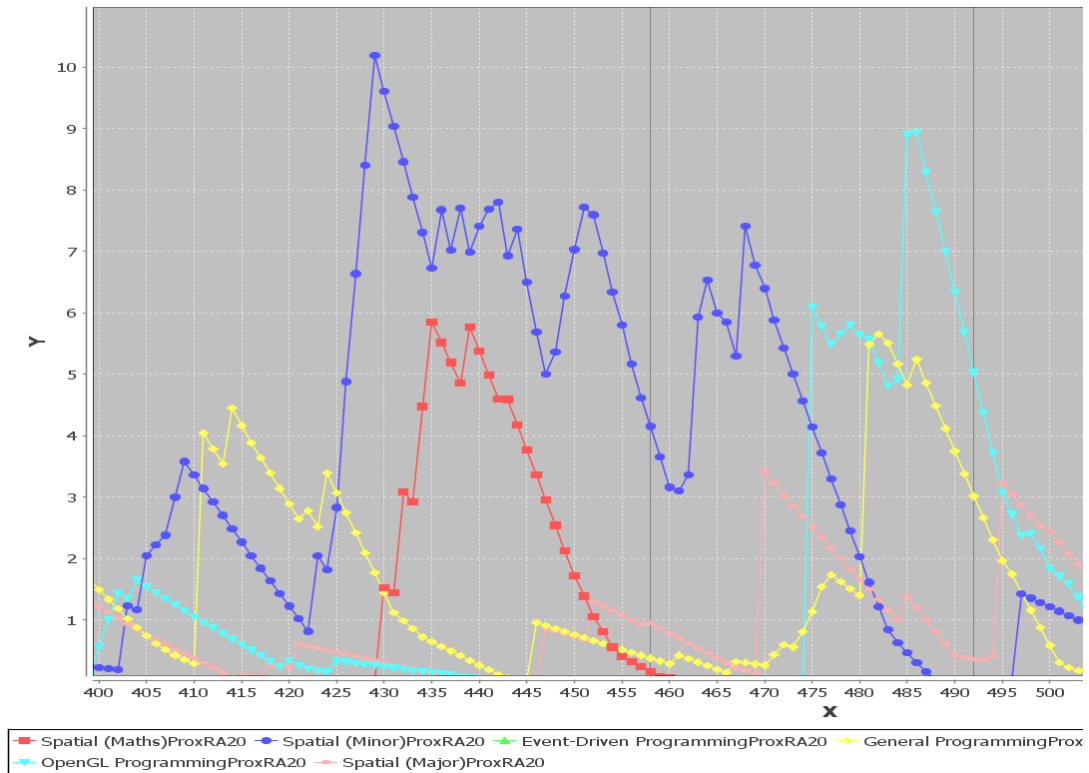
Figure 17: Change-Coding View (left) and Quick-Coding View (right)

The Change-Coding Panel contains one component for every Coding dimension specified in the Change-Coding setup xml file. Change-Coding data can be accessed by other SCORE components for further processing. For example, the user can utilise the SCORE Analyser's Graphing component to produce graphs showing the rolling average of changes in a certain category as described in the next section.

### 5.3.3 Graphing Change-Coding Results

The user can utilise the SCORE Analyser to produce graphs based on Change-Coding data. These graphs show the prevalence of different coding categories at different points of the Project History. At the moment, only dimensions coded using categories can be graphed. The most common graphing option is to graph rolling averages for individual categories. This produces peaks for periods where many Changes in the recent past fall into the same category. The researcher can then examine these periods in more detail to see whether they make up a larger problem. Graphs are generated using the open-source JFreeChart library<sup>34</sup>.

<sup>34</sup> JFreeChart: An open-source java library used for creating charts (<http://www.jfree.org/jfreechart/>)  
<http://db.apache.org/derby/>)



**Figure 18: A graph mapping rolling averages of the number of Changes belonging to different Coding classification categories**

### 5.3.4 Change-Coding Metrics

Graphing provides the user with a simple visualization of Change-Coding data and allows the user to identify areas associated with high levels of activity for a certain type of Change.

The user can also generate summaries of Change-Coding data as shown in Figure 19 using the SCORE Analyser. The view displays per category the total number of Changes that were coded with that category, as well as the total amount of time spent on Changes of that category. This data is also written to an .xml file.

<b>ErrorDropbox</b>	
None:	884 (73.73%) 05h:59m:48s 00.00.0000 (71.98%) Tpc: 24.42
General Programming:	121 (10.09%) 00h:53m:36s 00.00.0000 (10.72%) Tpc: 26.58
Bad:	69 (5.75%) 00h:26m:48s 00.00.0000 (5.36%) Tpc: 23.30
Major Spatial:	49 (4.09%) 00h:26m:26s 00.00.0000 (5.29%) Tpc: 32.37
Viewing or Projection:	25 (2.09%) 00h:13m:25s 00.00.0000 (2.68%) Tpc: 32.20
Unknown:	12 (1.00%) 00h:03m:29s 00.00.0000 (0.70%) Tpc: 17.42
Pipeline:	8 (0.67%) 00h:04m:16s 00.00.0000 (0.85%) Tpc: 32.00
Minor Spatial:	8 (0.67%) 00h:03m:42s 00.00.0000 (0.74%) Tpc: 27.75
Other:	7 (0.58%) 00h:03m:07s 00.00.0000 (0.62%) Tpc: 26.71
Event-Driven:	5 (0.42%) 00h:00m:55s 00.00.0000 (0.18%) Tpc: 11.00
Math Spatial:	4 (0.33%) 00h:01m:39s 00.00.0000 (0.33%) Tpc: 24.75
General OpenGL:	4 (0.33%) 00h:01m:26s 00.00.0000 (0.29%) Tpc: 21.50
Lighting:	2 (0.17%) 00h:00m:45s 00.00.0000 (0.15%) Tpc: 22.50
Animation:	1 (0.08%) 00h:00m:29s 00.00.0000 (0.10%) Tpc: 29.00

Figure 19: Coding Metrics View for a single dimension

### 5.3.5 Performing the Change-Coding Process using the SCORE Analyser

This section will provide a brief walk-through of how coding of Changes is carried out via the SCORE Analyser.

Before commencing analysis, the user develops a set of dimensions, each containing a set of coding categories. These categories are inserted into the SCORE Analyser xml configuration file.

Using the SCORE Analyser the user visits each Change in turn via the Diff View's navigation panel. At each Change, a category is selected for each dimension using either the Change-Coding Panel or the Quick Coding Panel (marked with arrows in Figure 20).

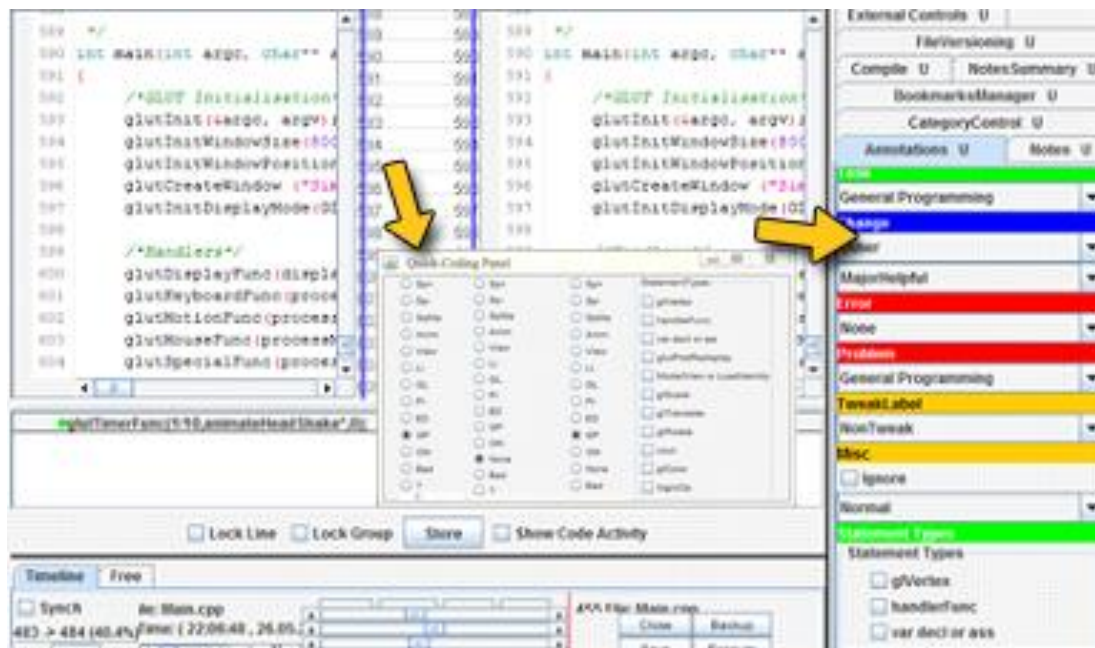


Figure 20: Coding using the Quick Coding and Coding Views

Once the user has coded all Changes the user utilises the Change-Coding Metrics panel to produce a count of the number of Changes falling into the different categories for each dimension, as well as the total time spent on Changes of that category. An example of data produced by this approach is shown in Table 10. Evaluation of data produced in this manner is presented in Section 5.3.

Table 10: Example of Change-Coding data

Category	Time	Changes	% Total Changes	% Total Time
None	13546	358	51.88%	51.44%
CombinedSpatial	5901	157	22.60%	22.56%
General Programming	2369	82	9.07%	11.78%
Viewing or Projection	1777	35	6.81%	5.03%
Animation	827	26	3.17%	3.74%
Pipeline	811	17	3.11%	2.44%
General OpenGL	520	12	1.99%	1.72%
Event-Driven	359	8	1.37%	1.15%
Lighting	0	1	0.00%	0.14%
Unknown	0	0	0.00%	0.00%
Other	0	0	0.00%	0.00%
	26110	696		

In addition to evaluation of the numerical Change data, the user generates a graph showing the rolling average of Changes belonging to different categories as shown in Figure 18. For each problem category type, the user further analyses any large peaks further through manual examination of the Changes involved. This provides a quantitative view of Change-Coding data through numerical analysis as well as a qualitative view through the detailed analysis of Changes at the source code level though, as will be discussed in Section 5.5, the method of graph analysis did not produce satisfactory results due to problems with interpreting the distribution of coded Changes. The Change-Coding method does not include any formal way of capturing insight into the role of Changes in the student problem-solving process produced by qualitative analysis. Instead, such analysis must be captured in notes associated with individual Changes.

The Change-Coding functionality is described in more detail in the SCORE Manual, Section 9.5.3.9. A video demonstration of the Change-Coding functionality is available at <http://www.youtube.com/watch?v=hYwMk-x8g5k&feature=plcp> and in the electronic appendix.

## 5.4 Change-Coding Results

This section presents results from the coding of individual Changes. Change-Coding analysis involves the classification of every Change in a Project History along Action, Error and Problem dimensions using the approach and classification scheme described earlier in this chapter. Analysis was carried out on the data consisting of ten Project Histories belonging to five students. The data and the selection method are described in detail in Section 3.6.2.2.

### 5.4.1 Action / Error / Problem Data

#### 5.4.1.1 Assignment 1

Table 11 presents the percentage of Changes and the percentage of time taken by those Changes that fall into different categories for the Action, Problem and Error dimensions for Assignment 1. These percentages are of a total of 6556 Changes spanning 83 hours.

**Table 11: Percentage of Changes falling into the different Coding classification categories for the Action, Error and Problem dimension in Assignment 1; the three top categories (including the most Changes) are marked in blue**

	Action Time	Action Changes	Error Time	Error Changes	Problem Time	Problem Changes
CombinedSpatial	21%	21%	7%	6%	11%	10%
Major Spatial	8%	8%	4%	3%	6%	5%
Minor Spatial	10%	11%	2%	2%	2%	3%
Math Spatial	3%	3%	1%	1%	3%	2%
General Programming	44%	47%	13%	12%	32%	32%
Event-Driven	17%	12%	2%	2%	6%	4%
General OpenGL	6%	6%	2%	2%	2%	2%
Pipeline	5%	5%	2%	2%	5%	5%
Other	3%	4%	1%	1%	1%	1%
Bad	3%	5%	4%	5%	3%	5%
None	-	-	68%	70%	40%	40%
Mean	12%	12%	4%	4%	7%	7%
StdDev	12%	13%	19%	19%	13%	13%

Raw data describing the total time spent and number of Changes rather than percentages can be found in the appendix, Section 9.6.2.

The largest number of Changes (44%) falls into the ‘*General Programming*’ category. The second-largest category is ‘*Spatial*’ with 21% of total Changes, with the third-most Changes falling into the ‘*Event-Driven*’ category, with 17%.

These results largely reflect the setting of assignment tasks. These tasks involved the creation of two-dimensional icons, the ability to place items on the drawing surface, rotation of one object about another, the functionality to create parent-child relationships between placed objects, the implementation of a functioning user interface to enable all of these actions as well as some minor subtasks. All tasks in the first assignment were restricted to two dimensions. The tasks are described in detail in Appendix Section 9.3.6.1. It does show that tasks of neither type were trivial to complete, otherwise students would not have required such a large number of Changes.

The large number of ‘General Programming’ related Changes is somewhat surprising. While the assignment skeleton did include an object-oriented framework for students to extend, aside from some relatively trivial object-oriented implementation of classes the assignment involved no explicit ‘General Programming’ problems. Part of the reason may be that students produce many syntax errors which are quickly fixed but still add up to a significant number of Changes overall. It should be noted that the figures presented in this Section do not give insight into how quickly an error was fixed; while time per Change of Changes falling into different categories is presented, an error may take a single Change to address, or it may take a hundred Changes. Therefore, the time taken per Change does not reveal how long errors of a certain category took to fix. Each of these Changes could have been fixing one single error or they could each have addressed an individual error. This shows the limitation of the Change-Coding method (discussed in detail in Section 5.5). The method lacks context as it does not explicitly store which Changes are associated with the solution of a particular problem. Such questions can be addressed by the Segment-Coding method presented in the next chapter.

The small number of ‘OpenGL’-related Changes suggests that OpenGL syntax and general concepts relating to the use of OpenGL did not play a significant role in student work on Assignment 1. This suggests that students did not find the OpenGL API difficult to use.

#### **5.4.1.2 Assignment 3**

Change-Coding data for Assignment 3 is presented in

Table 12. These percentages are of a total of 4016 Changes spanning 55 hours. Raw data on time and number of Changes can be found in the appendix, Section 9.6.2.



**Table 12: Percentage of Changes falling into the different Coding classification categories for the Action, Error and Problem dimension in Assignment 3**

	Action Time	Action Changes	Error Time	Error Changes	Problem Time	Problem Changes
CombinedSpatial	39%	35%	13%	11%	17%	16%
Major Spatial	25%	21%	11%	9%	14%	14%
Minor Spatial	12%	13%	1%	1%	1%	1%
Math Spatial	2%	1%	1%	1%	1%	1%
General Programming	27%	30%	8%	7%	20%	21%
Event-Driven	7%	7%	1%	0%	1%	1%
Viewing	6%	5%	3%	2%	3%	3%
Animation	6%	5%	1%	1%	4%	3%
General OpenGL	2%	3%	1%	1%	1%	1%
Pipeline	2%	1%	1%	1%	2%	2%
Lighting	3%	3%	0%	0%	0%	1%
Other	4%	5%	1%	0%	1%	0%
Bad	4%	5%	4%	5%	4%	5%
None	-	-	68%	71%	46%	46%
Mean	6%	7%	2%	2%	4%	4%
StdDev	7%	8%	19%	20%	13%	13%

The category containing the largest number of Changes is the ‘Spatial’ category, with 35% of all Changes involving spatial programming. The second-largest category is ‘General Programming’ involving 30% of all Changes. Thus, ‘General Programming’ and ‘Spatial’ are once more the largest categories, though their order is reversed in Assignment 3. The ‘Event-Driven’, ‘Animation Algorithm’ and ‘Viewing’ categories are each involved in slightly more than 5% of total Changes. The third assignment involved several major tasks. The first was the assembly of an avatar and the production of animations using the avatar (both spatial tasks). The second was the implementation of three views (viewing task). The third was the implementation of an animation algorithm (animation task). The fourth was the implementation of user interface functionality to activate animations and avatar movement (event-driven tasks) as described in more detail in Appendix Section 9.3.6.2. Most of the spatial tasks in Assignment 3 involved three-dimensional spatial programming.

Results from Assignment 3 are similar to those from Assignment 1. Again the 'Spatial' category makes up a large number of Changes which is unsurprising since most tasks involved 'Spatial' programming, though it does indicate these tasks are not of a trivial nature as they require many Changes to complete.

'General Programming' again makes up a substantial percentage of total Changes (though fewer than in Assignment 1) despite Assignment 3 not featuring even the simple object-oriented structure of the first assignment. This means that a substantial amount of the total student effort for both assignments was invested in non-Computer-Graphics related 'General Programming'. This is suboptimal given the Computer Graphics learning objectives of both assignments.

As with the first assignment, the small number of 'OpenGL' Changes suggests that the OpenGL API and concepts related to its use did not present a significant challenge to students. This suggests that OpenGL is a suitable API for teaching Computer Graphics Education and additional scaffolding for this is unnecessary.

## 5.4.2 Relationship between Action/Error/Problem Dimensions

### 5.4.2.1 Error/Action

This section examines the ratio of Changes marked as Error to those marked as Action. Coding data from both assignments is combined, but for purposes analysis of Spatial-coded Changes were divided into Spatial-coded Changes from Assignment 1, labelled 'Spatial A1', and those from Assignment 3 labelled 'Spatial A3' since Assignment 1 involved two-dimensional spatial programming, whereas Assignment 3 mainly involved three-dimensional spatial programming.

**Table 13: Time/Changes spent on Error Changes divided by Time/Changes spent on Action Changes**

	Time	Time-Mean	Changes	Change-Mean
Spatial A1	0.33	0.02	0.30	0.02
Spatial A3	0.32	0.02	0.31	0.03
General Programming	0.30	0.00	0.24	-0.04
Event-Driven	0.13	-0.18	0.12	-0.16
Viewing or Projection	0.52	0.21	0.47	0.19
Animation	0.18	-0.13	0.17	-0.11
General OpenGL	0.40	0.09	0.32	0.04
Pipeline	0.48	0.17	0.48	0.20
Lighting	0.10	-0.20	0.11	-0.17
Mean	0.31		0.28	
StdDev	0.14		0.13	

The Error/Action ratio describes how often work on a certain category of task causes an error to occur. The data is presented in Table 13. It combines data from both Assignment 1 and Assignment 3. Categories for which the ratio of Error to Action Changes falls below or above one standard deviation from the mean are marked in red or blue respectively.

As the table shows, there appears to be a higher than average proportion of 'View' and 'Pipeline' Errors to Actions, suggesting that 'View' and 'Pipeline' Actions cause an abnormally high number of errors. 'Event-Driven' and 'Lighting' ratios fall more than one standard deviation below the mean, suggesting that 'Event-Driven' and 'Lighting' programming actions cause comparatively fewer errors than average.

This may indicate that 'View' and 'Pipeline' tasks are difficult to solve whereas 'Event-Driven' and 'Lighting' tasks are more straightforward to implement. It could also mean that students may be spending more time debugging 'Event-Driven' and 'Lighting' problems utilising output-to-console statements or similar approaches which do not introduce new errors. On the other hand implementation of 'View' and 'Pipeline' problems may involve more direct problem-solving approaches, perhaps involving trial-and-error solution attempts.

#### 5.4.2.2 Error/Problem

Table 14 presents the ratio of Error to Problem Changes for each category. Again, data from Assignment 1 and Assignment 3 are combined for analysis purposes, but 'Spatial'-coded Changes were divided into 'Spatial'-coded Changes from Assignment 1, labelled 'Spatial A1', and those from Assignment 3 labelled 'Spatial A3'. The data describes how often students make errors when working on a problem of a certain category.

**Table 14: Time/Changes spent on Error Changes divided by Time/Changes spent on Problem Changes**

	Time	Time-Mean	Changes	Change-Mean
Spatial A1	0.63	0.04	0.61	0.05
Spatial A3	0.74	0.15	0.66	0.11
General Programming	0.41	-0.18	0.35	-0.20
Event-Driven	0.43	-0.16	0.39	-0.16
Viewing or Projection	0.87	0.27	0.78	0.23
Animation	0.27	-0.32	0.26	-0.29
General OpenGL	0.94	0.35	0.90	0.35
Pipeline	0.43	-0.16	0.43	-0.12
Lighting	0.60	0.01	0.60	0.05
Mean	0.59		0.55	
StdDev	0.21		0.20	

Both *'View'* and *'OpenGL'* Problem Changes appear to more often cause further Error Changes of the same category than average, whereas *'Animation'* Problems appear to cause fewer Errors than average. This suggests that *'View'* and *'OpenGL'* problems are less straightforward to solve, involving more incorrect attempts at problem-solving, whereas *'Animation'* problems are easier to solve in an iterative fashion, producing fewer errors.

The result for *'OpenGL'* problems seems unusual given that OpenGL tasks did not appear to cause significant difficulty for students measured by the prevalence of *'OpenGL'* actions as shown in 5.4.1. The results for the *'OpenGL'* category were probably skewed by one particular student problem by student Ida which involved logical operations. During this problem the student utilised a trial-and-error approach and produced many errors. It is the only large OpenGL Segment found during analysis and it involved many erroneous modifications. This problem is described in Appendix Section 9.7.6.1.4.3.3.

### 5.4.3 Discussion of Change-Coding data

As the analysis of the Action dimension in Section 5.4.1 showed, *'Spatial'* and *'General Programming'* tasks took up many more Changes than other types of tasks, with *'Event-Driven'* Changes also playing a significant role in the first assignment.

The large number of *'Spatial'* Changes in both assignments and *'Event-Driven'* Changes in the first assignment reflects the assignment specification's focus on tasks relating to spatial programming in both assignments and on user-interface driven programming in the first assignment. The large number of Changes does suggest that neither of the tasks is trivial.

The significant number of *'General Programming'* Changes (being the most common type of Change in Assignment 1 and the second-most common in Assignment 2) was somewhat unexpected. The first assignment contained only peripheral *'General Programming'* related tasks relating to the object-oriented extension of the assignment skeleton. The second assignment contained no explicit *'General-Programming'* tasks at all. The analysis of Segment contents and features which is described in the next chapter (Section 6.6 and Section 6.7) will shed more light on the role of *'General Programming'*, showing that some students encounter significant problems with *'General Programming'* concepts and algorithms not detailed in the assignment requirements. However, detailed examination of assignments also revealed many small errors such as syntax errors caused by typing errors usually corrected in one or two Changes. Such errors are likely to also contribute significantly to the high total of *'General Programming'* related errors since any programmer is likely to produce a large number of such errors. Since the programming language used was C++, students

did not have available the full range of highlighting features available in the Eclipse Java environment. Such assistance may help reduce such small errors, and this may help give students more time to spend on problems related to learning goals. The potential benefit of such assistance will be explored in future research. The use of C++ may also have contributed to student '*General Programming*' problems since C++ involves some terse and ambiguous syntax. The role of such issues is investigated in more detail using the Segment-Coding method of analysis in the next chapter in Section 6.6.

Examination of the relationship between dimensions showed that '*Event-Driven*' and '*Lighting*' programming led to fewer errors than average. On the other hand, '*Pipeline*' and '*View*' programming led to more errors than average. There are two possible explanations. One is that '*Pipeline*' and '*View*' problems are especially difficult to solve and '*Event-Driven*' and '*Lighting*' problems are easy to solve. Another is that students tended to utilise a trial-and-error approach to solve '*Pipeline*' and '*View*' problems and a more thought-out approach for solving '*Event-Driven*' and '*Lighting*' problems.

The ratio between Errors and Problems showed that many errors occurred during '*View*' and '*OpenGL*' problem solving, whereas few errors occurred during the solving of problems related to the implementation of '*Animation*' algorithms. This may indicate that '*View*' and '*OpenGL*' problems were solved via a trial-and-error approach with students trying different incorrect solution attempts whereas Animation algorithms were implemented in a more straightforward fashion, building up the solution iteratively with fewer errors or using debugging, which does not count as an error.

The next section will describe how the Change-Coding method could not answer some of the questions regarding the student programming process and student problem-solving raised during the analysis process, such as whether Error/Problem ratios indicated different styles of problem-solving or different levels of problem difficulty.

## 5.5 Limitations of the Change-Coding method

The application of the Change-Coding method discussed in this chapter revealed limitations of the method as a means of pinpointing and analysing student problems occurring during Computer Graphics programming.

It was assumed that Change-Coding could be completed quickly based on a scanning of the lines modified in that Change, perhaps involving an average of five or so seconds to code a single Change. At that rate, a Project History of 1000 Changes could be analysed in a little under an hour and a half. In practice, Change-Coding was time-consuming since examination of local modifications was often

insufficient to determine the appropriate category for a Change. For example, a Change involving a modification to a transform command such as `glVertex(1,0,0) -> glVertex(0,1,0)` is quite easy to code as a 'Spatial' action. On the other hand, a problem associated with the addition of a `glutPostRedisplay()` call (forcing a call of the OpenGL display function) is much more difficult to code. Students frequently added such calls when they were unsure what the problem was. The addition of this call could thus fall under a number of different categories depending on the context. Such Changes then required an examination of surrounding Changes until sufficient contextual understanding had been developed. This ran counter to the context-free nature of the Change-Coding approach. Instead of spending only a few seconds per Change as had been anticipated, tens of seconds or even minutes were required for the coding of individual Changes, leading to an analysis time of many hours per assignment. In practice, time taken per Change was probably closer to 60 seconds on average at the least, leading to analysis times of 16 hours for a single Project History. Re-coding as was required after moving from the first to the second version of the classification scheme as described in Section 5.2.1. This involved a similar time investment on top of the time spent on coding using the first version of the scheme.

The reason for this unexpected blow-out in analysis time was that the importance of context had not been taken into account during the initial development of the Change-Coding method. Coding of a Change requires identification of any errors caused by or fixed by the Change, which in turn requires an understanding of the context in which the modification occurs. It is that context which determines whether a particular modification is an error or a positive contribution towards a solution.

To determine the context for individual Changes required a large amount of time. As long as the researcher continues working on coding Changes one after another the researcher can usually remember enough of the previous Changes to establish this context. However, the Change-Coding method does not capture the context explicitly. It is indirectly captured via the 'Problem' classification dimension. However, it is not captured explicitly. Therefore, if the researcher loses track of the context of a particular Change (because he engaged in a different task, or because the Change was coded a long time ago, or because he was interrupted) he will have to re-establish the context by reading notes associated with Changes. This is required to correctly determine which Changes contribute to the context in which way and how the Changes relate to one another. This is particularly costly if any reclassification becomes necessary due to changes to the classification scheme. The Change-Coding method's failure to store Change context causes a large time-sink.

In addition to being very time-consuming, evaluation of data produced via the Change-Coding method also proved to be less expressive than was hoped. Using very specific categories (hence increasing the total number of categories) means coding requires more time as there are more categories to choose from at every Change. It also means that many categories include only a small number of items. This makes comparison of different categories difficult. On the other hand, more general and broad categories produce more items per category which allows for a more meaningful comparison and a more efficient coding but also means that problems with different features are lumped together, and the unique features are lost in the analysis. To make analysis feasible in terms of time required for coding, general categories were utilised.

The problem regarding the generality of categories was further compounded by the Change-Coding method not formally capturing or coding the context in which Changes occur. The context contains valuable information. It sheds light on the problem-solving approach used. It shows whether the problem was solved quickly or took many Changes to resolve. It also contains details relating to the nature of the problem and its features which cannot be derived from or associated with individual Changes, but which are vital to a detailed understanding of student problems and problem-solving during Computer Graphics programming.

To address the limitations regarding the context-free nature of Change-Coding an attempt was made to identify sequences of Changes sharing the same classification category, assuming they were related Changes addressing a task / problem. Such sequences of Changes could then be examined in more detail via qualitative analysis. To identify such sequences of Changes the coding results were visualized using graphs. This allowed identification of periods in which many Changes were assigned to the same error/problem/action category.

This approach was another reason for why categories had to be general, since if Changes related to the solution of a task or problem were coded with different categories then the Changes would not be detectable as a sequence of Changes sharing the same category. While producing sub-dimensions for each dimension would have addressed that problem (the higher-level code could have been used to detect related Changes) such a classification scheme would have required significantly more effort to apply.

Unfortunately approaches to identifying related Changes for qualitative analysis using Change-Coding data did not perform well in practice. Many small problems of the same type solved in close temporal proximity produce the same coding pattern (many Changes falling into the same error/problem/action category) as a single large (and hence more interesting) problem. On the other

hand, work on a large problem may frequently be interrupted by small intervals of work on other problems. In this case the sequence of versions will appear to be a multitude of small and apparently insignificant problems. Despite the significant effort involved in coding a Project History, there is no simple way to use this data to identify and qualitatively analyse problems made up of related Changes. Even if problems were identified, the Change-Coding method does not include any formal way of capturing data on the related Changes addressing the problem, making any further analysis problematic.

It is due to these limitations that the Change-Coding method of Project History analysis can only produce a high-level macro view of student Computer Graphics programming. The aim of this research project is to provide a complete picture including detailed understanding of student programming problems. This means the Change-Coding method cannot fully address the aims of this research project.

## 5.6 Change-Coding Conclusion

The Change-Coding analysis presented in this chapter provides a comprehensive overview of the sorts of actions, errors and problems that a sample of five students experienced and completed while writing two computer graphics programs. Analysis utilising the Change-Coding method showed that '*General Programming*' and '*Spatial*' problem-solving makes up the majority of Changes in the first and second assignments, providing some support for answering **RQ2** "Is 'spatial programming' a difficult area/topic of Computer Graphics programming for students?" in the affirmative.

Analysis of action to problem/error ratios showed that '*Lighting*' and '*Event-driven*' problems apparently produced fewer errors than expected, while '*Pipeline*' and '*View*' problems produced more errors than expected. This could mean that '*Pipeline*' and '*View*' problems are more difficult than other types of problems, or it could mean that students utilise different strategies to solve these types of problems.

While this sheds some light on student problem-solving, as described in Section 5.5 quantitative analysis of Change-Coding results only provided a shallow view of the student programming process. This view probably reflected the assignment specification as much as it did specific student problems. Change-Coding data also did not allow for the reliable identification of Changes related in the solution of problems in Project Histories. Thus, it did not allow for qualitative analysis of such problems to address questions left unanswered by quantitative analysis.



The Change-Coding method's limitations means that it does not provide a complete method of source-code level analysis of Project Histories. Hence, it does not address research question **RQ3** adequately. As discussed in Section 5.5, the limitations associated with the Change-Coding method are due to its context-free nature. The next chapter will introduce a new approach, the Segment-Coding Method, which was developed to address this limitation by coding sets of related Changes, thereby capturing the context in which a Change occurs.

Given the limitations of the Change-Coding approach, the results presented in this chapter should best be understood in the context of the findings arising from the analysis of Segment contents and features presented in the next chapter in Sections 6.6 and 6.7. They also serve as a rationale for the development of the Segment-Coding method presented in that chapter.

# 6 Segment-Coding

---

## 6.1 Segment-Coding Method Introduction

The Change coding method presented in the last chapter produced a macro view of Project Histories and student programming. However, it was limited in its ability to provide detailed insight into the nature of student programming problems and problem-solving processes. As a consequence, a second method of analysis was developed as part of this research project.

The problem with the Change-Coding method was rooted in its context-free nature. It was hard to code individual Changes. Furthermore, after coding was complete Change-Coding results did not allow for a reliable and effective pinpointing of individual periods of programming related to specific problems. This means that individual problems could not be analysed and no concrete statements could be made about the kind of problems faced by students.

To address the limitation of the Change-Coding method, a new context-aware method was developed. Instead of coding individual Changes, the new method identified sets of Changes related because they solve the same task or problem. These sets of Changes are captured as *Segments*. Each Segment includes all Changes involved in completing a task or solving a problem. This makes it possible to then conduct a qualitative analysis of Segment contents. It also allows for the analysis of quantitative features such as the time taken for Changes that are part of a Segment. This method will be called the *Segment-Coding* method.

The identification of related Changes to form Segments requires an understanding of the context in which Changes occur. It also requires an understanding of the relationship between Changes. Such relationships are difficult to determine using only the SCORE Analyser's core functionality. For this reason new SCORE Analyser views based on machine-generation of line histories<sup>35</sup> were implemented. These views allow the researcher to explore each individual source code Line History. They also allow the researcher to view summaries of modifications occurring over many Changes. The development of this new SCORE functionality is described in Section 6.2.

---

<sup>35</sup> Line History: A Line History stores all the modifications that are applied to a line of source code in the project history. Each line of source code in every version stored in the project history is assigned to exactly one line history.

Section 6.3 discusses the rules for the application of the Segment-Coding method, including a reworked one-dimensional version of the Change-Coding classification scheme to be used for classifying Segments. The section also includes the description of SCORE functionality to allow for the analysis and structuring of source code using Segments.

The results of segmenting ten Project Histories (the same data analysed via the Change-Coding method, described in Chapter 5) are presented in Section 6.5. Section 6.6 provides an in-depth analysis of the content of identified Segments. An analysis of Segment features is presented in Section 6.7.

## 6.2 Enabling Detailed Analysis with the SCORE Analyser

Part of the contribution of this research project to the field of Computer Science Education research is the development of a method for the efficient analysis of Project Histories at a source-code level.

Previous approaches analysing Project Histories have focused on metric-level (Mierle et al., 2005) or output-level (Jadud, 2006a) analysis. Such analysis can be automated through calculation of abstract metrics or machine-evaluation of compilation errors.

Source-code level analysis has the potential of providing a better, more in-depth contextual understanding of the student programming process. On the other hand, such analysis cannot be fully automated since it requires understanding of student actions in a Project History's Changes, which in turn requires human reasoning. This combined with the large number of Changes that make up a Project History means that source-code level analysis will be more work-intensive. A successful method of source-code level analysis must provide tools to reduce this workload to make the application of the analysis method feasible in an actual research project.

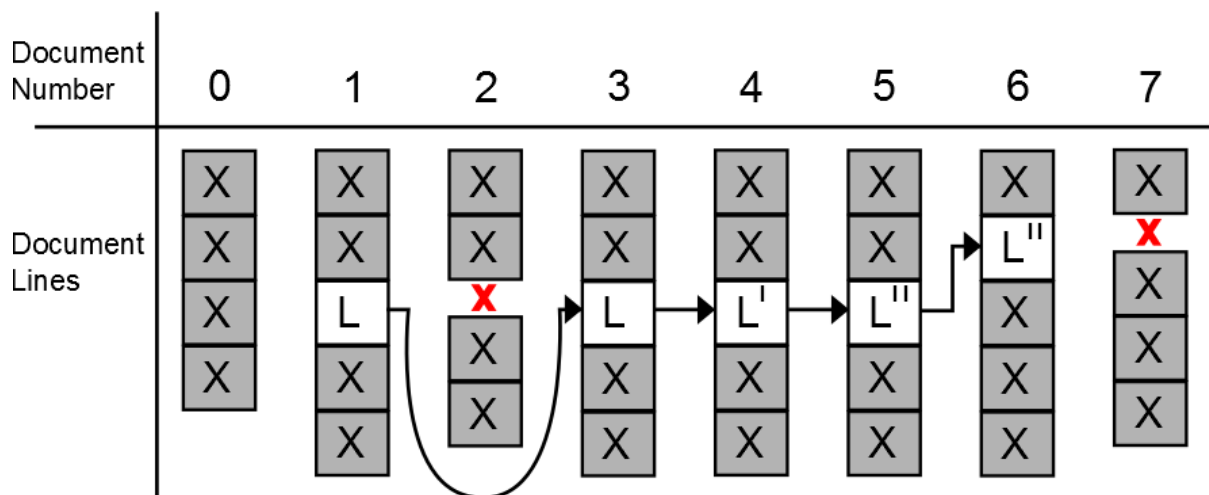
This section will present the development of functionality to enable effective source-code level analysis. In contrast to the basic functionality for browsing versions of source code presented in Section 4.4, these views provide the user with summaries which provide insight into the context in which individual Changes and modifications occur. This is essential for the application of the Segment-Coding method presented in this chapter which is based on understanding of context as compared to the non-contextual Change-Coding method presented in the previous chapter. This functionality is based on the concept of *Line Histories*. Every source code line in every version in the Project History is mapped to a Line History object. The development and evaluation of an algorithm for producing such Line Histories is presented in Section 6.2.1.2. Section 6.2.2 describes how these Line Histories are used to produce new SCORE Analyser functionality to enable detailed source-code

analysis. Section 6.4.2 presents a walk-through of how this functionality is used to facilitate detailed analysis of Project Histories.

## 6.2.1 Line Histories

### 6.2.1.1 Description of Line Histories

Line Histories are histories of lines. Each Line History consists of all the modifications to a single line which occur in a Project History. An example of a line being modified is shown in Figure 21. It shows eight sequential versions of a source-code document in which the line in question (marked with an L) is modified seven times. It is added in the second version, deleted in the third, re-added in the fourth, modified (the text is changed) in the fifth and sixth version, moved in the seventh version and finally deleted again in the eighth version. The Line History table shown in Figure 22 stores each of these modifications. Data stored include the modification type, the document in which the modification occurs, the line number of the line in that document and the text of the modification in that document.



**Figure 21: Example of modifications affecting the line marked 'L', including addition, deletion and modification to the line text marked as L-prime and L-prime-prime**

Document Id	Line Number	Modification Type	Modification Text
1	3	Added	L
2	-	Deleted	-
3	3	Ghost	L
4	3	Mutated	L <sup>i</sup>
5	3	Mutated	L <sup>ii</sup>
6	4	Moved	L <sup>ii</sup>
7	-	Deleted	-

**Figure 22: Example of the Line History table data structure associated with the line shown in Figure 21**

Line Histories used to implement much of the SCORE Analyser’s functionality. By generating Line Histories for all lines occurring in the Project History it becomes possible to view all modifications made to a line; the SCORE Analyser provides this functionality via the ‘Line History View’ presented in Section 6.2.2.2. Line Histories also make it possible to summarise Changes. The regular diff view shows all lines in a Change. The ‘Change Browser’ presented in Section 6.2.2.1 shows only the modified lines and the type of modification affecting those lines for each Change. Line Histories are also utilised in the machine-generation of Segments. Since both the machine-generation of Segments and the generation of Line Histories are software-engineering problems, algorithms for both are described and evaluated in detail in Chapter 7. This section serves as an introduction to the concept of Line Histories. The next section will describe the different types of modifications that make up Line Histories, whereas Section 6.2.2 will describe the ‘Line History View’ and the ‘Change Browser’ which are based on Line Histories.

### 6.2.1.2 Description of Modification Types and link to Problem Types

This section will describe the different types of line modifications detected by the line history generation algorithm. Modifications are changes to a line’s text or position, or add or remove a line.

#### 6.2.1.2.1 Maintained Modifications

The MAINTAINED modification type means that the line in question was not modified in this Change. That means that the line text remained the same (ignoring leading and trailing whitespace) and the line remained in the same relative position in the Longest Common Subsequence of lines. Line movement is detected using the Longest Common Subsequence algorithm. This is described in more detail in the Line History Generation Section 7.2.1.

#### 6.2.1.2.2 Addition and Deletion Modifications

Additions and deletions are the simplest type of line modification. They occur when a line is either added or removed from the previous version of the document.

Addition, Deletion and Maintained are the only modification detected by standard diff algorithms. These algorithms use the longest common subsequence to identify any differences between two versions of a document. All other modification types discussed later are detected as additions and deletions by standard diff algorithms.

A large amount of line addition usually occurs when a new problem is being approached; for example, a student may create new data structures or new functions as he moves to the next part of the assignment. Deletion of lines often occurs when a student decides to adopt a different solution approach. To do so, the student may remove lines belonging to the old approach. Deletions also occur as part of a clean-up / removal of dead and commented-out lines of code. In the latter case they can thus be a red herring when looking for significant problems.

Additions and deletions can be used to detect the creation of new programming constructs (classes / functions / data structures) and work on a new problem. However, they do not reveal problems that involve intense work on only a small number of lines without the frequent addition and deletion of lines.

#### 6.2.1.2.3 Mutation Modifications

Mutation modifications are modifications to a line of code which change the line's text as in Figure 23. A standard diff algorithm will detect a mutation modification as a deletion of the mutated line in the last document and an addition of the mutated line with the new text in the current document. For this reason, a standard diff algorithm loses valuable information when the context is the analysis of many Changes instead of two since it becomes impossible to trace the different 'versions' of a line.

The Line History Generation algorithm developed for SCORE recognises lines that have been modified from one Change to the next by comparing the text of modified lines. Figure 23 shows an example of a mutation, involving the change of two out of seventeen characters. Figure 24 shows an example of a Line History containing twelve modifications to a line.

```
guy.partsRot[PERSON_RLA][Z]+=INC; -> guy.partsRot[PERSON_LL][Z]+=10*INC;
```

Figure 23: A line mutation changes the line's text, as marked in yellow

(963): (209, 259, 266-271, 336, 339, 346, 349 : total = 12)
209, ADDED(O): guy.partsRot[PERSON_RLA][Z]+=INC;
267, MUTATED(O): guy.partsRot[PERSON_LLA][Z]+=10*INC;
268, MUTATED(O): guy.partsRot[PERSON_LLA][Y]+=10*INC;
269, MUTATED(O): guy.partsRot[PERSON_LLA][Y]-=INC;
270, MUTATED(O): guy.partsRot[PERSON_LLA][Y]-=2*INC;
271, MUTATED(O): guy.partsRot[PERSON_LLA][Y]-=INC2;
336, MUTATED(O): guy.partsRot[PERSON_LLA][X]+=INC;
339, MUTATED(O): guy.partsRot[PERSON_LLA][X]-=INC;

**Figure 24: A Line History composed of several mutations, modifying the axis of a transformation**

In practice the Mutant line modification type is the most significant of all modification types. The different mutant versions of a line stored in its line history can provide valuable insight into the student's solution attempts and errors.

Mutant line modifications allow identification of a wide range of problems by examination of the textual changes to the line and an understanding of the problem can often (but not always) be built up independent of the context in which the line occurs. For example, Mutant modifications can help identify Segments in which the student is working on an animation which will usually involve modification to lines containing transformation calls or to data structures storing data used with transformation calls. An example from a student assignment is shown in Figure 24. The student modifies the line, which is part of an animation 11 times, making three changes to the rotation axis.

#### **6.2.1.2.4 Moved Modifications**

Moved modifications are modifications in which a line is moved to a different position relative to its neighbouring lines in the previous Change (see Section 7.2.1 for details on how moved lines are detected). Standard diff algorithms using only calculation of the Longest Common Subsequence (LCS)<sup>36</sup> to detect additions and deletions would detect movement of lines as a combination of additions and deletions instead, meaning they would not correctly match lines from one version to the next.

In practice, lines are most often moved when the student is dealing with a program flow problem, and Line Histories with many moved modifications are probably part of a student's attempts to deal with program flow by resituating the line and observing the effects during debugging. Without the

<sup>36</sup> The longest common subsequence (LCS) problem is to find the longest subsequence common to all sequences in a set of sequences ([http://en.wikipedia.org/wiki/Longest\\_common\\_subsequence\\_problem](http://en.wikipedia.org/wiki/Longest_common_subsequence_problem)).

moved modification type such problems would not be detectable as they do not involve any change to the line's text.

One example is the Line History shown in Figure 25. Modifications to the Line History in context are shown in Figure 26. The line `menu.setLocation(-1,-1)` is moved several times as the student attempts to debug a problem with event-driven program flow. If the algorithm did not include detection of moved lines, then the Line History would have been incomplete, containing only a small number of modifications.

(2414): (759, 761-762, 767-768, 773 : total = 6)	
759, ADDED(O):	<code>menu.setLocation(-1,-1);</code>
761, MOVED(O):	<code>MOVED (menu.setLocation(-1,-1);)</code>
762, MOVED(O):	<code>MOVED (menu.setLocation(-1,-1);)</code>
767, MOVED(O):	<code>MOVED (menu.setLocation(-1,-1);)</code>
768, MOVED(O):	<code>MOVED (menu.setLocation(-1,-1);)</code>
773, MUTATED(O):	<code>menu.setLocation(-1,-1); /* make menu disappear */</code>

Figure 25: Line History from John A1; without moved-line detection, the Line History loses four of its six entries

761

```

if(_button==GLUT_RIGHT_BUTTON && _state==GLUT_DOWN)
    if(_state==GLUT_DOWN)
        menu.setLocation(_x, _y);
    else
        menu.setLocation(-1,-1);
    return;
}
mode = menu.hit(Point(_x, _y) ) + 1;

```

762

```

if(_button==GLUT_RIGHT_BUTTON) {
    if(_state==GLUT_DOWN)
        menu.setLocation(_x, _y);
    else
        return;
}
mode = menu.hit(Point(_x, _y) ) + 1;
menu.setLocation(-1,-1);

```

767



```

if(_button==GLUT_RIGHT_BUTTON) {
    if(_state==GLUT_DOWN)
        menu.setLocation(_x, _y);
    return;
}
menu.setLocation(-1,-1);
mode = menu.hit(Point(_x, _y) ) + 1;
if(mode>0)
    return;

```

768

```

if(_button==GLUT_RIGHT_BUTTON) {
    if(_state==GLUT_DOWN)
        menu.setLocation(_x, _y);
    return;
}
mode = menu.hit(Point(_x, _y) ) + 1;
if(mode>0) -> if(mode>0) {
    menu.setLocation(-1,-1);
    return;
}

```

Figure 26: A program-flow problem is debugged through moving of the line "menu.setLocation(-1,-1)"

#### 6.2.1.2.5 Ghost Modifications

A ghost modification is a re-addition of a line which was deleted in a previous Change. Detection of ghost modifications requires comparison of added lines in a Change to all the lines deleted in previous Changes. In practice only lines deleted in the recent past are to be included. Ghost lines are somewhat different from other modification types since their detection requires analysis of modifications spanning several Changes, whereas other modification types are based solely on the analysis of a pair of versions.

Ghost lines can occur for many different types of problems. They usually occur when a student identifies a problem line and temporarily deletes it during debugging to observe whether the removal impacts the problem, but then re-introduces the line again in a subsequent Change.

While ghost modifications serve to help identify such problems, they also play another (more important) role by ensuring that modifications made to the line after the ghost modification are added to the same Line History as those made to the line before the modification. Without ghost modifications, two separate Line Histories would be created instead. This would then break up an 'interesting' Line History into two smaller and thus apparently less significant line histories.

An example of ghost modifications occurring in Line Histories is shown in Figure 27. Without the ability to detect Ghost modifications, the Line History would have been split into four separate Line Histories, with the largest only containing half as many entries as the Line History resulting from the inclusion of Ghost modifications.

(1638): (104-110, 113, 128-130, 132, 465 : total = 13)
104, ADDED(O): glBegin( GL_POLYGON); /* draw filled triangle */
105, DELETED(O): DELETED (glBegin( GL_POLYGON); /* draw filled triangle */)
106, GHOST(O): GHOST (glBegin( GL_POLYGON); /* draw filled triangle */)
107, DELETED(O): DELETED (glBegin( GL_POLYGON); /* draw filled triangle */)
108, GHOST(O): GHOST (glBegin( GL_POLYGON); /* draw filled triangle */)
109, DELETED(O): DELETED (glBegin( GL_POLYGON); /* draw filled triangle */)
110, GHOST(O): GHOST (glBegin( GL_POLYGON); /* draw filled triangle */)
113, MUTATED(O): glBegin( GL_LINE); /* draw filled triangle */
128, MUTATED(O): glBegin(GL_LINE); /* draw filled triangle */
129, MUTATED(O): glBegin(GL_LINES); /* draw filled triangle */
130, MOVED(O): MOVED (glBegin(GL_LINES); /* draw filled triangle */)
132, MOVED(O): MOVED (glBegin(GL_LINES); /* draw filled triangle */)
465, DELETED(O): DELETED (glBegin(GL_LINES); /* draw filled triangle */)

**Figure 27: Example from Michael A1. Without ghost detection, the line history would be split into four separate Line Histories**

### 6.2.2 SCORE Detailed Analysis Features

This section will present SCORE Analyser functionality based on the Line History concept introduced in the last section. SCORE Analyser Views and components described in this section receive a Line History collection object. They then disseminate the information contained in the collection object to populate their View. In addition to the Views described here, the main Diff View (Section 4.4.1) also benefits from the introduction of Line Histories, since it now uses modifications as stored in Line Histories to map between the lines of the previous and current version rather than the mapping produced by a simple diff algorithm, which includes only added and deleted lines. This makes the mapping of lines much more meaningful. For example, through the use of the Line History table the Diff View can mark modified lines as ‘Mutated’ instead of marking the old version as deleted and the new version as added.

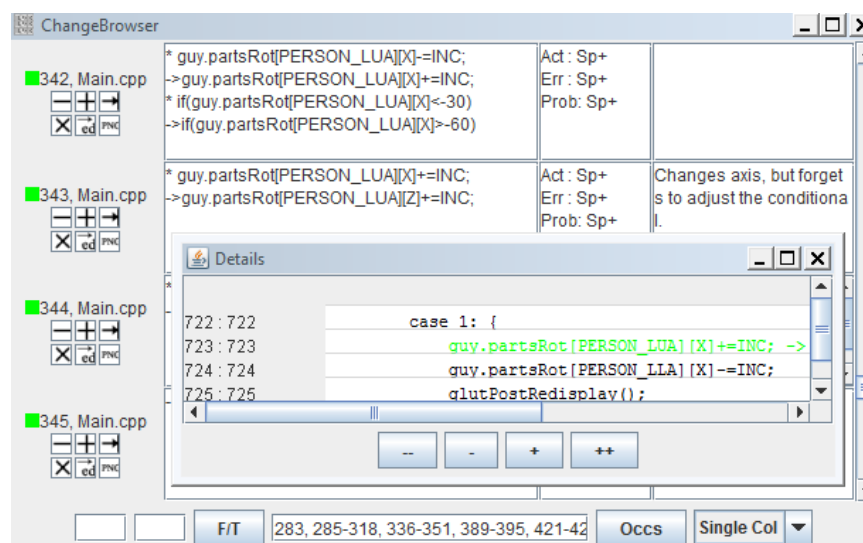
The combination of these views and tools enable the researcher to gain an understanding of the context in which a Change occurs much more effectively than would be possible by manual

comparison of different versions of source code. This reduces the large cognitive burden of remembering the content of the many different versions that can make up a unit of work.

### 6.2.2.1 Change Browser

The core ‘Diff View’ (Section 4.4.1) provides the user an easy visual comparison between one Change comprising two versions of source code. This is useful for comparing very complex changes in detail. However, most Changes involve the modification of a small number of lines (often a single line). Requiring the user to navigate each of these Changes in turn is unnecessarily cumbersome. It requires more time. It also requires the researcher to remember the last several Changes. Only then can the researcher place the Change currently being examined into the larger context of the problem.

The ‘Change Browser’ (Figure 28) allows the user to examine several changes at once by showing only the lines that were added, deleted or mutated from one document to the next (Figure 28, left). It also shows Change-Coding data (Figure 28 middle, see ‘Coding Description’, Section 5.3) and the Note (Figure 28 right, from the Note View, Section 4.4.2) for that Change.

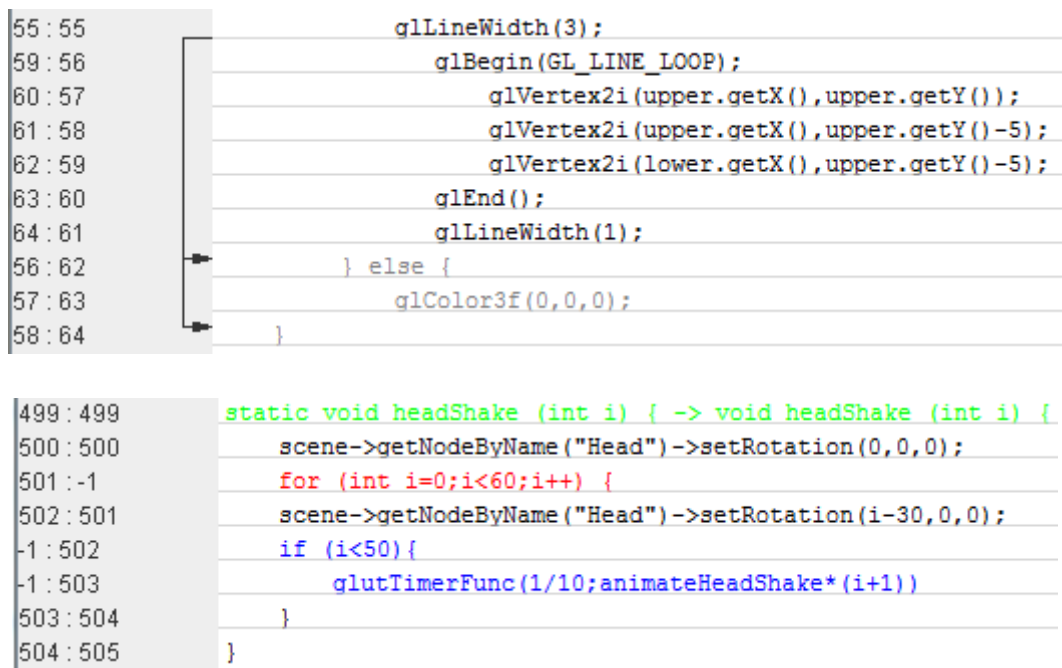


**Figure 28: Change Browser and Details Window**

Each Change also includes buttons (on the left) which allow the user to navigate to that change in the Diff View, execute the Change’s executable file or launch a source code editor for that Change (see Section 4.4.3.1).

In addition, the user can open a Details window for each change (see Figure 29) which shows all of the version’s source code lines along with visual representations of all line modifications that occurred during that Change. Moved lines are shown using arrows indicating the code’s current and

previous position. Additions and deletions are shown in blue and red respectively. Mutations are shown in the form of 'OldMutationText -> NewMutationText'. Compared to the standard Diff View, this view summarises all modifications to a Change in a single view, rather than showing them as mappings between two separate views. This more compact form provides the user a quick overview of a given Change. The user can also change the mode of the browser to display these Detail windows in the main browser for every Change rather than a list of modifications as shown in Figure 30. This provides context for modified lines, thereby making it easier for the user to understand a given Change at the cost of requiring more space per Change displayed.



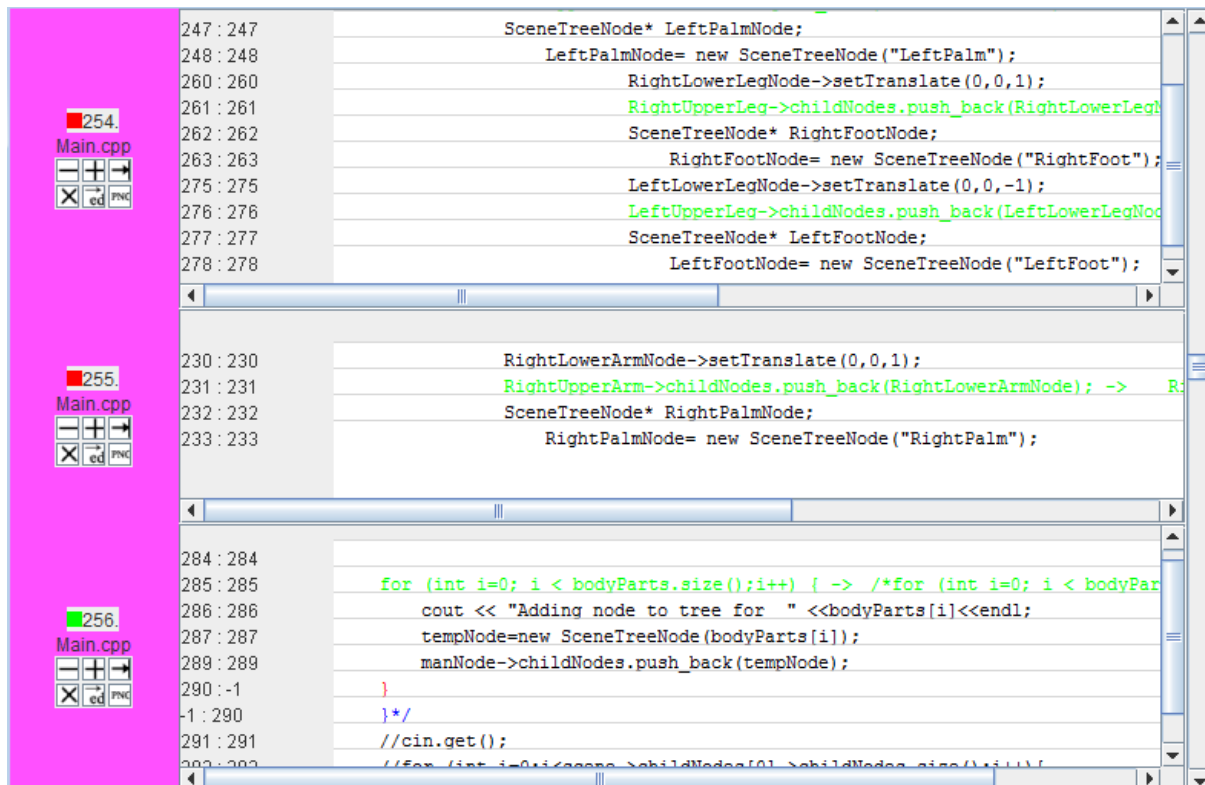
```

55:55      glLineWidth(3);
59:56      glBegin(GL_LINE_LOOP);
60:57      glVertex2i(upper.getX(),upper.getY());
61:58      glVertex2i(upper.getX(),upper.getY()-5);
62:59      glVertex2i(lower.getX(),upper.getY()-5);
63:60      glEnd();
64:61      glLineWidth(1);
56:62      } else {
57:63      glColor3f(0,0,0);
58:64      }

499:499  static void headShake (int i) { -> void headShake (int i) {
500:500      scene->getNodeByName("Head")->setRotation(0,0,0);
501:-1      for (int i=0;i<60;i++) {
502:501      scene->getNodeByName("Head")->setRotation(i-30,0,0);
-1:502      if (i<50){
-1:503          glutTimerFunc(1/10;animateHeadShake*(i+1))
503:504      }
504:505  }

```

Figure 29: Two examples of Details Window Views; moved lines are indicated via grey text and arrows, mutated lines using green text, deleted lines using red text and added lines using blue text



**Figure 30: The Change browser in Details display mode (the right-hand side panel is colored pink due to the Changes having been 'selected' by being surrounded with brackets in the input window)**

The user can also print Change details for all Changes (as seen in the 'Details' view) to a PDF document (see Appendix Section 9.7.4 for an example), providing a portable history of all modifications occurring in the Project History. These documents were found to be extremely useful in practice for analysing student programming processes, either for use alongside the SCORE Analyser tool during analysis, or for 'offline' analysis of Project Histories.

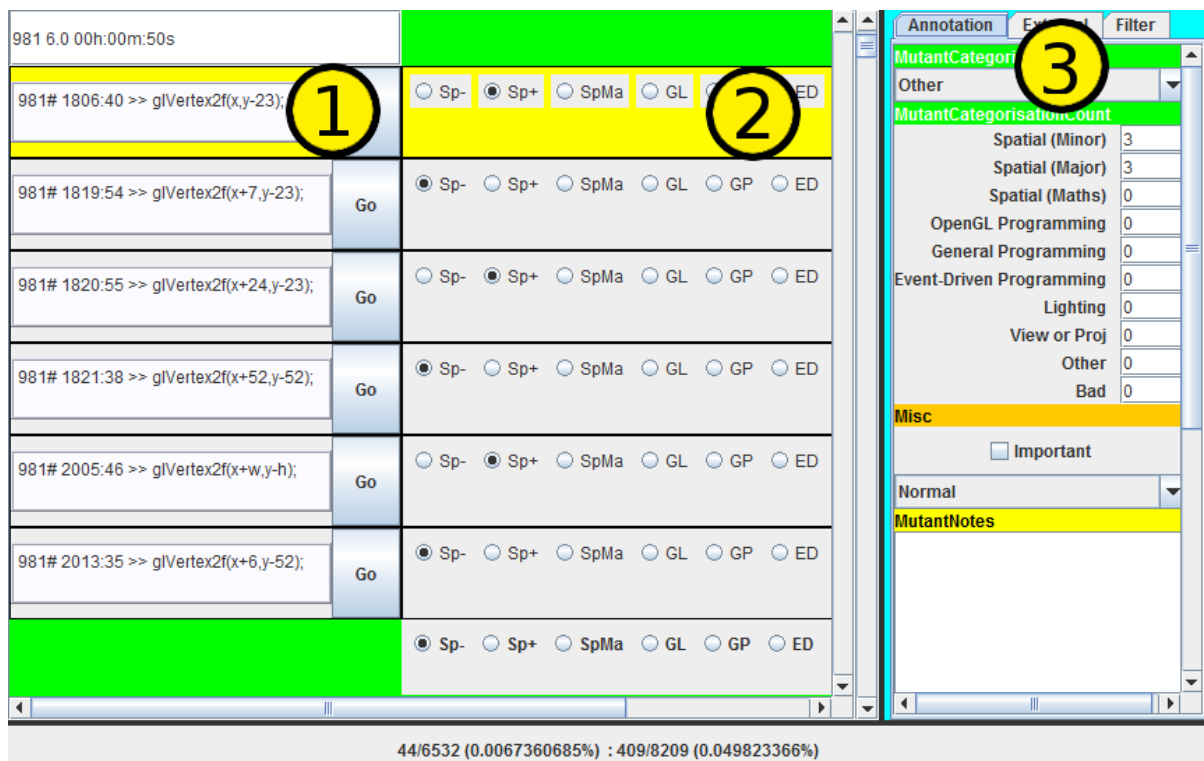
The 'Change Browser' is described in more detail in the SCORE Manual, Section 9.5.3.8. A video demonstration of the 'Change Browser' is available at <http://www.youtube.com/watch?v=O2InvGxaFqg&feature=plcp> and in the electronic appendix.

### 6.2.2.2 Line History View

The Line History View allows the user to view all of a line's modifications occurring throughout the Project History as calculated by the Line History Generation algorithm described in Section 6.2.1.2.

The Line History View is shown in Figure 31. The user can browse modifications which are presented as a list in order of occurrence as seen in Figure 32. For each line state, the line's text is presented along with the Change in which the modification occurs and the line number at which the line is located in that Change. Yellow parts of the line text are those modified since the last Change as

detected via an algorithm which calculates the Longest Common Subsequence. The View also shows whether the Change belonging to that line version compiled or not (the box is red for non-compiled Changes, green for compiled Changes). When the user presses the 'G' button next to each line modification will make the Diff Viewer display the Change associated with that modification. Some additional information including the time taken for each modification is displayed in the text field above the list of modifications. The user can also write all Line Histories to a text file.



**Figure 31: The Line History View with the Line History's states (1) the categorisation panel (2) and the summary and additional controls panel (3)**

1949 10 8.666667(0.33333334, 1.0, 1.0, 1	Write Occ	Write All
1949# 556:39 >> return "("+x+","+y")";	G	X
1949# 558:39 >> return "("+x+","+y+";	G	X
1949# 559:39 >> return "("+x+","+y+";	G	X
1949# 561:39 >> return "("+x+","+y+";	G	X
1949# 562:40 >> s="("+x+","+y+";	G	X
1949# 563:40 >> s="("+str(x)+","+y+";	G	X
1949# 564:40 >> s="("+itoa(x)+","+y+";	G	X
1949# 565:40 >> s="("+itoa(x)+","+itoa(y)+";	G	X
1949# 567:41 >> s="('"+itoa(x)+'+'+itoa(y)+'+'+";	G	X
1949# 569:-1 >> DELETED (s="('"+itoa(x)+'+'+itoa(y)+'+'+";	G	X

**Figure 32: Line History in detail**

The user can reorder or filter the Line Histories displayed using the panel shown in Figure 33. Line Histories can be filtered by size to show only the Line Histories with the largest number of modifications. They can also be filtered by time, showing only the Line Histories whose modifications add up to the longest total time. Finally they can be filtered using a regular expression. Only Line Histories containing at least one modification matching the regular expression are shown. The latter option is especially useful as it allows the user to identify Line Histories involving function calls the researcher is interested in. The View also allows the user to reorder the presentation of Line Histories, ordering them by size, time taken for all modifications or the Line History's position in the Project History.

<input type="checkbox"/> Filter Compile Status
.*glVertex.*
Filter RegExp
0.1
Filter Largest
Filter Annotation
Desc. Size
Asc. Size
Desc. Adj Size
Asc. Adj Size
Desc. Pos
Asc. Pos
Desc. Time
Asc. Time
00:05:00-00:00:00
Boundary
00:05:00-00:00:00
Replace

**Figure 33: Mutant Filter with some items removed for clarity**

The 'Line History View' is described in more detail in the SCORE Manual, Section 9.5.3.7. A video demonstration of the 'Line History View' is available at <http://www.youtube.com/watch?v=izoU03k5hIU&feature=plcp> and in the electronic appendix.

### 6.2.3 Performing the Segment-Coding Process using the SCORE Analyser

To conduct detailed analysis of an assignment, the user begins by analysing the modifications occurring in Changes by use of the Change Browser (Section 6.2.2.1). This provides the user a concise view of modifications and allows her to gain an understanding of most of the student's programming actions. During this analysis, the user takes notes in the Change Browser's integrated note field, describing observed student actions and memoing insights into student problems.

When the purpose behind a sequence of Changes seems unclear the user either views the Modification Summary View produced by the Change Browser or navigates the individual Changes using the Diff View (Section 4.4.1) to analyse the modifications in the full source code context.

Some sequences of Changes involve a small set of lines which are frequently modified. The user can utilise the Line History View (Section 6.2.2.2) which allows her to compare all the modifications applied to the line. The researcher can then examine some of the Changes in which the line is modified using the Diff View. This enables the researcher to see whether the context in which the line occurs changes during the line's lifetime.

When the user is unsure as to the purpose or effect of a modification, she can execute the project at that and at the previous Change to observe differences in the program. If the intention of a



modification is still not apparent, the user can search for the Change in which the problem is resolved and then compare the two Changes via the *Arbitrary Diff View* (end of Section 4.4.1), narrowing in on the differences between the correctly working and the error versions. The user can then utilise the *Compile Editor* (Section 4.4.3.1) to debug the version containing the error until the nature of the error becomes clear. This allows the user to understand the intention behind the modifications leading up to the error's correction.

Using this approach the user can build up a qualitative understanding of the assignment. The user gains a detailed view of the Project History on a source code level. Through this process, she gains insights into student problems, which are unavailable through abstract metrics or output approaches to analysis described in the literature review. To formalise the results of this analysis process, the user can then segment the Project History and codes Segments as will be described in the next two sections.

## 6.3 Segment-Coding Classification Method

### 6.3.1 Application of the Change-Coding Classification Scheme to Segment-Coding

As was discussed in Section 5.2, the Change-Coding method which was based on a method for the analysis of syntax errors (Jadud, 2006a) was found to be lacking in analytical power when applied to Project Histories.

To resolve these problems, a new analysis method called Segment-Coding was developed. This new method is similar to the approach taken in a project investigating the debugging methods and problems of professional software developers (Eisenstadt, 1993). Instead of analysing individual Changes or modifications in a Project History in isolation, anecdotes relating to programmers' debugging of software were analysed. The focus was on the programming process, with each such anecdote spanning large periods of a programmer's development work. These anecdotes were then categorised along several dimensions describing the nature of the underlying problem and how the developer eventually solved the problem.

Based on that approach a method to store not only data relating to individual Changes but also to encapsulate the context in which those Changes occur was developed. In this method sets of related Changes are grouped in *Segments*. These Segments fulfil a role similar to anecdotes in a study by Eisenstadt (1993) since a Segment's Changes contain both the problem and the eventual solution to the problem. In contrast to the use of anecdotes, analysis of Segments is rooted in the actual source

code actions rather than in anecdotes recalled long after the event and lacking the detail of the actual source code modifications.

The produced Segments are then coded along a single dimension identifying the type of task being carried out in the Segment. This mirrors the Action category of Change classification. The set of categories used in classification is the same as the coding classification scheme described in Section 5.2.2. Changes in the same Segment by definition share the same Problem category. The Error category used in Change-Coding is not explicitly replicated. Instead, errors are qualitatively discussed in the textual description associated with a Segment. A Segment's size gives indication as to the difficulty of the problem being solved, which allows differentiation between frequent but easy-to-solve problems (e.g. syntax errors) and less frequent but more challenging problems (e.g. errors occurring during the implementation of an animation) which was not possible with the individual-Change classification scheme.

The number of Segments is far smaller than the number of individual Changes, allowing a much more comprehensive description of each Segment, describing in detail the problem and student solution approaches involved. The smaller number of Segments also makes recoding much less time-intensive if changes to the classification scheme occur. Segments also make it easy to access the context in which a Change occurs since this context consists of the other Changes in the Segment.

Segments can also be parented to other Segments, creating Segment hierarchies which can provide additional context. For example, a Segment involving the avatar lifting a lamp may be parented to the same Segment as the Segment in which the lamp's upward movement is implemented.

In summary, Segments contain the following task/problem related information:

- Segment Problem Type: What task is student working on during Segment?
- Segment Change Set: What other Changes are related to the Change because they are related to the solving of the same problem?
- Segment Hierarchy: What other assignment problems (Segments) is the problem related to?
- Segment Size: How many Changes did the problem/task take to resolve (Indicates difficulty of the problem/task)?

The Segment-Coding approach here was utilised to classify the Segments presented in Section 6.5 and analysed in Sections 6.6 and 6.7.

The next section will describe rules used in the Segmenting of Project Histories.

## 6.3.2 Approach to Categorisation and Segmenting

### 6.3.2.1 Related Changes that involve different classification Categories

If a Segment contains related Changes which are part of different classification categories, split up the Segment into separate Segments each containing only Changes of the same classification category.

If syntax errors occur while a student is working on a problem, Changes fixing these syntax errors should be excluded from the Segment unless the whole Segment involves the fixing of a syntax error. If the number of Changes involved in fixing the syntax error is large enough a separate Segment containing those Changes should be formed.

#### **Examples:**

Example 1)

Situation: A related set of Changes includes Changes involving the implementation of a data structure to hold animation data as well as an animation algorithm implementation.

Approach: Two separate Segments should be formed for these Changes. The Segment relating to the creation of the data structure should be classified as '*General Programming*', whereas the Segment relating to the animation algorithm should be classified as '*Spatial*'.

Example 2)

Situation: A related set of Changes implementing a data structure includes several simple syntax errors resulting from the misspelling of variables and functions. Each of these errors is fixed in less than ten Changes.

Approach: All Changes involved in fixing the simple syntax errors should be excluded from the Segment.

Example 3)

Situation: A related set of Changes implementing a data structure includes a syntax error in which the student is trying to add an array to the data structure. The student requires 15 Changes to produce the correct two-dimensional array syntax.

Approach: Two separate Segments should be formed for the Changes implementing the data structure and the Changes involved in adding the array to the data structure. Both should be classified as '*General Programming*' Segments.

### 6.3.2.2 Segments containing separable units of work

If a Segment consists of different, separable units of work then break it up and put those Segments under same parent Segment.

#### **Examples:**

Example 1)

Situation: When implementing functionality to allow the user to pick up a lamp with the avatar, the student implements both a pickup animation and adds code which 'lifts' the lamp by translating it upward along the y-axis.

Approach: Changes implementing the animation should be put into a separate Segment to those implementing the lamp movement. Both these Segments should then be parented to the same 'Avatar Lamp Pick-up' Segment.

### 6.3.3 Approach to the Analysis of Segments

#### 6.3.3.1 Description of Segments

A Segment's textual description was used to describe all of the Segment's major development milestones as well as the student's problem-solving approaches. It was also used to note whether they were successful or not. The description was also used to describe any remaining uncertainties relating to any Changes, such as questions regarding why a student performed a certain modification.

Segment descriptions (also called Segment reports) are located in the appendix, Section 9.7.6. In practice, Segments dealing with straightforward problems were not described in more detail since the categorisation offered sufficient description for such Segments. Also, in some cases part of the description of Segments is contained in the notes associated with Changes belonging to that Segment. Change notes, which in some cases contain additional Segment description, are not contained in the thesis appendix. They can however be found in the electronic appendix and can be accessed using the SCORE Analyser which is also part of the electronic appendix.

#### **Examples**

Example 1) (An actual unedited Segment description from John A1, Segment 'Rotation around Parent'. The grammar is somewhat informal as the text is unchanged from the original memo):

“Working on rotation around parent. Again gets calculating centre wrong by using `upper.x-lower.x/2`, and uses bad formula for calculating angle which does not allow for 360 degree rotation.

Tries casting to an integer and using debug messages to print out variable values. Fixes centre calculation (but uses parent centre instead of child centre), but is moving centre instead of lower-left-corner, so moving too far (hence moving the child away from the parent when rotated).

After fixing that problem, runs into problems because the integer position of objects loses decimal point values, leading objects to get closer to parent as they're rotated (very common problem amongst students). Eventually solves the problem by storing parent-child distances instead of using changing child centres (fixed 684).

686-714 Tries to find a formula for rotating smoothly around 360 degrees, again gets distance formula wrong ( $(lx - sx/2)$ ), instead of  $(lx - (lx - sx)/2)$ . Finds a good formula, but still rotates the child's lower-left corner around the parent rather than its centre, so the rotation isn't entirely correct and remains that way.

Note that students aren't really drawing out points of rotation etc much... that would help... a more constrained environment with such functions, or teaching them how to do it.”

### 6.3.3.2 Discovering the purpose of a Change belonging to a Segment

When producing a Segment description it was helpful to look at significant Changes in the Segment. Such a Change could be the Segment's final Change, or it could be a Change fixing a particular problem. That Change represents a correct implementation and preceding Changes could be compared to it to describe whether they are steps towards or away from the correct implementation.

On a line level, this was accomplished by analysing a line's Line History, in particular the final modification that is still part of the Segment under investigation. This helped determine whether a modification of a particular line was correct or not. However, it is important to verify whether the context the line is in changes in the meanwhile. A change in the context may modify the meaning of subsequent modifications.

#### Examples:

Example 1)

One example can be found in Ida's implementation of a Star Jump animation in Assignment 3. Figure 34 shows the line history for the addition of an upper-arm rotation to the star jump animation. Figure 35 shows screen captured for the associated Changes. Given only the screen capture, it is impossible to determine whether the rotation added at (456) is correct, since perhaps the student intends to have the avatar raise its arms for the jump animation. However, when examining the line's history one can spot that two more axis modifications occur in the history, after which all the remaining modifications are only tweaks to the amount of rotation. Examining the associated screen captures, it becomes clear that the third modification to an x-rotation at (459) is what the student intended, meaning that the modifications at (457-458) were both errors.

(1433): (456-460, 462, 471 : total = 7)	
456, ADDED(O):	$\text{rotation}[\text{lu\_arm}][\text{yc}] = 90 + \text{rotation}[\text{ru\_leg}][\text{xc}];$
457, MUTATED(O):	$\text{rotation}[\text{lu\_arm}][\text{zc}] = 90 + \text{rotation}[\text{ru\_leg}][\text{xc}];$
458, MUTATED(O):	$\text{rotation}[\text{lu\_arm}][\text{xc}] = 90 + \text{rotation}[\text{ru\_leg}][\text{xc}];$
459, MUTATED(O):	$\text{rotation}[\text{lu\_arm}][\text{xc}] = -90 - \text{rotation}[\text{ru\_leg}][\text{xc}];$
460, MUTATED(O):	$\text{rotation}[\text{lu\_arm}][\text{xc}] = -85 - 2 * \text{rotation}[\text{ru\_leg}][\text{xc}];$
462, MUTATED(X):	$\text{rotation}[\text{lu\_arm}][\text{xc}] = -85 - 1.8 * \text{rotation}[\text{ru\_leg}][\text{xc}];$
471, MOVED(X):	MOVED ( $\text{rotation}[\text{lu\_arm}][\text{xc}] = -85 - 1.8 * \text{rotation}[\text{ru\_leg}][\text{xc}];$ )

Figure 34: Line History implementing an upper arm transform for the star jump animation

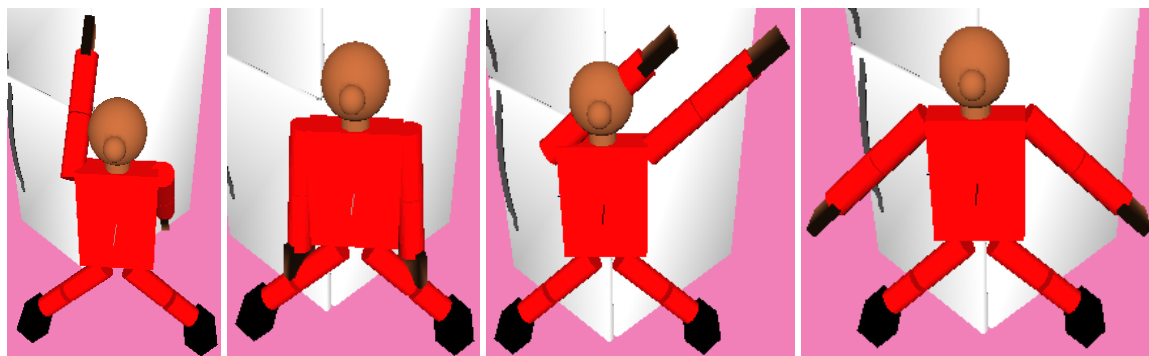


Figure 35: Implementation of the Star Jump animation at Changes 456-459

#### 6.3.4 Sub-categorisation of Segment classification categories

As discussed in Section 6.3.1, the classification scheme developed for use in Change-Coding was also used in the categorisation of Segments. During the in-depth analysis of Segments (described in Section 6.6) more detailed descriptions of underlying problems were developed and these more detailed descriptions allowed for the additional differentiation of problems belonging to a classification category. For example, problems involving object-oriented programming and problems involving pointer syntax would both be classified as 'General Programming' tasks, but the

breakdown into these more precise categories makes the classification provide more insight into the nature of the underlying problem.

Since many such sub-categories have relatively few items in them, classification on that level would not yield enough Segments per category for quantitative analysis. Also, despite belonging to different sub-categories the most effective way of qualitatively examining problems belonging to the same classification category (such as 'C++ Object-Oriented Programming' and 'C++ Pointer' problems belonging to 'General Programming') is to examine them together. This provides a more manageable and understandable break-down and also allows for exploration of what features of a certain classification category's items make for especially challenging problems and how such features are different to features found in the same category that do not.

In order to benefit from both a more general classification and a more detailed break-down, Segments were initially classified as belonging to one of the more abstract classification categories discussed in Section 5.2.2. They were then further deconstructed into the specific classification categories pertaining to the precise nature of the problem/error listed next. In GT terms, the classification categories serve as categories, whereas the sub-categories serve as codes. These are outlined below.

#### **Spatial:**

##### **1. Two-Dimensional Coordinate Drawing / Primitive Creation (2D)**

Segments involving the implementation of an icon or object as an OpenGL graphics primitive (usually using glBegin/glEnd) in two dimensions or the use of 2D coordinates to position an object.

##### **2. Two-Dimensional Coordinate Manipulation (2D)**

Segments involving the dynamic manipulation of 2D coordinates. Segments usually involve the implementation of object move or rotate functionality not using OpenGL transformations in Assignment 1.

##### **3. Three-Dimensional Coordinate Drawing / Primitive Creation (3D)**

Segments involving the implementation of an icon or object as an OpenGL graphics primitive (usually using glBegin/glEnd) in three dimensions or the use of 3D coordinates to position an object.

##### **4. Order of Transformations**

Segments involving students learning how to develop compound transformations which can produce hierarchical models. These Segments occur in Assignment 3 during the avatar assembly.

## 5. Three-dimensional Transformations (3D)

Segments primarily involving the modification of three-dimensional transformations to position objects. Implementation of animations falls into this category. It is different to the *Order of Transformation* category in that the implementation work shows the student already understands how to composit / order transformations.

## 6. Mathematical (Formulas, etc)

Student problems caused by mathematical errors, including the use of incorrect formulas. Usually seen in Assignment 1 when implementing object rotation and performing trigonometry to calculate angles.

## 7. Tweaking

Tweaking describes actions which manipulate spatial coordinates or transformations by small amounts and do not change the direction of the coordinate or transformation. While few Segments will be comprised entirely of such Changes (and hence few Segments will be classified as 'Tweaking') many 'Spatial' Segments falling into one of the other sub-categories will contain some 'Tweaking' Changes.

### **General Programming:**

#### 1. Loss of Precision

Student problems caused by a loss of precision for stored coordinate values, angle values or dimension values. Usually due to incorrectly storing some or all values in INT format.

#### 2. Syntax / Semantics

Segments relating to programming syntax or semantics. Problems in this category are dependent on the programming language used, so in this research project, they relate specifically to C++ syntax/semantics. However, many of the problems apply to other programming languages with similar syntax. Several different cases are listed below:

##### 2.1. Switch statement

Student problems caused by the incorrect use of the switch statement, often being caused by students not being aware that once a condition evaluates as true further cases will be executed until a break statement is found.

##### 2.2. C++ Function Pointers

Student problems related to passing function pointers to functions. Usually occurs when students attempt to pass pointers to their display function to the GLUT function responsible for binding it



### 2.3. C++ String Concatenation

Student problems concerning C++ string concatenation syntax, such as problems concatenating numbers with strings.

### 2.4. C++ Data Type Syntax

Student problems to do with the C++ data type syntax for arrays (including problems with array access or initialisation), queues, stacks or other primitive or complex standard C++ data types.

### 2.5. Mixing up assignment and identity (= instead of ==)

Student problems caused by incorrect use of the assignment operator where the equals operator was meant to be used.

## 3. C++ Pointers

Segments involving the use of C++ pointers. Includes two types of problems:

### 3.1. C++ Pointer syntax

Student problems caused by a poor understanding of C++ pointer syntax or syntax for accessing non-pointer member variables, and confusion between the two.

### 3.2. C++ Pass-by-value / Pass-by-reference

Student problems caused by the C++ pass-by-value / Pass-by-reference mechanism, such as passing an object by value to a function but expecting changes to be applied to the object being provided to the function (when in fact changes are made to a copy of the object).

## 4. C++ Memory

Problems related to memory management in C++.

### 4.1. C++ Memory Management

Student problems caused by C++ memory errors, resulting in crashes or unpredictable behaviour.

### 4.2. C++ silent function crash

Student problems to do with the fact that C++ will sometimes silently exit a function when suffering a memory fault instead of crashing without completing the execution of that function.

## 5. C++ Object-Oriented Programming

Student problems related to implementing object-oriented classes or methods, such as improperly overriding a parent method without the 'virtual' keyword.

6. Computer Science Concepts

Student problems in correctly implementing or applying standard computer science algorithms such as algorithms for traversing graphs or trees.

7. Inefficient Data Structure or Algorithm

Segments involving problems relating to an inefficient algorithm which produces poor performance (e.g. it slows the execution of the program significantly) or the implementation of an algorithm which is inefficient in terms of development time required to produce it, such as the repeated copying of a code block when a simple loop would suffice.

8. Oversight

Segments involving the addressing of problems relating to an oversight.

9. Misleading Error Message

Segments involving a problem which produces a misleading error message.

10. WaveFrontImporter Usage

Segments involving students learning how to use the WaveFrontImporter library package (developed by the lead researcher) which was used to import models in Assignment 3.

**Pipeline / Program Flow:**

1. Pipeline order

Student problems due to the incorrect order of OpenGL calls (excluding transformations), usually resulting in parts of the drawn scene being overdrawn, the user interface having lighting applied to it, or commands not having any effect (for example, when being drawn after the final glFlush call in the display function).

2. Logical Operator Pipeline

Student problems related to correctly using logical operators and correctly achieving effects such as the 'rubber rectangle'<sup>37</sup> effect.

3. Drawing instead of storing object state / Drawing outside display

Student problems caused by not storing the state of an object (an object's new position, or its status as selected/not selected, for example) and instead directly drawing the object with its new state, usually in the init function or an input handler.

4. Forgetting to call display / glutPostRedisplay

Student problems caused by forgetting to ensure the scene is re-rendered after attributes of an

---

<sup>37</sup> Rubber Rectangle: The use of the logical operation XOR to draw and erase a shape on screen without re-rendering the screen.

object are changed. Usually occurs when students forget to add `glutPostRedisplay()` calls to their mouse or keyboard handling functions.

### **Event-Driven:**

#### **1. Screen-Window Conversion**

Student problems involving the failure to convert from GLUT screen coordinates as used in handlers (y-values starting at the bottom of the screen) to GL coordinates (y-values starting at the top of the screen). This problem type falls into both the 'Event-Driven' and the 'OpenGL' categories, as the inconsistency is caused by the differences between screen and window coordinates in OpenGL and GLUT.

#### **2. Event-Driven Program flow**

Student problems associated with a failure to understand the program flow and state of variables in their event-driven code, leading to incorrect actions being executed.

#### **3. Hit Code Error**

A Segment involving an error in 'hit code', a conditional which is to calculate when a coordinate falls into the bounds of an object.

### **OpenGL:**

#### **1. Colour Tweaking**

This category involves experimentation with different colour values, which becomes problematic when too many changes are spent choosing the correct colour value.

#### **2. Mixing up handlers (special with keyboard, etc)**

Student problems arising from the use of incorrect mouse or keyboard handlers, for instance use of the standard keyboard handler to attempt to capture arrow keys (which require the special keyboard handler) or use of the passive mouse handler to capture mouse events when the mouse is 'dragged' (a button is pressed while it is being moved) which requires the dragged mouse handler.

#### **3. GL Fonts**

Student problems or experiments in choosing a font from the GLUT font set, or attempting to use incorrect macros to access non-existent fonts, as well as any mathematical calculations involved in correctly positioning a font.

#### **4. Line Stipple / Fill patterns**

Student problems to do with creating working stipple or fill patterns.

## 5. OpenGL Syntax

Any Segments relating to student problems with OpenGL syntax; for example, a student attempting to use `glVertex3` instead of `glVertex3i/f/d`.

### **Animation Algorithm (Assignment 3 Only):**

#### 1. Animation Algorithm

Student problems to do with the implementation of an incorrect animation algorithm, such as using loops without displaying animation frames between or not using any timing mechanism, which leads to the animation running too fast or not being displayed at all.

### **View (Assignment 3 Only):**

#### 1. Setting up Camera

Student problems and work in setting up views, either through `gluLookAt` or manual transformations.

### **Lighting (Assignment 3 Only):**

#### 1. Lighting

Student problems with lighting, including not enabling any lights, positioning the lights wrongly or too far away or having too large an attenuation factor.

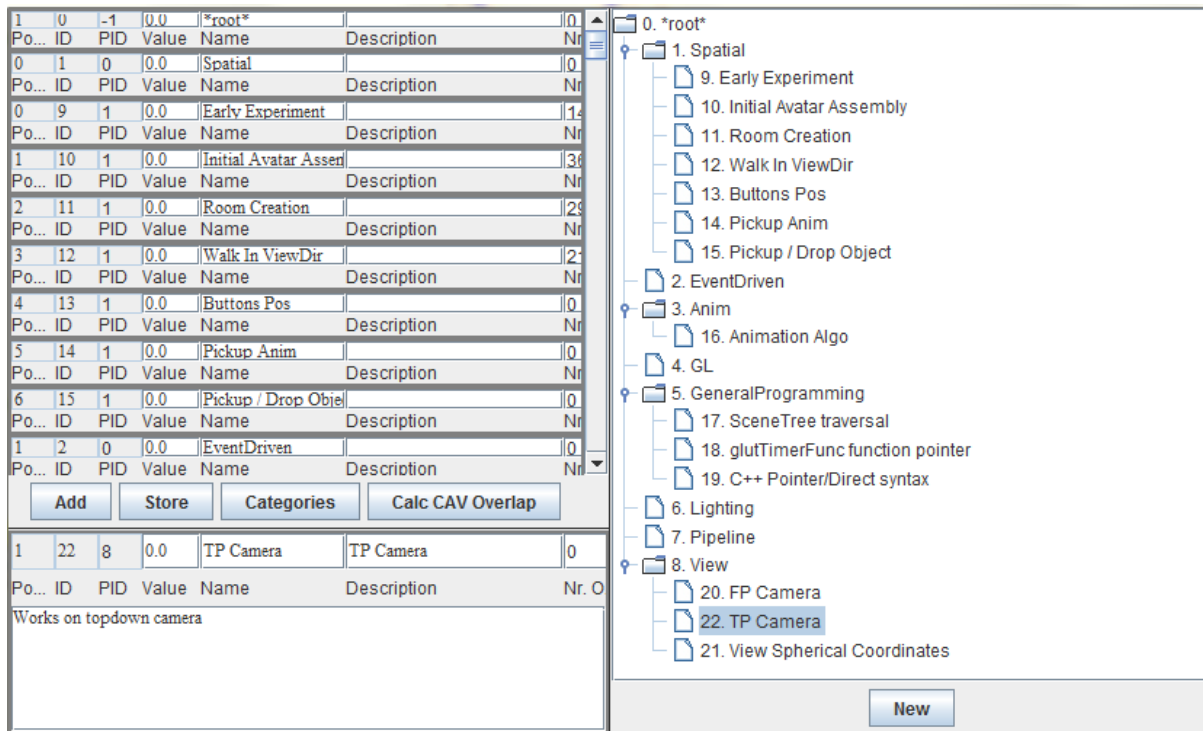
## **6.4 Segment-Coding with SCORE**

### **6.4.1 SCORE Analyser Segment-Coding Facilities**

The previous sections laid out rules for performing the Segment-Coding of Project Histories. This section will showcase the SCORE Analyser Views and functionality implemented to enable the user to apply the Segment-Coding method to Project Histories, as well as functionality implemented to enable the user to produce Segment-Coding-related data for quantitative analysis of Segment features.

#### **6.4.1.1 Manual Segment-Coding View**

The Segmenting-Coding View allows the user to create Segments consisting of sets of Changes to describe the student programming process captured in the Project History. The Segment-Coding View is shown in Figure 36.



**Figure 36: The Segment-Coding View, showing the database summary (top-left), the Segment summary (right) and the currently selected Segment's details (bottom-left)**

The lower-left panel shows a Segment's details. A Segment is described by a unique ID, a name, a short description, a long comment, a set of Changes that make up the Segment and an integer problem difficulty value that stores the researcher's judgement on how difficult the underlying problem was to resolve. The upper-left panel shows a summary list of all Segments.

When the user adds a Segment it appears in both the upper-left list and in the tree view located on the right. The tree view enables the user to add Segments to other Segments as 'Sub-segments'. The Segment hierarchies produced in this way can be of an arbitrary depth. This functionality was used to assign individual Segments to categories such as 'Spatial' or 'Lighting' by making them children of top-level category Segment nodes. In addition to using the hierarchical structure for categorisation of Segments, the user can utilise it to further break down Segments into Sub-segments. For example, the implementation of a pickup animation may involve movement of the avatar as well as movement of the object to be picked up. These two tasks can then be added to separate Segments, which are parented to the same 'Pickup Animation' Segment. The main limitation with the hierarchical segmenting structure as currently implemented is that sometimes a larger problem involves Sub-segments belonging to different categories. Such relationships cannot be modelled because the Sub-segments need to be parented to different top-level Segments representing the different categories.

The user can also utilise the Segment-Coding View to convert Segments to a human-readable text file, where each Segment is listed according to its position in the hierarchy along with the detailed information associated with it. Segments are stored in a database using the open-source Apache Derby database software.

The Segment-Coding View allows for the creation of multiple Segmentation versions. For example, the user might create a new version every time she makes any considerable modifications to the coding scheme. Such different versions could also involve completely different coding schemes, developed to probe different research objectives.

#### 6.4.1.2 Segment Time/Modification Evaluation Output

The user can also generate time information for stored Segments using the Segment-Coding View. The primary measurement of Segment significance (as discussed in Section 9.7.5) is the number of Changes associated with Segments. This number can be calculated easily. However it may also be useful to evaluate data regarding the time spent on different types of Segments (given by the sum of time spent on the Segment's Changes). As will be discussed in Section 9.7.5, analysis of time spent on Changes is far more complex. Very large gaps will appear between Changes between work sessions. Large gaps between Changes can also occur during a work sessions. These gaps may be due to any number of activities. Some activities may be related to work on the Segment, such as looking up information. Other activities are not, such as chatting to a friend or browsing the internet. The algorithm used to calculate time spent per Segment heuristically circumvents these issues.

One situation in which time between two versions is inaccurate is if the student stops working for a period of time. For example, the student may go to bed and continue working another day. In this case, the time difference between two versions could be hours or even days, but this is not student time spent working on the Change. Such Changes can be identified by the very large time between versions, since a student is unlikely to have tens of hours working on the assignment program without making any changes to the source code. The heuristic algorithm sets the time for such Changes to be equal to a user-defined period of time. In practice, the average time spent by students on a Change was used. Such Changes will be referred to as *Suspend* Changes.

Another situation may involve a smaller interruption of student work, such as the student chatting or going to the bathroom. While the time will be lower than that of a suspension of work, such small interruptions are likely to occur more often. To identify such periods heuristically, the algorithm can be set to cap time values smaller than the *Suspend* boundary but higher than another value which will be called the *Max* boundary. The value to which the time is capped can be defined by the user.

In practice, it was set to ten minutes since such values are significant outliers to the distribution of Change times.

There is one further situation discovered during analysis in which there was an apparent inaccuracy in measuring time between Changes using timestamps. Changes with unreasonably small time values were discovered. These Changes always involve different files being modified. These Changes are most likely due to several files being modified at once. Once a save all/compile action is executed, each file is added to the Project History in turn, generating multiple Changes, which should in fact only be a single Change. The small time values are entirely due to the speed of compilation. If such small Changes were included, they would significantly decrease averages measured for sets of related Changes. The heuristic algorithm can be set to exclude Changes whose time between versions falls below a certain threshold. In practice a value of 5 seconds was chosen. This is close enough to the time it would take for the compiler to compile the program that the student could not have done any useful work in the meanwhile. Such Changes are referred to as *Small Changes*.

The settings which the algorithm uses to determine whether Changes are normal, *Suspend*, *Max* or *Small* are entered using the 'Segment Time Generation settings' panel shown in Figure 37. The time calculated using this heuristic algorithm will not precisely calculate student time spent working on the Change (which is impossible without monitoring students constantly). However, the results should reflect student work patterns well enough to enable quantitative analysis of Segment features.

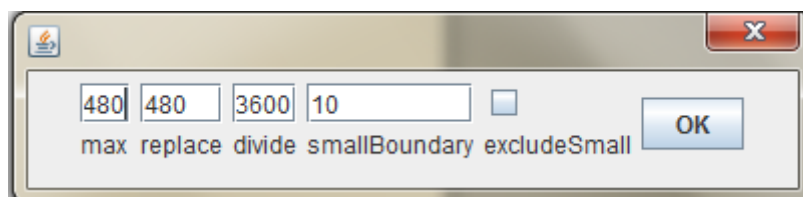


Figure 37: Segment Time Generation settings panel

The algorithm calculates average time and median Change time per Segment. It also calculates the average deviation of Change Time from the Change time median for each Segment, and the total number of modifications of different types (Added, Deleted, Moved, Ghost, Mutated). It presents those data along with data on the size of the Segment, the problem difficulty (as assigned by the researcher through the Segment-Coding View interface) and the category to which the Segment was assigned. The output also includes the number of *Suspend*, *Max* and *Small* Changes contained in the

Segment. The output of the Segment Time/Modification generation algorithm for one assignment is presented in Table 15.

**Table 15: Example of calculated Segment Time and Modification Data**

Seg ID	Name	Category	ProbDiff	Avg	Median	AvgDev	Small
1.1	Door Impl.	"Spatial"	1	126	79	108	0
1.2	Attempts GL transformations	"Spatial"	5	108	43	119	0
1.3	Arrow Icon	"Spatial"	1	119	80	92	0
2.1	Screen-Window convo	"ED"	2	174	139	113	3
2.3	GLUT Menu	"ED"	3	89	82	62	0
3.1	Virtual function and Constructor	"GP"	2	96	67	79	1
3.2	Data type for RGB	"GP"	2	77	49	59	0

Seg ID	Max	Susp	Added	Deleted	Moved	Ghost	Mutated	AllMod	size
1.1	2	0	7	1	0	0	7	70	14
1.2	3	1	5	4	0	0	9	75	15
1.3	1	1	4	1	0	0	9	65	13
2.1	3	2	6	4	0	0	11	75	12
2.3	1	0	11	5	0	0	12	115	23
3.1	0	0	5	1	0	1	7	55	10
3.2	0	0	3	0	0	0	8	50	10

This data can then be used to perform quantitative analysis of Segment features using statistical analysis software such as R. An analysis of features of identified Segments will be presented in Section 6.7.

### 6.4.2 Segment-Coding using the SCORE Analyser

This section will describe how the SCORE Analyser was used to perform Segment-Coding in this research project. The process of manually segmenting a student assignment began with Detailed Analysis as was described in Section 6.2, intended to roughly segment the assignment according to the tasks set out in the assignment specification. The aim during this phase was not to produce accurate Segments. Neither was it to assign individual Changes to Segments. Some areas of student programming will be poorly understood after completion of this step. While analysing Changes, notes were taken via the SCORE Analyser interface regarding key Changes and the overall assignment structure.

Once this rough segmenting was complete, the second phase of Detailed Analysis involved examining each of the rough Segments in turn to develop an understanding of student programming actions occurring in the Segment's Changes. At this point, actual Segments were produced via the Segment-Coding View. While the researcher was still taking notes regarding individual Changes via the main SCORE Analyser interface's Notes view, notes regarding identified Segments are entered



into the Segment's comment section in the Segment-Coding View. In this research project, only Segments larger than 10 Changes were added as 'interesting' Segments, both to cut down workload and because problems involving fewer Changes are unlikely to require any further analysis as they did not pose a substantial problem to the student. At the beginning of a new analysis session the researcher can utilise the SCORE Analyser to read notes and view a Segment's Changes in the Change Browser. This allows the researcher to refresh her understanding of student problems and actions underlying the Segment's Changes. During the second analysis phase, an understanding of all parts of the Project History was developed.

These first two steps can be time-consuming since they require significant manual analysis of individual Changes. Part of this research project involves the proposal of algorithms for Machine Segmenting of Project Histories which will be discussed in detail in Chapter 7. This Machine-Segmenting can augment the first two steps by providing 'candidate segments' of Changes that are likely to be related. These 'candidate segments' are then analysed, merged and/or broken apart to produce real Segments. If machine-generated Segments do a reasonable job of describing underlying problem Segments (as will be demonstrated in Chapter 7) then this will reduce researcher workload for initial Segment discovery significantly.

The researcher then qualitatively analysed the contents of each Segment produced by the previous steps. Each Segment's Changes were examined in detail to uncover the underlying problems and solution strategies involved in the completion of the task underlying the Segment. During this process inaccuracies in the initial Segment-Coding will be uncovered and corrected; some Changes may be found to have been incorrectly assigned to a Segment, for example. Other Segments may upon closer examination be dealing with a different problem than first thought and will be modified and/or assigned to a different classification category to reflect the new, more accurate understanding of student actions. The product of the detailed analysis of Segment contents is presented in Section 6.6.

After the analysis of Segment contents was completed time data was generated for Segments as described in Section 6.4.1.2. This data was then analysed quantitatively. Results for the largely quantitative analysis of Segment features are presented in Section 6.7.

One of the practical limitations of Segment-Coding in the current version of the SCORE Analyser is that Changes can only be added to or removed from Segments via the Manual Segment View's details panel. A future extension to the SCORE Analyser will allow individual Changes to be associated with Segments directly via the main SCORE Analyser interface.

The 'Segment-Coding View' is described in more detail in the SCORE Manual, Section 9.5.3.10. A video demonstration of the 'Segment-Coding View' is available at [http://www.youtube.com/watch?v=C\\_aO5tDxpil&feature=plcp](http://www.youtube.com/watch?v=C_aO5tDxpil&feature=plcp) and in the electronic appendix.

## 6.5 Segmenting Results

This section provides a summary of all Segments identified in the ten Project Histories analysed as part of this research project. They were identified using the SCORE Analyser's Segment-Coding functionality and the Segment-Coding rules and classification scheme described in Section 6.3, 'Segment-Coding Classification Method'.

### 6.5.1 Measuring Segment Significance

The size of Segments and the problem categories in which the largest segments occur provide a rough idea on what areas of programming students encountered most difficulty with during their implementation of the first and third assignments.

To determine whether Segment size in terms of the number of Changes comprising the Segment, total time taken by the Changes in the Segment or average time taken by Changes in the Segment was the best indicator of how much of a problem the Segment had posed to the students, a statistical analysis of these metrics was conducted. The researcher first classified all Segments manually according to the perceived difficulty of the underlying problem based on how often students made serious errors while working on the problem and then performed analysis of variance, correlating this metric to each of the other metrics. The best correlation was found to be with Segment size (null hypothesis  $p=2.23E-15$ ), with total time also providing a good correlation with statistical significance (null hypothesis  $p=5.91E-08$ ), whereas average time was a very poor predictor of problem difficulty (null hypothesis  $p=0.128$ ). Based on these results Segment difficulty is the best metric for measuring the difficulty of a problem underlying a given Segment. The full statistical analysis can be found in Appendix Section 9.7.5.

There is validity to the argument that the manual rating of Segments according to difficulty perceived by the researcher introduces subjective bias. However, it should be noted that these ratings were only used to discover which of a number of metrics seems to best correlate with a Segment's difficulty. Each of the proposed metrics does involve measuring the amount of work carried out on a particular Segment by the student. Hence, the bias could at worst have led to the choice of a metric which is less good than another at accurately measuring student effort. The metric itself is still objective in the sense that it is derived directly from data about the Segment. Future research should eliminate potential researcher bias by using students' evaluation of the difficulty of

Segments. This was not possible as part of this research project due to time constraints. It is also important to note that since this research project is based mainly on qualitative research methods, a degree of subjectivity on the part of both the researcher and research participants is unavoidable. As Glaser and Strauss write, GT “is designed to allow, with discipline, for some of the vagueness and flexibility that aid the creative generation of theory” (Glaser & Strauss, 1967, p. 103). And when seen in a constructivist light as Charmaz does, “a constructivist approach means more than looking at how individuals view their situations. It not only theorizes the interpretive work that research participants do, but also acknowledges that the resulting theory is an interpretation” (Charmaz, 2006, p. 130).

Another valid critique of the rating of Segments according to the number of Changes is that some tasks intrinsically require more source code changes despite not being especially difficult. The detailed qualitative analysis presented in this thesis ameliorates this concern somewhat. This is because contents of Segments are analysed in detail (change-by-change) and this analysis uncovers whether a Segment includes many rote modifications.

In order to better address this issue in future research, two modifications will be considered for future research.

The first modification is a change to the method of Segment analysis. In initial Change-Coding, each Change was coded as correct or erroneous. This method should be introduced into the Segment-Coding method. In addition to existing analysis, each Change within a Segment could be tagged as rote (not requiring problem-solving), productive (moving towards a correct solution) or unproductive (either an error or an action which leaves the solution attempt in the same state as before). A proposal including these ideas is presented in Section 6.8.

The second modification involves measuring Segment difficulty with a metric which takes into account the nature of modifications occurring during the Segment, rather than simply utilising a count of Segment Changes. As an example, such a metric might be the number of modifications per line of source code in the Segment. This would mean that Segments involving many Changes which require little or no modification (rote Changes) would be ranked as relatively easy, whereas those Segments requiring many modifications to the lines of source code produced would be ranked as hard. A proposal for such a metric is provided in Section 6.8.2.

Since assignment tasks were not designed to include an equal number of tasks from each classification category, such a comparison does not in itself show that one particular classification category which is over-represented was more challenging than a different classification category. For

example, there was only a single fairly simple ‘Lighting’ task in Assignment 3 while both assignments included multiple ‘Spatial’ tasks, hence the number of ‘Spatial’ Segments is likely to be much higher than the number of ‘Lighting’ Segments.

On the other hand since any task could be solved in a small number of Changes (even a single Change) by simply writing out the solution source code the size of Segments does indicate the level of challenge of the underlying task. Thus, the occurrence of very large Segments (measured by the number of Changes) can be taken as an indication that students struggled with tasks of that type.

Ultimately analysis at the level of abstraction of only looking at the number of Segments falling into a certain classification category or sub-category provides only limited insight. In addition, it risks missing important internal features of the Segment which cannot be fully represented by a simple classification scheme. For example, a Segment may involve the interplay of different types of tasks or problems that cannot be sufficiently described via its classification category. Such features can only be properly explored under the microscope of in-depth qualitative analysis of the contents of the Segments described in this section. This in-depth analysis will be presented in Section 6.6.

## **6.5.2 Method of Segment Referencing**

### **6.5.2.1 Change Notation**

When identifying an individual Change or a range of Changes, the Change(s) will be presented in round brackets. The number identifying a Change corresponds to the Change’s overall position in the Project History. It can be used to look up that Change using the SCORE Analyser by navigating to the overall position associated with the Change. As an example, (64) identifies the 64<sup>th</sup> Change occurring in a Project History. Sometimes a Change is named explicitly, as in Change 64 referring to the same 64<sup>th</sup> Change. A range of Changes or set of ranges is identified by specifying the set of ranges inside brackets. For example, (10-15, 20-25) refers to the 12 Changes that occupy the overall positions between Change 10 and Change 15 and between Change 20 and Change 25 (inclusive).

### **6.5.2.2 Segment Notation**

To reference Segments (which in turn consist of a set of Changes) a unique string containing the student, assignment, classification category as well as its relative position inside that classification category is used. The Segment’s name and number of Changes involved in the Segment are also listed:

`STUDENTNAME_ASSIGNMENT_NR.CATEGORY_ID.SEG_ID(.SEG_SUBID)* “SEGNAME” [CHANGENR]`

For example, [JOHN\\_A1.ED.5 "SomeSeg" \[10\]](#) identifies the 5th Segment classified as 'Event-driven' in John's Assignment 1 submission. It is called "SomeSeg" and consists of 10 Changes. The CATEGORY is one of the classification categories presented in Section 5.2.2.

Segments can be further broken down by adding sub-positions. For example, [CHRISTOPHER\\_A3.GP.2.1 "First Part" \[16\]](#) is the 1<sup>st</sup> Sub-segment of the 2<sup>nd</sup> Segment classified as 'General Programming' in Christopher's Assignment 3 submission. [CHRISTOPHER\\_A3.GP.2.2 "Second Part" \[21\]](#) is the 2<sup>nd</sup> Sub-segment of that same Segment. In theory, such hierarchies can be of any depth, though only a single level of Sub-segmenting was used in the identification of Segments in this research project.

These Segment identifiers can be used to look up Segment details in the appendix to this thesis. The Segment reports listing details for each Segment are located in Appendix Section 9.7.6. For example looking up [JOHN\\_A3.SP.2."Pickup Teapot anim " \[14\]](#) yields the Segment's details as shown in Figure 38. The details include the Segment's full name, a free-text description (which was used to classify Segments into the sub-categories listed in Section 6.3.4), a list of the Changes which make up the Segment as well as a free-text detailed description (sometimes lengthy) describing the content of the Segment, including solution approaches applied or student misconceptions where applicable.

<p><b><u>JOHN_A3.SP.2."Pickup Teapot anim " [14]</u></b>  Name: Pickup Teapot anim  Description: Spatial  Occurences: 396-407, 439, 611 : total = 14 (+)  Detailed Description:  Adds code to animation modifying teapot position. Spends changes tweaking rate of teapot ascent to match avatar animation, Tweaks teapot y-translation (423-424),</p>
--

**Figure 38: Details of Segment JOHN\_A3.SP.2 as found in the appendix**

### 6.5.3 Identified Segments

The following table (Table 16) presents all Segments identified during analysis of Assignment 1 and Assignment 3. Assignment 1 Segments are preceded by A1, whereas Assignment 3 Segments are preceded by A3. Since Segments are sorted by student (each student's Segments are shown in a separate column) student names were removed from the Segment descriptions in order to reduce the size of the table. Each table row shows all Segments belonging to one of the sub-categories discussed in Section 6.3.4. Row heading colours indicate the overarching category the sub-category belongs to. Sub-categories belonging to the same parent category are grouped together and their header cell is of the same colour (a darker or lighter shade of blue). When the header cell colour changes this indicates that the following rows deal contain sub-categories belonging to a different

parent category. A total of 163 Segments were identified across the ten Project Histories. The average size of identified Segments was 23.64 Changes, with a median of 18 Changes per Segment. Segments of size  $\geq 30$  falling considerably above the median are deemed to be 'large' Segments which are likely dealing with particularly difficult problems.

**Table 16: Table showing all Segments (for all students/assignments) belonging to a category**

\*1) did not attempt related tasks

\*2) attempted but did not complete related tasks

	John	Ida	Christopher	Michael	Thomas
2D Coordinate Drawing	A1.SP.1. [14] A1.SP.2. [15] A1.SP.3. [13] A1.SP.4. [11] A1.SP.5. [11] A1.SP.6. [14]	A1.SP.1. [21] A1.SP.2. [41]  A3.SP.1. [17] A3.SP.2. [16]	A3.SP.3.4. [14]	A1.SP.1. [36] A1.SP.2. [10] A1.SP.3.1. [21] A1.SP.4. [18] A1.SP.5. [26] A1.SP.6. [56] A1.SP.7. [23] A1.SP.8. [18] A1.SP.12. [21] A1.SP.13. [11] A1.SP.14. [21] A1.SP.15. [18] A1.SP.16. [14] A1.SP.17. [11] A1.SP.18. [11]	A1.SP.2. [10] A1.SP.5. [20] A1.SP.6. [13] A1.SP.7. [12]
2D Coordinate Manipulation	A1.SP.7. [56]	A1.SP.3. [23]	*1)	A1.SP.3.1. [21] A1.SP.11. [18] *1)	A1.SP.3. [60] A1.SP.8. [13]
3D Coordinate Draw	A3.SP.3. [12]		A3.SP.3.2. [29]	A3.SP.2. [12]	A3.SP.3. [27]
Order of Transformations	A3.SP.4.1. [26] A3.SP.4.2. [33]	A3.SP.3.1. [24] A3.SP.3.2. [28]	A3.SP.1.1. [36]	A3.SP.1. [47] A3.SP.3. [71]	A3.SP.5.1. [39] A3.SP.5.2. [55]

3D Transformation	A3.SP.1. [12] A3.SP.2. [14] A3.SP.5.1. [19] A3.SP.5.2. [99] A3.SP.5.3. [120]	A3.SP.4.1. [66] A3.SP.4.2.1. [79] A3.SP.4.2.2. [14] A3.SP.4.3. [30]	A3.SP.2.1. [24] A3.SP.3.1. [14] 2*)	*1)	A3.SP.1. [10] A3.SP.6.1. [36] A3.SP.6.2. [21] A3.SP.6.3. [20]
Mathematical	A1.SP.7. [56 ]		A3.SP.3.3. [25] A3.SP.3.5. [47] 1*)	A1.SP.3.2. [26] A1.SP.9. [26] A1.SP.10. [18] 1*)	A1.SP.1. [20] A1.SP.4.1. [19] A1.SP.4.2. [10]  A3.SP.2.1. [12] A3.SP.2.2. [36] A3.SP.4. [36]
Loss of Precision		A1.GP.7.[56]			A1.GP.6. [31] A1.GP.7. [43]
Syntax / Semantics	A1.GP.2. [10]  A3.GP.1. [13]	A1.GP.2. [19] A1.GP.4. [21]  A3.GP.1. [11]	A1.GP.3. [22] A1.GP.4. [47]  A3.GP.1. [13] A3.GP.3. [25]	A1.GP.1. [14] A1.GP.3. [12] A1.GP.4. [11]	A1.GP.1. [10] A1.GP.3. [14]  A3.GP.3. [14]
C++ Pointers			A1.GP.1. [10] A1.GP.2. [12]  A3.GP.4. [12]		A1.GP.2. [114] A1.GP.4. [20]
C++ Memory	A1.GP.3. [20]				
C++ Object-Oriented	A1.GP.1. [11]  A3.GP.3. [23]	A1.GP.5. [17]	A1.GP.5. [31] A1.GP.8. [15]		

General Computer Science Concepts		A1.GP.6. [20]	A3.GP.2. [121]	A1.GP.2. [24] A3.GP.1. [23] A3.GP.2. [28]	A1.GP.5. [54]
Oversight		A1.GP.1. [21] A1.GP.3. [18]	A1.GP.6. [10]		A3.GP.2. [8]
Misleading Error Message			A1.GP.7. [25]		
WaveFrontImpor ter Usage	A3.GP.2. [12]	A3.GP.2. [21]			A3.GP.1. [39]
Pipeline order				A1.PI.3. [10] A1.PI.6. [11] A1.PI.7. [19] A1.PI.8.2. [13]	
Logical Operator Pipeline		A1.ED.3. [37]			
Draw not Store		A1.ED.2. [19]		A1.PI.1. [19] A1.PI.4. [24] A1.PI.5. [45] A1.PI.8.1. [65]	
No Display				A1.PI.2. [10]	
Screen-Window Conversion	A1.ED.1. [15]	A1.ED.1. [11]	A1.ED.1. [9]		A3.ED.2. [21]
Event-Driven Program flow	A1.ED.2. [11] A1.ED.3. [23]	A1.ED.2. [19] A1.ED.3. [37] A1.ED.4. [15] A1.ED.5. [22]	A1.ED.5. [11] A1.ED.6. [12]	A1.ED.1. [45] A1.ED.2. [30] A1.ED.3. [28] A1.ED.4. [10]	A1.ED.1. [20]
Hit Code Error			A1.ED.2. [18] A1.ED.3. [21] A1.ED.4. [11]		A3.ED.1. [10]
Colour Tweaking				A1.GL.2. [14] A1.GL.3. [11] A1.GL.4. [11]	



Mixing up handlers					A3.GL.1. [10]
GL Fonts					
Line Stipple / Fill patterns		A1.GL.1. [31]			
OpenGL Syntax				A1.GL.1. [14]	
Animation Algorithm	A3.ANIMATION.1. [19]	A3.ANIMATION.1. [15]	A3.ANIMATION.1. [18]	A3.ANIMATION.1. [60]	A3.ANIMATION.1. [46]
View / Setting up Camera	A3.VIEW.1. [37] A3.VIEW.2. [21] A3.VIEW.3. [15]	A3.VIEW.1. [17]	A3.VIEW.1. [47] A3.VIEW.2. [15] A3.VIEW.3. [14] *2)	A3.VIEW.1. [35] *2)	A3.VIEW.1. [21] A3.VIEW.2. [20]
Lighting		A3.LIGHTING.1. [26] A3.LIGHTING.2. [21]		A3.LIGHTING.1. [60] A3.LIGHTING.2. [19]	A3.LIGHTING.1. [13]

While Table 16 listed all Segments, the following table (Table 17) shows only the size of the largest Segment. The size is shown as the number of Changes of the largest Segment containing a problem of that sub-category for each student. Cells are marked in red when the size of the Segment is larger or equal to 30 Changes. Such Changes are considered ‘large’ and indicate substantial difficulty faced by the student in solving the underlying problem.

**Table 17: Table showing the size of the largest Segment per category for each student**

\*1) did not attempt related tasks

\*2) attempted but did not complete related tasks

	John	Ida	Christopher	Michael	Thomas
2D Coordinate Drawing	15	41	14	36	20
2D Coordinate Manip	56	23	*1)	21 *2)	60 (6)

3D Coordinate Draw	12		29	12	27
Order of Transformations	33	28	36	71 (7)	55
3D Transformation	120	79 (7)	24 *2)	*1)	36
Mathematical	56		47 1*)	26 *1)	36
Loss of Precision		56			43
Syntax / Semantics	13	21	47	14	14
C++ Pointers			12		114 (14)
C++ Memory	20				
C++ Object-Oriented	23	17	31		
General Computer Science		20	121 (12)	28	54
Oversight		21	10		8
Misleading Error Message			25		
WaveFrontImporter Usage	12	21			39
Pipeline order				19	
Logical Operator Pipeline		37			
Draw not Store		19		65 (6)	
No Display				10	
Screen-Window Conversion	15	11	9		21
Event-Driven Program flow	23	37	12	45	20
Hit Code Error			21		10
Colour Tweaking				14	
Mixing up handlers					10
GL Fonts					
Line Stipple / Fill patterns		31			

OpenGL Syntax				14	
Animation Algorithm	19	15	18	60 (6)	46
Setting up Camera	37	17	47 *2)	35 *2)	21
Lighting		26		60 (6)	13

The remainder of this section will summarise and discuss the distribution and size of Segments as presented in Table 19 and

Table 20 without discussing Segment internals in detail. A detailed qualitative discussion of Segment contents will be given in the next Section 6.6.

It should be noted once again that only Segments larger than or equal to 10 Changes were analysed<sup>38</sup>. Some problems caused many errors that were solved in less than 10 Changes; for example, many students encountered OpenGL syntax problems that were solved in one or two Changes. Such small problems are excluded as they are not deemed significant.

Results presented in this section should be seen in the light of the way tasks were laid out in the assignment specification. It is possible that in a different context or given different scaffolding, problems which would fall into the same category would pose a significantly greater or lesser problem to students. This means the results are not necessarily generalizable. The analysis of Segment contents presented in Section 6.6 delves into student problems in detail. This type of analysis takes into account the structure of tasks. That means it produces results which are more generally applicable.

### 6.5.3.1 Spatial

Most of the '*2D Coordinate Drawing*' category Segments involve the production of 2D icons and occur in A1. Almost all of these Segments are small (> 20 Changes). The two large exceptions are [IDA\\_A1.SP.2."Rotate Icon" \[41\]](#) and [MICHAEL\\_A1.SP.6."Clipping" \[56\]](#) but these are only 3 of the total 30 (3/30) Segments. Overall the problems categorised as '*2D Coordinate Drawing*' did not appear to have been a major issue for the students analysed.

The '*2D Coordinate Manipulation*' category involves parent-child rotation. Some Segments from the '*Loss of Precision*' category are also interwoven with these Segments and hence will be included in the discussion. The large Segments (> 30 Changes) are [JOHN\\_A1.SP.7."Rotation around Parent" \[56\]](#), [THOMAS\\_A1.SP.3."Wall Move/Resize" \[60\]](#), [THOMAS\\_A1.GP.6."Loss of Precision" \[31\]](#) and [THOMAS\\_A1.GP.7."Unintentional Rounding" \[43\]](#), [IDA\\_A1.GP.7."Child object rotate"\[56\]](#). All students who attempted the task produced large or even very large Segments (>50 Changes). Two students did not attempt the task at all. Problems that fall into the '*2D Coordinate Manipulation*' category were apparently very hard to solve.

---

<sup>38</sup> There are one or two legacy exceptions to this rule; these are segments which once included more than ten Changes but were later reclassified to include less than ten Changes; removing them would have been troublesome, hence these segments are still included.

Most of the '*3D Coordinate Draw*' category problems involve the production of rooms consisting of walls, a ceiling and a floor in Assignment 3. Only four such Segments exist, and they are all < 30 Changes, suggesting that problems in this category were not a major issue for students.

The avatar assembly task falls into the '*Order of Transformations*' category; 4/5 students produced large Segments (> 30 Changes) and two students did not successfully complete the task. This category of problem was apparently difficult for students to resolve.

In the context of the third assignment the '*3D Transformation*' category applies to the development of avatar animations. Two students did not complete this task successfully and hence did not produce large Segments. All other students produced large Segments (>30 Changes). Ida and John both have very large Segments, e.g. [IDA\\_A3.SP.4.2.1."Pickup Animation" \[79\]](#) and [JOHN\\_A3.SP.5.3."Swim Anim" \[120\]](#). Both of these are among the largest Segments found in Project Histories. The *3D transformation* category of problems was apparently very hard to resolve.

In the mathematical category, 3 students had large Segments involving >30 Changes; one student had no Segments at all, though this is in part due to the Segment [IDA\\_A1.GP.7."Child object rotate"\[56\]](#) being coded as 'General Programming', since that Segment also involved an underlying mathematical problem and could also have been classified as mathematical. One student who had no large Segments did not attempt the parent-child rotation task which is the major mathematical task required in assignments. Problems in the mathematical category were apparently hard to very hard to resolve.

### 6.5.3.2 General Programming

The '*Loss of Precision*' sub-category's Segments are all part of the object-rotation task, with the loss of rotation problem only occurring in that context. Two students have large Segments (>30 Changes) with one student's Segment being very large, exceeding 50 Changes. Two students did not attempt the associated task, meaning only one student who worked on the task did not encounter a significant problem. Segments in this category were apparently hard to resolve.

The '*Syntax/Semantics*' category involves only 1/15 Segments with a large size (>30 Changes) and only 4/15 Segments of a moderate size (>20 Changes). The only large Segment is [CHRISTOPHER\\_A1.GP.4."Macro" \[47\]](#) which involves macro syntax. It appears to have been a significant issue for Christopher only and did not present a major issue for the other students.

Three students produce no problem Segments which fall into the '*C++ Pointer*' category. Christopher's '*C++ Pointers*' Segments are all small (<20 Changes). Thomas produced a very large

Segment [THOMAS\\_A1.GP.2."Child-Parenting Algorithm" \[114\]](#), by far the largest Segment in that Project History. The '*C++ Pointers*' category apparently did not produce issues for most students given the assignment set-up, but ended up being a very hard issue to resolve for one student.

Only one student (John) attempted to free memory; he produced one small Segment involving a '*C++ Memory*' problem in which he unsuccessfully attempted to free memory before giving up. This category was not an issue in this assignment, but given that no student successfully worked on freeing memory it might have been if freeing memory had been an assignment task.

The category '*C++ Object Oriented Programming*', which includes problems relating to C++ object-oriented syntax and semantics, produces only one large and one moderate Segment, and only five Segments in total, suggesting it was not a major issue.

The '*General Computer Science*' category includes Segments to do with the implementation of Computer Science algorithms. John has no Segments that fall into this category. Ida and Michael have some moderate-sized (< 20 Changes) Segments. Thomas has a very large Segment totalling 54 Changes ([THOMAS\\_A1.GP.5."Circular Parent-Child" \[54\]](#)) whereas Christopher produced a very large Segment ([CHRISTOPHER\\_A3.GP.2."SceneTree traversal" \[121\]](#)) totalling 121 Changes relating to the implementation of a graph algorithm. Problems relating to the '*General Computer Science*' category appear not to have been an issue for three students, while being a hard issue for Thomas in one instance and a very hard issue for Christopher.

Of the four Segments that fall into the '*Oversight*' category, only one is of moderate size (<20 Changes), suggesting that this category of problems did not present a major issue for students.

Only a single instance of a Segment involving a '*Misleading Error Message*' occurred, and it is of moderate size (<20 Changes) indicating that misleading error messages were not a major issue for students.

Three students produce Segments related to problems with the '*WaveFrontImporter*'. One is large (>30 Changes) and two are of moderate size (>20 Changes). Overall, use of the WaveFrontImporter does not appear to have presented a significant issue for students.

### 6.5.3.3 Pipeline

Only one student produced Segments that fall into the '*Pipeline Order*' category, but all are small. Problems of this category did not appear to present a major issue. The same is true of the '*No Display*' category.

The *'Logical Operator'* category includes only one Segment. This Segment is large (>30 Changes). However, that student was the only student to utilise Logical Operations, so while logical operations were not a significant issue in the analysed assignments they might have been had other students also attempted the task.

Two of five students produced *'Draw-Not-Store'* problem Segments. One produced a Segment of moderate size, while the other produced two very large Segments, one involving 65 Changes ([MICHAEL\\_A1.PI.8.1."Wall Drawn Not Stored" \[65\]](#)). *'Draw-Not-Store'* was apparently not an issue for students in general, but was a very hard issue for one student.

#### 6.5.3.4 Event-Driven

Four students produced Segments involving problems related to the *'Screen-Window Conversion'* category. Only one was of moderate size (>20 Changes). This category did not appear to present a major issue to students. The same is the case for the *'Hit Code Error'* category which involved only two students and only one Segment of moderate size.

All students produced several Segments falling into the *'Event-Driven Program Flow'* category. Ida and Michael produced large Segments (>30 Changes), with Michael producing a very large Segment of 45 Changes ([MICHAEL\\_A1.ED.1."Button highlight" \[45\]](#)). Of the 13 Segments, 7 are of moderate size (>20 Changes) and 3 are large (>30 Changes). The category appears to have presented a hard problem to Michael only. It was not a major issue for other students but was nevertheless pervasive.

#### 6.5.3.5 OpenGL Syntax / Semantics

In the OpenGL category, the *'Colour tweaking'* category involves three small Segments (>20 Changes). The *'Mixing Up Handlers'* category involves only a single small Segment. The *'Line Stipple Pattern'* category involves a single large (<30 Changes) Segment. Finally, the *'OpenGL Syntax'* category involves a single small Segment. None of the *'OpenGL'* sub-categories appear to have presented a significant issue to students.

#### 6.5.3.6 Animation Algorithm

When implementing animation algorithms three students produced only a single small Segment, Thomas produced a large Segment of 46 Changes ([THOMAS\\_A3.ANIMATION.1."Animation Algorithm" \[46\]](#)) and Michael produced a very large Segment of 60 Changes ([MICHAEL\\_A3.ANIMATION.1."Anim Algo" \[60\]](#)) and also did not implement a working algorithm. While the *'Animation Algorithm'* implementation appeared not to be an issue for three students, it appeared to have been a hard to very hard issue for two students.

### 6.5.3.7 View

All students produced Segments relating to '*Viewing / Projection*'. Of these John, Christopher and Michael produced large Segments (>30 Changes) with Christopher producing a very large Segment of 47 Changes ([CHRISTOPHER\\_A3.VIEW.1."FP Camera" \[47\]](#)). Christopher and Michael also did not successfully complete all tasks related to Viewing. Viewing appeared not to have been a major issue for two students, a hard issue for John and Michael and a very hard issue for Christopher.

### 6.5.3.8 Lighting

Two students do not have any Segments relating to '*Lighting*' despite completing the lighting task successfully. Ida produced two moderate-sized Segments (<20 Changes) and Michael produced a very large Segment ([MICHAEL\\_A3.LIGHTING.1."Lighting Attenuation" \[60\]](#)) of 60 Changes. The implementation of lighting was apparently not a major issue for 4/5 students, but was a very difficult issue for Michael.

When examining the size of Segments, four categories/sub-categories were found to contain Segments larger than average for all students who had attempted the associated tasks. The first was the 'Order of Transformations' sub-category (involved in avatar assembly in Assignment 3). The largest Segments for each student falling into this category were of an average size of approximately 45. The second were the '2D Spatial Manipulation' / 'Loss of precision' sub-categories (both of which were involved in the solution of the same problem) which together involved an average Segment size of 65 for the largest Segments falling into this category, with two students not attempting the associated task. The third was the '3D Transformations' category, for which three students' largest Segment on average consisted of approximately 85 Changes, with two students not completing the task successfully. Finally, the 'Setting up Camera' sub-category (which maps to the 'View' category) involved an average largest Segment size of approximately 31, with two students not completing the task. The aforementioned problems caused difficulty to all students. On the other hand, some students produced no large Segments relating to the 'General Programming' category whereas others produced very large Segments. For example, in the 'General Computer Science' sub-category one student produced a large Segment of size 121 and one a large Segment of size 54. Two other students produced moderate Segments of size 28 and 20. One student did not produce any Segment falling into this category at all. Another example is the 'C++ Pointers' sub-category (also belonging to the 'General Programming' category) for which one student produced a Segment of size 114, one a Segment of size 12 and three students produced no Segments at all falling into this category.



#### 6.5.4 Discussion of Segmenting Results

When examining the distribution of Segments by type, the categories with which the largest number of Segments are the ‘Spatial’ category with 48% of Segments (79 Segments), the ‘General Programming’ category with 25% of Segments (42 Segments) and the ‘Event-Driven’ category with 13% of Segments (22 Segments). However, the absolute number of Segments does not reflect the difficulty associated with individual Segments. Segment size in terms of the number of Changes associated with Segments is likely to be a better metric for judging the difficulty associated with student work on a Segment. Judging by Segment size, four categories jump out as having caused significant issues across students. The avatar assembly task (‘Order of Transformations’) involved four students producing large or very large Segments (average size of the largest Segment was 45 Changes), and two students did not complete the task successfully. The avatar animation task (‘3D Transformations’) saw John and Ida producing very large Segments and Thomas producing a large Segment for an average largest-Segment size of approximately 85 Changes, whereas Michael and Christopher did not complete the task correctly. The rotation-about-parent task (‘2D spatial manipulation’, ‘Loss of precision’) caused all three students who attempted it to struggle significantly and produce large or very large Segments (average size of the largest Segments was approximately 65 Changes<sup>39</sup>) relating to dealing with problems associated with the task. The ‘View/Projection’ category caused only minor problems for two students, but three students produced large Segments (<30 Changes) and two of these students did not complete all required Viewing functionality (average size of the largest Segment was approximately 31 Changes). Three of the four categories (‘Order of Transformations’, ‘3D Transformations’ and ‘Viewing’) involve spatial programming, suggesting that spatial programming is the most challenging topic encountered by students while working on their assignments.

Several categories are notable as they did not produce major errors across most students but produced significant problems for only one or two students. These are the ‘C++ Pointers’ category (114 Changes for Thomas), the ‘General Computer Science’ category (54 and 121 Changes for Thomas and Christopher respectively), the ‘Animation Algorithm’ category (60 and 46 Changes for Michael and Thomas respectively), the ‘Lighting’ category (60 Changes for Michael) and the ‘Draw-Not-Store’ category (65 Changes for Michael).

This cursory examination of problem Segment size and distribution provides some insight into the nature of student problems and which problems might be particularly hard as well as showing that some problems affect only a subset of students. However, it provides no detailed insight into why

---

<sup>39</sup> For Ida, two Segments (a ‘Loss of Precision’ Segment and a ‘2d Coordinate Manipulation’ Segment are combined as they solve the same task)

students are struggling with these problems. It also does not uncover what commonalities between problems might be implicated in stymying student problem-solving. The next section (Section 6.6) will present results of the detailed analysis of problem Segments which will help answer these questions.

## **6.6 Analysis of Segment Contents**

This section will describe results of a detailed analysis of Segments. In contrast to the categorisation of Segments presented in the last section (Section 6.5) which gave only a brief overview of the results of the application of the Segment-Coding method this analysis will examine Segment contents, focusing on student problem-solving approaches and pitfalls.

Two methods are used in this analysis, one involving the qualitative analysis of Segments and one involving a quantitative analysis of spatial programming based on data developed from the qualitative analysis. These methods are discussed in Section 6.6.1. Results are presented in Sections 6.6.1.1 and 6.6.1.2.

Based on results from the analysis of Segment contents themes<sup>40</sup> were developed. These themes are issues that affect student problem-solving. They are presented in Section 6.6.6. These issues were used as the basis of a process model of student programming problem-solving which is based on identified problem-solving issues which is presented in Section 6.6.7.

### **6.6.1 Method of Segment Content Analysis**

#### **6.6.1.1 Qualitative Analysis of Segment Contents**

Qualitative analysis of Segment contents began with the initial segmenting of Project Histories. Project Histories were examined using the detailed analysis approach described in 6.2.3. This produced a growing understanding of which Changes were related and what task/problem they were related in solving. These relationships were captured through the production of Segments, whereas memoing using Change and Segment notes stored insights gained into the nature of student problem-solving and issues observed with student solution attempts.

As Segments were discovered, analysed, coded and re-coded according to the classification scheme presented in Section 5.2.2 insights captured in memos began to provide deep insight into the nature of student problems and student problem-solving approaches. This insight included the root causes of observed problems and the student misconceptions and issues that led to the problems requiring significant effort to solve. Misconceptions relate to students not having developed a concept or

---

<sup>40</sup> Theme (Grounded Theory): A theme underlies a set of Grounded Theory categories; themes are developed through the analysis of commonalities and relationships in coded data

having developed an incorrect concept whereas issues relate to broader problems with student problem-solving. Student misconceptions in fact form one of the identified issues. It also led to the development of the sub-categories described in Section 6.3.4. For logistical reasons, these categories were presented before the main body of the analysis, which is presented in this section. However, the definition of these categories arose from the detailed analysis of Segment contents discussed in this section. The qualitative analysis presented here which led to the development of these categories was then used to discover themes underlying Computer Graphics programming. These themes in turn were unified by the development of a model of student Computer Graphics programming.

As detailed examination of Segment contents progressed different problems were compared. Commonalities in student problems, both within and between students, were identified. A summary of this analysis is presented in Section 6.6.4. Due to the verbosity of the analysis process the bulk of the results of this analysis (analysis results comprise a total of roughly 172 pages) are located in Appendix Section 9.7.7.1. Notes which served as part of the analysis and memoing process are located in Appendix Section 9.7.6. One example detailed analysis of a student problem is presented in Section 6.6.3.

The final step of analysis involved extracting underlying themes from the analysed Segments. The themes underlie the different types of problems encountered by students and encapsulate their commonalities. These themes, which are issues affecting student problem-solving, are presented in Section 6.6.6. Analysis of the themes led to the development of a theoretical model of student programming problem-solving, presented in Section 6.6.7. Together these themes and the model are the main theoretical contribution towards better understanding the problems of student Computer Graphics programming.

### **6.6.1.2 Quantitative Analysis of Spatial Programming in Animations**

In addition to the detailed qualitative analysis of Segment contents, quantitative analysis on the number of errors introduced by students during spatial programming was carried out. The analysis was applied to Segments involving the implementation of avatar animations (see Appendix Section 9.3.6.2 for a description of the avatar animation task).

The animation task was designed to give students experience with compound transformations, requiring students to use their assembled avatar and through time-driven rotation of limbs produce three simple animations. The animation task requires the task involving the assembly of the avatar using transformations (see Appendix Section 9.3.6.2 for a description of the avatar assembly task) to

have been completed first and completion of the avatar assembly task requires students to have understood and applied concepts relating to the effect of compounding transformations. Hence errors occurring during the animation task are not related to misconceptions of how to produce compound transformations. They are due to incorrect spatial visualization of transformations. If students make many errors when implementing animations this shows that they are struggling with spatial programming. This can be assumed to be due to their difficulty cognitive issues with mentally visualizing compound transformations. The concept of a transformation is clearly linked to spatial ability as discussed in the literature review (see Section 2.3, especially Section 2.3.2) meaning that such problems, if they arise, are due to a spatial ability bottleneck.

Since the assignment specification did not include step-by-step instructions to building animations to determine whether any given rotation added to the animation is correct or not, the Line History View was used to examine whether that rotation was deleted or changed to a rotation with a different axis or whether the sign (direction) for the rotation was changed. Context of modifications was also examined to detect whether modifications might be rooted in non-error-related activities such as the restructuring of an animation.

The execution of the assignment at key points of the animation was used to visually determine the correctness of animations, taking into consideration the final form of the student's submitted animation as observed in the relevant change at which work on the animation is completed.

For each analysed animation, a summary table (see Figure 39) showing each rotation implemented during the creation of that animation was produced. It summarises the implementation of rotation transformations only (it excludes scale and translate transforms) since rotations modify the orientation of local coordinate system axes and are hence likely to cause the most difficulty to students, and they are also the most common transformation applied during implementation of animations.

Limb	Transform Modifications	Type	# (1-3)
Upper Leg I (L1)	+xc (144), +zc (145)		2 Axis
Lower Leg I (L2)	Continued from Upper Leg (+zc 147)	Continued	-

Lower Arm II (L1)	Copied from first experiment (+xc (157))	Copied	-
Upper Arm I (L2)	-zc (160), -yc (161), -xc (211), -yc (213)	Composite	4 Axis
Lower-Upper Arm I (L2)	+zc (210), LowerArm->UpperArm (216)		1 Axis, 1 Limb,

**Figure 39: Example table summarising rotations produced during implementation of an animation**

The summary table lists each implemented rotation in the 'Limb' column. The name listed in the 'Limb' column describes which limb the transformation was applied to. The 'Transform Modifications' column lists each modification to the rotation, including the axis and direction (sign) and the Change at which it occurs. The 'Type' column shows the type of the rotation. 'Continued' means the rotation was continued from an upper limb. For example, applying an x-rotation to the lower leg after applying an x-rotation to the upper leg earlier would count as a 'Continued' rotation. 'Copied' means the rotation was copied from a previously completed animation or from an earlier frame/phase in the same animation. Both 'Continued' and 'Copied' transformations are excluded from the final results as they do not require spatial visualization by the student. 'Composite' means the rotation acts on a limb that already has a rotation applied to it, or acts on a limb that is the child of a limb that has a rotation applied to it, causing the compositing of rotations around different axes. It is hypothesised that such rotations would be more difficult for students to visualize and hence implement.

Table 18 shows an example of such data. Two metrics are derived from this summary. The 'Initially Incorrect / Total Transformations' metric gives the percentage of rotations that were incorrect when they were first added. Given that there are three axes and one is the 'correct' axis, on average simple guessing would provide a percentage value of 66%, whereas successful spatial reasoning would reduce the number of errors. A student with a perfect spatial understanding of transformations and their composition would make no errors and achieve a 0.0 value, whereas a student making an error for every single rotation would achieve the worst possible value of 1.0.

**Table 18: Table showing the "Initially Incorrect / Total" and "Rotation/Mod" metrics derived from the evaluation of rotation transformations**

Animation	# of	# of Initially	Initially	# of Axis	Rotation/
-----------	------	----------------	-----------	-----------	-----------

	Rotations	Incorrect Rotations	Incorrect / Total	Mods	Mod
Ida A1	4	2	0.5	8	2
Ida A2	4	2	0.5	8	2
Ida A3	4	3	0.75	10	2.5

The 'Rotation/Modification' (Rotation/Mod) metric divides the total number of axis changes to rotations that occurred during implementation of the rotations by the total number of rotations. If a rotation is added correctly then it involves one 'axis modification' at the time of its implementation, meaning a value of 1.0 means that students made no errors while implementing an animation; a higher ratio indicates a higher number of errors. For example, a ratio of 2.0 means a student requires two axis modifications per rotation, which means that on average the student corrected every implemented rotation once.

Results from the examination of animations are discussed in Section 6.6.5.

## 6.6.2 Presentation of Project History data

### 6.6.2.1 Presentation of Line Histories

In order to illustrate changes to a line of source code occurring in the Project History, Line Histories are sometimes presented in the format shown in Figure 40. The top, purple line shows overall data for the Line History. Its identifier is shown in brackets on the left, whereas the numbers in brackets on the right are the Changes in which modifications to the line occurs. The total number of modifications is shown as *total = X*.

The rows below the heading row summarise the individual modifications. The first number shows the overall position of the Change at which the modification occurs. The next item shows the modification type, one of the types (Added, Mutated, Ghost, Deleted or Moved) described in Section 6.2.1.2. The item after this is either (O) or (X), indicating that the Change associated with the modification compiled or did not compile respectively. What follows is the textual content of the line at that Change. By comparing the textual content of the modification with the modification above it, one can see the textual changes made by the student in that Change.

(833):	(658, 664-667, 673 : total = 6)
658, ADDED(O):	float angle = asin(float(newy-centery)/float(newx-centerx));

664, MUTATED(O):	float angle = atan(float(newy-centery)/float(newx-centerx));
665, MUTATED(X):	float angle = atan2(float(newy-centery)/float(newx-centerx));
666, MUTATED(X):	float angle = atan2(double(newy-centery)/double(newx-centerx));
667, MUTATED(O):	float angle = atan(float(newy-centery)/float(newx-centerx));
673, MUTATED(O):	float angle = atan(float(newy-py)/float(newx-px));

**Figure 40: An example Line History**

### 6.6.2.2 Presentation of Changes

Sequences of Changes including the modification made to Changes are presented in the format shown in Figure 41. In this format, the numbers (in this case 386, 387, and 388) indicate the position of the overall Change being shown. Instead of showing all of the source code for that Change, only modifications and their surrounding lines (for context) are shown. These images are generated via the SCORE Analyser (they come from the Change Browser's Details views as discussed in Section 6.2.2.1).

Modifications to line text are shown in green, with the original text being shown on the left and the new text on the right of the '->' arrow. Moved lines are shown using a grey arrow pointing from the original to the new position. Deleted lines are shown in red, and added lines are shown in blue.

**386:**

```
glPushMatrix();
glTranslatef(0.0, 4.9, -3.7); -> //glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
//glTranslatef(0,0.75,0); -> glTranslatef(0,0.75,0);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();
```

**387:**

```
glPushMatrix();
//glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.75,0); -> glTranslatef(0,0,0);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();
```

**389:**

```

glPushMatrix();
//glTranslatef(0.0, 4.9, -3.7); -> glTranslatef(0.0, 4.5, -3.3);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.4,-0.4);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();

```

Figure 41: An example sequence of Changes showing modifications to source code for those Changes

### 6.6.3 Example of in-depth Qualitative Analysis of Segment Contents

This section will present the detailed analysis of one Segment to illustrate the method of analysis utilised in detailed analysis of Segments. The most detailed in-depth analysis of Segments of special significance includes screen captures to show the state of the program, line histories and code excerpts to ground the analysis in the actual source code, and a detailed description of important programming actions including Changes at which modifications occur. Since Segments may include tens of Changes such descriptions can be very lengthy, and hence the remainder of descriptions are presented in the appendix, Section 9.7.7.1. This analysis provides insight into student problems, enabling a description of the underlying themes as described in Section 6.6.6, as well as the refinement of classification categories into sub-categories as discussed in Section 6.3.4. The detailed analysis can best be followed by launching the associated Project History (Assignment 3, Student Michael) in the SCORE Analyser, both of which are included in the electronic appendix, and examining student actions directly while reading the analysis results. Items in round brackets refer to specific positions in the Project History.

The student creates a naïve avatar construction in Segment [MICHAEL\\_A3.SP.1."Initial Assembly"](#) [47] from (143-189) (see Figure 42), making only a single error early on at (145). Other than that, all limbs are correctly assembled, with many tweaking changes occurring in the Segment (see Figure 43, 159-182).

```

301 glPushMatrix();
302 glTranslatef(0.0, 4.9, -1.5);
303 glObjects.at(objectAt)
304     ->getObjectsByName()["LeftUpperArm"]->draw();
305 glPopMatrix();

```

Figure 42: The student's initial naïve construction at (182)



```

28, ADDED(O): glTranslatef(0.0, 1.0, 2.0);
44, MUTATED(O): glTranslatef(5.0, 1.0, 2.0);
45, MUTATED(O): glTranslatef(0.0, 1.0, 2.0);
156, MUTATED(O): glTranslatef(0.0, 0, 0);
158, MUTATED(O): glTranslatef(0.0, 0, -2.0);
159, MUTATED(O): glTranslatef(0.0, 5.0, -2.0);
160, MUTATED(O): glTranslatef(0.0, 7.0, -1.5);
161, MUTATED(O): glTranslatef(0.0, 6.0, -1.5);
162, MUTATED(O): glTranslatef(0.0, 5.7, -1.5);
163, MUTATED(O): glTranslatef(0.0, 5.9, -1.5);
182, MUTATED(O): glTranslatef(0.0, 4.9, -1.5);

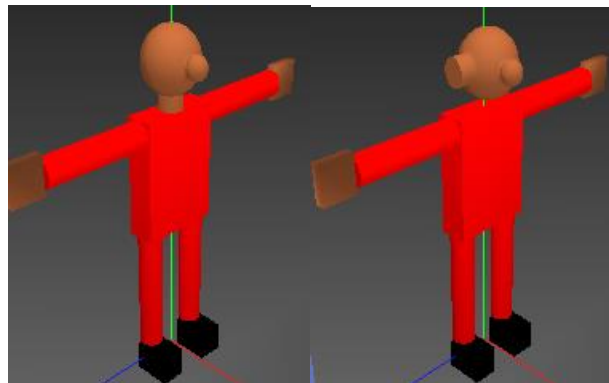
```

**Figure 43: Line History showing the upper arm's assembly modifications from (28-182)**

The student also places limbs one dimension at a time (see Figure 43, changes 156-159) which may have to do with the student having difficulty thinking about more than one dimension at a time.

### 6.6.3.1 Semi-Naïve Assembly

The student becomes aware that the assembly is incorrect after implementing keys for rotating the head since the head rotates about its own centre as shown in Figure 44.



**Figure 44: The incorrect assembly becomes apparent at (310-317) after the student implements keys to rotate the head**

The student begins by adding a second transform to the head after the orientation block. The student then tweaks the first translate until the two translates add up to roughly the same value as the original head translate from (319-321) (see Figure 45 and Figure 46). However, the first translate does not correctly position the head on the head-torso joint, which is obvious since the pre and

post-translate do not add up to the value the student had used earlier to position the head. This shows the student does not yet understand how to properly position the head onto the joint for rotation since the student knows the dimensions of the joints from the initial naïve assembly and could work out the necessary translate statements if he understood the method.

```
glPushMatrix();
//glTranslatef(0,5.9,0);
glTranslatef(0.0, 4.0, 0.0);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,1,0);
glObjects.at(objectAt)->getObjectsByName()["Head"]
->draw();
glPopMatrix();
```

Figure 45: First attempt at rotation about a joint at (321)

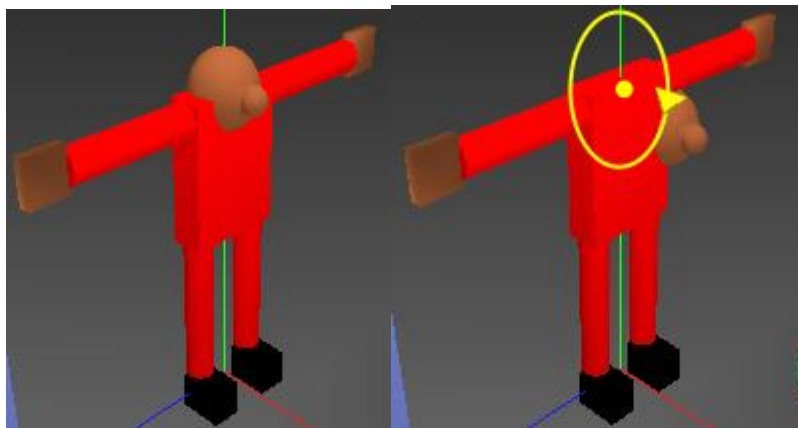


Figure 46: Rotation of the head about the misplaced joint at (321)

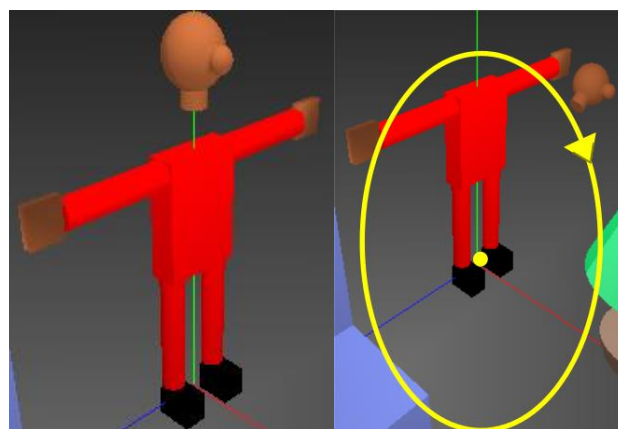


Figure 47: Rotation about the origin at (334)

The student experiments with removing the second translate as well as with tweaking the y-value of both the first and the second value, but always uses incorrect translate statements (322-333). At (334-335) the student tries adding back the original translate in front of the orientation block where it is added to the existing pre-translate, translating the head even higher than in the initial naïve construction.

The student tweaks this translation from (336-338), moving the head to the correct torso-head joint and allowing for proper rotation (see Figure 48). Note that the yellow circles showing the rotation of the limb in that Change have been added by the researcher. However, while the student has in fact created a correct joint rotation, the student has not implemented a hierarchical transformation for the avatar, meaning that limbs will not rotate together (the head is rotating about the global point at which it meets the torso in the initial assembly, not about the local point at which it meets the torso).

```
glPushMatrix();
//glTranslatef(0,5.9,0);
//glTranslatef(0.0, 1.0, 0.0);
glTranslatef(0.0, 5.15, 0.0);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
//glTranslatef(0,5.9,0);
glObjects.at(objectAt)
    ->getObjectsByName() ["Head"]->draw();
glPopMatrix();
```

**Figure 48: The corrected assembly, with the pre and post-translate adding up to the head's original translate (first commented-out translate)**

From change (339-374) the student implements manual rotation of all limbs via key strokes. This allows the student to observe that the avatar's construction is incorrect and non-hierarchical, since the rotation of an upper limb will not rotate the associated lower (child) limbs.

The student works on the proper joint orientation of the palm from (375-390) (Figure 49 shows screen captures for some of these changes and Figure 50 shows the text of the modifications), still without implementing a hierarchical model. However, since the student is working on extremities this problem does not become apparent. There are no child limbs attached to extremities which would be put out of place by the extremity's rotation.

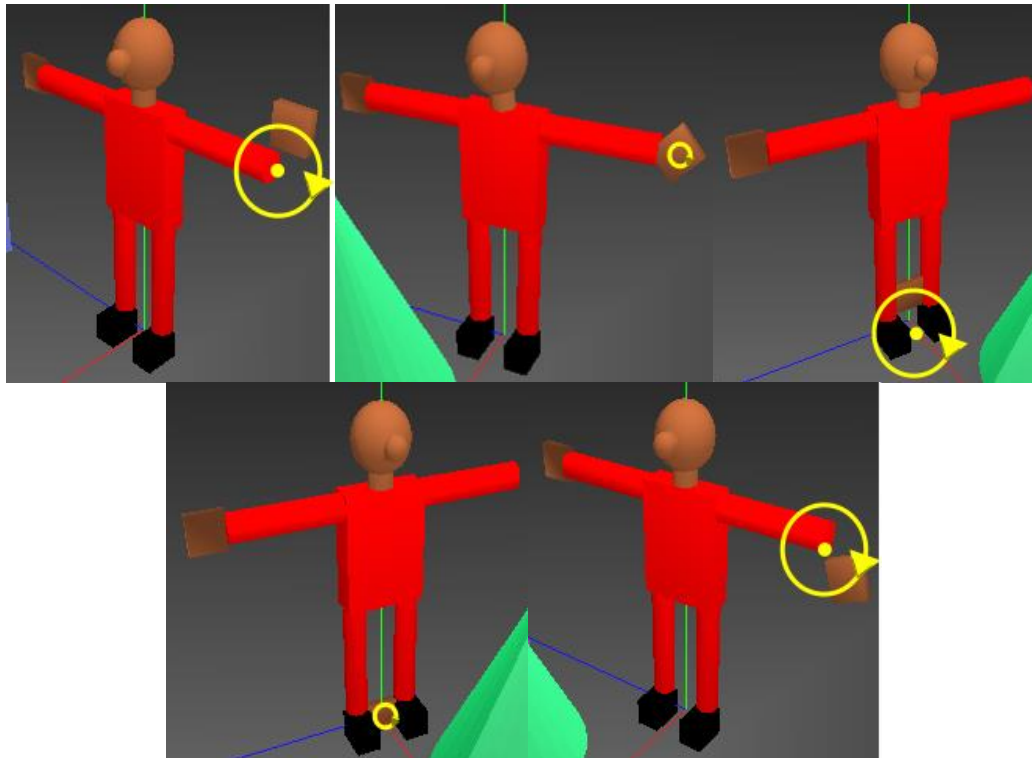


Figure 49: Effect of the moving of translation statements at (380 & 384, 385, 386, 387, 389)

**380:**

```
glPushMatrix();
glTranslatef(0.0, 5.15, 0.0); -> glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.75,0);
//glTranslatef(0.0, 4.9, -3.7);
gObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();
```

**385:**

```
glPushMatrix();
glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.75,0); -> //glTranslatef(0,0.75,0);
//glTranslatef(0.0, 4.9, -3.7);
gObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();
```

**386:**

```

glPushMatrix();
glTranslatef(0.0, 4.9, -3.7); -> //glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
//glTranslatef(0,0.75,0); -> glTranslatef(0,0.75,0);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();

```

**387:**

```

glPushMatrix();
//glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.75,0); -> glTranslatef(0,0,0);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();

```

**389:**

```

glPushMatrix();
//glTranslatef(0.0, 4.9, -3.7); -> glTranslatef(0.0, 4.5, -3.3);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.4,-0.4);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();

```

**Figure 50: Experimentation with addition and removal of pre and post-translates in the placing of the left palm at (380 & 384, 385, 386, 387, 389)**

From (375-380) the student copies the head's orientation rotate block and the pre/post translate calls across to the palm. After realising that this places the palm at the head position, the student replaces the first translate call with the palm's original translate call (without taking into account the need to translate the palm onto the joint). The student keeps the second translate call intact. This shows a poor spatial understanding of the method the student had developed to position and orient the head. The resulting incorrect rotation is shown in the first panel of Figure 49.

After implementing keys for orienting the palm from (381-384) the student removes the palm's second translate call (that is still unchanged from when it was copied from the head translate call, and hence wrong), thereby having the palm rotate about its own centre again rather than about the correct joint. The student removes the palm's pre-translate and adds back the post-translate in

(386), then removes the post-translate in (387), all of which lead to incorrect rotations as shown in Figure 49.

In (388) the student adds back the post-translate, correctly including a z-dimension coordinate which translates in the correct direction but incorrectly adding a y-coordinate. This shows the student is not properly visualising the positioning of the limb (see Figure 51). This post-translate will position the palm at the lower end of the arm, as becomes apparent after the student adds back the pre-translate (correctly deducting the amount added at the post-translate) at (389). The student never corrects this error, instead copying the assembly to the right palm at (390).

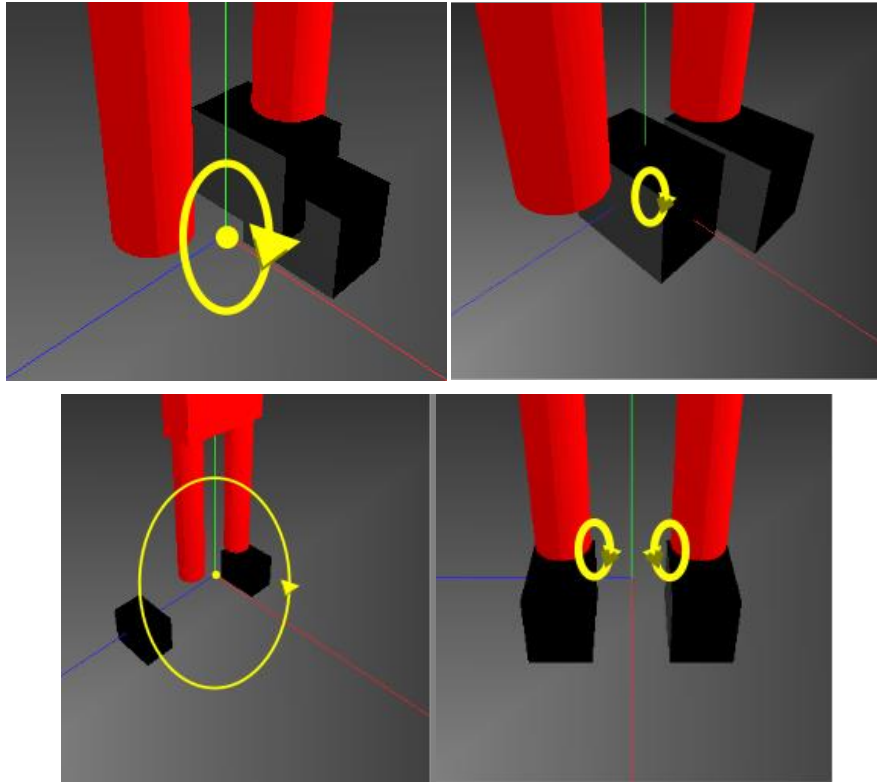
```
glPushMatrix();  
//glTranslatef(0.0, 4.9, -3.7);  
glRotatef(lpx, 1, 0, 0);  
glRotatef(lpy, 0, 1, 0);  
glRotatef(lpz, 0, 0, 1);  
glTranslatef(0,0,0); -> glTranslatef(0,0.4,-0.4);  
//glTranslatef(0.0, 4.9, -3.7);  
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
```

**Figure 51: Adding a y and z dimension to the post-translate at (388)**

(Feet, 404-411) The student works on applying the palm construction technique to the feet (see Figure 53). Initially at (404) (see Figure 52), the student removes the pre-translate and adds a post-translate; however, the translate incorrectly translates along the y-axis. The student removes the post-translate at (405), leaving the foot to rotate about the origin, before adding it back at (407) with an incorrect x-translate and a correct z-translate. After tweaking these values from (408-410) the student adds a pre-translate to place the feet at the correct position in relation to the legs.

```
glPushMatrix();  
glTranslatef(0.3, 0.0, 0.65); -> //glTranslatef(0.3, 0.0, 0.65);  
//glTranslatef(0.0, 4.5, -3.3);  
glRotatef(lpx, 1, 0, 0);  
glRotatef(lpy, 0, 1, 0);  
glRotatef(lpz, 0, 0, 1);  
glTranslatef(0,0.4,-0.4);  
  
glObjects.at(objectAt)->getObjectsByName()["RightFoot"]->draw();  
glPopMatrix();
```

**Figure 52: Foot assembly at (404)**



**Figure 53: Rotation of the foot caused by different assemblies at (404, 405, 407 and 411)**

The student does not attempt to enable rotation for the non-extremity limbs (the upper/lower arms and legs). It is likely that he realises that the rotation of limbs with child limbs would not rotate the child limbs properly, but he does not attempt to find a correct (hierarchical) solution to the problem, and the proper construction of other limbs is never attempted.

### 6.6.3.2 Avatar Movement (Part of Semi-Naïve assembly)

(Avatar movement, 428-435) The student spends (412-427) working on Viewing before returning to avatar assembly, captured in Segment [MICHAEL\\_A3.SP.3."Partially Proper Assembly"](#) [71]. The student attempts to implement avatar movement by summing the x/z coordinates of limb's translate statements with move variables as shown in Figure 54. This moves each limb into place individually before it is then individually rotated. This approach shows the student has still not developed an understanding of how to utilise `glPush()`/`glPop()` statements to build hierarchical models.

```

glPushMatrix();
//glTranslatef(0,5.9,0);
//glTranslatef(0.0, 1.0, 0.0);
glTranslatef(0.0+movex, 5.15, 0.0+movez);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
//glTranslatef(0,5.9,0);
gObjects.at(objectAt)->getObjectsByName()["Head"]->draw();
glPopMatrix();

```

**Figure 54: Enabling avatar movement at (434); move variables highlighted in yellow**

*Whole-body*

*orientation/rotation*

*(436-532)*

The student next attempts to add a whole-body orientation/rotation block to the avatar. This would be simple to achieve using hierarchical transformations as shown in Figure 55 by adding a whole-body translate and rotate call.

```

PUSH BODY STACK
TRANSLATE TO BODY_POSITION
ROTATE BODY

(For each limb, do: )
PUSH LIMB STACK
TRANSLATE TO LIMB JOINT
ROTATE LIMB JOINT
TRANSLATE TO LIMB POSITION
POP LIMB STACK

POP BODY STACK

```

**Figure 55: An approach to proper whole-body positioning and orientation**

Since the student does not have a conceptual understanding of how to composite transformations to create hierarchical transformations, he does not develop this straight-forward solution.

At (440-441) the student begins by surrounding the head's transformation calls with an if/else conditional block (see Figure 56), which uses the head's standard translation into place (developed earlier to orient the head about the head-chest join) for any orientation mode except for body orientation mode, and uses a simple post-translate when in body-orientation mode. This leads to the head being oriented about the origin when in body-orientation mode, with the rest of the body staying in place, as shown in the left panel in Figure 57. The student removes the if/else clause again at (445), removing all the changes made from (440-445).



```

if (sltbody)
{
    glRotatef(headx, 1, 0, 0);
    glRotatef(heady, 0, 1, 0);
    glRotatef(headz, 0, 0, 1);
    glTranslatef(0+movex,5.9,0+movez);
}

else {
glTranslatef(0.0+movex, 5.15, 0.0+movez);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
}

```

Figure 56: The if block enables body-orientation, the else block enables orientation of the head only at (444)

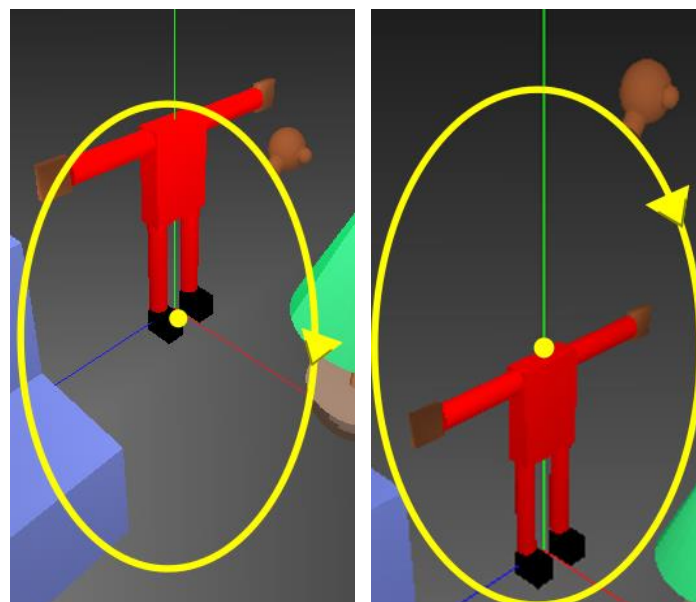


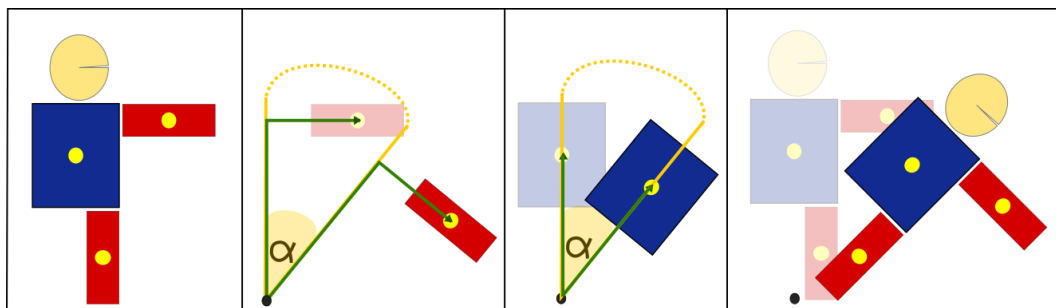
Figure 57: Effect of the body orientation on the head's rotation at (444) and (446)

At (446) the student changes the post-translate from the correct value of (0,0.75,0) to (0,5.9,0). This new value is a summation of the pre and post translate calls. As a result the head is placed at twice the correct height, but still orienting about the correct point. This error is a repeat of the same experiment in (437).

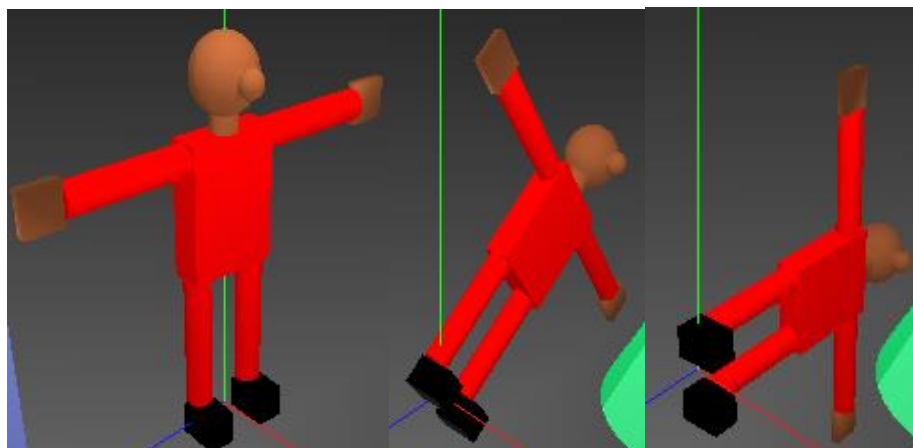
From (447-450) the student again adds an if/else if block, copying the head's correct transform block (pre/post translates and orientation block) into both the if and the elseif conditional clauses, which means that the head is positioned and oriented as usual if in head or body mode, but is left at the origin and not oriented when in any other limb mode.

At (451-452) the student again combines the pre/post translates for the body's conditional clause as he did in change (444), leading to the same incorrect result. (The student switches from using an elseif block to using an else block in [452]).

The student addresses the problem of the head rotating by itself from (453-457) by adding an orientation block using the head's rotate variables. The same block is then added to all other limbs from (458-487). The resulting assembly will apply the same pre-rotation to all limbs before the limb's individual translation call (when body orientation is selected). This is shown in Figure 58. This has the same effect as a single rotation call with the same value preceding all statements would have had, rotating the whole avatar about the origin as shown in Figure 59. The student does not understand how compositing of transformations works, so he does not apply this simpler and better solution.



**Figure 58: Individual rotation and translation of limbs to achieve rotation of the whole avatar**



**Figure 59: Rotation of the body by individual rotation and translation of limbs at (487)**

This approach allows for both the rotation of the whole avatar OR for the rotation of individual extremities but not both at the same time. In what is likely an attempt to fix this shortcoming from (489-506) the student makes the keys that modify the head rotation variable simultaneously modify the body rotation variable by the same amount as shown in Figure 60. The student also switches

from using the head rotate variables to using body rotate variables in the body orientation as shown in Figure 61.

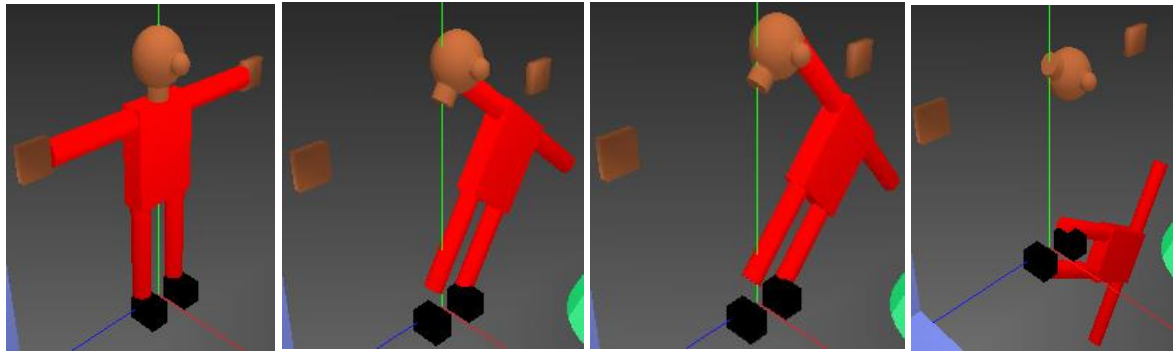
```
glPushMatrix();
if (sltbody){
glRotatef(headx, 1, 0, 0); -> glRotatef(bodyx, 1, 0, 0);
glRotatef(heady, 0, 1, 0); -> glRotatef(bodyy, 0, 1, 0);
glRotatef(headz, 0, 0, 1); -> glRotatef(bodyz, 0, 0, 1);
glTranslatef(0+movex,4.9,3.7+movez);
}
else {
glTranslatef(0.0+movex, 4.5, 3.3+movez);
glRotatef(rpx, 1, 0, 0);
glRotatef(rpy, 0, 1, 0);
glRotatef(rpz, 0, 0, 1);
glTranslatef(0,0.4,0.4);
}
glObjects.at(objectAt)->getObjectsByName()["RightPalm"]->draw();
glPopMatrix();
```

Figure 60: Construction method at (506) using a universal 'body' rotate value for whole-avatar rotation

```
else if(key == '4')
{
    if (slthead || sltbody)
    {
        headx += 5;
        bodyx += 5;
    }
    else if (sltlpalm) lpx += 5;
    else if (sltrpalm) rpx += 5;
    else if (sltlfoot) lfx += 5;
    else if (sltrfoot) rfx += 5;
}
```

Figure 61: Rotation of the head or body increments both the head's and body's rotate value at (506)

This means that in body orient mode, the extremities are rotated along with the body. However, when another limb is selected the non-extremities maintain their body orientation while the extremities use their own (head, palm, foot) orientations, leading to the extremities remaining at the origin while the non-extremities are oriented about the origin as shown in Figure 62.



**Figure 62: As the head rotates about its global 'joint' with the body, the body rotates away from the head at (505)**

By modifying the extremity and body variables by the same amount the student is trying to achieve body rotation for both types of limbs, while simultaneously maintaining the separate rotation of extremities. The correct and simple solution to this problem would be to utilise a hierarchical assembly method, but the student does not develop an understanding of this concept. His alternative approach of manually applying the same transformations to individual limbs cannot be successful; an understanding of the effect of transformations on local coordinate systems would have informed the student that this approach is a dead end.

From (507-511) the student reverts to the previous solution approach from (487), using the head rotate variable for all limbs in body orient mode, which allows either extremities or the whole body to be rotated.

```

glPushMatrix();
if (sltbody){
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0+movex,5.9,0+movez);
}
else{
glTranslatef(0.0+movex, 5.15, 0.0+movez);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
}
gObjects.at(objectAt)->getObjectsByName() ["Head"]->draw();
glPopMatrix();

glPushMatrix();
if (sltbody){
glRotatef(bodyx, 1, 0, 0);
glRotatef(bodyy, 0, 1, 0);
glRotatef(bodyz, 0, 0, 1);
}
glTranslatef(0.0+movex, 4.0, 0.0+movez);
gObjects.at(objectAt)->getObjectsByName() ["Torso"]->draw();
glPopMatrix();

```

**Figure 63: Assembly at (522)**

From (513-522) the student goes back to using the body variable for all non-extremities, and also modifies the keyboard keys to modify the body variable when in body mode and the head/palm/foot variable when in head/palm/foot mode.

From (523-532) the student implements the final construction approach (see Figure 64) by removing the if/else conditionals and applying the body orientation block to all limbs (extremities and non-extremities), while applying the individual orientation blocks to extremities. This allows the extremity orientation to be applied at the same time as the body orientation as shown in Figure 65. However, the student cannot apply this approach to non-extremities since it is not a hierarchical construction. This is masked by the fact that extremities have no children, and hence their rotation will not break the avatar.

```

glRotatef(bodyx, 1, 0, 0);
glRotatef(bodyy, 0, 1, 0);
glRotatef(bodyz, 0, 0, 1);
glTranslatef(0.0+movex, 5.15, 0.0+movez);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
glObjects.at(objectAt)->getObjectsByName()["Head"]->draw();
glPopMatrix();

glPushMatrix();
glRotatef(bodyx, 1, 0, 0);
glRotatef(bodyy, 0, 1, 0);
glRotatef(bodyz, 0, 0, 1);
glTranslatef(0.0+movex, 4.0, 0.0+movez);
glObjects.at(objectAt)->getObjectsByName()["Torso"]->draw();
glPopMatrix();

```

Figure 64: Final construction at (532)

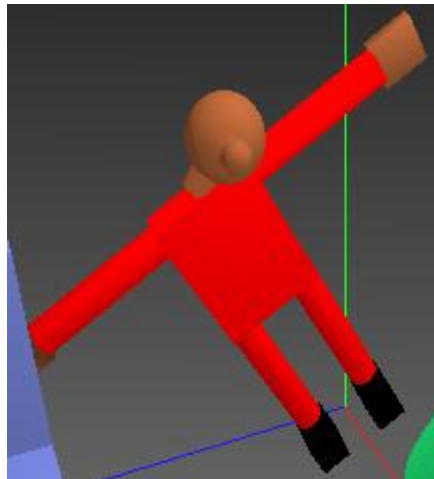


Figure 65: Rotation can be applied separately to the extremities and the body at (532)

The student shows his continued lack of development of spatial concepts from (533-601) while attempting to develop an animation. Given the construction approach used, a proper animation is impossible to achieve, so the student settles on applying a simple rotation to the entire avatar. However rather than using the existing assembly, he implements an entirely new assembly with a fixed rotate value which is called when the animation button is pressed (see Figure 66).

```

glPushMatrix();
glRotatef(50, 0, 1, 0);
glTranslatef(0,5.9,0);
gObjects.at(objectAt)->getObjectsByName()["Head"]->draw();
glPopMatrix();

glPushMatrix();
glRotatef(50, 0, 1, 0);
glTranslatef(0.0, 4.0, 0.0);
gObjects.at(objectAt)->getObjectsByName()["Torso"]->draw();
glPopMatrix();

```

**Figure 66: Animation attempt involving the implementation of an entirely new avatar assembly at (589)**

While the student discovers the correct method for rotating limbs around their parent-child joints, the student never develops a hierarchical assembly preventing him from applying the approach to non-extremity limbs. His assembly method precludes any rotation of non-extremity limbs as such rotation would cause the limb to rotate by itself away from the avatar and hence visibly break the avatar assembly.

In addition, the avatar's rotation occurs before its translation into place, leading to rotations applied to the avatar incorrectly rotating it about the origin from its global position.

The student spent considerable time in attempting to properly assemble the avatar. This included lengthy experimenting with the positioning of transformation calls. However, the student does not develop an understanding of the underlying concepts of local and global coordinate systems and the effect of the compositing of transformations. Because of this the student ended up utilising an incorrect dead-end assembly approach by applying individual rotations to each limb in order to orient the avatar.

#### **6.6.4 Summary of Qualitative Analysis of Segment Contents**

Qualitative analysis of Segment contents provided insight into the nature of problems occurring during student programming of Computer Graphics assignments. The analysis of these problem Segments also showed why certain problems proved especially challenging to students. A summary of the analysis is presented in this section, whereas the detailed (and lengthy) descriptions can be found in Appendix Section 9.7.7.1. In addition to the detailed analysis of Segments presented here, the development of sub-categories for the classification of Segments and the description of these

categories is also part of the results of the detailed qualitative analysis process. The sub-categorisation scheme is described in more detail in Section 6.3.4.

#### 6.6.4.1 Spatial

Two-dimensional spatial tasks (those falling into the ‘Two-Dimensional Coordinate Drawing / Primitive Creation’ sub-category) were generally solved relatively quickly by students, and frequently without error. When students did make errors during their implementation of two-dimensional icons, in some instances they required a long time to identify the error. This suggests that spatial understanding of spatial problems is built up iteratively as students work on these icons. An error in their mental spatial model requires considerable reconstruction of that spatial model.

Some two-dimensional problems falling into the ‘Two-Dimensional Coordinate Drawing / Primitive Creation’ sub-category appear to have caused significantly more problems than would be expected. One such example is [CHRISTOPHER\\_A1.SP.2."Hit Code" \[17\]](#). In the Segment, the student spends 17 Changes attempting to fix the hit conditional that makes buttons recognise when their rectangular area has been clicked on. A hit conditional consists of a conditional which tests whether a point (the point that was clicked by the user) falls within the rectangular boundary of the button. That the student requires so many Changes for such a simple problem is most likely due to the lack of feedback. The hit box is not drawn to screen, thus the student cannot identify which part of the formula is incorrect. A more detailed analysis of two-dimensional spatial tasks can be found in the appendix, Section 9.7.7.1.2.1.

One specific two-dimensional spatial task was found to be different to other such problems. It required substantial effort from all students who worked on it. The rotation of child objects (‘Two-Dimensional Coordinate Manipulation’ sub-category) about parent objects (see Appendix Section 9.3.6.1 for a description of the task) was significantly challenging to all three students who attempted it. Detailed analysis of the task highlighted the role of intersecting sub-tasks. These sub-tasks fell into different classification categories. This in turn made the problem confusing and more difficult to solve, as students ended up not correctly identifying the actual problem with their program. The task involved spatial, mathematical and event-driven programming and also provoked a loss-of-precision error (‘General Programming’ category, ‘Loss of precision’ sub-category) in all students who worked on it. The intersection of sub-tasks as well as students’ inability to visualize the output from their mathematical formulas (‘Mathematical’ sub-category) prevented students from correctly identifying errors in their implementations correctly, causing them to introduce additional errors in their incorrect solution attempts rather than addressing the existing errors. A detailed examination of the parent-child rotation task can be found in the appendix, Section 9.7.7.1.1.1.



For three-dimensional tasks, the avatar-assembly task ('Spatial' category, 'Order of Transformations' sub-category) caused a significant problem for all five students. In the avatar assembly task, students were asked to assemble an 'avatar' from a set of limbs initially centred at the origin using transformation commands. This assembly was to be hierarchical; that is, if a lower limb is moved, the upper limb should be moved as well and stay attached to the lower limb. This is achieved by putting the child limb's transformations after the parent limb's transformations and not removing the parent limb's transformations via a `glPop()` before the child limb has been rendered. A construction not meeting this constraint is called *non-hierarchical*. The assembly should also cause the limb to rotate about the point at which it meets its parent limb, instead of about its own centre or the parent limb's centre, which requires the limb to be moved in two steps: onto the joint, rotated, and then into its correct relative position. Assemblies not meeting this requirement are termed *joint-naïve*.

Three of five students initially produced non-hierarchical assemblies ([MICHAEL\\_A3.SP.1."Initial Assembly" \[47\]](#), [JOHN\\_A3.SP.4.1."Initial naïve assembly" \[26\]](#), [CHRISTOPHER\\_A3.SP.1.1."Initial Avatar Assembly" \[36\]](#)), while four of five students ([JOHN\\_A3.SP.4.1."Initial naïve assembly" \[26\]](#), [CHRISTOPHER\\_A3.SP.1.1."Initial Avatar Assembly" \[36\]](#), [THOMAS\\_A3.SP.5.1."Naïve Assembly" \[39\]](#), [IDA\\_A3.SP.3.1."Naïve Assembly" \[24\]](#)) initially did not compose transformations to correctly rotate limbs around the correct 'joint' joining the limb to its parent. No student initially produced an assembly that was both hierarchical and rotated limbs around the correct joint. One student (Michael) never develops the concept of how to utilise push and pop statements and order transformations to produce hierarchical transformations, whereas a different student (Christopher) does not develop the concept of ordering transformations to correctly rotate limbs about the parent-child joint.

All students spent a substantial amount of time experimenting with the order of transformation during avatar assembly, especially when while working on rotating limbs about the correct parent-child joint. Students experimented with many different positions and combinations of translate statements before and after the block of rotation calls which implements the child's orientation. This shows that students had substantial problems in visualizing the effect of different orderings of transformations. A detailed analysis of students' development of avatar assembly is located in the appendix, Section 9.7.7.1.1.2.

Four of five students implemented animations ('Spatial' category, 'Three-dimensional Transformations' sub-category) using their assembled avatars, with one student (Michael) being unable to do so because of his incorrect avatar assembly. When implementing animations, students

had previously completed avatar assembly and were thus familiar with the effect of compositing of transformations. This means that since students understood the underlying context at this stage of the assignment implementation, the only challenge to this task was the correct visualization of transformations which would create the intended orientation of limbs during the animation. Analysis of Segment contents reveals that the implementation of animations is nevertheless very challenging for all students. Analysis of student animation implementations shows that the students make a substantial number of errors when implementing rotation calls, performing only a little better than if they randomly chose an axis for rotations. When including only composite rotations, students choose the incorrect axis much more often than would be expected by random chance. This suggests that students are unable to properly, lack level of cognitive spatial ability necessary to visualize transformations, especially composite transformations. This phenomenon is examined in more detail in Section 6.6.5.

Despite struggling with the animation problems, students do not implement any visual aids to help them better understand or visualize their spatial actions. The exception to this are two students (John and Ida) who draw out local coordinate axes for individual limbs after this is suggested in a practical class. Appendix Section 9.7.7.1.1.4 provides a detailed analysis of students' work on animations.

#### 6.6.4.2 Viewing

The implementation of Views ('View' category) presented a moderate to significant challenge to students, but Views required less effort to implement than Avatar Assembly or Avatar Animations; this may be due to the view tasks being comparatively simple and not involving any composite transformations.

Three students (John, Christopher, Michael) initially utilised an incorrect conception of Views, attempting to utilise fixed coordinates with a `gluLookAt` call to produce a 'look-ahead'. One student (Michael) never developed a correct approach. Christopher also did not fulfil all required Viewing tasks, neglecting to include the additional rotation call which would have allowed the third-person View to orbit the avatar.

Both Thomas and John encountered errors related to accidental stacking of projections onto the projection matrix (forgetting to reset it to the identity matrix before applying the projection). A detailed analysis of View Segments can be found in the appendix, Section 9.7.7.1.1.5.

### 6.6.4.3 Tweaking

Tweaking ('Spatial' category, 'Tweaking' sub-category) involves minor modification to spatial coordinates to achieve pixel-perfect positioning. Tweaking occurred in both two-dimensional and three-dimensional spatial problems and often took up many Changes. While tweaking in all likelihood did not present a significant problem to students, tweaking Changes do take up time unnecessarily meaning that approaches minimizing the need for students to tweak spatial programming actions may give them more time on core assignment tasks. A detailed analysis of Segments involving tweaking activity is located in the appendix, Section 9.7.7.1.2.2.

### 6.6.4.4 Mathematical

Student application of the simple mathematics ('Spatial' category, 'Mathematical' sub-category) required by tasks in the first and third assignment was almost always initially incorrect. Furthermore, in several cases students did not recognise the errors in their mathematical formulas and spent considerable time in debugging the related problems without success.

Student errors include incorrect versions of the circle equation, incorrect formulas for calculating the mid-point between two points, incorrect functions used to calculate the angles as well as incorrect formulas to calculate distances. A more detailed exploration of problem Segments involving mathematical errors can be found in the appendix, Section 9.7.7.1.2.3.

### 6.6.4.5 General Programming

In terms of non-spatial programming problems, 'General Programming' problems presented a significant challenge to several students despite neither assignment containing any explicit General Programming task.

As mentioned in the discussion of 'Spatial' Segments all three students who worked on implementing parent-child rotation encountered a 'Loss-of-precision' problem, requiring significant effort until the error was correctly identified. In this example, the error's degree of challenge was compounded because it arose unexpectedly and in conjunction with other assignment tasks.

Most students also encountered some 'General Programming' problems related to C++ syntax or semantics. The most common such error involved C++ Object-Oriented syntax and semantics ('C++ Object-Oriented Programming' category), especially the omission of the 'virtual' keyword which produced moderate problems for several students (see Table 16, 'C++ Object-Oriented'). Other cases like Christopher's incorrect implementation of macros ([CHRISTOPHER\\_A1.GP.4."Macro" \[47\]](#), 'Syntax/Semantics' sub-category) or Thomas's problems relating to C++ pass-by-value / pass-by-reference semantics ([THOMAS\\_A1.GP.2."Child-Parenting Algorithm" \[114\]](#), 'C++ Pointers' sub-

category) proved extremely difficult for these students to resolve, largely because they were unable to identify the precise cause of the underlying problem despite knowing from which part of the source code the problem originated.

While the assignment specification did not explicitly call for the implementation of algorithms or data structures based on general Computer Science concepts, students who did decide to implement such approaches did sometimes encounter significant difficulty. One example is Christopher's implementation of a Scene Graph ([CHRISTOPHER\\_A3.GP.2."SceneTree traversal" \[121\]](#), 'Computer Science Concepts' sub-category). In attempting to implement a Scene Graph data structure the student implements an incorrect graph traversal algorithm which returns wrong values. The student then requires a very long time to implement the proper combination of conditional statements and recursive calls to correctly return graph nodes.

While the loss-of-precision error ('Loss-of-precision' sub-category), which was caused by the design of the assignment skeleton, occurred predictably for all students who attempted the task, other 'General Programming' errors occurred unpredictably based on particulars of the students' assignment implementations. Some students expended significant effort due to such problems (especially Christopher in [CHRISTOPHER\\_A3.GP.2."SceneTree traversal" \[121\]](#) and Thomas in [THOMAS\\_A1.GP.2."Child-Parenting Algorithm" \[114\]](#)), whereas other students whose implementations did not elicit these errors expended very little time on 'General Programming' problem Segments overall (e.g. John in both A1 and A3). A detailed examination of 'General Programming'-related problems can be found in the appendix, Section 9.7.7.1.2.4.

#### 6.6.4.6 Event-Driven and Pipeline

Analysis of 'Event-Driven' and 'Pipeline' problems revealed a potential conceptual source of confusion. Two students ([IDA\\_A1.ED.2."Dynamic Object Storing" \[19\]](#), [MICHAEL\\_A1.PI.1."Init render" \[19\]](#), [MICHAEL\\_A1.PI.5."Not storing objects" \[45\]](#), [MICHAEL\\_A1.PI.4."Highlight not stored" \[24\]](#), [MICHAEL\\_A1.PI.8.1."Wall Drawn Not Stored" \[65\]](#)) initially attempted to draw the output from programming actions to the screen directly instead of storing the output in the appropriate data structure in what has been termed the 'Draw-not-store' conceptual error ('Pipeline' category, 'Drawing instead of storing object state / Drawing outside display' sub-category).

Ida also encountered significant pipeline issues when attempting to use logical operations ([IDA\\_A1.ED.3."Logic Ops" \[37\]](#)).

All students also encountered problems with program flow during their implementation of event-driven functionality ('Event-driven' category, 'Event-Driven program flow' sub-category). Such

problems involve the student misunderstanding the program flow, leading to variables being in the wrong state or being modified at the wrong time, meaning that interface actions do not execute correctly. Most of these problems were of small to moderate size, with some such as Segment [MICHAEL\\_A1.ED.1."Button highlight" \[45\]](#) requiring many Changes to address.

In some instances, a misunderstanding of the GLUT event-handling mechanism created event-driven problems when the student did not realise that mouse clicks and keyboard presses create two separate events (one up-event and one down-event), as occurred in Segment [JOHN\\_A1.ED.3."GLUT Menu" \[23\]](#) ('Event-Driven' category, 'Event-Driven Program Flow' sub-category).

A detailed examination of event-driven and pipeline issues is presented in the appendix, Section 9.7.7.1.2.5.

#### 6.6.4.7 Animation Algorithm

When implementing animation algorithms three students ([CHRISTOPHER\\_A3.ANIMATION.1."Animation Algo" \[18\]](#), [JOHN\\_A3.ANIMATION.1."Animation Algorithm" \[19\]](#), [THOMAS\\_A3.ANIMATION.1."Animation Algorithm" \[46\]](#)) initially implemented a naïve method based on a loop which executed different frames of the animation ('Animation Algorithm' category/sub-category). John and Michael's ([JOHN\\_A3.ANIMATION.1."Animation Algorithm" \[19\]](#), [MICHAEL\\_A3.ANIMATION.1."Anim Algo" \[60\]](#)) implementations involve errors because of their misconception of how `glutPostRedisplay()` redraws the screen. Implementation of the animation algorithm was completed relatively quickly by four of the five students, but Michael's implementation contained both a program flow error and GLUT issues which the student never managed to untangle. This led to Michael investing a large amount of time into his unsuccessful and non-functional implementation of an animation algorithm.

The detailed analysis of students' work on animation algorithms can be found in the appendix, Section 9.7.7.1.1.4.

#### 6.6.4.8 OpenGL

'OpenGL syntax and semantics' were generally not a significant problem for students. When syntax errors did occur, they were fixed in less than the ten Changes required for them to have been included in a Segment.

Most of the problems caused by OpenGL were related to inconsistencies between the semantics of OpenGL, GLUT and C++ ('OpenGL' category, 'OpenGL syntax' sub-category). The fact that OpenGL uses degrees whereas C++ mathematics functions use radians to measure angles caused confusion

for several students. This confusion was most detrimental when it intersected with other problems. While working on child-parent rotation several students (especially Ida in Segment [IDA\\_A1.GP.7."Child object rotate"](#) [56]) incorrectly converted angles while debugging loss-of-precision and mathematical errors. This introduced further errors into their implementations.

The inconsistency in how y-coordinates are mapped between GLUT mouse and keyboard events and OpenGL window coordinates also caused errors for most students ('Event-Driven' category and 'OpenGL' category, 'Screen-Window Conversion sub-category'), requiring more than ten Changes to address in one case (e.g. [THOMAS\\_A3.ED.2."Forgets screen-window conversion"](#) [21]) but in other cases requiring less than ten Changes. Part of the reason why students managed to quickly identify and address this problem may be that students were repeatedly informed of this inconsistency in lectures and practical classes.

GLUT event-handling caused a handful of small problem Segments ('OpenGL category', 'Mixing up handlers' sub-category) relating to mouse clicks causing two separate events (a mouse-down and mouse-up event) instead of one event (e.g. [JOHN\\_A1.ED.3."GLUT Menu"](#) [23]) and relating to the different keyboard handlers for normal alphanumeric and 'special' keys such as arrow keys (e.g. [THOMAS\\_A3.GL.1."Avatar movement keys"](#) [10]).

While the OpenGL API did not appear to present a major stumbling block to students, the elimination of inconsistencies between OpenGL, GLUT and the programming language would have prevented a number of student problems. A detailed look at Segments involving OpenGL-related issues can be found in the appendix, Section 9.7.7.1.2.6.

#### 6.6.4.9 Lighting

The lighting task ('Lighting' category/sub-category) was simple and did not cause significant errors for three of five students. Ida ([IDA\\_A3.LIGHTING.1."Lighting"](#) [26], [IDA\\_A3.LIGHTING.2."Lighting 2"](#) [21]) struggled with OpenGL pipeline issues related to lighting. She was unable to create a light source, leaving the scene unlit, and spent many Changes attempting to modify other lighting-related calls which did not relate to the cause of the problem.

Michael was the only student to encounter significant problems during implementation of OpenGL lighting, caused by his use of a large attenuation value. This reduced the strength of the light so significantly that it did not provide any lighting on object surfaces (see [MICHAEL\\_A3.LIGHTING.1."Lighting Attenuation"](#) [60], [MICHAEL\\_A3.LIGHTING.2."Final Lighting"](#) [19]). The student required many Changes modifying other parts of the lighting source code before

identifying the source of the problem. A detailed examination of lighting-related Segments is given in the appendix, Section 9.7.7.1.2.7.

### 6.6.5 Quantitative Analysis of Spatial Programming in Animation Segments

Having presented a summary of the detailed analysis of Segments in the previous section, this section presents results from the detailed examination of spatial programming occurring during the programming of animations in Assignment 3. The method is discussed in more detail in Section 6.6.1.2. This section presents a summary of the results across all animations implemented by students. A detailed description of animations and an individual breakdown of these animations can be found in the appendix, Section 9.7.7.1.1.4.

When analysing the animation implementations of the four students who made a reasonable attempt at implementing at least one animation, it becomes clear that all of these students encountered difficulty with rotations implemented as part of their work on animations. Many of the transformations require one or even two corrections (trying two or even three out of three axes). It is more common for a transformation to initially be incorrect rather than correct, unless it is copied or continued from a previous transformation on the same or a lower limb.

Some students side-stepped these problems by implementing very simple animations including no composite rotations, essentially just continuing a single rotation along all limbs as is the case in Thomas's Waving animation ([THOMAS\\_A3.SP.6.3."Waving animation" \[20\]](#)).

Table 19 presents statistics relating to the implementation of rotation calls across animations.

Table 20 presents statistics relating to the implementation of only the subset of rotation calls that were Composite rotations. Each row contains data pertaining to one animation implementation. The ‘# of Rotations’ column shows the total number of unique (non-continued or copied) rotations added by the student during the implementation of the animation. The ‘# of initially incorrect rotations’ column shows the number of these added rotations that rotated about the wrong axis when first added. The ‘Initially Incorrect / Total’ column shows the percentage of rotations that were incorrect when first added. The ‘# of Axis mods’ column shows the total number of axis modifications that were required to produce the animation. In the ideal case if the student makes no errors then this number will equal the number of total rotations added, meaning the only axis modification required is the one present in the addition of the rotation call. The ‘mod/ rotation’ column shows the total number of modifications required per rotation for that animation. Since the number of required modifications for a correctly added rotation is 1, the total number of corrections required on average is (mod/rotation – 1). A ‘mod/rotation’ value of 1 would indicate the student did not need to correct any of the added rotations.

**Table 19: Summary statistics for all implemented rotations for each animation**

Animation	# of Rotations	# of Initially Incorrect Rotations	Initially Incorrect / Total	# of Axis Mods	Rotation/ Mod
Ida A1	4	2	0.5	8	2
Ida A2	4	2	0.5	8	2
Ida A3	4	3	0.75	10	2.5
Thomas A1	5	2	0.4	8	1.6
Thomas A2	2	0	0	2	1
Thomas A3	3	2	0.67	8	2.67
Christopher A1	2	1	0.5	4	2
Christopher A2	2	2	1	6	3
John A1	5	4	0.8	21	4.2
John A2	4	3	0.75	7	1.75
John A3	5	4	0.8	17	3.4
Avg			0.61		2.37



**Table 20: Summary statistics for composite transformation rotations only for each animation; animations marked as N/A did not contain compound transformations**

Animation	# of Rotations	# of Initially Incorrect Rotations	Initially Incorrect / Total	# of Axis Mods	Rotation/ Mod
Ida A1	1	1	1	4	4
Ida A2	2	2	1	5	2.5
Ida A3	2	1	0.5	5	2.5
Thomas A1	1	1	1	3	3
Thomas A2	0	0	N/A	0	N/A
Thomas A3	1	1	1	4	4
Christopher A1	0	0	N/A	0	N/A
Christopher A2	0	0	N/A	0	N/A
John A1	3	3	1	18	6
John A2	1	1	1	2	2
John A3	2	2	1	11	5.5
Avg			0.94		3.69

As Table 19 shows, approximately 60% of rotations added are initially incorrect. Given a trial and error approach and three dimensions to choose from, the average would be 66%. This means students did not perform much better than guessing at the correct axis for rotation. Students also required 2.37 axis modifications per rotation, showing that they on average needed to correct each rotation 1.37 times.

Students add composite rotations incorrectly (shown in

Table 20) 93.7% of the time; their first attempt is almost never correct. The results are worse than would be expected based on a random guess. Students also require 3.68 axis modifications and hence 2.68 corrections per rotation.

When taken together, these results indicate students have significant problems visualising the result of rotation calls. The much higher incidence of errors for composite rotations lends support to the hypothesis that composite rotations are much harder for students to visualise and implement correctly than non-composite rotations.

Animations are implemented after avatar construction is complete. At this point students have already had significant exposure to OpenGL transformations and the coordinate system. They have also had significant exposure to as spatial reasoning in the third and second assignments, as well as in lectures. Despite this, the work analysed suggests students are still having difficulties in their spatial understanding of transformations in general and composite transformations in particular. Their general tendency towards implementing simple animations also suggests they are uncomfortable with attempting spatial reasoning.

This suggests that it may be beneficial to give students more exposure to tasks requiring spatial reasoning to build up their spatial ability. Such tasks could involve composite transformations and perhaps specially designed applications or learning materials. These materials could then become part of the core Computer Graphics syllabus.

Furthermore, students did not implement visual aids to help in their development of animations. The only example occurred with John who drew out the local x/y/z axes for limbs, but only after intervention from the instructor who demonstrated this technique. This suggests that students might benefit from a greater emphasis given to teaching students debugging strategies for visual programming. Such debugging strategies might include the development of simple tools to allow the visualization of spatial concepts such as local coordinate axes. Such tools could also be provided to students in the form of libraries or as part of assignment skeletons to provide additional scaffolding, especially when students are first introduced to three-dimensional spatial programming.

### **6.6.6 Identified Issues**

This section will describe the issues affecting student problem-solving (themes in Grounded Theory parlance) identified based on the qualitative analysis of Segment contents (see Section 6.6.4) as well as the analysis of spatial programming (see Section 6.6.5), in line with the Grounded Theory approach to analysis adopted for this study.

For each issue, solutions based on observations of how that issue occurred in practice are proposed. The evaluation of these solutions falls outside the scope of this thesis and is left to future work. To evaluate the proposed solutions student work on assessment material implementing these solution approaches can be captured and analysed using the SCORE framework. During this analysis, the researcher can focus on student interaction with problems of a certain type and investigate whether the issue occurs in these problem contexts. With solutions that are student-driven (such as students learning better debugging techniques) rather than educator-driven (solutions that involve restructuring of the assignment specification, for example) the researcher can also look for artefacts pertaining to the solution approach, such as students using a particular debugging technique. In those contexts, the effectiveness of these approaches can then be evaluated; in the debugging technique example, evaluation may examine how soon after application of the debugging technique the student resolves the underlying problem.

#### **6.6.6.1 Conceptual**

Analysis of Segments identified several concepts which caused students to produce errors until they were properly understood. Without an understanding of an important concept underlying the task, students were unable to address the problem effectively or to develop a solution approach. As discussed in the last section, this led to several cases in which students did not complete core assignment tasks. Several different concepts and misconceptions were identified. These will be discussed as sub-sections to this section.

To address conceptual issues, students could be provided with learning material that clearly outlines these concepts and illustrates them with examples which allow students to master and then apply the concept.

It may also help to structure assignments in such a way that core tasks are each directly related to the development of one core concept. Students could then be provided with ways of verifying the correctness of their solution for these tasks. This will allow students to be certain they have fully understood a concept before they move on so they can apply it with confidence to further tasks.

##### **6.6.6.1.1 Transformation Concepts**

Transformation concepts relate to the use of three-dimensional transformations ('Spatial' category, 'Order of Transformations' sub-category). Errors related to transformation concepts occurred for all students as described in Section 6.6.4, and when judged by the number of Changes involved in the

related Segments as well as by the large amount of experimentation and errors that were uncovered during qualitative analysis these concepts were the most difficult for students to develop.

There are two distinct transformation concepts. The first is the '*Hierarchical Transformations*' concept, which involves understanding how to composite transformations and apply push/pop calls to create hierarchical models. For most students it was the easier concept to learn and apply in terms of time spent on it, though one student (Michael, [MICHAEL\\_A3.SP.1."Initial Assembly" \[47\]](#)) never developed the concept and hence never implemented a hierarchical avatar assembly.

The second concept is related to '*Compositing of Transformations*'. Students were required to understand how compositing of transformations would lead to changes in the local coordinate system of a limb which would allow them to rotate the limb about a parent limb. Students spent a long time experimenting with different combinations and orderings of transformation calls before producing correct implementations, indicating this concept was hard to grasp. Christopher never implemented a proper compositing of transformations ([CHRISTOPHER\\_A3.SP.1.1."Initial Avatar Assembly" \[36\]](#)) and Michael's implementation ([MICHAEL\\_A3.SP.1."Initial Assembly" \[47\]](#)) was only partially correct.

In addition to students struggling with concepts relating to transformations, students struggled with the implementation of transformations for their animations even after having developed these concepts, indicating that they also struggled with the cognitive task of visualizing the effect of transformations which is an activity related to their visio-spatial ability. It is likely that this also contributed to their difficulty with developing the concepts in the first place as they were unable to visualize the effect of different orderings of transformations.

The following Segments involve problems related to a transformation concept ('Spatial' category, 'Order of Transformations' sub-category):

- [MICHAEL\\_A3.SP.1."Initial Assembly" \[47\]](#)
- [THOMAS\\_A3.SP.5.1."Naive Assembly" \[39\]](#)
- [THOMAS\\_A3.SP.5.2."Proper Assembly" \[55\]](#)
- [IDA\\_A3.SP.3.1."Naive Assembly" \[24\]](#)
- [IDA\\_A3.SP.3.2."Proper assembly" \[28\]](#)
- [JOHN\\_A3.SP.4.1."Initial naive assembly" \[26\]](#)
- [JOHN\\_A3.SP.4.2."Final assembly" \[33\]](#)
- [CHRISTOPHER\\_A3.SP.1.1."Initial Avatar Assembly" \[36\]](#)

#### **6.6.6.1.2 Mathematical Concepts**

The category of mathematical concepts involves all incorrect applications of mathematical formulas ('Spatial' category, 'Mathematical' sub-category). While neither assignment included difficult mathematical tasks, students demonstrated misconceptions related to mathematical formulas or their application almost every time they approached a new problem involving mathematics.

As one example, two students attempted to implement distance-based pick-up functionality in the third assignment ([CHRISTOPHER\\_A3.SP.3.5."Pickup / Drop Object" \[47\]](#), [THOMAS\\_A3.SP.4."Carrying Lamp" \[36\]](#)). Neither student used the correct distance formula  $d = \sqrt{(xu-xl)^2 + (yu-yl)^2}$  in their initial implementation. Even more significantly despite working on the problem for a long time, neither student ever developed the correct distance formula.

These Segments contain problems related to a mathematical concept ('Spatial' category; mostly 'Mathematical' sub-category, but also other sub-categories when only a minority of the Changes are mathematical in nature):

- [JOHN\\_A1.SP.7."Rotation around Parent" \[56\]](#)
- [JOHN\\_A3.SP.5.3."Swim Anim" \[120\]](#)
- [CHRISTOPHER\\_A3.SP.3.3."Walk In ViewDir" \[25\]](#)
- [CHRISTOPHER\\_A3.SP.3.5."Pickup / Drop Object" \[47\]](#)
- [MICHAEL\\_A1.SP.3.2."Button Spatial Coloring" \[26\]](#)
- [MICHAEL\\_A1.SP.9."Circle Draw" \[26\]](#)
- [MICHAEL\\_A1.SP.10."Circle resize" \[18\]](#)
- [THOMAS\\_A1.SP.1."Button Spatial Modulus" \[20\]](#)
- [THOMAS\\_A1.SP.4.1."Object rotate maths" \[19\]](#)
- [THOMAS\\_A1.SP.4.2."Object rotate angle Math Formula" \[10\]](#)
- [THOMAS\\_A3.SP.2.1."Angle Conversion" \[12\]](#)
- [THOMAS\\_A3.SP.2.2."Avatar Movement" \[36\]](#)
- [THOMAS\\_A3.SP.4."Carrying Lamp" \[36\]](#)
- [IDA\\_A1.GP.7."Child object rotate"\[56\]](#)
- [IDA\\_A1.SP.3."Object rotation" \[23\]](#)

#### 6.6.6.1.3 View Concepts

View concepts are related to the 'View' category/sub-category. Three students ([CHRISTOPHER\\_A3.VIEW.1."FP Camera" \[47\]](#), [JOHN\\_A3.VIEW.2."FP view" \[21\]](#), [MICHAEL\\_A3.VIEW.1."View" \[35\]](#)) initially produced View implementations which used fixed lookAhead coordinates. An example would be the call `gluLookAt(avatarX, avatarY, avatarZ, avatarX+5, avatarY, avatarZ+5, 0, 1, 0)`. Such a call by itself produces a fixed View instead of a view which looks in the direction the avatar's head is facing required by the assignment specification. This shows a lack of understanding regarding the way in which the View

and Model concepts inherent in the ModelView matrix interact. One student (Michael) never develops a correct solution.

These Segments containing problems related to a view concept ('View' category/sub-category):

- [CHRISTOPHER\\_A3.VIEW.1."FP Camera" \[47\]](#)
- [CHRISTOPHER\\_A3.VIEW.2."TP Camera" \[15\]](#)
- [CHRISTOPHER\\_A3.VIEW.3."View Spherical Coordinates" \[14\]](#)
- [MICHAEL\\_A3.VIEW.1."View" \[35\]](#)
- [THOMAS\\_A3.VIEW.1."Third-Person Camera" \[21\]](#)
- [JOHN\\_A3.VIEW.2."FP view" \[21\]](#)

#### ***6.6.6.1.4 Draw-not-store Misconception***

The draw-not-store misconception involves the student immediately drawing out the effect of actions to screen without storing the effect of the action ('Pipeline' category, 'Drawing instead of storing object state / Drawing outside display' sub-category). For example, if an object is created a student may draw the object directly to screen without storing it in a data structure, meaning that it will be cleared and lost at the next window clear call.

This misconception occurred early in the first assignment and affected two students. Ida struggled with this misconception early on in the first assignment ([IDA\\_A1.ED.2."Dynamic Object Storing" \[19\]](#)). After she had resolved her misconception, she did not produce any further errors. Michael implemented erroneous object addition ([MICHAEL\\_A1.PI.1."Init render" \[19\]](#), [MICHAEL\\_A1.PI.5."Not storing objects" \[45\]](#)), highlighting ([MICHAEL\\_A1.PI.4."Highlight not stored" \[24\]](#)) as well as wall-drawing source code ([MICHAEL\\_A1.PI.8.1."Wall Drawn Not Stored" \[65\]](#)). Interestingly Michael produced errors related to the misconception even after resolving it in different contexts.

The following Segments contain problems related to a draw-not-store misconception ('Pipeline' category, 'Drawing instead of storing object state / Drawing outside display' sub-category):

- [IDA\\_A1.ED.2."Dynamic Object Storing" \[19\]](#)
- [MICHAEL\\_A1.PI.1."Init render" \[19\]](#)
- [MICHAEL\\_A1.PI.5."Not storing objects" \[45\]](#)
- [MICHAEL\\_A1.PI.4."Highlight not stored" \[24\]](#)
- [MICHAEL\\_A1.PI.8.1."Wall Drawn Not Stored" \[65\]](#)

#### ***6.6.6.1.5 Time-based behaviour concepts***

The 'Time-based Behaviour concepts' refer to concepts relating to the correct implementation of an algorithm for animations which can draw frames to screen in a way in which frames will not be overdrawn before they are visible, occurring in Segments belonging to the 'Animation algorithm'

category/sub-category. Three students ([CHRISTOPHER\\_A3.ANIMATION.1."Animation Algo" \[18\]](#), [JOHN\\_A3.ANIMATION.1."Animation Algorithm" \[19\]](#), [THOMAS\\_A3.ANIMATION.1."Animation Algorithm" \[46\]](#)) initially attempted to utilise a loop with no timers for drawing animations which led to non-functional animation algorithms. All of these students managed to develop a correct concept and resolve this issue and move to functional approaches using either timers or busy loops in a moderate amount of time. Michael ([MICHAEL\\_A3.ANIMATION.1."Anim Algo" \[60\]](#)) never produced a functional concept of time-based behaviour and never corrected errors in his implementation of an animation algorithm.

These Segments contain problems related to time-based concepts ('Animation category/sub-category):

- [CHRISTOPHER\\_A3.ANIMATION.1."Animation Algo" \[18\]](#)
- [JOHN\\_A3.ANIMATION.1."Animation Algorithm" \[19\]](#)
- [MICHAEL\\_A3.ANIMATION.1."Anim Algo" \[60\]](#)
- [THOMAS\\_A3.ANIMATION.1."Animation Algorithm" \[46\]](#)

#### **6.6.6.2 Difficult-to-Visualize**

The Difficult-To-Visualize issue occurred in mathematics and spatial contexts ('Spatial category', all sub-categories, some cases also occurred in the 'Event-Driven' and 'Lighting' categories). In a mathematics context, students were unable to visualize the effect of their application of mathematical formulas. For example, Ida never discovered the problem underlying her parent-child rotation formula. This problem could have been detected easily had the student drawn out the circular path about which the object's lower and upper points were rotated. Thomas and Michael would likely have realised their distance calculation formulas (used to calculate distance from the avatar to objects) was incorrect if they had drawn out the boundary of their distance test. In the spatial context, better visualization methods might have helped John understand and correct the gimbal lock problem without the considerable time spent on trial-and-error experimentation.

This issue mirrors the finding that the most challenging problems for novice programming students involve the learning and application of concepts where low feedback is provided to the student as a result of the programming action (Butler & Morgan, 2007). Butler and Morgan's work (2007) showed that student ranking of the difficulty of topics was found to closely track the level of feedback provided when working on the topic. For example, algorithms which without additional student intervention such as addition of print statements provide almost no feedback were ranked as hardest, whereas syntax, which provides a high level of feedback through detailed error messages, was ranked as the easiest topic. Many Computer Graphics tasks can provide very poor feedback, with students being unable to determine how their source code produced the results on

screen. On the other hand, a Computer Graphics program would be the ideal visualization tool to provide a high degree of feedback if used correctly. The challenge to the Computer Graphics educator is how to train students to utilise Computer Graphics programming to turn low-feedback Computer Graphics concepts into high-feedback concepts.

Students sometimes attempted to use `print/cout` statements to print out coordinates, especially when dealing with mathematical problems. This approach did not in most cases lead to a quick resolution of the problem, presumably because while the students turned to printing out the coordinates because they had problems visualizing the effect of their source code, they also had problems visualizing the coordinates being printed to the screen. The only other approach utilised by students when left to their own devices involved trial and error. Students did not implement any proper visualization aids. However when an instructor introduced the simple visualization aid of drawing out local coordinate axes both John and Ida used the approach extensively. This suggests these students were looking for ways in which to better use their application as a visualization tool but were unable to develop methods of doing so by themselves.

Students' inability to utilise their Computer Graphics applications to visualize their spatial and mathematical programming is regrettable. This is especially true since such visualization is an important real-world application of Computer Graphics programming. Explicitly teaching students visio-spatial debugging techniques and/or providing them with debugging tools may help resolve student issues relating to visualization of spatial or mathematical programming actions. Such techniques or tools could include some of the approaches used by the researcher in debugging student programs such as drawing out rotation paths, drawing out local coordinate axes or drawing out the effect of every transformation in a series of transformations. It may also include more complex techniques. One example is the drawing out of viewing volumes in an independent View. This makes it possible to observe the effect of modifications to the viewing volume when view-related OpenGL calls are modified. Students could implement tools to perform such debugging tasks as part of their practical or homework exercises, thereby providing both the benefit of the application of the tool to allow better visualization, as well as a deeper understanding of visualization techniques.

These Segments contain problems related to the difficult-to-visualize issue:

-all the Segments listed in the 'Cognitive Difficulty of Spatial programming' issue

-several mathematical Segments ('Spatial' category, 'Mathematical' sub-category):

- [JOHN\\_A1.SP.7."Rotation around Parent" \[56 \]](#)
- [CHRISTOPHER\\_A3.SP.3.3."Walk In ViewDir" \[25\]](#)



- CHRISTOPHER\_A3.SP.3.5."Pickup / Drop Object" [47]
- THOMAS\_A1.SP.4.1."Object rotate maths" [19]
- THOMAS\_A1.SP.4.2."Object rotate angle Math Formula" [10]
- THOMAS\_A3.SP.2.2."Avatar Movement" [36]
- THOMAS\_A3.SP.4."Carrying Lamp" [36]
- IDA\_A1.GP.7."Child object rotate"[56]
- IDA\_A1.SP.3."Object rotation" [23]

-event-driven Segments relating to hit code (mostly 'Event-Driven' category, 'Hit code' sub-category):

- MICHAEL\_A1.ED.2."Select ll > ur" [30]
- CHRISTOPHER\_A1.SP.2."Hit Code" [17]
- CHRISTOPHER\_A1.ED.2."Hit Code" [18]
- CHRISTOPHER\_A1.ED.3."Hit Code 2" [21]
- CHRISTOPHER\_A1.ED.4."Hit Code 3" [11]
- THOMAS\_A3.ED.2."Forgets screen-window conversion" [21]

-lighting attenuation ('Lighting' category/sub-category):

- MICHAEL\_A3.LIGHTING.1."Lighting Attenuation" [60]

### 6.6.6.3 Cognitive Difficulty of Spatial programming

Students frequently exhibited problems when working on three-dimensional transformations, most apparent during the implementation of avatar assembly and avatar animation ('Spatial' category, 'Three-dimensional transformations' sub-category), with students requiring a significant amount of experimentation and producing a large number of incorrect modifications. As was discussed in Section 6.6.5, this was also the case for animation tasks at which point students had already learned and applied the required concepts. Students did not do much better than chance when adding new rotations, showing they lacked sufficient spatial ability to correctly visualize these transformations. This lack of spatial ability will in turn prevent students from understanding spatial concepts and from working with spatial concepts.

The visualization tools proposed for the 'Difficult-to-Visualize' issue may be an effective way of addressing spatial ability weaknesses. These tools could help augment students' visio-spatial ability by using their own program as a visualization tool. In addition, it may be beneficial to improve students' spatial ability through specially designed learning material and assessment tasks. Such an approach has already been shown to be successful in engineering Computer-Graphics: (Alias et al., 2002; Blade & Watson, 1955; Hsi et al., 1997; Leopold et al., 2001; Martín-Dorta et al., 2008) (see Literature Review Section 2.3.4 for more details). Application of these approaches to Computer Graphics Education would require the crafting and evaluation of new materials as such materials must be task-relevant and well-crafted to be effective (Baenninger & Newcombe, 1989).

These Segments contain problems related to the 'Cognitive Difficulty of Spatial programming' issue:

- [CHRISTOPHER\\_A3.SP.1.1."Initial Avatar Assembly" \[36\]](#)
- [CHRISTOPHER\\_A3.SP.2.1."Pickup Anim" \[24\]](#)
- [JOHN\\_A3.SP.4.1."Initial naive assembly" \[26\]](#)
- [JOHN\\_A3.SP.4.2."Final assembly" \[33\]](#)
- [JOHN\\_A3.SP.5.1."Walk anim" \[19\]](#)
- [JOHN\\_A3.SP.5.2."Pickup Anim" \[99\]](#)
- [JOHN\\_A3.SP.5.3."Swim Anim" \[120\]](#)
- [MICHAEL\\_A3.SP.1."Initial Assembly" \[47\]](#)
- [MICHAEL\\_A3.SP.3."Partially Proper Assembly" \[71\]](#)
- [THOMAS\\_A3.SP.5.1."Naive Assembly" \[39\]](#)
- [THOMAS\\_A3.SP.5.2."Proper Assembly" \[55\]](#)
- [THOMAS\\_A3.SP.6.1."Walk Animation" \[36\]](#)
- [THOMAS\\_A3.SP.6.2."Pickup Animation" \[21\]](#)
- [THOMAS\\_A3.SP.6.3."Waving animation" \[20\]](#)
- [IDA\\_A3.SP.3.1."Naive Assembly" \[24\]](#)
- [IDA\\_A3.SP.3.2."Proper assembly" \[28\]](#)
- [IDA\\_A3.SP.4.1."The Walk Animation" \[66\]](#)
- [IDA\\_A3.SP.4.2.1."Pickup Animation" \[79\]](#)
- [IDA\\_A3.SP.4.3."Star Jump" \[30\]](#)

#### 6.6.6.4 OpenGL Pipeline Black-box

The ‘OpenGL Pipeline Black-Box’ issue relates to problems based on students’ misunderstanding of the state of the OpenGL state machine, or an unawareness of the way in which an OpenGL action is processed in the graphics pipeline. Issues with the OpenGL pipeline acting as a ‘black box’ occurred in many different problem Segments. One example is Ida’s omission of a call to place a light when implementing lighting ([IDA\\_A3.LIGHTING.1."Lighting" \[26\]](#), [IDA\\_A3.LIGHTING.2."Lighting 2" \[21\]](#), ‘Lighting’ category/sub-category). Problems during the implementation of animation algorithms relating to the way in which the `glutPostRedisplay` call refreshes the screen ([JOHN\\_A3.ANIMATION.1."Animation Algorithm" \[19\]](#), [MICHAEL\\_A3.ANIMATION.1."Anim Algo" \[60\]](#), ‘Animation Algorithm’ category/sub-category) are also examples of students’ lack of insight into the workings of the OpenGL pipeline causing problems. Problems related to these issues were hard to resolve because students did not have tools available to pinpoint the source of the problem.

Solutions to the black-box problem might include teaching students to access OpenGL pipeline state using `glGet(...)` calls or providing them with a library of functions to query OpenGL pipeline states related to different OpenGL functionality used in the assignment. To address cases such as that of the `glutPostRedisplay()` call students could be provided with a debugger including breakpoints to allow them to understand the way in which OpenGL directs program flow during execution.

These Segments contain problems related to the ‘OpenGL Pipeline Black-box’ issue:

-Logical Operators (‘Event-Driven’ category, ‘Pipeline order’ sub-category):

- IDA\_A1.ED.3."Logic Ops" [37]
- Render-Not-Store ('Pipeline' category, 'Drawing instead of storing object state / Drawing outside display' sub-category):
- MICHAEL\_A1.PI.1."Init render" [19]
  - MICHAEL\_A1.PI.5."Not storing objects" [45]
  - MICHAEL\_A1.PI.4."Highlight not stored" [24]
  - MICHAEL\_A1.PI.8.1."Wall Drawn Not Stored" [65]
  - IDA\_A1.ED.2."Dynamic Object Storing" [19]
- Projection Stacking ('View' category/sub-category):
- JOHN\_A3.VIEW.1."Minimap" [37]
  - THOMAS\_A3.VIEW.2."Ortho/Top-down Camera" [20]
- Lighting ('Lighting' category / sub-category):
- IDA\_A3.LIGHTING.1."Lighting" [26]
  - IDA\_A3.LIGHTING.2."Lighting 2" [21]
  - MICHAEL\_A3.LIGHTING.1."Lighting Attenuation" [60]

### 6.6.6.5 Program-Flow Understanding

Issues related to program-flow understanding occurred for all students in an event-driven context ('Event-Driven' category, 'Event-Driven Program Flow' sub-category) and sometimes also contributed to pipeline and animation algorithm problems. In problems related to program-flow understanding, students do not understand the order in which their source code is executed when the program is run and what state variables will be in. While all students encounter issues related to program-flow understanding, problems related to program-flow understanding usually take only a short or moderate number of Changes to address in the context of the Project Histories analysed as part of this research project.

To assist students with dealing with program-flow problems it may be helpful to teach students better debugging techniques and provided with better debugging tools to debug these types of problems. Most students rely on the `cout` statements, and the results have shown this approach to be inadequate. One such debugging tool/technique is the use of breakpoints.

These Segments contain problems related to the 'Program-Flow understanding' issue:

- JOHN\_A1.ED.2."Deleting Children" [11]
- JOHN\_A1.ED.3."GLUT Menu" [23]
- CHRISTOPHER\_A1.ED.5."ED program flow" [11]
- CHRISTOPHER\_A1.ED.6."ED Program flow 2" [12]
- MICHAEL\_A1.ED.1."Button highlight" [45]
- MICHAEL\_A1.ED.2."Select ll > ur" [30]
- MICHAEL\_A1.ED.3."GLUT Menu" [28]
- MICHAEL\_A1.ED.4."Resizing circle" [10]
- THOMAS\_A1.ED."Event-Driven"
- IDA\_A1.ED.2."Dynamic Object Storing" [19]
- IDA\_A1.ED.3."Logic Ops" [37]

- [IDA\\_A1.ED.4."Flower pot move" \[15\]](#)
- [IDA\\_A1.ED.5."Selection" \[22\]](#)

#### 6.6.6.6 Interplay of different problems

The 'Interplay of different problems' issue arises in any context in which different problems intersect, making them harder to solve (a mixture of any number of sub-categories). There are many such examples involving event-driven, spatial and mathematical problems in the first assignment. The best example in the current data is that of parent-child rotation. In solving this problem, students had to implement different mathematical formulas for calculating angles, measuring distance and performing rotation as well as maintaining a spatial understanding of the parent object and child object while also implementing event-driven code to enable smooth rotation. Students were also unsure of whether to use radians or degrees with C++ mathematical functions. In addition, student solutions caused loss of precision, introducing a 'General-Programming' problem. The intersection of these problems made it extremely difficult to debug and resolve them, since even correcting part of the problem would not produce correct results as long as other errors were present meaning they lacked feedback during the debugging process.

One strategy which may help address this issue relates to assignment specifications. If tasks are structured in a way that allows students to build up solutions to complex problems iteratively while at the same time being able to verify the correctness of their solution to each part of the problem then students are less likely to face a situation in which multiple problems overlap in a detrimental manner. Also, evaluation of assignment specifications as part of a review of teaching practices using a tool like SCORE will reveal flaws in the assignment specification that introduce unintended problems such as the loss-of-precision problem, and will allow the educator to identify during which tasks problems tend to overlap to allow for a better structuring of tasks in the next iteration of that course.

Another strategy that may help students cope with overlapping problems is making sure students have debugging strategies for each of the different problem types that occur. It is also important to ensure that students are able to identify different types of problems in the first place. This may help students to untwine the different problems and address each in isolation using a fitting debugging strategy.

The following Segments are significantly impacted by problems related to the 'Interplay of different problems' issue (the issue is also present to a lesser extent in many other Segments):

- [JOHN\\_A1.SP.7."Rotation around Parent" \[56 \]](#)
- [THOMAS\\_A1.SP.4.1."Object rotate maths" \[19\]](#)

- THOMAS\_A1.SP.4.2."Object rotate angle Math Formula" [10]
- THOMAS\_A1.GP.6."Loss of Precision" [31]
- THOMAS\_A1.GP.7."Unintentional Rounding" [43]
- IDA\_A1.GP.7."Child object rotate"[56]
- IDA\_A1.SP.3."Object rotation" [23]
- JOHN\_A1.ED.2."Deleting Children" [11]
- JOHN\_A1.ED.3."GLUT Menu" [23]
- MICHAEL\_A1.SP.6."Clipping" [56]
- MICHAEL\_A1.PI.7."Scissor test" [19]
- MICHAEL\_A1.PI.8.1."Wall Drawn Not Stored" [65]
- MICHAEL\_A1.PI.8.2."Wall Drawing ED" [13]
- CHRISTOPHER\_A3.GP.2."SceneTree traversal" [121]
- CHRISTOPHER\_A3.SP.1.1."Initial Avatar Assembly" [36]

### 6.6.6.7 Programming Language Syntax and Semantic Concepts

The 'Programming Language Syntax and Semantic' issue captures all those concepts that relate to the use of the language itself (relating to 'General Programming' sub-categories' such as 'Syntax/Semantics', 'C++ Pointers' or 'C++ Object-Oriented Programming'. For example, a student may have mastered object-oriented programming using Java, but nevertheless encounter problems when attempting object-oriented programming in C++. In the data reviewed, this was usually the case because students omitted the 'virtual' keyword. In this case, it is the concept of C++ Object-Oriented semantics that is at issue. Were the problem to do with the actual object-oriented programming model, it would be a logic error and not fall into this category of misconceptions.

The issue relates to a range of concepts; in the data analysed issues relating to C++ Object-Oriented semantics, memory management, C++ pointers and a range of other issues captured in the catchall 'semantics' classification sub-category are related to the 'Programming Language Syntax and Semantic' issue.

The individual topics making up this issue vary based on the programming language. For example, use of Java would remove the 'C++ Pointers' category as Java does not use pointers. Even if two programming languages share a topic, the way that the syntax and semantics of the language generate problems are likely to be different. For example, the use of Java would have eliminated student errors related to the 'virtual' keyword but may have introduced different issues relating to the specifics of Java semantics such as the use of interfaces.

The best way to address the issue of 'Programming Language Syntax and Semantics' is to ensure that all students possess sufficient familiarity with the programming language and its syntax and semantics to encounter few such problems, or to be able to resolve them quickly when they are encountered.

Since this is not always possible given the myriad different programming languages students may learn during their study and the rapidity with which details of syntax and semantics can be forgotten, a language with conceptually simple and clear semantics should be chosen. Languages with confounding syntax or semantics (such as the ‘virtual’ keyword) should be avoided.

Alternatively (or in addition), the assignment specification can steer students away from known problem areas, and libraries can be provided to ameliorate some issues. For example, several C++ libraries provide conceptually simpler and safer pointers which may have prevented pointer-related problems such as the issues encountered by Thomas relating to ‘C++ pass-by-reference/pass-by-value’ semantics. In addition, students can be reminded of common errors at the outset of their assignment; a list of common errors can be generated based on analysis of previous semesters’ work using a method such as the SCORE-based analysis method proposed in this thesis.

The following Segments are related to programming language misconceptions:

-Object-Oriented (‘General Programming’ category, ‘C++ Object-Oriented Programming’ sub-category):

- JOHN\_A1.GP.1."Virtual function and Constructor" [11]
- JOHN\_A3.GP.3."Object-Oriented" [23]
- IDA\_A1.GP.5."Hit code and methods" [17]
- CHRISTOPHER\_A1.GP.5."Virtual Keyword" [31]
- CHRISTOPHER\_A1.GP.8."Virtual keyword 2" [15]

-Memory (‘General Programming’ category, ‘C++ Memory’ sub-category):

- JOHN\_A1.GP.3."Delete Mode Memory Problems" [20]

-Semantics (‘General Programming’ category, ‘Syntax/Semantics’ sub-category):

- THOMAS\_A3.GP.3."Accidental Octal" [14]
- CHRISTOPHER\_A1.GP.3."C++ String" [22]
- CHRISTOPHER\_A1.GP.4."Macro" [47]
- CHRISTOPHER\_A3.GP.3."glutTimerFunc function pointer" [25]
- JOHN\_A3.GP.1."Static function pointer" [13]
- THOMAS\_A3.GP.3."Accidental Octal" [14]
- IDA\_A1.GP.2."Switch statement" [19]
- IDA\_A1.GP.4."Static functions" [21]

-Misleading Error Message (‘General Programming’ category, ‘Misleading Error Message’ sub-category):

- CHRISTOPHER\_A1.GP.7."Confusing function bracket error" [25]

-C++ Pointers (‘General Programming’ category, ‘C++ Pointers’ sub-category):

- CHRISTOPHER\_A1.GP.1."this keyword" [10]
- CHRISTOPHER\_A1.GP.2."C++ this. syntax / static functions" [12]
- THOMAS\_A1.GP.2."Child-Parenting Algorithm" [114]
- THOMAS\_A1.GP.4."Pointer syntax" [20]
- CHRISTOPHER\_A3.GP.1."Setting NULL / This keyword" [13]
- CHRISTOPHER\_A3.GP.4."C++ Pointer/Direct syntax" [12]

#### 6.6.6.8 Inconsistencies (degree-radian, screen-window)

Inconsistency issues are caused by an inconsistency between the operation of different APIs or libraries. In the analysis presented in this chapter, the two clearest examples involve the inconsistency between OpenGL and GLUT window y-coordinates and the inconsistency between the unit used for angle measurement in C++ (radians) and OpenGL (degrees). Such inconsistencies served to confuse students and to introduce errors. They were most detrimental when they co-occurred with other problems. In this case, students would either not identify the problem related to the inconsistency, or they would identify working code as faulty, suspecting that the inconsistency had caused an error when it had not. This caused affected students to attempt to address a non-existent error, which actually introduced further errors.

The simplest solution is to attempt to prevent inconsistencies by choosing materials that are pedagogically sound and do not include inconsistencies. When this is not possible one potential solution is to provide students with a library that prevents inconsistencies by replicating functionality (for example, a mathematics library that also uses degrees). The downside is that students may not be prepared when exposed to these inconsistencies when working on real-world applications. Hence it may be best to simply ensure students are well-informed about all the inconsistencies present in any libraries they might use. Students could also be given a cheat-sheet listing all inconsistencies and the contexts in which they occur, as well as simple debugging techniques which will allow them to identify whether their source code contains an error relating to an inconsistency. Another potential approach would be to design practical laboratory tasks involving such inconsistencies to give students experience with identifying and resolving related issues.

These Segments contain problems related to the Inconsistencies issue:

-Screen-Window (mostly 'Event-Driven' category, 'Screen-Window conversion' sub-category):

- [CHRISTOPHER\\_A1.ED.1."Screen-Window" \[9\]](#)
- [JOHN\\_A1.ED.1."Screen-Window convo" \[15\]](#)
- [MICHAEL\\_A1.SP.10."Circle resize" \[18\]](#)
- [IDA\\_A1.ED.1."Hit Code " \[11\]](#)
- [THOMAS\\_A3.ED.2."Forgets screen-window conversion" \[21\]](#)

-Angle-Conversion ('Spatial' category, 'Mathematical' sub-category):

- [IDA\\_A1.SP.2."Rotate Icon" \[41\]](#)
- [IDA\\_A1.SP.3."Object rotation" \[23\]](#)
- [THOMAS\\_A3.SP.2.2."Avatar Movement" \[36\]](#)
- [THOMAS\\_A3.SP.2.1."Angle Conversion" \[12\]](#)

#### 6.6.6.9 Tweaking

Tweaking is an issue which occurred mainly during spatial programming when students spent time on pixel-perfect placement of coordinates ('Spatial' category, 'Tweaking' sub-category). Unlike other issues, tweaking is not in itself a difficult task. It consists of simple trial-and-error modification of variables. Though tweaking is not in itself a significant problem and will not lead to students getting 'stuck', it does use up time that students could be using for core assignment tasks.

To reduce time spent on spatial tweaking programming, students could be provided with visualization and measurement tools via a library. This would allow them to measure distances and angles on-screen.

Tweaking can also occur in non-spatial contexts such as the choosing of a colour. For example, Segment [MICHAEL\\_A1.GL.3."Color tweak" \[11\]](#) involves a student spending 11 Changes experimenting with different colour values. Such tweaking was not common in the Project Histories analysed in this research project, but other types of tweaking may need to be addressed in different settings depending on tweaking behaviour observed in students which is in turn driven by the assessment task specification. Approaches to support student tweaking would be based on the particular setting.

These Segments contain problems related to the tweaking issue. All these Segments are categorised as 'Spatial' in their category. Most are not classified as 'Tweaking' in their sub-category since only a minority of Changes are tweaks:-almost all spatial Segments contain some amount of tweaking

-the following Segments are comprised almost entirely of tweaking:

- [MICHAEL\\_A1.SP.4."Line buttons" \[18\]](#)
- [MICHAEL\\_A1.SP.5."Positioning button text" \[26\]](#)
- [MICHAEL\\_A1.SP.7."Positioning status text" \[23\]](#)
- [MICHAEL\\_A1.SP.16."Onto Icon" \[14\]](#)
- [JOHN\\_A3.SP.3."Furniture positioning, tweaking" \[12\]](#)
- [IDA\\_A3.SP.1."Viewport" \[17\]](#)
- [IDA\\_A3.SP.2."GUI Background" \[16\]](#)
- [IDA\\_A3.SP.4.2.2."Teapot Animation" \[14\]](#)

-these Segments contain a substantial amount of tweaking:

- [JOHN\\_A3.SP.4.1."Initial naive assembly" \[26\]](#)
- [JOHN\\_A3.SP.2."Pickup Teapot anim " \[14\]](#)
- [JOHN\\_A3.SP.5.3."Swim Anim" \[120\]](#)
- [MICHAEL\\_A3.SP.1."Initial Assembly" \[47\]](#)
- [THOMAS\\_A3.SP.4."Carrying Lamp" \[36\]](#)
- [THOMAS\\_A3.SP.5.2."Proper Assembly" \[55\]](#)
- [THOMAS\\_A3.SP.6.1."Walk Animation" \[36\]](#)
- [THOMAS\\_A3.SP.6.2."Pickup Animation" \[21\]](#)
- [IDA\\_A3.SP.4.1."The Walk Animation" \[66\]](#)



- IDA\_A3.SP.4.2.1."Pickup Animation" [79]

### 6.6.7 A Model of Student Problem-Solving during Computer Graphics Programming

The last section described different issues that were identified during the analysis of student programming. This section describes a simple model (shown in Figure 67) of student programming during Computer Graphics programming. It outlines the phases at which these issues usually arise. This follows the Grounded Theory dictum that “process must be built into theory” (Corbin & Strauss, 1990), in this case by breaking the observed phenomena down into phases.

The model adds value to the theoretical themes discussed in the last section by providing a framework in which these themes can be understood as part of a single, integrated process. It can also aid further research by allowing the researcher to establish at which stage of the model a particular issue occurs. This is useful because it suggests the type of problem-solving break-down the researcher should be looking for.

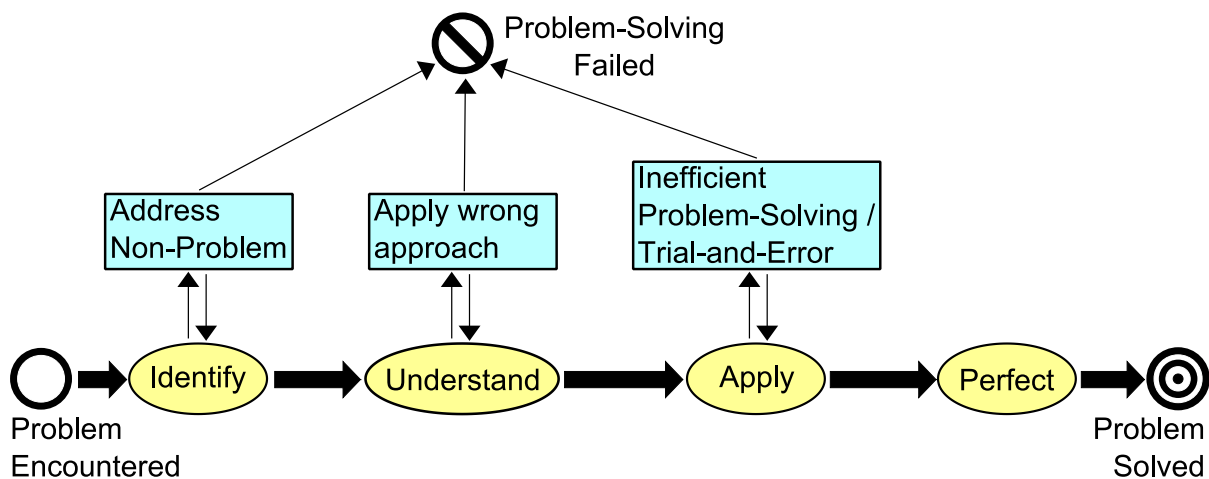


Figure 67: The four phases of the Computer Graphics student programming problem-solving process

#### 6.6.7.1 Identify Phase



Figure 68: The 'Identify' phase of problem-solving

The model consists of four phases. The first phase is the *Identify* phase (see Figure 68). The *Identify* phase involves locating the problem correctly in the source code. Misidentification leads to the student working on an unrelated piece of code without addressing the source code actually causing the problem. Successful resolution of the *Identify* phase requires the student to be able, either

through previous experience or via a process of excluding potential problem sources until the actual problem found, to identify the problem source. Ida's implementation of parent-child rotation ([IDA\\_A1.GP.7."Child object rotate"\[56\]](#)) is an example of a Segment involving a problem at the *Identify* phase. The student frequently misattributes errors relating to the loss of precision involved in the rotation to unrelated tasks, including the unit of angle measurement, leading her to several times incorrectly convert angles from radian to degree, introducing an additional problem while not addressing the actual problem.

#### 6.6.7.2 Understand Phase



Figure 69: The 'Understand' phase of problem-solving

The second phase is the *Understand* phase (see Figure 69). In the *Understand* phase the student must understand the nature of the problem and remember or develop a useful/correct concept which can solve the problem. If the student does not understand the problem, or does not know what concept to apply to resolve it, the student will apply incorrect solution approaches that do not solve the problem but may waste considerable student time and effort. A good example of a Segment involving a problem at the *Understand* phase is Michael's avatar assembly ([MICHAEL\\_A3.SP.3."Partially Proper Assembly" \[71\]](#)). The student knows the source of the problem (the transformations assembling the avatar) but does not conceptually understand how to make the assembly hierarchical by adding the child's transforms to the parent's transforms inside the parent's push/pop statements. The student spends a long time experimenting and finally producing a poor work-around which does not function correctly, hence not resolving the problem.

#### 6.6.7.3 Apply Phase



Figure 70: The 'Apply' phase of problem-solving

The third phase is the *Apply* phase (see Figure 70). The *Apply* phase requires the student to apply the concept or solution approach correctly, which requires skill in utilising the concept or solution approach. If the student does not apply the concept correctly the student cannot solve the problem efficiently and may have to resort to a trial-and-error approach. To apply concepts, the student must be skilled at their application. The type of skill varies from domain to domain. In a spatial

programming setting, the skill is spatial ability. When considering event-driven programming, the skill involves understanding program flow. When attempting to implement a Computer Science algorithm, the skill involved is algorithmic thinking. An example of a Segment involving a problem during the *Apply* phase is John's animation implementation ([JOHN\\_A3.SP.5.2."Pickup Anim" \[99\]](#)). Despite having correctly understood concepts relating to compound transformations, the student still struggles with correctly compounding transformations to achieve the desired result as he has difficulty visualizing the effect of individual transformations on the local coordinate system.

#### 6.6.7.4 Perfect Phase



**Figure 71: The 'Perfect' phase of problem-solving**

The final phase is the *Perfect* phase (see Figure 71), which involves modifying a correct solution in minor ways to make it more aesthetically or conceptually pleasing. In spatial programming, this may involve small modifications to coordinates for pixel-perfect positioning. In other programming contexts, it may involve simplification of convoluted source code, for example. Students cannot fail at this phase as their solution is already essentially correct (unless they unwittingly introduce a new error). Instead, they will simply continue tweaking until they are pleased with the quality of their solution, at which point they will move on to the next task or problem.

#### 6.6.7.5 Moving through the model during problem-solving

When a student works on a problem the student moves through each of the phases of the model in turn. Not all of the phases or even any of the phases need cause any issues, in which case the student moves from problem to solution quickly and without error. Also, some phases may be completed without any programming such as when a student identifies a known error without the need for debugging. If any phase does cause an issue, the student will experience the associated difficulty (addressing a non-problem, applying a wrong solution approach, or solving the problem inefficiently). The student will then either overcome the associated difficulty and move to the next phase, eventually solving the problem, or the student will give up and either leave the problem unsolved or the student will 'cheat' by circumventing the problem (e.g. Christopher's animation implementation in Segment [CHRISTOPHER\\_A3.SP.2.1."Pickup Anim" \[24\]](#) only uses transformations that do not show the incorrect assembly).

If a student does not know a concept, this may also affect the student's ability to identify the problem in the *Identify* stage since the student may know what they are looking for. This is not

always the case. For example, if a student has only just introduced some source code, the student may know for certain that it is the source of the problem but not know how to go about solving it, which means the student is stuck at the *Understand* phase.

A Segment or task may involve many smaller sub-problems, each involving the student moving through the phases of problem-solving. For example, when implementing animations students tended to develop animations for each limb in turn. The students would first experiment with transformations until the correct result was achieved (*Apply* phase), then tweak the amount of transformation until the result was aesthetically pleasing (*Perfect* phase). As a result, such Segments often consisted of many Apply-Perfect phases.

Such problems also often did not cause any issues during the *Identify* or *Understand* phases, as the students understood the source of the problem (the transformations they had just introduced) as well as the concepts involved in addressing it (the production of compound transformations that they had already applied during avatar assembly). This is often the case; many times students will be aware of the root of the problem and know a solution approach for the problem in which case they will not produce any programming actions related to the 'Identify' or 'Understand' phases. The next section will showcase an example of a problem which involved explicit programming actions for all phases of the model.

Issues described in the previous section usually occur in a particular phase of the student problem-solving process.

Issues that affect the Identification phase:

- Difficult-to-Visualize
- OpenGL Pipeline
- Interplay of Problem Types
- Inconsistencies

Issues that affect the concept phase:

- Conceptual
- Programming Language & Syntax

Issues that affect the Apply phase:

- Cognitive Difficulty of Spatial programming
- Program-Flow Understanding
- Algorithm Understanding

Issues that underlie the Perfect phase:

- Tweaking

#### 6.6.7.6 Illustration of the Model through an example

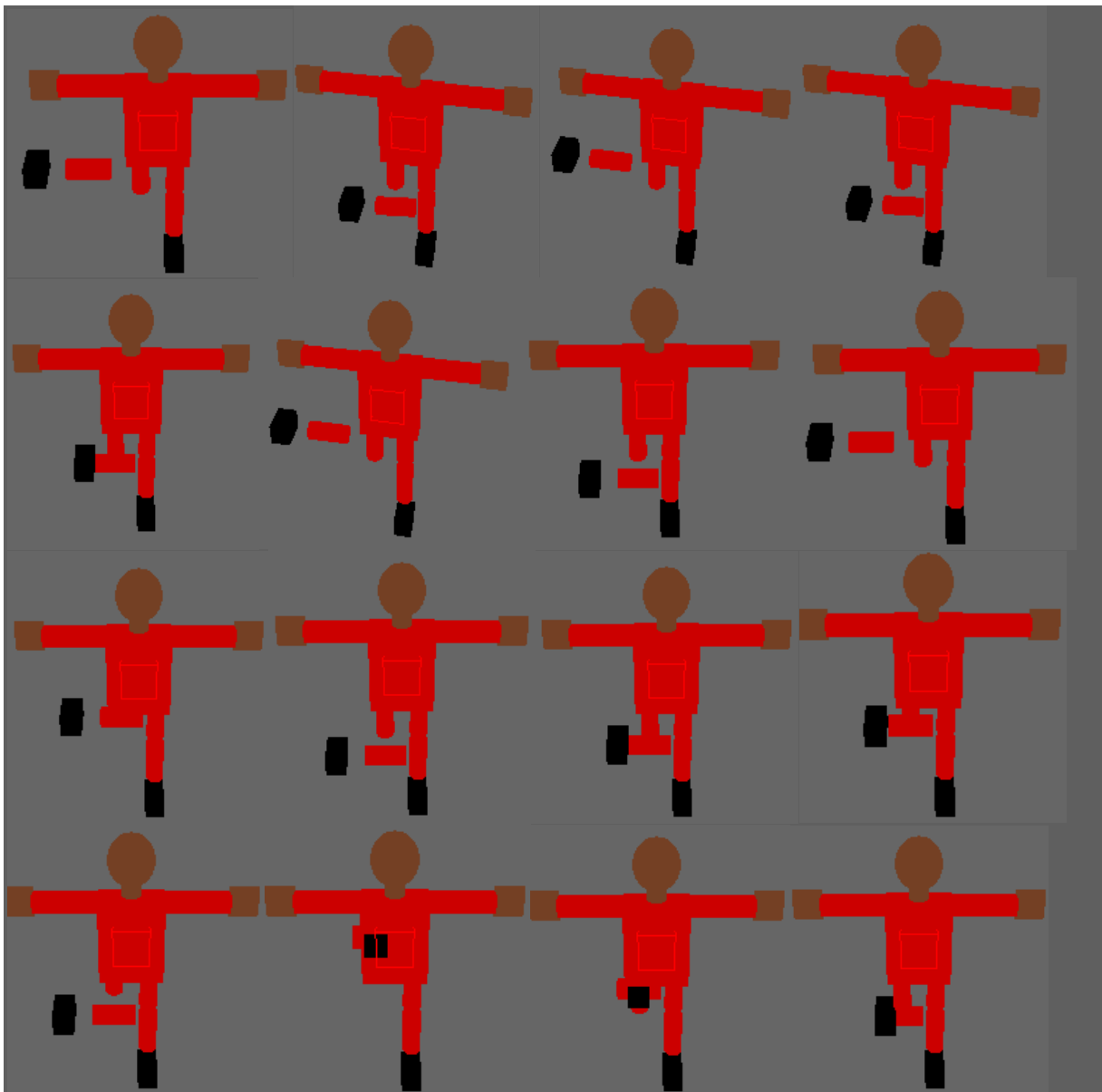
The following is an example of two related tasks which elicit programming actions for each phase of the proposed model. While the example involves each of the phases of the model, most Segments do not in fact contain actions for each model stage. In these cases several phases are completed implicitly. For example, the student may start working on a new problem and hence know which instructions are related to the problem. The student will then skip the 'Identify' phase. The student may then remember the correct solution approach or develop such an approach quickly in her head. In this case, the student will skip the 'Understand' phase. The solution approach may then be applied quickly and effectively, thereby producing only very few actions related to the 'Apply' phase. The implemented solution may be simple enough to be of sufficient quality after implementation to not require any action in the 'Perfect' phase. In such a case, the only trace of the problem would be a small number of Changes (perhaps only a single Change) associated with the 'Apply' phase.

Thomas first works on assembly in Segment [THOMAS\\_A3.SP.5.1."Naive Assembly" \[39\]](#); at that point the student produces a naïve assembly in which movement of lower limbs will not move upper limbs as should be the case. In Segment [THOMAS\\_A3.SP.1."Initial Anim Experiment" \[10\]](#) the student attempts to implement a first animation. Since the assembly is naïve the animation does not work correctly and the avatar 'breaks apart' during the animation as shown in Figure 72. The student tweaks the transformations to discover what's wrong. This is the 'Identify' phase in which the student tries to understand the origin of the problem. At (309) the student has correctly identified the assembly as the problem (rather than the animation transformations) and commences work on repairing the assembly.



**Figure 72: Limbs break apart during the "Initial Anim Experiment" Segment due to a non-hierarchical assembly, leading the student to pinpoint the problem as relating to the assembly transformation calls in the 'Identify' phase**

Thomas then moves on to the proper assembly in Segment [THOMAS\\_A3.SP.5.2."Proper Assembly"](#) [55]. From (311-332) the student attempts different approaches to assembling the avatar, using different orderings of transformations as shown in Figure 73. This is the 'Understand' phase in which the student seeks to develop a correct concept to apply to solve the problem. At (332) the student develops a correct assembly for the leg. This means the student has developed a correct concept for assembly, thereby completing the 'understand' phase.



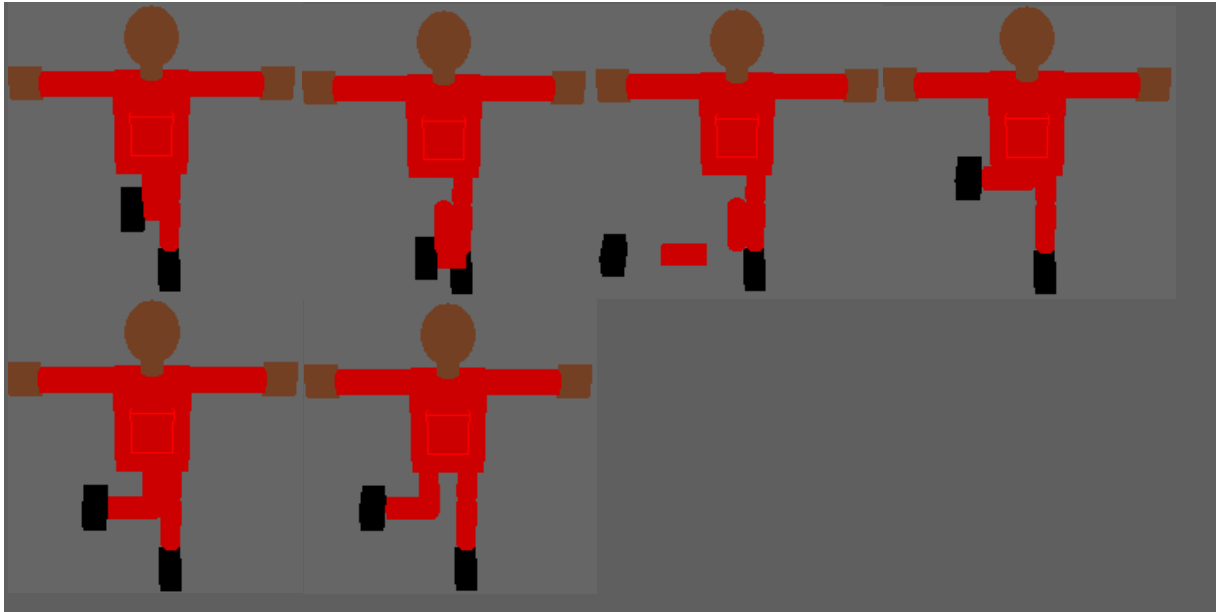


Figure 73: During the 'Understand' phase, the student trials different assembly approaches from (309-330) lead to different incorrect assemblies as shown, with the lower leg finally rotating correctly about the upper leg in the final Change at (331)

The student then moves to the 'Apply' phase, applying the developed assembly method to the other limbs during Changes (334-336, 338-367), producing a correct assembly. Application of the method to the right arm, requiring some experimentation during the 'Apply' phase, is shown in Figure 74.

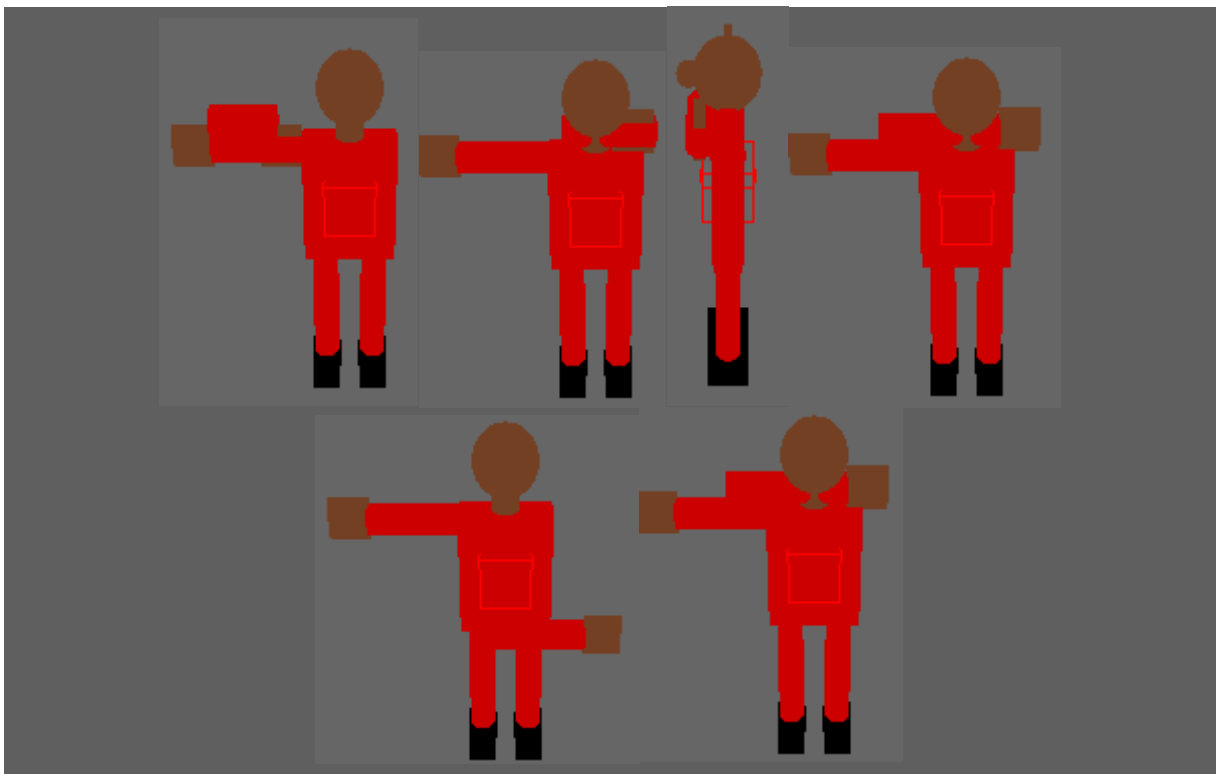
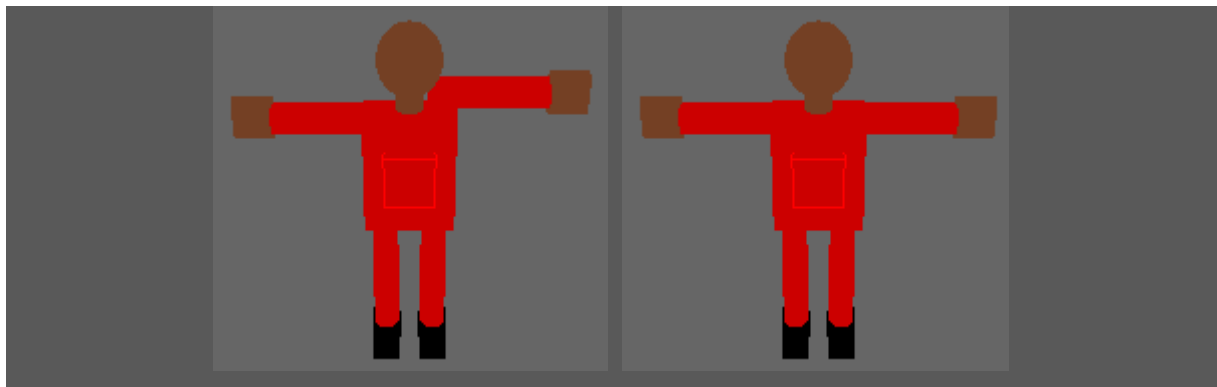


Figure 74: During the 'Apply phase' the approach developed while assembling the leg is applied to the arm, again requiring some experimentation in the 'Apply' phase

During the ‘Understand’ and ‘Apply’ phases as the student works on limbs he frequently corrects the position by small amounts to tweak the limb into place, placing those actions into the ‘Perfect’ phase. An example of a ‘Perfect’ action, slightly moving the already essentially correctly placed arm to slot it into place, is shown in Figure 75. As this shows, larger problems often consist of sub-problems (such as the assembly of a limb) which involve their own progression through the phases of the model. Such progression is often implicit for most phases; for example, while assembling other limbs using a developed method the student will only have to move through the ‘Apply’ and ‘Perfect’ phases since the other phases have been successfully completed for the first assembled limb and can be carried over to the assembly of the new limb.



**Figure 75: In an example of a ‘Perfect’ phase the arm is moved into place at Change (358), resulting in a correct avatar assembly as shown on the right**

When the student begins work on a walk animation utilising the previously implemented assembly in Segment [THOMAS\\_A3.SP.6.1."Walk Animation" \[36\]](#) the student moves straight to the ‘Apply’ phase as the problem-source is well known (the transformations involved in the assembly). The student can thus implicitly progress through the ‘identify’ phase. Since the student has developed an understanding of how to compound transformations, the student can also implicitly move through the ‘Understand’ phase without any further work. Implementation of animations hence involves cycles of ‘Apply’ and ‘Perfect’ in which the student creates an animation for some limbs, correcting errors in the ‘Apply’ phase and then tweaking the values during the ‘Perfect’ phase to make the outcome aesthetically pleasing.

As the number of screen captures for the different phases graphically illustrate, the development of the concept during the ‘Understand’ phase was the most difficult part of the problem-solving process in this instance, with the ‘Apply’ phase also requiring some experimentation. Both the



'Identify' and 'Perfect' phases were completed relatively swiftly. In the case of the 'Perfect' phase, this was because part of the 'perfection' occurred during the initial assembly in [THOMAS\\_A3.SP.5.1."Naive Assembly" \[39\]](#). The 'Identify' phase was relatively brief because the only two potential problem sources were the implementation of the assembly and the implementation of the animation, and after excluding the animation as the problem source the student was left with only the correct option of identifying the assembly as the problem source. In other problems, the distribution was different. For example in Segments related to implementation of animations such as [JOHN\\_A3.SP.5.2."Pickup Anim" \[99\]](#) the great majority of Changes were spent in the 'Apply' phase. In a problem Segment dealing with a difficult-to-understand error message relating to a misplaced bracket ([CHRISTOPHER\\_A1.GP.7."Confusing function bracket error" \[25\]](#)) the student spent most of the time in the 'Identify' phase; after identification of the faulty code the problem was soon fixed. While students in the examined Project Histories rarely spent the majority of their time in the 'Perfect' phase, some Segments (usually smaller ones) such as [MICHAEL\\_A1.SP.7."Positioning status text" \[23\]](#) relating to the positioning of a text box do consist mostly of 'Perfect' Changes, in this case occurring as the student nudges a text box into position.

## 6.7 Analysis of Segment Features

The previous section presented results of the primarily qualitative analysis of Segment contents. This form of analysis makes good use of the richness and depth of source-code level data. This section will complement the analysis of Segment contents with a mainly quantitative analysis a Segment feature, the difference in average time spent on Segment Changes belonging to different classification categories (*Segment-Change Average Time*). Because the findings were deemed weaker than those that form the remainder of the body of this thesis the section was shortened considerably; the full content of the section is available in the appendix, Section 9.7.8.

The SCORE Analyser produces Excel spreadsheets containing data for both individual Changes as well as data for Segments. Segment data is derived by calculating average and total values for Change metrics such as time or number of modifications (only time data is analysed). The complete set of Segments does not necessarily (or usually) include all Changes that are part of a Project History, only those that form sets containing ten or more related Changes since only such sets are used to produce Segments. Changes that are not part of a Segment are not included in any analysis of Segment metrics. These Changes are included in the analysis of individual Change metrics presented in Appendix Section 9.7.8.1. The mechanism used for generation of this data is described in technical detail in Section 6.4.1.2. As is discussed in that section, the calculation of Change time involves pitfalls relating to student work patterns since there is no way to be sure whether a student was

working on the project at any given time between two Changes. The heuristic algorithm designed to circumvent these issues is described in Section 6.4.1.2. As the calculation of time data involves heuristics, the time metric is an estimate and not a precise measurement.

In the Segment classification scheme used to categorise Segments relating to spatial programming are captured in the same ‘Spatial’ category. In the following quantitative analysis ‘Spatial’ Segments were split into two sub-categories, ‘Transform Spatial’ and ‘Non-Transform Spatial’. The ‘Transform Spatial’ category includes the Segments belonging to the ‘Order of Transformations’ and ‘Three-dimensional transformations’ category whereas the ‘Non-Transform Spatial’ category includes Segments belonging to all other ‘Spatial’ sub-categories (most of them involving two-dimensional spatial programming) as described in Section 6.3.4. ‘Transform Spatial’ and ‘Non-Transform Spatial’ Segments were analysed separately. This is because of an intuition that two-dimensional spatial problems may present a different challenge level. This may mean that difference in dimensionality leads to different problem-solving approaches. Classification categories which include less than ten Segments are excluded from the ANOVA analysis presented in Section 9.7.8.3.2 due to the low sample size; the excluded categories are ‘General OpenGL’, ‘Lighting’ and ‘Animation’.

Descriptive statistics for Average Time for different classification categories are shown in Table 91. Pipeline and Spatial Segments have the shortest Average Time with 57.55 and 60.61 seconds respectively, with ‘General Programming’ and ‘TransformationSpatial’ Segments having means close to the global mean at 64.77 and 69.58 seconds respectively. ‘Event-Driven’ Segments have a high average ‘Average Time’ of 80.82 seconds. Segments of type ‘View’ have the highest average ‘Average Time’ of 92.28 seconds, 25.01 seconds above the global mean. This means that assuming that time spent between Changes involves the student thinking about the problem being solved, ‘View’ Segments and (to a lesser extent) ‘Event-Driven’ Segments seem to involve the most cognitive effort. ‘Pipeline’ Segments and ‘Spatial’ Segments seem to frequently be solved more often using trial-and-error approaches with shorter thinking breaks.

**Table 21: Descriptive Statistics for Segment Average Time per Change broken down by Category**

	N	mean	diff. mean	Sd	Median	Range	Skew	kurtosis	se
Event-Driven	22	80.82	13.55	39.55	79.06	181.29	1.64	3.27	8.43
GeneralProg	42	64.77	-2.5	32.59	61.29	129.36	0.46	-0.75	5.03
Pipeline	10	57.55	-9.72	14.07	56.94	49.31	-0.24	-0.78	4.45
Spatial	59	60.61	-6.66	32.72	50.85	152.99	1.34	1.63	4.26
TransSpatial	20	69.58	2.31	30.14	74.07	110.69	0.41	-0.83	6.74

View	10	92.28	25.01	34.71	80.72	94.21	0.27	-1.77	10.98
------	----	-------	-------	-------	-------	-------	------	-------	-------

While there seems to be a fairly large difference in means, standard deviation is also quite high for most categories. This may suggest that problems belonging to the same category involve different levels of challenge, or that different students take different approaches to solving problems of the same category, or that the classification categories are not precise enough or a combination of such factors.

To discover whether the difference in mean 'Average Times' between Segments of different classification categories is significant, Analysis of Variance comparing the means of Average Times between categories was conducted. Since analysis of Segment Average Time per Change was shown to be logarithmically distributed (the full analysis is presented in the appendix, Section 9.7.8.7.1) a logarithmic transformation was applied to the data since normal distribution is a prerequisite for the application of the ANOVA test.

Analysis of Variance between Segment classification categories yields a p-value of 0.018 as shown in Table 95. Residuals are presented in Table 93, whereas t and p-values for all categories are presented in Table 94. Since the p-value is  $< 0.05$ , the null hypothesis that there is no difference in Average Time means between different classification categories is rejected. The adjusted  $r^2 = 0.0532$ , meaning the effect is rather small, and much of the variance in Average Times is not explained by Segment classification category.

**Table 22: ANOVA Residuals**

Residuals:				
Min	1Q	Median	3Q	Max
-0.47598	-0.14114	0.00192	0.14597	0.5144

**Table 23: ANOVA t and p-values**

	Estimate	Std. Error	t-value	Pr(> t )
(Intercept)	1.86631	0.04464	41.804	$> 2E-16$
General Programming	-0.11412	0.05511	-2.071	0.04002
Pipeline	-0.11958	0.07986	-1.497	0.1363
Non-Transform Spatial	-0.13812	0.05231	-2.64	0.00911
TransformSpatial	-0.06479	0.0647	-1.001	0.31814

View	0.07064	0.07986	0.885	0.37777
------	---------	---------	-------	---------

**Table 24: ANOVA r-squared and p-value**

Residual standard error: 0.2094 on 157 degrees of freedom  
Multiple R-squared: 0.08247  
Adjusted R-squared: 0.05325  
F-statistic: 2.822 on 5 and 157 DF  
p-value: 0.01806

While the correlation is weak it is nevertheless statistically significant. This means that there is a difference in the means of Average Time based on Segment classification category.

To determine for which category 'Segment-Change Average Time' means differ significantly from other categories a Tukey's Range Test (aka Tukey's Honest Significant Difference test) was performed on the data. Results are presented in Table 96. The results show that the mean of View Segments is significantly different to (greater than) that of Spatial Segments (at  $p \leq 0.05$ ). While the p-value of 0.13 for between 'View' and 'General Programming' is not significant, it is fairly small. Given that ANOVA has already shown that Average Time varies by Segment classification category it is likely that 'View' Segments on average have a higher mean Average Time than 'General Programming' Segments. The only other near-significant difference ( $p=0.09$ ) is between 'Event-Driven' Segments (second-largest mean Average Time) and 'Non-Transform Spatial' Segments (smallest mean Average Time). Again, while not reaching significance, given that there is a significant difference between means based on ANOVA the small p-value is at least indicative that 'Event-Driven' Segments have a higher Average Time on average than Spatial Segments.

**Table 25: Tukey's Range Test (aka Tukey's Honest Significance Difference)**

	Event-Driven	GeneralProg	Pipeline	Spatial	TransSpatial
General Programming	0.31	-	1.00	0.99	0.95
Pipeline	0.67	1.00	-	1.00	0.98
Spatial	0.09	0.99	1.00	-	0.75
Non-Transform Spatial	0.92	0.95	0.98	0.75	-
View	0.95	0.13	0.33	0.05	0.55

In summary, it seems that View Segments (and probably 'Event-Driven' Segments) involve a more deep-thinking solution approach, while 'Non-Transform Spatial' Segments (and probably 'General

Programming' Segments) involve a somewhat more trial-and-error approach, with 'Pipeline' and 'TransformSpatial' Segments involving more balanced approaches.

This leaves open the question as to why 'View' Segments have such a high mean average time (the highest), high enough to be significantly higher than that of 'Non-Transform Spatial' Segments, whereas 'Transform Spatial' Segments, which also involve three-dimensional spatial programming have a much lower average time. It could indicate that students use deeper-thinking approaches when working on 'View' Segments than when working on other Segments involving three-dimensional spatial programming. A detailed examination of selected Segments of both types was carried out, but due to time limitations this examination could be carried out for only a small set of Segments. Since the results were deemed weaker than those that form the body of the thesis they were placed in the appendix, Section 9.7.8.4. This analysis lent support to the hypothesis that students perform more deep-thinking programming actions when working on 'View' Segments. However, given the limitations of the analysis further research is necessary to validate this hypothesis.

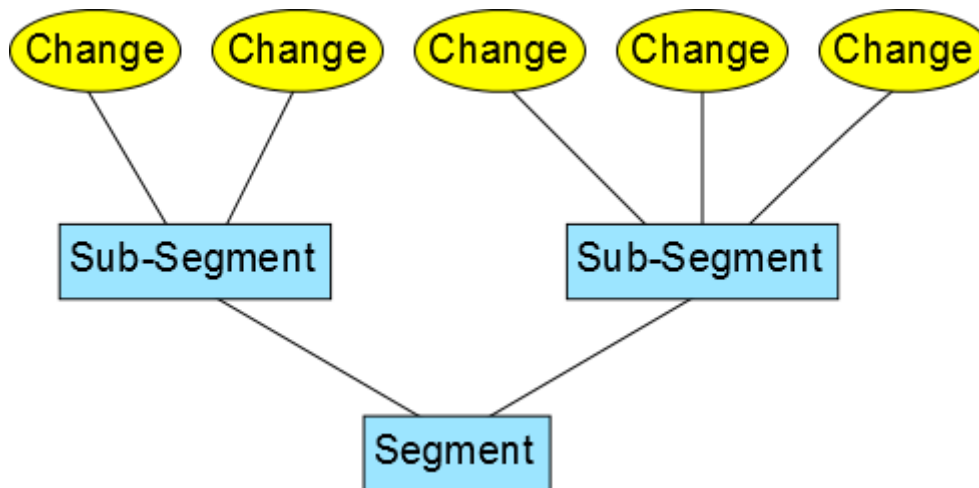
The analysis did not produce clear-cut results as would have been desirable because of the relatively small data set. Also, the small data set did not permit for a break-down into the more precise classification sub-categories. Nevertheless the results do serve to illustrate the potential of the analysis of Segment features which in other contexts may prove more fruitful.

## **6.8 Updating the Segment-Coding method for future research**

This section proposes changes to the coding model utilised in the Segment-Coding method based on experiences with the method gained during its application during this research project.

### **6.8.1 Updating the coding method**

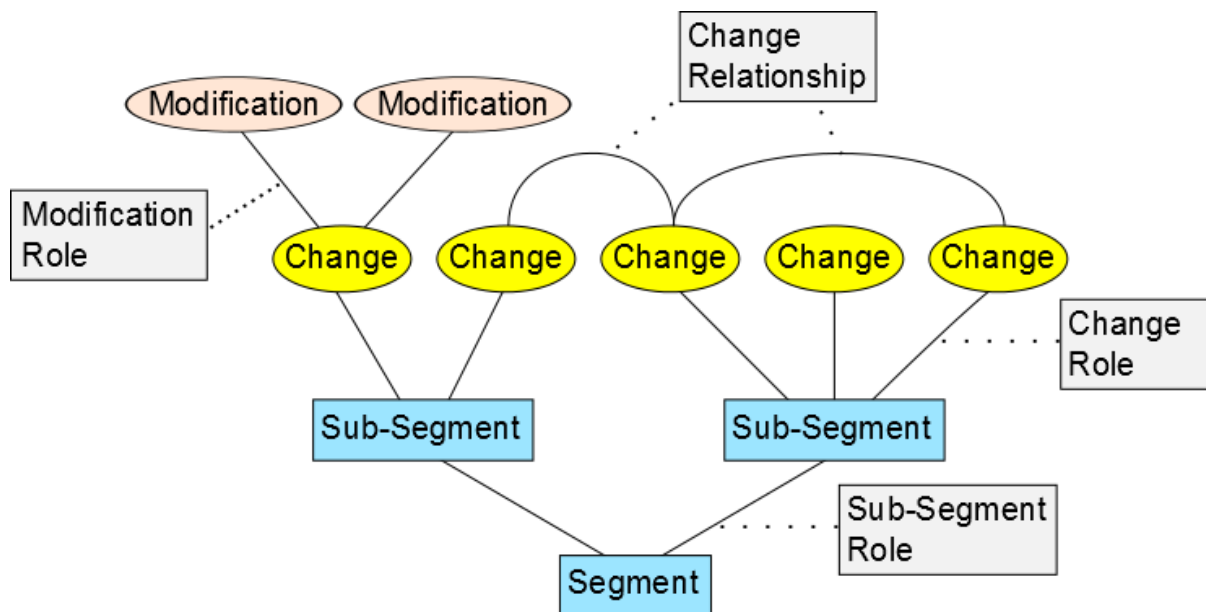
While the Segment-Coding method outlined in this chapter proved very useful in structuring and breaking down Project Histories for analysis, it did have some limitations. The current model used in storing Segment and Change data is shown in Figure 76. The only formal relationship is the relationship between Changes and the Segment to which they are assigned. Other data must be captured in the Change's or Segment's notes. One example of such data is the relationships between Changes such as when one Change fixes the problem caused by another Change. Another is the role of individual Changes in the Segment. For example, spatial Segments include tweaking Changes that make minor modifications to coordinates. It would be useful to be able to separate these Changes from the non-tweaking Changes which make major modifications, such as changing an axis of rotation.



**Figure 76: The currently implemented Change-Segment data model**

This informal method of storing the details of relationships of individual Changes or of a Segment's structure using notes/memos was found to be lacking. Using this approach understanding the content of a Segment requires reading and understanding of an informal textual description. The informal nature of the textual description makes it likelier that the level of detail and style of description will vary from Segment to Segment. Also, reading the description to extract Change numbers which contain significant modifications is cumbersome. Textual descriptions are also not machine-accessible and thus cannot be used in data visualizations or machine-evaluation of any kind.

To address these problems a future classification method should include formal mechanisms for capturing such relationships. A new data model fulfilling this requirement to be implemented in a future version of the SCORE Analyser is shown in Figure 77. In addition to Changes and Segments, the new model also incorporates individual Modifications (changes to lines occurring from one version to the next).



**Figure 77: A new, more fine-grained Modification-Change-Subsegment-Segment data model**

This new model explicitly stores individual Modifications as children of the Changes in which they occur. The *Modification Role* which is associated with this relationship is used to describe the role of the modification in the Change. This allows the researcher to formally note important modifications as they occur.

The model also includes *Change Roles* that are assigned as a relationship between a Change and its parent Segment or Sub-Segment. These *Change Roles* can capture the role that the Change plays in the Segment. For example, Changes which produce key errors or new problem-solving approaches could be classified as such, and textual descriptions detailing their role could be assigned to them. Such Changes could then also be linked to other related Changes. For example, a key error Change could be associated with a 'Problem Solved' link to another Change in which the problem introduced in that Change is solved. Such descriptions are currently stored informally as text in the parent Segment's description, but their formalisation would make the application of the classification scheme more reliable. This would also allow for graphical representation of such relationships to give the researcher an overview of the inner workings of a Segment.

Relationships between Segments and Sub-Segments are currently also stored as text in either the parent's or Sub-segment's description. In the new model, these are stored as *Sub-Segment roles*. For example, a Segment may consist of a Sub-segment containing the 'Initial experimentation', a 'First Solution attempt', a 'second solution attempt' and finally a 'clean-up and minor modifications' Sub-segment.

The third type of relationship available in the new model is the *Change relationship*. This encodes relationships between individual Changes. For example, one Change may be connected to another Change via a 'corrects-problem' relationship, indicating that the Change addresses an error introduced by the second Change.

In this new model Segments still play the same role of providing high-level context by grouping Changes related to a specific problem or task. However, the new relationships allow the capturing of more fine-grained relationships. Such relationships could then also be coded individually, allowing for the coding at different levels of abstraction, which in turn will provide better resolution for the application of the GT methodology. Coding would occur at the level of abstraction appropriate for the question to be answered. For example, high-level analysis may lead the researcher to want to investigate the way in which Segments of a certain type are developed, leading him to code these Segments at the Change level or even at the Modification level. Other Segments not associated with 'interesting' problems may be described only at the Segment level, thereby saving time. The practical implementation and evaluation of this new coding model is left to future research.

### 6.8.2 Evaluating other metrics for measuring Segment significance

Utilising number of Changes as a measure of Segment significance might be misleading for some rote programming activities. For example, the implementation of a simple algorithm or data structure may involve the production of a considerable amount of source code over a large number of Changes. However, if the source code is in working or near-working condition after being added, then it probably should not be classified as 'difficult', even if the production does take several Changes. It is still reasonable to suggest that the number of Changes or the time spent on these Changes measures effort, since students had to invest time and energy in creating the source code. However, effort may not always correlate with significance since some tasks are more 'verbose' in terms of requiring the production of more (but simpler) code.

An alternative way of measuring Segment significance would be to evaluate the amount of Change each line of source code produced during the Segment undergoes, and to weigh the contribution of lines of source code based on how frequently they change. For example, a Segment which involves the addition of ten lines of source code which are on average modified twice would be classified as less difficult than a Segment which involves the addition of five lines of source code which on average are modified five times.

A simple way of calculating this metric would be to count up the total number of modifications. For ten added lines of source code modified twice this count would be  $10 * 2 = 20$ , whereas for five lines



modified five times the count would be  $5 * 5 = 25$ . Intuition suggests that the relationship might not be linear, that is to say if a line of source code is modified many times (say, a line is modified ten times) then it may be much more important than a larger set of lines modified fewer times (say, five lines modified twice each) so perhaps the metric should be weighted in some way to give a larger weight to more modifications. This could be done by multiplying the modification factor by a constant to the power of the number of line modifications, thereby exponentially valuing modifications. The formula for this would be:

$$\text{NUMBER\_OF\_MODIFICATIONS} * \text{CONSTANT}^{\text{NUMBER\_OF\_MODIFICATIONS}}.$$

For example, using a constant of 1.1 a source code line modified ten times could be calculated as being worth  $(10 * 1.1^{10}) = 26$  modifications. Utilising this approach would over-value ‘tweaking’ Segments (see Section 6.6.4.3) in which small ‘perfecting’ modifications are made to lines of source code, since there might be many such modifications, but they in fact rote modifications.

Another problem for approaches utilising modifications is that these approaches will greatly overvalue any use of bulk modifications, such as those produced by the ‘search and replace’ functionality which is sometimes strategically used by students. Such bulk modifications produce large numbers of modifications to individual lines, but these modifications are all inherently part of a single underlying logical modification. To counter-balance this, modifications could be weighted based on how many modifications occur in the Change. For example, if ten modifications occur in the same Change, each of these could be assigned one-tenth of the value as compared to a single modification occurring during a single Change.

These proposed alternatives to the current Segment significance metric may prove to perform better when properly evaluated. On the other hand, they are somewhat more complex and more prone to failure in certain contexts, such as the bulk-modification and tweaking contexts mentioned earlier (however, the tweaking example also undermines the number of Changes metric to some extent).

## 6.9 Limitations of the Segment-Coding method

### 6.9.1 Measurement of Time Taken per Change

One limitation of the current Segment-Coding approach lies in the way that Time between Changes is calculated. Currently it is calculated as the difference between the timestamp of the current and the previous version, and it is meant to measure the ‘mental effort’. However, in practice time taken will include the manual effort required to type lines of source code.

In order to exclude this manual effort from the time taken measurement, a future approach to calculating time taken could deduct a value from the time based on the number of characters typed in that Change or the number of lines modified. This value could be the average time taken to type a character. This value could be based on calculating the total number of typed characters and the total time taken for all Changes in the Project History and then dividing the time by the number of characters.

The implementation of such an adjustment algorithm would require evaluation of the resulting time taken to ensure that it better reflects the probable mental effort invested into a Change than the original raw difference between timestamps.

### **6.9.2 Time-intensive nature of analysis**

The second limitation relates to the time-intensive nature of the Segment-Coding method. While the Segment-Coding method required significantly less time for analysis of an assignment, it was still time-intensive to carry out. In part, this was due to the method still evolving during its application and due to initial application being carried out without the full set of tools and Segment-Coding rules having been developed. In addition, comprehension of student programming is a challenging task and it will inevitably be more time-consuming than fully automated but shallower approaches such as machine-identification of syntax errors used in previous work such as work by Jadud (2005). On the other hand, the method produces much richer data.

If the method is very time-consuming then it can feasibly be used only on a small set of Project Histories and researchers may decide against applying the method, choosing a different method with less overhead. Improvements to the method which decrease this time and effort overhead will increase the number of Project Histories that can be analysed. Chapter 7 presents the development of such a method. The method machine-segments Project Histories and can hence reduce the workload placed on the researcher by reducing the time required for Segment-Coding of the Project History.

## **6.10 Segment-Coding Conclusion**

This chapter described a method for the analysis of Project Histories called Segment-Coding. Compared to the Change-Coding method presented in the last chapter which involves the coding of individual Changes, this method involves coding of 'Segments' of related Changes. The chapter described the method, presented rules for the application of the method, results produced by the application of the method as well as interpretation of said results.

The Segment-Coding method utilises Detailed Analysis (see Section 6.2) which includes the Change Browser and Line History views based on the concept of Line Histories, which enable the user to perform context-aware analysis effectively for large Project Histories (Section 6.2).

The approach to analysis involved rules for the application of the Segment-Coding method as well as the development of sub-categories of the classification categories discussed in Section 5.2.2 developed during analysis of Segment contents discussed later in the chapter (see Section 6.3 for details). An example is the break-down of the Spatial category into the sub-categories of 'Two-Dimensional Coordinate Drawing / Primitive Creation', 'Two-Dimensional Coordinate Manipulation', 'Three-Dimensional Coordinate Drawing / Primitive Creation', 'Order of Transformations', 'Three-dimensional Transformations' and 'Mathematical'. SCORE Analyser facilities for the segmenting of Project Histories and application of classification categories and sub-categories was presented and complemented by a walk-through of how these facilities were used in practice to perform Segment-Coding (see Section 6.4).

The Segment-Coding method was applied to the ten Project Histories selected for analysis (see Section 3.6.3 for a description of these Project Histories). An initial examination of the results based on the size of Segments and the distribution of Segments between the different classification sub-categories showed that 'Spatial' problems, especially those relating to three-dimensional spatial tasks, proved difficult for all students to address. The examination also showed that problem Segments falling into the 'General Programming' category caused significant difficulty for some students, but in a less predictable fashion. The difficulties encountered by students arose from details of their own particular implementation of the assignment rather than from the assignment specification itself as was the case with 'Spatial'-categorised problems (see Section 6.5).

A more thorough qualitative analysis of the contents of identified Segments (see Section 6.6 ) produced detailed insights into the problems that students implementing Computer Graphics programs face. Analysis showed that all students struggled most significantly with three spatial tasks. Two of these tasks occurred in Assignment 3 and involved the assembly of an avatar involving the use of transformations to produce a hierarchical model and the implementation of animations utilising the assembled avatar. The third occurred in Assignment 1 and involved the implementation of child objects about parent objects, but compared to the other two tasks this task only involved two-dimensional spatial programming. Detailed examination showed that the difficulty of this problem was compounded by the fact that several problems co-occurred while students attempted to implement this functionality. Comparison and detailed analysis of Segment contents revealed themes underlying the problems faced by students in these Segments. These themes were distilled

into a set of 'issues' relating to student problem-solving, presented in Section 6.6.6. The following nine issues were identified: Conceptual, Difficult-to-Visualize, Cognitive Difficulty of Spatial Programming, OpenGL Pipeline Black-box, Program-Flow Understanding, Interplay of different Problems, Programming Language Syntax and Semantic Concepts, Inconsistencies and Tweaking.

Furthermore, several different types of concepts and misconceptions relating to the 'Conceptual' issue in a Computer Graphics context were discovered (other contexts would give rise to different concepts): Transformation Concepts, Mathematical Concepts, View Concepts, Draw-not-Store Misconception, Time-based Behaviour Concepts.

The discussion section also presented several scaffolding approaches which may potentially help students and improve learning outcomes.

The identified issues in turn form the basis of a model of student problem-solving (see Section 6.6.7). Each issue affects a particular phase of this model. The model has four phases, the 'Identify' phase in which the root cause of the problem is identified. The 'Understand' phase in which a solution concept / approach is remembered or developed. The 'Apply' phase during which this approach or concept is applied. Finally, during the 'Perfect' phase the (already correct and functional) solution is improved. Failure to complete a phase leads to a break-down in problem-solving associated with that phase which the student must resolve to progress towards solving the problem. The analysis of Segment contents provides answers to research question **RQ1** regarding the nature of student problems with Computer Graphics programming, as well as providing evidence for answering research question **RQ2** regarding the challenging nature of spatial programming in the affirmative.

An analysis of Segment features (see Section 6.7) delved into the difference between different types of spatial programming problems. The analysis showed that 'View' problems have the longest average time spent per Segment Change. This time is longer than that of other three-dimensional spatial problems. A subsequent investigation of selected Segments in detail (Appendix Section 9.7.8.4) suggested that this is due to the difficulty of utilising OpenGL to visualize a viewing volume, whereas it is relatively simple to visualize the effect of a transformation simply by applying it. This research method did not provide the same quality of results produced by the analysis of Segment contents. It is nevertheless interesting that based on results of this limited analysis the use of problems which students cannot utilise their applications to visualize may force them to perform more mental visualization. This may be beneficial in terms of improving their spatial programming ability. On the other hand it may also cause students with insufficient spatial ability for such

visualization to give up. Further research should be conducted to verify these results, and to clarify the extent to which the different styles of problem-solving approach affect student learning.

Experiences with application of the Segment-Coding analysis method led to the development of an updated model of Segment-Coding to be used in future research which complements the current contextual nature of the method with ways of storing fine-grained data relating to relationships between individual Changes (see Section 6.8).

The Segment-Coding analysis method was found to have two primary limitations in practice (see (Section 6.9). One relates to the way that time per Change is calculated, but given that the analysis of Segment contents, which forms the backbone of this chapter, does not rely on this metric this limitation did not provide a significant stumbling block during this research project. The second limitation relates to the time-intensive nature of analysis using the Segment-Coding method. This limitation meant that only a subset of collected Project Histories could be analysed during this research project. The Segment-Coding method was successful in producing these results. This provides evidence answering research question **RQ3** concerning the development of an effective and reliable method of fine-grained source-code level Project History analysis in the affirmative. However, the time-consuming nature of the method may make it unfeasible for some researchers, and may impose too strict a limit on the number of student projects that can be analysed in a given timeframe. To address this limitation in order to be able to answer the research question regarding a successful method of Project History analysis in the affirmative, the next chapter (Chapter 7) describes a machine approach to Segment-Coding designed to decrease the time required for analysis.

# 7 Line History Generation and Machine-Segmenting

---

## 7.1 Line History Generation and Machine-Segmenting Introduction

This chapter will present the two software engineering contributions to Computer Science Education made by this thesis. These are based on software-engineering research as discussed in the Literature Review (see Section 2.5). The first is the Line History generation algorithm which was introduced in the Segment-Coding chapter in Section 6.2.1. It tracks all modifications made to individual line of source code which occur in a Project History. Line Histories form the basis of the most effective Machine-Segmenting methods discussed later in this chapter. However, their application is not limited to the machine-generation of Segments. Line Histories also enable the SCORE Analyser to provide summary views of how programs have changed (as presented in 6.2.2) as well as enriching the main Diff view presented in Section 4.4.1 by enabling it to show line modifications in terms of moved lines, changed lines and ‘ghost’ lines. A normal diff algorithm would only show lines as added, deleted or maintained. These views allow the researcher to see the context in which a modification occurs, thereby significantly reducing the effort required to understand Changes and the individual modifications that comprise them. Line Histories may also have a range of other potential applications such as forming the basis for a method of plagiarism detection. This chapter presents both the algorithm used in the generation of Line Histories as well as an evaluation of the generation algorithm. The evaluation of the Line History generation algorithm is based on the *Information Retrieval* metrics of precision and recall (as described in Section 7.2 and Section 7.3).

The second software-engineering contribution discussed in this chapter is Machine-Segmenting. The Segment-Coding method’s most significant drawback is its time-intensive nature. The longer it takes to analyse Project Histories, the fewer Project Histories can be analysed during the course of a research project. In order to make the Segment-Coding method an effective research approach for source-code level analysis of Project Histories, the time required to analyse Project Histories should be shortened as much as possible. Since the development of an effective method for source-code level Project History analysis is one of the contributions this thesis intends to make to the body of Computer Science Education research, this chapter will present a method to support analysis through the development of machine methods for the generation of Segments. Such a method

would automatically produce Segments consisting of sets of Changes such as the Segments produced and analysed as part of the application of the Segment-Coding method described in Chapter 6. A successful approach for the Machine-Segmenting of Project Histories would significantly speed up the initial segmenting phase of a research project utilising the Segment-Coding analysis method. Such a Machine-Segmenting method would automatically group sets of related Changes in Segments much as a human researcher would, based on features of the source code modifications made to Changes.

In fact, the first methods developed were not true Machine-Segmenting algorithms. These algorithms instead attempted to identify ‘interesting’ Changes without grouping them into Segments, whereas true Machine-Segmenting algorithms both group Changes into Segments and then (based on Segment size) indicate which of these are likely to be ‘interesting’. Such algorithms will be referred to as algorithms for identification of ‘interesting’ Changes or Change-Identification algorithms<sup>41</sup>; this label applies to Machine-Segmenting algorithms as well, but when discussing Machine-Segmenting algorithms in particular then the ‘Machine-Segmenting’ label will be used.

The Machine-Segmenting and Change-Identification approaches described in this chapter are not intended to replace the human researcher. Some Segments of related Changes are extremely difficult to detect using machine approaches as will be discussed in Section 7.11, and machine-generated Segments are unlikely to perfectly fit the ‘real’ Segments present in the Project History. Furthermore, even if a machine-generation approach did produce a perfect set of Segments identical to those the human researcher would propose, a human researcher would still be required to analyse and evaluate Segments to discover underlying issues and problems. Rather than replacing the human researcher, machine-generated Segments are intended to serve either to provide a researcher with an overview of a Project History and the student’s programming process and/or to serve as the basis for a full Segment-Coding of the Project History. In either case, the actual analysis of Segment content still requires a human researcher to utilise the methods of detailed analysis described in Sections 4.4 and 6.2.2.

The most effective of the Machine-Segmenting and Change-Identification algorithms implemented is a Machine-Segmenting algorithm called LH-Graph and is based on the software-engineering concept of co-change as introduced by Hassan & Holt (2004), utilising the co-modification of lines of source code to predict which Changes are related. The evaluation of Machine-Segmenting methods required the development of an evaluation framework which is also presented in this chapter.

---

<sup>41</sup> The terms algorithm, approach and method will be used interchangeably to describe Machine-generation and Change-Identification algorithms

Extension algorithms for LH-Graph which seek to augment results produced by the core algorithm are also presented and evaluated in this chapter.

The method developed for the evaluation and comparison of Machine-Segmenting algorithms is described in Section 7.4. Section 7.4.5 describes the SCORE Analyser functionality implemented to generate and evaluate machine-generated Segments. Several approaches to Machine-Segmenting and Change-Identification developed as part of this research project will be described in Section 7.6. Evaluation of different Machine-Segmenting and Change-Identification approaches is presented in Section 7.7. Extension algorithms to the best-performing Machine-Segmenting algorithm are presented in Section 7.8 and evaluated in Section 7.9. Section 7.10 discusses a second phase of evaluation on an independent data set. Section 7.11 describes types of problems/tasks which are detected well and those which are detected poorly by the best Machine-Segmenting approach. Limitations to the evaluation approach are discussed in Section 7.12. Future extensions to the Machine-Segmenting method are proposed in Section 7.13.

The data used in the initial development and evaluation of machine-generated Segments is discussed in Appendix Section 9.8.1.

## **7.2 Implementation of Line History Generation**

This section will present the design and evaluation of a method for the generation of Line Histories. Line Histories were described in detail in Section 6.2.1. A Line History stores all the modifications that occur to a source code line inside the project's history. This requires an accurate mapping of source code lines from version to version to build up the history.

Line Histories can be used to assist manual analysis of source code. They give the researcher access to the history of a line with all of its modifications without having to manually navigate between many different Changes to find the line in question. Having to do so would prove extremely time-consuming for lines with many modifications. It also allows for the provision of a more powerful diff-style view which can highlight not just additions and deletions but also moved and modified lines. Furthermore, Line Histories allow for a summary view which can summarise all modifications occurring over a range of Changes, allowing the researcher to quickly establish an understanding of context and relationship between those Changes.

The software engineering literature contains an example of work utilising a similar line history concepts. In work by Canfora et al. (2007) the generation of a similar data structure producing a mapping of lines from one version to another is outlined. It has a reported error rate below 5%. The method presented in this section was developed without knowledge of this existing method and



hence utilises a different algorithm. It also includes additional functionality not found in this previous approach, since it also attempts to detect ‘ghost’ lines, lines that are added after having been recently deleted.

There are five different modifications that lines can undergo: *Added*<sup>42</sup>, *Deleted*, *Moved*, *Ghost* and *Mutated*. These are discussed in more detail in Section 6.2.1.2. These are the *Added*, *Deleted*, *Moved*, *Ghost* and *Mutated*. The algorithm for detection of these modifications which generates Line Histories is described in Section 7.2. An evaluation of the Line History algorithm’s accuracy is presented in Section 7.3

### 7.2.1 Generation Algorithm Description

This section describes the Line History Table generation algorithm. To build a Line History Table, the algorithm processes each versioned file in the Project History in turn. Each file is composed of an arbitrary number of Changes to that file. The generation algorithm iterates through all of the file’s Changes in order of their timestamp, matching lines between the previous and the current version and adding the line in the current version to the Line History belonging to the line from the previous version with which it was matched, or creating a new Line History for newly created lines. Leading and trailing whitespace is generally removed for all string comparisons occurring as part of the algorithm. At each Change, the following generation steps for the detection of modifications are executed:

#### **Step 0) Generating Matching Line Groups**

The first step involves detecting groups of lines which are unchanged from the previous to the current version. Each line in such a group in one version will match a line in the other version. The groups are detected by applying the Longest Common Substring<sup>43</sup> algorithm until no more ‘substrings’ (group of lines) are found indicating that all remaining lines do not have a matching line in the other version. All identified groups of lines could consist either of Maintained or Moved lines.

#### **Step 1) Detecting Maintained Lines**

Maintained lines are detected through application of the Longest Common Subsequence (LCS) algorithm. The sequences compared are the groups of lines identified in the previous step. All groups of lines belonging to the LCS are marked as *Maintained* and are connected. This is done by traversing

---

<sup>42</sup> *Italics* in this chapter generally denote that the italicised name refers to a class, method or parameter name from the actual implementation of the SCORE Analyser; for example, the *Added* modification type is used as a name in the actual implementation of the Line History generation algorithm; sometimes these names are slightly shortened or modified for readability.

<sup>43</sup> Longest Common Substring: An algorithm which finds the longest common (continuous) substring between two or more input strings

the LCS and at every item of the LCS matching the location of the item (line) in the previous version's sequence with the location (line) in the current version's sequence. All matched source code lines are removed from the set of lines to be matched. This step will process most of the lines unless a very large amount of modifications occurred between the two versions.

### **Step 2) Detecting Moved Lines**

To detect moved lines, each remaining unmatched group of lines (those lines that were not part of the LCS) is marked as *Moved*, meaning that every individual line in these groups is marked as moved.

After the *Moved* line detection step, no lines remaining in the previous version's set of unmatched lines are textually equal to any lines in the current version's set of unmatched lines since such lines would have been detected either as *Maintained* or *Moved* lines.

### **Step 3) Detecting Mutant Lines**

To detect *Mutant* lines (lines that have had their text modified) all unmatched lines from the old document are compared to unmatched lines from the current document. However, instead of performing a string comparison to determine textual identity between lines, a Levensthein distance comparison is performed.

Levensthein distance is a measure of difference between strings. It counts the number of insertions, substitutions and/or deletions of characters that would be required to change the first string into the second string. The Levensthein distance is divided by  $(m + n) / 2$ , where  $m$  is the length of the first string and  $n$  is the length of the second string. This generally produces a value between 0.0-1.0 with 0.0 being exactly identical strings and 1.0 being completely different strings. In some cases, the value can exceed 1.0, for example with the strings "aaa" and "b", since two deletions and a substitution are required resulting in a value of  $3 / ((3+1)/2) = 3 / 2 = 1.5$ ; this could be dealt with by using the length of the longer string instead of the average of the length of both strings.

The result of the comparison of each of the previous version's remaining lines with each of the current version's remaining lines is stored as *MutantCandidates*. If lines are in close proximity as measured by their line numbers ( $\text{proximity} = |\text{PrevLineNr} - \text{CurLineNr}|$ ) then the distance is reduced by multiplication with a constant value to ensure that close-proximity lines are preferred as *Mutants*. Line pairs are then visited in ascending order of Levensthein distance. If a pair's distance falls below the *MutantLevenstheinDistance* threshold (set in a configuration file) then the pair is accepted as a mutant. The lines are matched with a *Mutant* modification and removed from the set of lines to be matched; any pairs containing a matched line are also removed from the *MutantCandidates* vector.

This process continues until all *MutantCandidate* pairs have been visited or removed or until a pair's distance value falls above the *MutantLevenstheinDistance*.

#### **Step 4) Detecting Ghost Lines**

Each recently deleted line is considered a *GhostCandidate*. The window in which lines are considered 'recently deleted' and hence *GhostCandidates* is set via the *GhostWindow* parameter in a configuration file. The *GhostWindow* parameter specifies how long *GhostCandidates* are stored before being removed. For example, a setting of *GhostWindow*=5 would mean that at each step all *GhostCandidates* more than five Changes old would be culled from the set of *GhostCandidates*. The *Ghost* line detection step goes through each *Ghost Candidate* in the order of the candidate's age, with younger more recently deleted candidates being visited first. The Levensthein distance between the candidate and each remaining unmatched line in the current version is calculated. If any distances fall below the *GhostLevenstheinDistance* threshold that is set by the researcher, then the candidate is matched with the unmatched line from the current document with the lowest distance using a *Ghost* modification type. The line is then removed from the set of unmatched lines of the current document, and the ghost candidate is removed from the set of *GhostCandidates*. This process continues until either the set of unmatched lines is empty or all *GhostCandidates* have been visited.

#### **Step 5) Detecting Added and Deleted Lines**

In the final step, all unmatched lines from the previous version are marked as *Deleted* while all unmatched lines from the current version are marked as *Added*. All Deleted lines are added to the set of *GhostCandidates* to be used for Ghost detection in subsequent iterations of the algorithm.

#### **Algorithm Result**

By iterating over the Project History's Changes in turn Line Histories are built up; when the algorithm completes, every source code line in every version will be assigned to exactly one Line History. Each Line History will consist of anywhere between one and many entries. Each entry is associated with a Modification Type, a version in which it occurs and the line number at which it is located in that version. Line Histories consist of one *Added* modification at the Change in which the line was first added and any number of *Mutated*, *Maintained*, *Moved*, *Deleted* and *Ghost* modifications. The relationship between *Ghost* and *Deleted* modifications must be  $n:n$  or  $n:n+1$  since there can be only one *Ghost* modification for every *Deleted* modification.

Figure 78 shows modifications occurring to a line `circleCount++` (the middle line in every panel) over its lifetime while Figure 79 shows these modifications stored in the line's Line History. As discussed

in Section 6.2.2.1, blue text in Figure 79 indicates added lines or ghost lines (the Change Browser Details View shown does not differentiate between these two types), red text indicates deleted lines and green text indicates mutated lines, with the original text on the left of the ‘->’ arrow and the new text on the right.

351: Added

```
point2.y = clickUp.y;  
newFlowerVase[circleCount] = new DiagramObject1("Flower Vase", point1, point2);  
circleCount++;  
//newPieceOfFurniture[tableCount]->render();  
tableCount++;
```

352: Deleted

```
point2.y = clickUp.y;  
newFlowerVase[circleCount] = new DiagramObject1("Flower Vase", point1, point2);  
circleCount++;  
//newPieceOfFurniture[tableCount]->render();  
tableCount++;
```

354: Ghost

```
point2.y = clickUp.y;  
newFlowerVase[circleCount] = new DiagramObject1("Flower Vase", point1, point2);  
circleCount++;  
//newPieceOfFurniture[tableCount]->render();  
tableCount++;
```

355: Mutated

```
point2.y = clickUp.y;  
newFlowerVase[circleCount] = new DiagramObject1("Flower Vase", point1, point2);  
circleCount++; -> //circleCount++;  
//newPieceOfFurniture[tableCount]->render();  
tableCount++;
```

356: Mutated

```
point2.y = clickUp.y;  
newFlowerVase[circleCount] = new DiagramObject1("Flower Vase", point1, point2);  
//circleCount++; -> circleCount++;  
//newPieceOfFurniture[tableCount]->render();  
tableCount++;
```

357: Mutated

```
display();  
//newFlowerVase[circleCount] = new DiagramObject1("Flower Vase", point1, point2);  
circleCount++; -> //circleCount++;  
//newPieceOfFurniture[tableCount]->render();  
tableCount++;
```

389: Deleted

```

display();
//newFlowerVase[circleCount] = new DiagramObject1("Flower Vase", point1, point2);
//circleCount++;
newPieceOfFurniture[tableCount]->render();
tableCount++;

```

Figure 78: Sequence of modifications to a line (the line in question is the 'circleCount' line)

(2258): (351-352, 354-357, 389 : total = 7)
351, ADDED(O): circleCount++;
352, DELETED(O): DELETED (circleCount++;)
354, GHOST(O): GHOST (circleCount++;)
355, MUTATED(O): //circleCount++;
356, MUTATED(O): circleCount++;
357, MUTATED(O): //circleCount++;
389, DELETED(O): DELETED (//circleCount++;)

Figure 79: Line History showing modifications to a line

In order to facilitate easy access to all Line History modifications at a given Change, a *HistoryDocument* data structure storing all Line Histories present in that Change and their modification status in that Change is also created. Pseudo-code for the Line History Generation algorithm is presented in Appendix Section 9.7.1.

## 7.2.2 Generation Algorithm Example Run

An example will illustrate how the Line History Generation algorithm operates by demonstrating its mapping of lines from one version to the next. The example uses a Mutant detection Levensthein distance limit and Ghost detection Levensthein distance limit of 0.2.

In step 1 (see Figure 80) groups are generated based on lines whose text is identical, which are paired. In this example, four such groups are generated, one consisting of two line pairs, and two consisting of one line pair each. The LCS algorithm is then applied to find the longest common subsequence of these groups. In this case, the LCS includes the three line pairs which are marked with black connectors. These line pairs are connected and their modification type is set to *Maintained*. The remaining line pair which falls outside the LCS is marked as *Moved*.

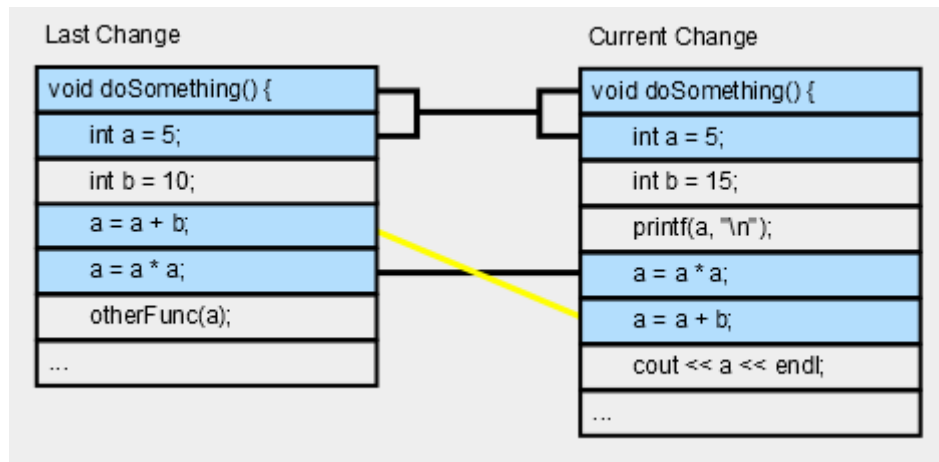


Figure 80: Matching of MAINTAINED lines (step 1)

In step 2 (Figure 81) the two unmatched lines of the previous document are compared to the three unmatched lines of the current document, resulting in  $2 \times 3 = 6$  comparisons. Of these comparisons, only the comparison between *int b = 10* and *int b = 15* falls below the mutant detection threshold of 0.2 as is shown in Figure 81 in the top table. These two lines are therefore linked and their type is set to MUTANT.

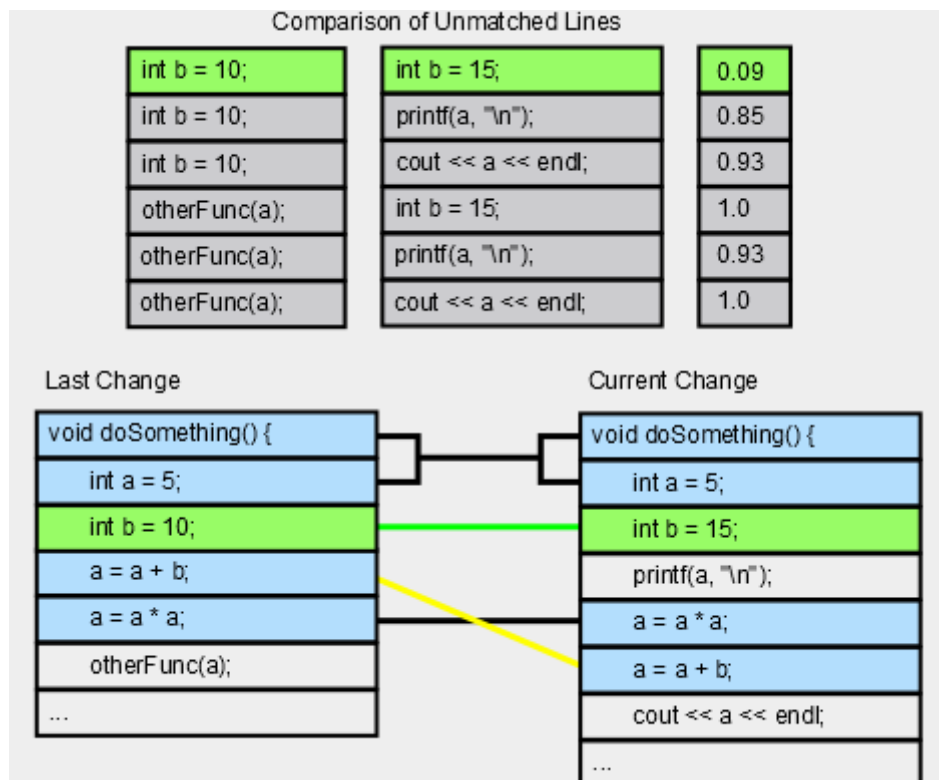
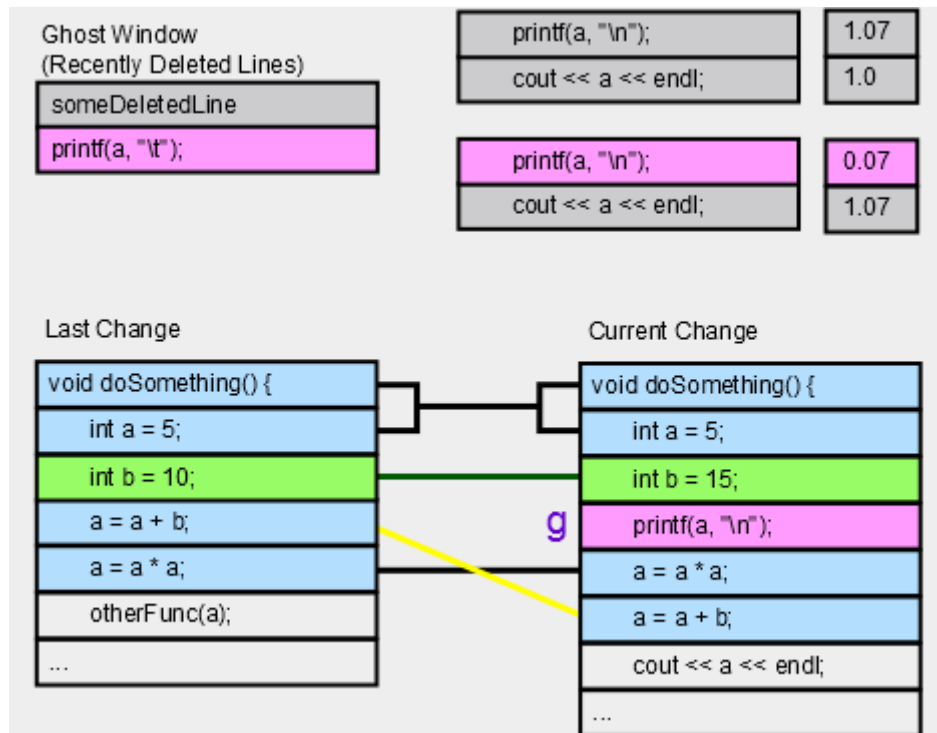


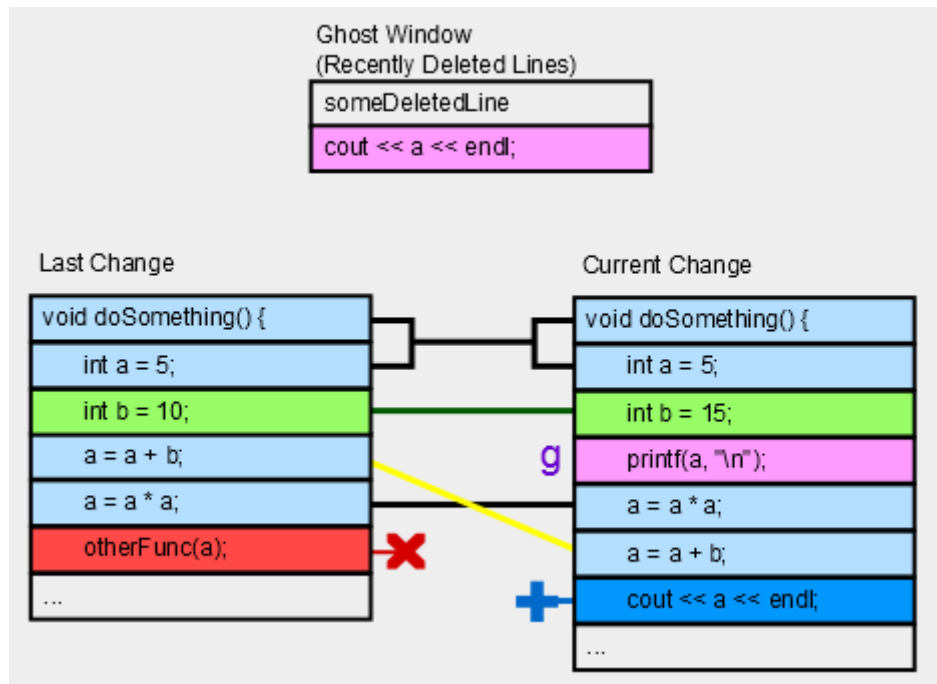
Figure 81: Matching of Mutated Lines (Step 2)

In step 3 (Figure 82), the two lines of the current document that remain unmatched are each compared to the two lines currently in the Ghost line candidate window, for a total of  $2 * 2 = 4$  comparisons. The comparison of `printf(a, "\t");` and `printf(a, "\n");` falls below the threshold, therefore `printf(a, "\n");` is accepted as a *Ghost* modification of `printf(a, "\t");` and is added to that line's Line History.



**Figure 82: Matching of Ghost lines (step 3)**

In step 4 (Figure 83), the remaining unmatched line from the previous document is marked as *Deleted* and added to the ghost candidate window. The remaining unmatched line from the current document is marked as *Added* and a new Line History is created for that line (the only new Line History created during the comparison of these two Changes).



**Figure 83: Generation of Added and Deleted lines (step 4)**

The algorithm has now completed generation of Line Histories for this Change, and has identified one *Deleted*, one *Added*, one *Mutated*, one *Moved*, one *Ghost* and three *Maintained* lines. The Line Histories present in this Change are also stored in a new *HistoryDocument*.

### 7.3 Evaluation of the Line History Generation Algorithm

This section presents the evaluation of the proposed Line History generation algorithm. The evaluation approach involves producing line histories for selected Project Histories and then manually examining each Change in the Project History to determine how many lines were correctly matched, how many were incorrectly matched and how many lines that should have been matched were not. This data will then be used to calculate the recall and precision of the algorithm in terms of 'correctness'. The problem with this approach is that it is extremely time-consuming.

Part of the evaluation of the Line History generation algorithm involves finding good values for the algorithm's mutant and ghost detection Levenshtein distance and for the size of the ghost window, but calculating recall and precision for multiple settings would require too much time. For this reason, the initial evaluation phase involved testing different settings for the Line History generation algorithm with a Machine-Segmenting method. Since machine generation has not yet been discussed and this evaluation would be difficult to follow it has been placed in the appendix, Section 9.7.2.



The following evaluation will be based on the manual analysis of generation results utilising the best settings discovered using evaluation with the Machine-Segmenting method. To measure the precision and recall of the generated Line History Table, every Change in a sample of four Project Histories was manually examined. At every Change, detected mutants and ghosts were evaluated for correctness, with the number of errors being recorded. The number of mutants that were not detected was also noted. The number of non-detected ghosts was not noted as that would have required comparison of each added line with every line that was deleted in recent history. While theoretically possible, this would have required too much time.

Using these data, precision for ghost and mutant detection can be calculated as:

$$Precision = \frac{Correct}{Correct + Incorrect}$$

Furthermore, for mutant detection recall can be calculated as:

$$Recall = \frac{Correct}{Total}$$

Recall cannot be calculated for ghost detection as the total number of ghosts is difficult to determine.

Since this type of manual evaluation is very time-consuming, it was conducted on four Project Histories only. Two were selected from the set of Project Histories analysed in detail in this research project, and two were selected from among the other Project Histories not analysed in detail in this research project. The selection was based on Project History size, with two shorter Project Histories comprising relatively fewer Changes having been chosen for time reasons. These Project Histories consist of 626 (Student6) and 677 (Student7) Changes, whereas the other two Project Histories consist of 717 (John A1) and 1075 (Christopher A3) Changes.

These non-analysed Project Histories were selected since the first two Project Histories were involved in the implementation of the Line History Generation algorithm; the non-analysed Project Histories serve as the independent data set. Two of the Project Histories are chosen from Assignment 1 and two are chosen from Assignment 3.

A summary of the results is presented below. Table 26 shows the total number of mutants to be identified based on analysis by the human researcher (Total To Identify), the total actually identified by the algorithm (Total Identified), the number of mutants of that total which were correctly identified (Identified Correctly) and the total of mutants that should have been but were not

identified (Not Identified) as well as the precision and recall metrics calculated based on these values. The data (see Table 26) shows that Mutant detection produces a very high recall with precision values  $\geq 0.95$  for all Project Histories. Precision is also high at  $> 0.9$  for all Project Histories.

Ghost detection is also precise (see Table 27), with precision values  $\geq 0.95$  level for three Project Histories. It does somewhat worse for one Project History, achieving only 86.7% precision.

**Table 26: Precision / Recall for line history mutant generation**

	Total To Identify (TOI)	Total Identified (TOT)	Identified Correctly (COR)	Not Identified (NOT)	Precision (COR / TOT)	Recall (COR / TOI)
John A1	911	962	874	37	0.91	0.96
Christopher A3	1043	1022	992	51	0.97	0.95
Student6 A1	775	772	747	28	0.97	0.96
Student7 A3	822	870	801	21	0.92	0.97

**Table 27: Precision for line history ghost generation**

	Correct (COR)	Incorrect (INC)	Precision (COR / (COR+INC))
John A1	19	1	0.95
Christopher A3	61	3	0.95
S6 A1	30	0	1.0
S7 A3	75	10	0.87

Detailed examination of the results did highlight some weaknesses in the generation algorithm, specifically in the way that lines in close proximity are weighted, which can produce incorrect matches especially at the lax *MutantLevenstheinDistance* = 0.5 boundary that performed best in the evaluation using the machine-generation algorithm presented in Section 9.7.2. However the results suggest that the Line History Generation method proposed here produces accurate results as-is.

As the results show the Line History generation algorithm is effective in correctly matching mutant and ghost lines, producing high precision and accuracy. Previous work on generating Line Histories , (Canfora et al., 2007) produced an error rate of below 5%. Using the proposed line history generation algorithm a 5% error rate is achieved for *Mutated* modifications, which are the most difficult modifications to detect; the total error rate for the algorithm proposed here across all

modification types including *Maintained*, *Added*, *Deleted*, *Moved* and *Ghost* modifications would be much lower than 5%. In addition, the Line History Generation algorithm proposed in this thesis can also detect *Ghost* modifications, whereas the Line History generation algorithm proposed by Canfora et al. (2007) cannot.

While precision and recall were found to be high for the setting utilised in the analysis, the manual examination of the generated Line History Table found errors associated with the lax threshold that would not have occurred given a tighter threshold. Despite this, the laxer threshold produced better performance with the Machine-Segmenting algorithm. It is possible that this is because the tighter threshold would have failed to detect more mutants (hence reducing recall). However, intuition suggests that some of the erroneous links created may in fact have improved results because the lines matched were associated by line proximity rather than textual proximity. When a line is deleted in a certain area of the source code and a new line is added in close proximity to the line that was deleted, there is a high likelihood that these lines are related, though they should not be matched as 'the same' line.

The results suggest that the Line History Generation approach described is effective which justifies the use of Line Histories in Project History analysis, both for its use in providing the Line-History based SCORE Analyser functionality presented in Section 6.2.2, as well as its use in machine-generation of Segments as discussed later in this chapter. The following sections will describe methods for Machine-Segmenting of Project Histories. Two of the approaches, the LH and LH-Graph methods, are based on Line Histories. These approaches will be shown to be the most effective of the different methods trialled.

## **7.4 Method for the Evaluation of Machine Segmenting and Change-Identification Algorithms**

Following on from the discussion of Line History Generation the following sections will present a description and evaluation of Machine-Segmenting and Change-Identification algorithms. Before the presentation of the actual algorithms and their evaluation, this section will present a method of evaluation of machine algorithms developed as part of this research project.

## 7.4.1 Precision, Recall and A/E Ratio

### 7.4.1.1 Precision and Recall

The identification of ‘interesting’ Changes or Segments is an Information Retrieval<sup>44</sup> task. In the context of this research project, Changes are ‘documents’ and a successful algorithm would identify as many relevant Changes as possible.

The most commonly used measures of performance of Information Retrieval algorithms are recall and precision. Precision and (where possible) recall are utilised as the primary measures of performance in most of the Software Engineering literature presented in the literature review (e.g. Adams et al., 2010; Canfora et al., 2006; Ying, Murphy, Ng, & Chu-Carroll, 2004; Zimmermann et al., 2005).

The formal definition of recall and precision are (Manning et al., 2008, Chapter 8):

$$precision = \frac{\#(relevant\ items\ retrieved)}{\#(retrieved\ items)}$$

$$recall = \frac{\#(relevant\ items\ retrieved)}{\#(relevant\ items)}$$

In other words, precision describes how many of the retrieved documents are relevant and recall describes how many of the relevant documents were retrieved.

Recall and precision are usually found to be negatively correlated. If one desires to achieve a very high degree of recall (detect as many ‘interesting’ changes as possible) one may include more marginal identified items. This reduces the precision, meaning that a larger ratio of ‘uninteresting’ items is included compared to ‘interesting’ items. The trivial case of including all items in the result guarantees a recall score of 1.0 but makes the actual precision equal the expected precision, for example.

On the other hand, only including the best items may produce very high actual precision but very low actual recall (since even if all identified items are correct, if the number of selected items is very small that may still be a small percentage of total items to be identified). This would provide the researcher with too few of the ‘interesting’ items to be useful. A good method should be able to produce a useful balance of high precision and recall.

---

<sup>44</sup> Information Retrieval: An area of study involving the searching and retrieval of documents or the content of documents.

### 7.4.1.2 Actual / Expected Ratio

A measure based on actual versus expected recall / precision was also utilised for evaluation of algorithms.

Let:

$$Target = \{relevant\ documents\}$$

$$Total = \{all\ documents\}$$

$$Selected = \{retrieved\ documents\}$$

$$Correct = \{relevant\ documents\} \cap \{retrieved\ documents\}$$

The actual precision for a given algorithm is:

$$Actual\ Precision = \frac{|Correct|}{|Selected|}$$

The actual recall for a given algorithm is:

$$Actual\ Recall = \frac{|Correct|}{|Target|}$$

The expected precision for a random algorithm at a number of *Selected* items chosen will be:

$$Expected\ Precision = \frac{|Target|}{|Total|}$$

The expected recall for a random algorithm will be:

$$Expected\ Recall = ExpectedPrecision \times \frac{|Selected|}{|Target|} = \frac{\frac{|Target|}{|Total|} \times |Selected|}{|Target|} = \frac{|Selected|}{|Total|}$$

To determine how well an algorithm is doing compared to how well an algorithm performing as expected would do, one can calculate the ratio of actual to expected accuracy/recall.

$$A/E\ Ratio = \frac{ActualPrecision}{ExpectedPrecision} = \frac{|Correct| \times |Total|}{|Selected| \times |Target|} = \frac{\frac{|Correct|}{|Target|}}{\frac{|Selected|}{|Total|}} = \frac{ActualRecall}{ExpectedRecall}$$

As the formula above shows, the ratio of *ActualRecall* / *ExpectedRecall* equals the ratio of *ActualPrecision* / *ExpectedPrecision* and can be calculated as  $(Correct / Target) / (Selected / Total)$ .

This ratio will be called the *Actual to Expected* or simply *A/E ratio*. The *A/E ratio* together with the *p-value* will be the primary measurements used to evaluate the effectiveness of algorithms.

When speaking of *average precision* in this chapter, the term is taken to mean the average of precision values calculated for several Project Histories instead of its conventional meaning in Information Retrieval as the area below the precision-recall curve as items from a ranked list are selected (Zhu, 2004).

Average precision in the Information Retrieval sense would have provided a good method for the evaluation of Machine-Segmenting algorithms. Unfortunately, it is not possible to use it to evaluate the Line Metric and File Metric approaches using average precision since these methods do not produce ranked lists.

For this reason, an approach calculating the precision and recall for a set percentage of total Changes selected was utilised instead. Since these algorithms will later be shown to be less effective at identifying ‘interesting’ Segments, future research on Machine-Segmenting methods could utilise the Information Retrieval concept of *average precision* as an evaluation metric rather than choosing a set percentage of Changes to be selected.

## 7.4.2 Calculating P-Values

### 7.4.2.1 Definition and use of p-value in evaluation

Recall and precision provide a sense of how well the machine approach is working for a particular assignment. However, by themselves they do not provide concrete evidence as to how likely such a result is to occur randomly and hence how secure we can be in preferring it over randomly choosing Changes.

A good algorithm run may be the result of random chance. This is especially likely if only a very small number of Changes are selected. For example, if a single change is selected and 50% of all changes are ‘interesting’, there is a 50% chance of the identified change being ‘interesting’. In this case the result would produce an actual precision of 100% compared to an expected precision of 50%, but this ‘good’ result would occur 50% of the time. On the other hand, if 25 out of 25 changes are correctly selected, the likelihood that this has occurred due to random chance is very slim.

### 7.4.2.2 Calculating p-values calculation using a hypergeometric distribution

A p-value can be calculated using the hypergeometric binomial test:

$$P(X = k) = \frac{\binom{m}{k} \binom{N-m}{n-k}}{\binom{N}{n}}$$

It enables calculation of the probability of selecting a specified number of marked elements from a set of  $N$  elements of which  $m$  elements are marked and  $n$  elements are chosen. By using the upper tail the probability of choosing at least  $k$  marked elements is calculated.

In evaluating an algorithm for the machine-detection of 'interesting' Changes of an assignment, the set  $N$  is the set of all changes in that assignment project,  $m$  is the set of all 'interesting' / marked elements as identified by a human researcher, and  $k$  is the number of elements correctly identified by the algorithm when comparing the algorithm's output to the human-identified output. The produced p-value shows how likely a result is to have arisen from random choosing of  $n$  Changes from the set of all Changes.

#### **7.4.2.3 Inadequacy of the hypergeometric distribution for calculation of true p-values**

The p-value produced by the selection of individual changes using the hypergeometric distribution does not accurately reflect the way in which 'interesting' Changes are distributed in the ordered set of Changes which makes up a Project History. Groups of 'interesting' Changes tend have Changes belonging to that group clustered in close temporal proximity.

An algorithm randomly choosing Segments of Changes may randomly choose such a group. It would then achieve what would appear to be an extremely unlikely outcome according to the hypergeometric distribution. Therefore, instead of comparing algorithms to a random algorithm picking individual changes the algorithms should be compared to an algorithm that randomly picks sequences of changes. Because each sequence of changes includes many individual changes, this reduces the total number of 'picks'. This in turn means that results achieving a certain number of 'correct' picks are more likely than if individual changes were selected.

For example, if we have 100 total changes, 50 of which are 'interesting' and we select 30 changes of which 20 are identified correctly, then the p-value for ( $N=100$ ,  $m=50$ ,  $n=30$ ,  $k=10$ ) is  $p = 5.09352745672187e-07$ . On the other hand take the case in which the number of total changes, 'interesting' changes, selected changes and correctly selected changes are divided by 10. In this case we have the same ratio of correctly identified to total selected changes, but ( $N=10$ ,  $m=5$ ,  $n=3$ ,  $k=1$ ) yields a much higher (non-significant at  $p < 0.05$ ) p-value of  $p = 0.138$ .

However, it is also not possible to simply count Segments as individual items in the distribution, as Segments may be interleaved with non-segment changes and Segment size varies.

Precisely calculating the p-value for reaching a correctly identified number of changes  $\geq k$  is likewise hard, since it involves testing all possible combinations of Segment selection given by:

$$\binom{n}{k} = \frac{n!}{k! \times (n - k)!}$$

This number becomes very large for even relatively small values for  $n$  and  $k$ , such as  $n=50$  (50 total Segments) of which  $k=10$  are chosen, resulting in  $1.02722781e+10$  combinations to be calculated. Since this method of calculation is not feasible, a simulation-based method of p-value calculation was developed.

#### 7.4.2.4 Calculation of p-values using simulation with random Segments

Since calculation of all possible combinations and evaluation of whether they exceed the desired  $k$  is not computationally feasible, a p-value was instead produced via a simulation approach. The approximate percentage of combinations that produce larger  $k$ -values than the algorithm being analysed was determined by running a large number of trials using a random Segment selection algorithm. The number of trials producing larger  $k$ -values is then divided by the number of total trials. This yields an approximate p-value for a given value of  $k$ . The algorithm used to produce random runs is the same random algorithm which will be described in Section 7.6.1. The resulting value is the approximate p-value for a given value of  $k$ . If an approach has a low p-value (significant at  $> 0.05$ ) and high *A/E ratio*, then it can be concluded that the approach is likely to reliably produce good results.

### 7.4.3 Fit, Spread and Average Segment Size

#### 7.4.3.1 Quality of Segments

In addition to identifying a large number of ‘interesting’ Changes a good approach would also group related Changes (into Segments); the fewer Segments related Changes are spread over, the better. Having related Changes grouped in Segments makes it easier for the researcher to discover the underlying problem or task the student is solving during these Changes.

It would also be preferable if related Changes were not mixed with unrelated Changes. In the following examples, bracketed items represent Segments, letter-number identifiers (such as A1) represent Changes, and Changes with the same alphabetic identifier are related. Given three related sets of Changes (each containing three Changes), the best approach to segmenting them would be to identify related Changes in their own Segments:

(S1: A1, A2, A3) (S2: B1, B2, B3) (S3: C1, C2, C3)

Mixing unrelated Changes in Segments would make it much more difficult to identify underlying problems:



(S1: A1, B2, C3) (S2: A2, B2, C3) (S3: A3, B3, C3)

Identifying all Changes in a single Segment also would not provide useful guidance to the researcher:

(S1: A1, A2, A3, B1, B2, B3, C1, C2, C3)

Finally, identifying each Change in its own Segment is as unhelpful as identifying all Changes in a single Segment:

(S1: A1), (S2: A2), (S3: A3), (S4: B1), (S5: B2), (S6: B3), (S7: C1), (S8: C2), (S9: C3)

The examples above show that useless segmenting outcomes can take many forms ranging from monolithic Segments containing all Changes to fragmented Segments containing single Changes. Since in this example all the outcomes have identified the same set of Changes the *A/E ratio* and *p-value* for these approaches would be equal. Neither of these measures is useful in measuring the quality of segmenting achieved by the algorithm.

Since *A/E ratio* and *p-value* are not useful in measuring segmenting quality, two different metrics were developed. These metrics are *fit* and *spread*.

It should be noted that since Change-Identification algorithms do not produce Segments the fit and spread metrics cannot be applied to Change-Identification algorithms. They are useful in evaluation Machine-Segmenting algorithms only.

#### 7.4.3.2 Fit Metric

The *Fit* metric measures how well sets of related Changes are ‘fitted’ by machine-generated Segments, that is to say what proportion of the Changes co-occurring in Segments in which Changes of a set of related Changes occur are related Changes, and how many are unrelated Changes. The *fit* metric is calculated for every set of related Changes. A set of related Changes will also be called *related Segment*, as compared to the segments produced by an algorithm which are called *machine-generated Segments* or simply *Machine Segments*. What follows is the formula for calculating fit.

One way of calculating the *fit* metric would be:

*hs* = human-identified Segments

*ms* = machine-identified Segments

$|hs \cap ms|$  = number of Changes which overlap between the human-identified and machine-identified Segment, calculated between every human-identified and machine-identified Segment which shares at least one Change

$|ms \text{ or } hs|$  = the number of Changes in the corresponding machine-identified/human-identified Segment

$$simplefit = \sum_{ms \in msecs \text{ where } |hs \cap ms| > 0} \frac{|hs \cap ms|}{|ms|}$$

However, this way of calculating the metric would be too harsh in cases in which a small number of Changes from the set of all related Changes occurs in a Machine Segment.

This is why the calculation method used weights the fit according to what proportion of a related Segment's Changes occur in the machine-generation Segment:

$$weightedfit (fit) = \frac{\sum_{ms \in msecs \text{ where } |hs \cap ms| > 0} \frac{|hs \cap ms|}{|ms|} \times \frac{|hs \cap ms|}{|hs|}}{\sum_{ms \in msecs \text{ where } |hs \cap ms| > 0} \frac{|hs \cap ms|}{|hs|}}$$

The fit value can range from 0.0-1.0 with low values indicating a poor fit and high values indicating a good fit.

### 7.4.3.3 Spread Metric

The *Spread* metric measures how 'spread apart' an actual Segment is; that is, how many Machine Segments it is spread over. It is calculated as:

$mSegsNr$  = number of Machine Segments containing at least one of the related Segment's Changes

$rSegChanges$  = number of Changes in the related Segment

$$spread = 1.0 - \frac{\frac{mSegsNr - 1}{rSegChanges}}{\frac{rSegChanges}{rSegChanges - 1}}$$

The spread value can range from 0.0-1.0, with low values indicating that the related Changes are spread over many Segments, while a high value indicates that related Changes are contained in few Segments. A value of 1 indicates that all related Changes are grouped in a single Machine Segment.

For example, if we have 20 changes and:

- 1 Machine Segment, *spread* =  $1 - (1-1) / (20-1) = 1$
- 5 Machine Segments, *spread* =  $1 - (5-1) / (20-1) = 0.79$
- 20 Machine Segments, *spread* =  $1 - (20-1) / (20-1) = 0$

#### 7.4.3.4 Using Spread and Fit to determine segmenting performance

*Spread* and *Fit* values will tend towards 1 if related Segments are very well described by machine-identified Segments. If this is not the case then one or the other or both values will be lower; however, it is possible to achieve very high *fit* and very low *spread* and vice versa, both of which indicate poor results. For example, consider this example from earlier:

(S1: A1, A2, A3, B1, B2, B3, C1, C2, C3)

It has a spread value of 1 for any of the related Segments, since all related Changes are part of a single Machine Segment. However, the fit value will be  $(3/9) * (3/9) = 9/81 = 1/9 = 0.11$  which is in fact the poorest value possible for this example since all non-relevant Changes are grouped with related Changes. On the other hand, consider this example:

(S1: A1), (S2: A2), (S3: A3), (S4: B1), (S5: B2), (S6: B3), (S7: C1), (S8: C2), (S9: C3)

It produces a fit value of 1 since every related Change is not grouped with any unrelated Changes (since they are grouped with no other Changes at all) but the spread value will be  $1 - ((9-1) / (9/1)) = 1 - 1 = 0$  since related Changes are as spread as possible.

Hence, a good approach should produce a balance of good *spread* and *fit*. A very high value for one metric is meaningless if the other metric shows very poor performance. To measure segmenting performance with a single metric, the *fit\*spread* metric is derived by multiplying the fit and spread values.

This means that if one metric has a very low value, the *fit\*spread* metric will show poor performance even if the other metric is high and will hence filter out algorithms that produce trivial segmenting results such as monolith Segments or very fragmented Segments.

#### 7.4.4 Evaluating algorithms using Precision/Recall, Fit/Spread and P-Values

##### 7.4.4.1 Evaluation by p-value and ratio

Algorithms are evaluated in two ways. One is evaluation in terms of how many Changes they correctly identify. This evaluation is based on the comparison of precision/recall values and on the probability value associated with the result.

Proposed algorithms were evaluated in two phases. The first phase involves the ten Project Histories which were selected for detailed analysis as described in Section 3.6.3. Evaluation of Segmenting results for these Project Histories are discussed in Sections 7.7 and 7.9. However, these are also the Project Histories using which the Machine-Segmenting and Change-Identification algorithms were developed. Hence there is a risk that these algorithms are fitted to produce good performance for these Project Histories and that they would perform significantly worse for other Project Histories. For this reason a second phase of evaluation was used to test the algorithms with an independent data set as described in Section 7.10.

For each machine-segmented Project History a Change-Identification or Machine-Segmenting algorithm will produce precision/recall, A/E ratio and p-value metrics. In order to be able to compare among different algorithms, the metrics for individual assignments must be analysed together. In order to enable comparison, average values for precision/recall, A/E ratio and p-values as well as the total number of assignments for which the algorithm passed the significance test at significance levels ( $p \geq 0.05$ ) and ( $p \geq 0.01$ ) are calculated and presented (as discussed in Section 7.4.1 'Average Precision' in this context is simply the average of precision values calculated for each assignment, not as the area below the precision-recall curve).

The method by which a range of settings could be evaluated through a step-by-step reductive process is described in Appendix Section 9.8.1. The method by which consistent and repeatable results were achieved is described in Appendix Section 9.8.3.

#### **7.4.4.2 Evaluation by fit/spread**

The second method of evaluation utilises Fit and Spread (see 7.4.3) to measure how well a method groups related Changes. This evaluation approach can be applied only to methods that group related Changes. Some of the proposed methods evaluated in Section 7.7 do not group Changes and hence this evaluation method cannot be applied to them. Those methods essentially produce worst-case segmenting, since non-grouping of Changes can be interpreted as every Change having its own Segment or all Changes being part of the same monolith Segment. As with precision/recall and A/E ratios, an average value for the fit\*spread metric is calculated across assignments to provide an overview of how well a given method performs.

In the evaluation of different approaches, A/E ratio will be utilised as the primary measure of performance. When an algorithm is run with different settings, the setting maximising this ratio will be reported as the 'best' run unless the difference between several runs is very slight. In that case, the spread\*fit measure will be used to decide amongst the settings. Significant discrepancies such as

high A/E ratios being connected with especially low spread\*fit ratios will be reported if they are observed.

#### 7.4.4.3 Percentage of Changes chosen for evaluation

Another decision that must be made for the analysis of assignments is how many candidate Changes should be analysed for each method as a percentage of total Changes occurring in any given assignment. One approach would be to utilise Average Precision which is defined as the area under the recall-precision curve. This approach is often utilised to evaluate methods that return ranked lists such as the Machine-Segmenting methods developed as part of this research project. This approach was not chosen because it is not applicable to the evaluation of Change-Identification methods which do not produce Segments as explained earlier in Section 7.4.1.

Instead of utilising average precision over a range of Changes chosen, a fixed value of 25% of an assignment's Changes to be selected was used for all evaluation presented in this chapter. The reasoning behind this value is that an analysis of 25% of an assignment's Changes cuts down workload considerably when compared to analysing the whole assignment but is still likely to identify the assignment's most significant problems. This value will be referred to as the '*Segment cut-off*' level.

Another approach that was considered was utilising all Segments that were above a certain boundary value in size since this is an approach which would probably yield good results in practice. However, this would mean that the actual percentage of Changes would vary from assignment to assignment. This means that algorithms producing larger Segments would select many more Changes. This makes it difficult to compare different runs of the same algorithm to one another. Also, this method too would not have been applicable to Change-Identification algorithms. For these reasons, this approach was not utilised.

#### 7.4.5 Presentation of Results

In the body of the thesis data is presented as a summary for the average result of an algorithm across all tested Project Histories. The Overall Summary table (Table 28) summarises results over both assignments. It is used to present results since it is concise and allows for easy identification of the best run. Raw data is described with additional tables as described in Appendix Section 9.8.5. As is shown in Table 28, runs that perform well are highlighted. A yellow highlight means that that run produced the best A/E ratio (best performance in identifying 'interesting' Changes). A green highlight means that the run produced the best spread\*fit value (best segmenting performance). A blue highlight means that the run produced both the best A/E ratio and the best spread\*fit ratio.

**Table 28: Ratio, spread/fit and p-value values averaged over all students and both assignments**

Setting	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
Setting 1	1.594	66.1%	0.537	0.003	10	9
Setting 2	1.643	67.8%	0.515	0.003	10	9
Setting 3	1.604	66.1%	0.497	0.005	10	8
Setting 4	1.624	66.8%	0.475	0.004	10	9

## 7.5 SCORE Machine Segmenting Generation and Evaluation Facilities

This section will provide a brief overview of the SCORE Analyser facilities (functionalities and views) for Machine-Segmentation of Project Histories as well as a description of the output produced by Machine-Segmentation.

### 7.5.1 Generating Machine Segments

All Machine-Segmenting and Change-Identification algorithms are implemented using the same SCORE Analyser framework. This makes the Machine-Segmenting generation mechanism very flexible, allowing the user to implement new Machine-Segmenting methods by extending interfaces and abstract classes. This enables Machine-Segmenting generation algorithms implemented by the user to receive Line History and other Project History data from the SCORE Analyser to produce a list of ranked Segments, like the methods that will be presented in Section 7.6. Change-Segmenting algorithms are implemented using the same Machine-Segmenting framework, but they only produce a single monolith Segment containing all identified Changes.

Once a Machine-Segmenting method is implemented the only point of contact with its internal workings comes via the method's settings. The user enters these settings via the *Machine Segmentation Settings Window* shown in Figure 84. Each of the rows represents one run of the algorithm across and entire Project History, utilising the settings specified in that row. Settings can be composed of an arbitrary number of user-defined Integer and Floating-Point numerical values, Boolean values and String values, each of which is identified via a unique name. For each setting the *Machine Segmentation Settings Window* produces one entry field (or checkbox) for each row of settings. The input settings are specified as part of the implementation of a segmentation method.

Method4.3\_FriendMainMerged

5.0	5.0	0.0	0.0	0.0	0.0	0.0	0.0
adjacent#	maxDistance	lookInProximityChanges	lookInProximityLines	proxAdjSearchDist	parseSearchDist	parseAdjSearchDist	friendLookDistance

5.0	10.0	0.0	0.0	0.0	0.0	0.0	0.0
adjacent#	maxDistance	lookInProximityChanges	lookInProximityLines	proxAdjSearchDist	parseSearchDist	parseAdjSearchDist	friendLookDistance

Setting Initial shortLivedBoundary : 0.0, shortLivedAdjSearchDist : 0.0, outlierDensityAcceptBoundary : 0.0, lookInProximityLines : 0.0, parseSearchDist : 0.0  
Setting CAMetrics\_1 shortLivedBoundary : 0.0, shortLivedAdjSearchDist : 0.0, outlierDensityAcceptBoundary : 0.0, lookInProximityLines : 0.0, parseSearchDist : 0.0

LoadFromFirst Add Load OK WriteOut ReadIn DefaultLabel Label

**Figure 84: The Machine Segmentation Settings Window**

The *ASegmentationMethodFactory* provides functions which must be overridden to return the names and default values for all of the method's settings. This allows a researcher the flexibility of utilising any number of settings for the implementation of the Machine-Segmenting algorithm.

Along with settings for the Machine-Segmenting algorithm, the settings window also allows the user to input the Segment cut-off level. The largest Segments whose sum of Changes exceeds the percentage of total Changes specified by the cut-off level are used to calculate evaluation statistics discussed in Section 7.4.

The user can also import settings from or exported as plain text; the import/export text area is located below the setting rows, as can be seen in Figure 84. It contains the name of each setting (only if it is set to a non-default value) and the value assigned to it. This makes it easy to re-run previously applied settings, or to slightly modify settings.

The generation mechanism automatically stores the results of the Machine-Segmenting process as an HTML file containing evaluation data and as a .txt file containing a machine-readable list of Segments. This enables the researcher to load previously generated Segments into the SCORE Analyser. This circumvents the need to re-generate Segments during every analysis session. The format of data produced by Machine-Segment Generation for offline use stored in HTML documents is presented in detail in the appendix, Section 9.5.3.11.5. The next section will describe the way in which Machine-Segmenting output is presented in the SCORE Analyser.

## 7.5.2 SCORE Analyser Segment Output

The user can machine-generate Segments either directly in the SCORE Analyser for a single assignment in the SCORE Analyser Machine-Segmenting View, or they can be bulk-generated for sets

of student assignments via a stand-alone program called *GroupSegmentGenerator* that is provided as part of the SCORE Analyser.

### 7.5.2.1 SCORE Analyser Machine-Segmenting View

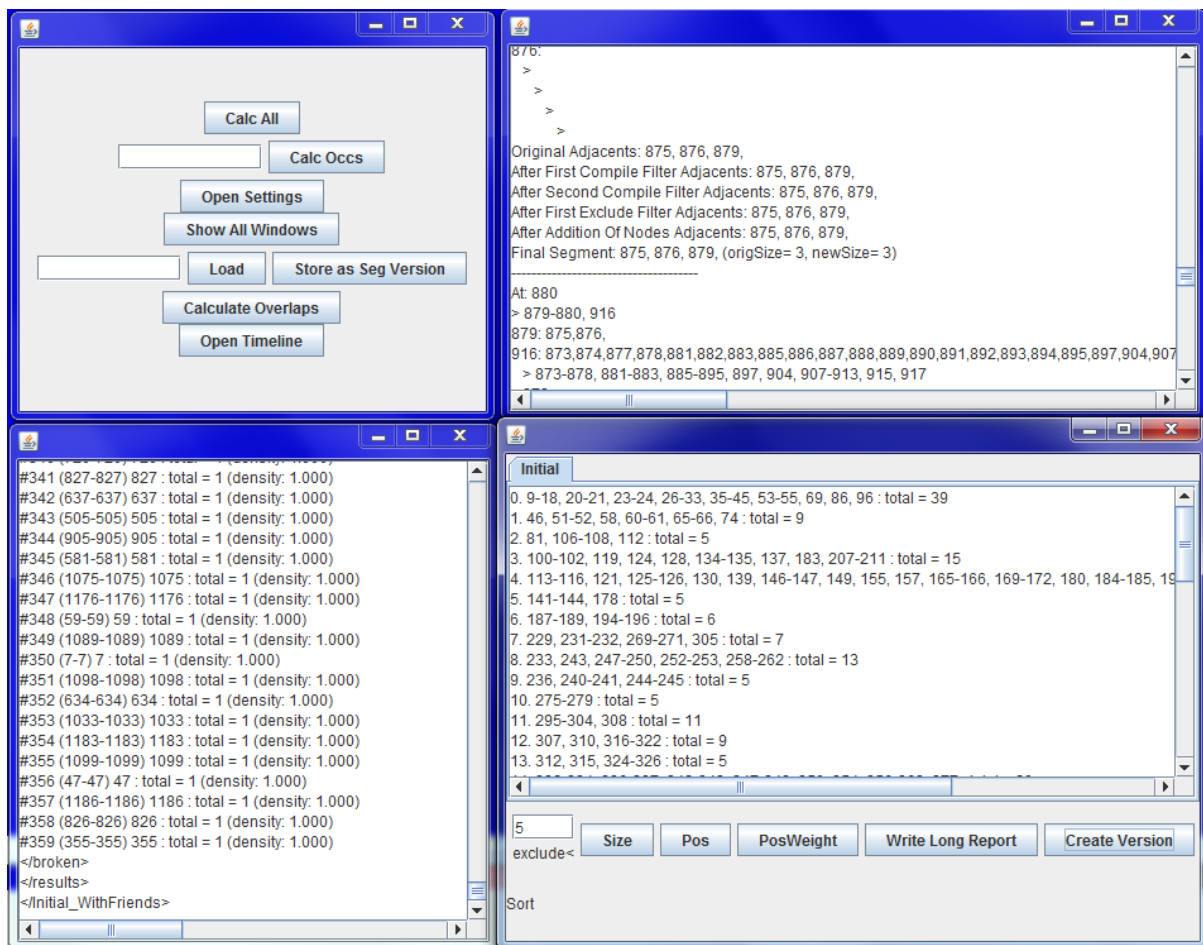
If Segments are produced for an individual student from inside a running SCORE Analyser (instead of using the stand-alone application) or a list of Segments is loaded from a machine-readable list of Segments<sup>45</sup>, then the *Machine-Segmenting View* will provide output relating to the Machine-Segmenting process.

The stand-alone *GroupSegmentGenerator* program generates candidate Segments from all Changes. The SCORE Analyser machine-generation mechanism can be instructed to generate Segments from all Changes, or from a range of Changes as specified via the control panel (Figure 85, top left). This allows for more targeted experimentation to identify which Changes are grouped by a particular Machine Segment generation algorithm for a particular Change or set of Changes.

---

<sup>45</sup> Such lists storing generated Segments are produced by both the SCORE Analyser Machine-Segment Generation View and the stand-alone external Segmenting applications packaged with the SCORE Analyser

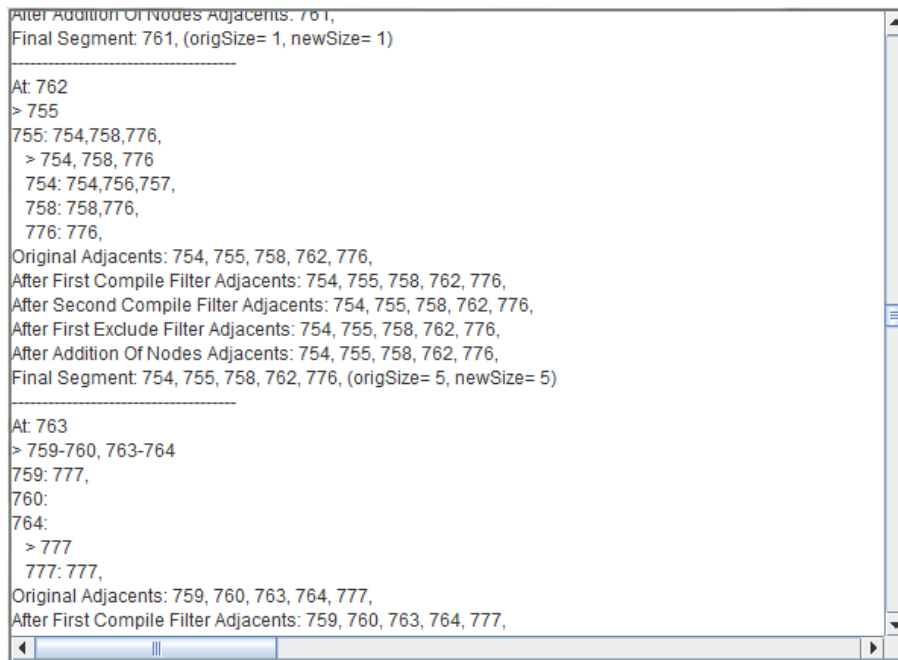




**Figure 85: SCORE Analyser Machine-Segment Generation Views; Control Panel (upper-left), Debug View (upper-right), Segment list (lower-left), Navigation View (lower-right)**

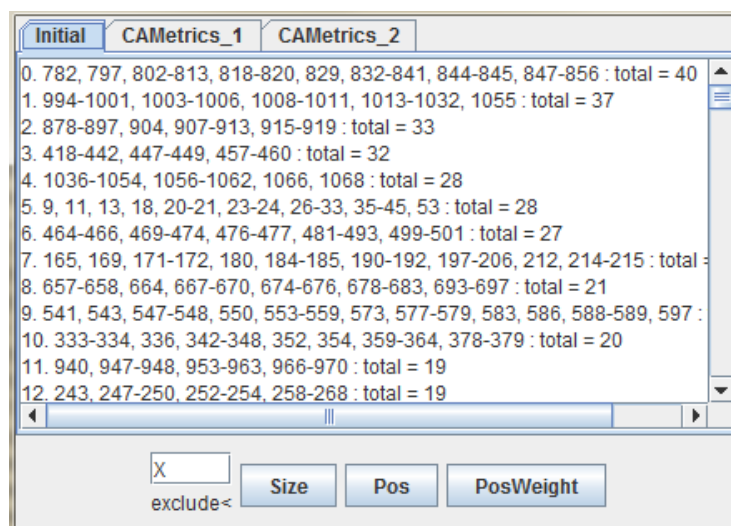
In addition to generated Segments being saved as HTML files as described in Appendix Section 9.5.3.11.5, generated Segments are also displayed in the Segment list which is part of the SCORE Analyser Machine-Segment Generation View (Figure 85, bottom-right).

Machine Segment generation via the SCORE Analyser also provides the user with debugging output not available when Segments are batch-generated via the stand-alone *GroupSegmentGenerator* as shown in Figure 85 (top-right) and Figure 86. This output is generated by the machine generation method writing to a special Debug object. This debug facility can be used to analyse the inner workings of a Machine-Segmenting algorithm, such as how Changes are identified or filtered out by different extension algorithms. Depending on the researcher's use of the Debug object these statements can be very verbose, in which case the debug functionality is best used with a limited number of Machine-Segmenting runs so that the text content of the window does not overload available memory.



**Figure 86: SCORE Analyser Machine-Segment Generation Debug View**

The Segment Navigation View (see Figure 87 bottom right, Figure 87) allows the user to load a Machine Segment's Changes into the Change Browser discussed in Section 6.2.2.1. This produces a view summarizing all the modifications occurring in the Changes belonging to the machine-generated Segment. This allows the user to efficiently gain an understanding of the content of a machine-generated Segment, as well as (through in-depth study of these modifications) an understanding of whether the Segment is actually properly describing related Changes.



**Figure 87: SCORE Analyser Machine-Segment Generation Segment Navigation View**

### 7.5.2.2 Segment Timeline View

Machine-Generated Segments can also be visualised via a separate Timeline View (see Figure 88) launched from the Machine-Generation View control panel. The Timeline View allows the user to perform a visual comparison between human-identified Segments and Segments from an arbitrary number of machine-generation runs.

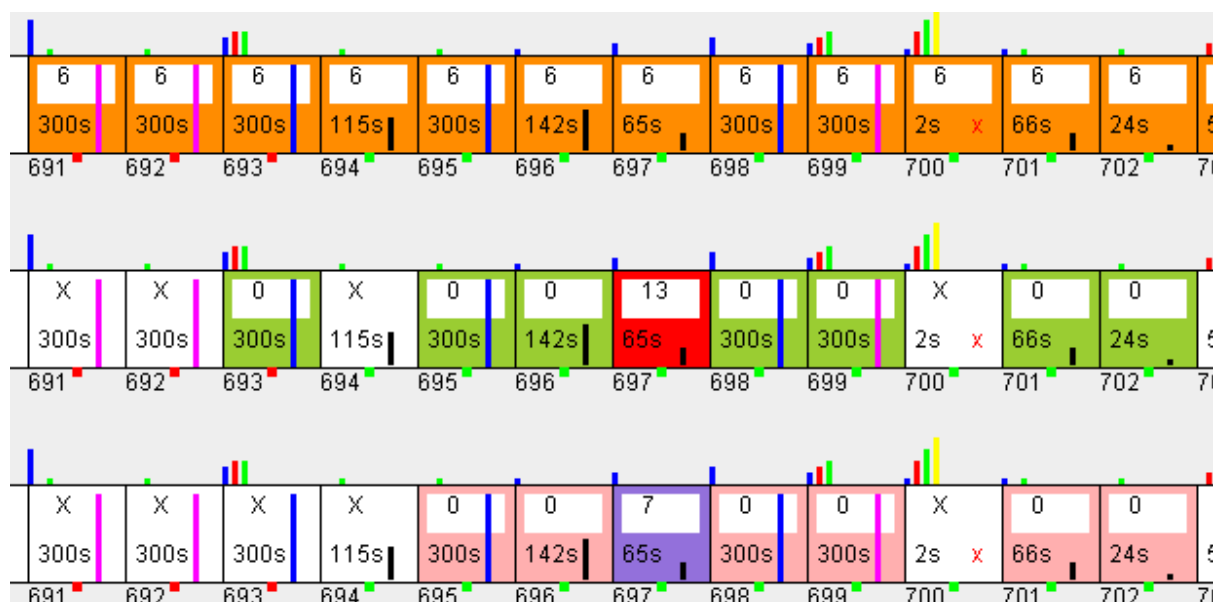


Figure 88: Excerpt from the Segment Timeline View

Each box in the View represents a Change (see Figure 89), with the overall position of the Change being displayed below the box. Next to the overall position, the small coloured box indicates the Change's compile status (red for not compiled, green for compiled). Inside the box, the time taken by the Change is displayed as a number of seconds, with the bar next to the number providing a visual representation of time taken. The bars on top of the Change box show the number of additions (blue), deletions (red), mutations (green), moved (yellow) and ghost (purple) modifications occurring in that Change. The white box containing a number shows the id of the Segment to which the Change is assigned. For example, in Figure 88 six Changes are assigned to the same Segment with id = 0. The Segment id is also visually represented via the box colour, allowing for fast identification of 'related' Changes making up a Segment.

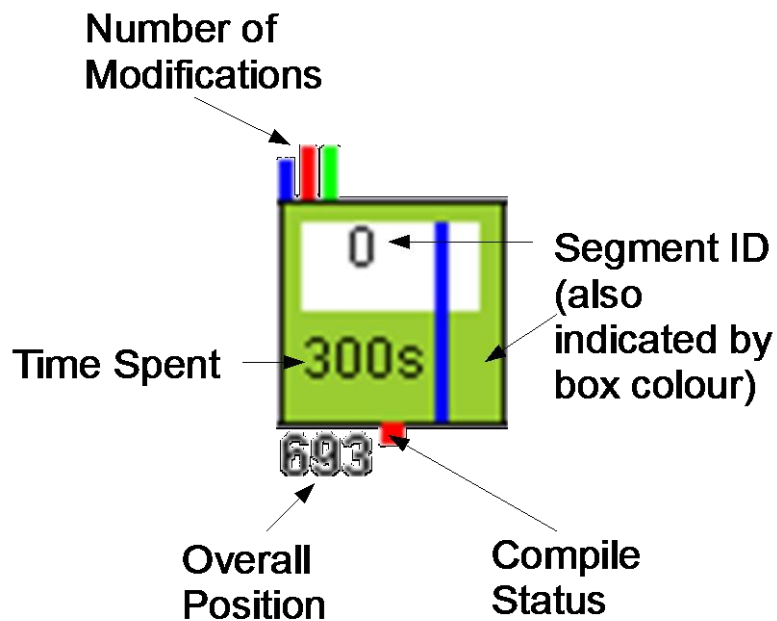


Figure 89: Representation of a Change in the Timeline

The top row in the Timeline View shows the human-identified Segments as entered by the user in the Segment View. Each row below that shows Segments produced by a machine-generation run carried out during the previous machine-generation of Segments. This allows the user to quickly visually compare how the Segments produced by different machine-generation runs match up with the ‘correct’ Segments identified by the human researcher.

In addition to being used for evaluation of machine-generation purposes, the user can also utilise the Timeline View to provide a visual representation of human-identified Segments, as well as the time taken by Changes and modifications occurring during Changes to aid in detailed analysis of a Project History.

### 7.5.3 Machine-Segmenting Walkthrough

In this section, a walkthrough for the two common applications of Machine-Segmenting functionality will be presented.

#### 7.5.3.1 Evaluation

The Machine-Segmenting functionality implemented in the SCORE Analyser can be used to evaluate the performance of a Machine-Segmenting algorithm. To enable this evaluation, the Project Histories to be evaluated against must first be manually segmented using the approaches outlined in Section 6.3.4 to produce ‘correct’ human-identified Segments.

With this comparison data available, the user can then machine-generate Segments as described in Section 7.5.1 with a range of different settings. The Machine-Segmenting generation algorithm will automatically produce data comparing the ranked list of machine-identified Segments to the set of human-identified Segments and will calculate the metrics described in Section 7.4 for evaluation purposes. The user can also use the *Segment Timeline View* to visually examine how well Machine Segments map to human-identified Segments. This will allow the user to evaluate methods, extension algorithms or individual settings to develop Machine-Segmenting algorithms that perform well on the evaluation data through fine-tuning and development of additional extension algorithms. Final evaluation of developed algorithms and settings should then be carried out on an independent data-set to verify good results were not achieved by over-tuning.

### 7.5.3.2 Segment Discovery

Once algorithms for Machine-Segmenting have been developed and shown to be effective through evaluation these algorithms can be utilised to support Segment discovery when engaging in *Segment-Coding* as described in Section 6.3.4.

Instead of having to start from scratch, machine-generated Segments provide a starting point for the researcher. The researcher can analyse machine-identified Segments by loading the Segment's Changes into the *Change Browser* (see Section 6.2.2.1), through manual examination of individual Changes and through examination of Line Histories being modified in the Segment.

The researcher can then utilise machine-generated Segments as a basis for creation of proper Segments in the *Manual Segmenting View*, either based on a single machine-generated Segment's Changes or through the merging and splitting of several machine-generated Segments containing related Changes. The Segments produced in this way will then be refined iteratively during analysis of Segment contents in the fashion described in 6.4.2.

Once machine-generated Segments have been produced, they can be sent to the Segment-Coding view by pressing the 'Store Version' button in the Segment Navigation View. This will create a new segment-coding version in that view and populate it with all the Machine Segments currently visible in the Segment Navigation View in the order that they appear. This means that any filtering by size or ordering of Segments in order of appearance will affect the way in which Segments are stored. Further development of the SCORE Analyser will provide functionality to split and merge such Segments to simplify the process of creating human-identified Segments from machine-identified Segments described above.

The SCORE Analyser Machine-Segmenting facilities are described in more detail in the manual, Section 9.5.3.11, and are demonstrated in the video at <http://www.youtube.com/watch?v=sBR5PRXrREQ&feature=plcp> which is also included in the electronic appendix.

## **7.6 Description of Machine-Segmenting and Change-Identification Algorithms**

This section will describe the algorithms for the Change-Identification or Machine-Segmenting of Project Histories developed as part of this research project. The algorithms are presented in increasing order of effectiveness as discovered during the evaluation, in order to demonstrate the overall process associated with the final algorithm development.

Two different types of algorithms are presented. The last two algorithms presented are ‘true’ Machine-Segmenting algorithms which divide Project Histories into Segments consisting of sets of Changes (Machine-Segmenting). The first three algorithms do not actually Segment the Project History but instead merely return a single set of Changes deemed to be ‘interesting’ (Change-Identification). In essence, the first three approaches return only a single ‘Segment’. The Change-Identification algorithms were developed first and the File Metrics and Line Metrics algorithms are based on activity of lines of code in recent history. The Machine-Segmenting methods were developed later and are based on the concept of co-change. While the Machine-Segmenting algorithms outperform the Change-Identification algorithms, these two types of algorithms present two different potential approaches to the machine analysis of Project Histories and hence evaluation and comparison of both types is presented in the body of the thesis. Evaluation of the algorithms is presented in Section 7.7.

### **7.6.1 Identification of interesting Changes by Random Selection**

The Random Segmenting algorithm chooses random Segments of size  $x$  until Segments totalling size  $y$  have been chosen. If  $(y \% x \neq 0)$  the final Segment chosen will be of size  $y \% x$  instead of size  $x$  so that exactly  $y$  Changes will be chosen.

The Random algorithm is not a viable candidate algorithm since it only does as well as chance. However, it is useful as a benchmark for evaluating other algorithms as discussed in Section 7.4.2.

### **7.6.2 Identification of interesting Changes by Measuring Recent Modification of Files (File Metrics Method)**

The File Metrics (FM) algorithm to identification of ‘interesting’ Changes utilises diff comparisons between the file’s current and previous version.

The reasoning underlying this algorithm is that a large amount of modifications to files indicates significant student effort. Hence, Changes for which the file has recently been heavily modified are more likely to be ‘interesting’. These ‘interesting’ Changes are returned as a set and are not structured in any way.

The algorithm processes the File History of each file occurring in the Project History in turn. For each File History, each Change (pair of the current and previous version) is visited in turn. A diff-comparison using a standard diff algorithm (see Figure 90) is performed between the text of the previous and the current version. This produces a count of lines added to and deleted from the current version. The number of added and/or deleted lines (which modification is used can be set via parameters) is used to calculate a rolling average of modification counts. The current value of that rolling average is stored along with the Change. After this process completes for all File Histories and their associated Changes, a rolling average value containing a measure of the number of lines added and/or deleted in the Change and the Changes immediately preceding it will be stored for every Change in the Project History. A graph showing these rolling averages for an example Project History is shown in Figure 91.

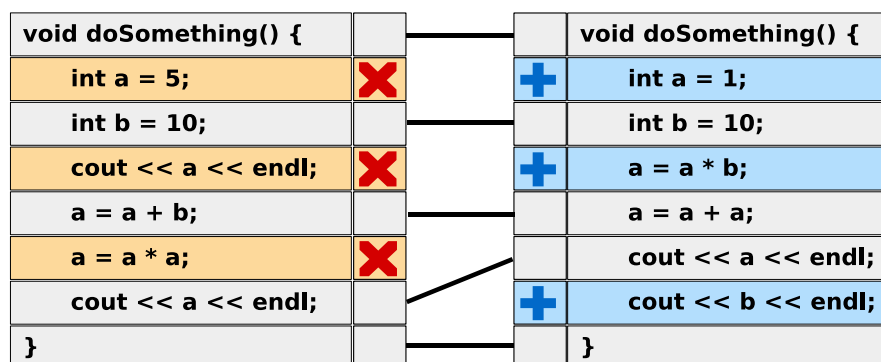
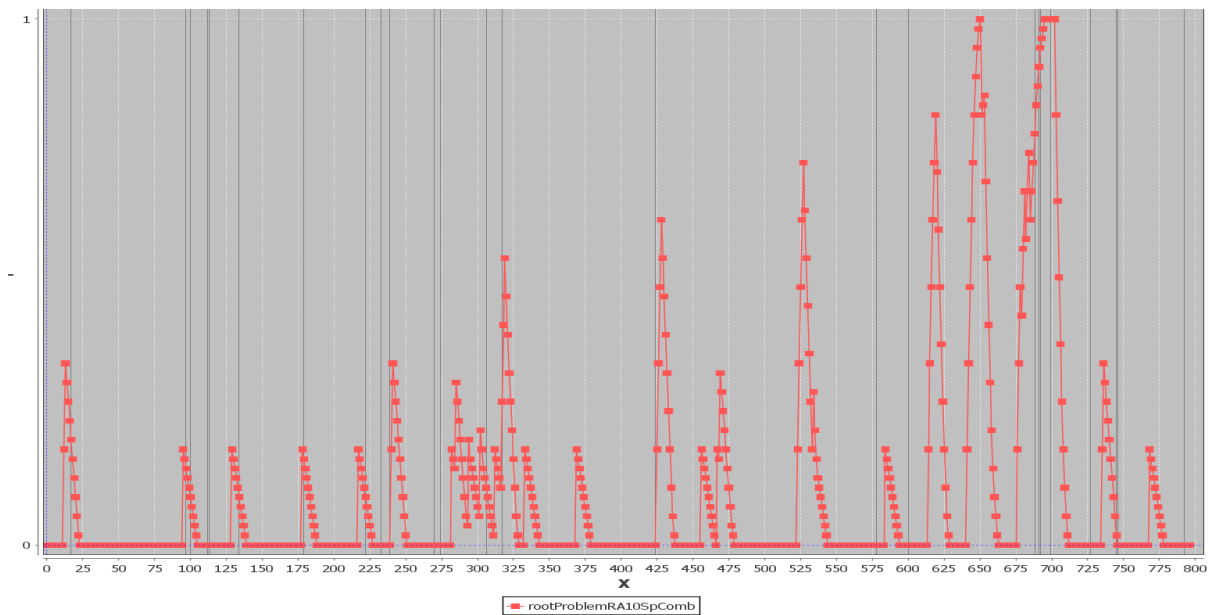


Figure 90: Diff algorithm comparing two Changes.



**Figure 91: Graph showing the rolling average of modifications (deletions and additions) in the recent past.**

Changes are then sorted by this metric which measures recent activity. The *SegNrToSelect* (a variable parameter) Changes with the largest or smallest value are selected. Whether largest or smallest Changes are selected can be set via the X parameter. These Changes are then returned by the algorithm as candidate ‘interesting’ Changes.

### 7.6.3 Identification of interesting Changes by Measuring Recent Modification of Lines (Line Metrics Method)

The Line Metrics (LM) algorithm to identification of ‘interesting’ Changes utilises specialised diff comparisons to detect modification between the lines of each current and previous version in the Project History. The difference between the diff algorithm required for the Line Metrics approach and a standard diff algorithm is that the LM diff algorithm needs to detect not just addition and deletion of lines but also modification and movement of existing lines. Standard diff algorithms would instead represent these modifications as pairs of deletion/addition. In this evaluation, the diff algorithm used is that developed for Line History Generation described in Section 6.2.1.2 which detects Addition, Deletion, Mutation, Ghost and Moved modifications.

The reasoning underlying this algorithm is similar to that of the FM algorithm. A large number of modifications to lines is hypothesized to indicate significant student effort. Hence, Changes containing lines undergoing significant recent modifications are more likely to be ‘interesting’. Compared to the FM method, the LM method’s line-level granularity is hypothesized to be more accurate since modification of the same line may be likely to indicate an error with the line being addressed, meaning the modification is likely to be more significant than the adding of a new line.



More modification of a line in recent history would then indicate a significant error being worked on. Just as with the FM method ‘interesting’ Changes are not grouped in any way.

Like the FM algorithm the LM algorithm also traverses all Changes of File Histories in turn. However, instead of calculating a single diff value from the sum of additions and deletions occurring between the previous and current versions, the algorithm calculates a separate activity value for each of the current version’s lines.

Activity values are calculated by matching lines from the previous to the current version’s document. If the line in the current document has been added (it’s not matched by any previous line) it is assigned a fixed value *AddedValue*. If it has been modified or moved, then it is assigned the value that it had in the previous version plus the *MutatedValue* or *MovedValue*. If a line is unchanged between the two versions, then it is assigned its line value from the preceding document multiplied by a *DecayValue* ( $> 1.0$ ) ensuring that the value of lines not being modified decreases towards 0 over time. Once the algorithm completes for all File Histories and their Changes, each Line in every Change will have a value measuring its recent activity in terms of modifications applied to it.

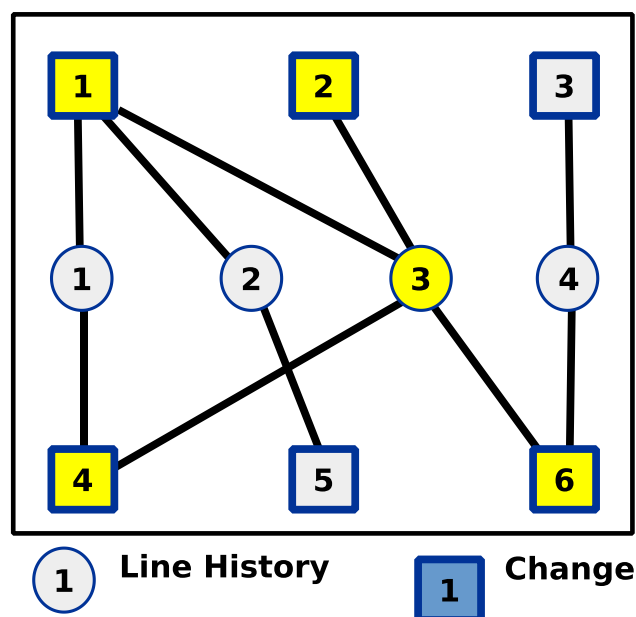
Changes are then sorted in descending order either by the sum of all of a Change’s line metric values or by the Change’s maximum line metric value, depending on the boolean parameter *UseValueSum*. The *SegNrToSelect* Changes with the largest value are selected and returned as candidate ‘interesting’ Changes.

#### 7.6.4 Generation of Segments using Line Histories (LH Method)

The Line History (LH) algorithm for identification of ‘interesting’ Segments of Changes mines Line Histories to detect and group related Changes. The output is not a set of Changes as with the algorithms discussed in the preceding sections. Instead, related Changes are grouped into Segments. Hence, it is the first real Machine-Segmenting approach.

While the LM algorithm also works at a line-level granularity, it is restricted to detecting modifications occurring in recent history. In addition, the LM algorithm does not group identified Changes. There is no straight-forward way of moving from activity values of individual lines to the grouping of the Changes to which these lines belong. The LH algorithm overcomes these limitations. By accessing modifications to a line, the LH algorithm can detect Changes related to that line anywhere in the Project History (instead of only in recent history) and can group those Changes based on the line that is modified in all of these Changes. These sets of Changes are then returned as Segments. The LH algorithm is aware of the full temporal context in which modification to lines occurs rather than the limited temporal context analysed by the LM approach.

The first step is to generate a Line History Table as described in Section 6.2.1.2. To generate Segments, each Line History is visited in turn. A set of Changes in which modifications to the line occur is generated by retrieving the modification data from the Line History. Which modification Changes are included is specified via the *ModificationFlags* parameter. For example, the algorithm can be set to include only Changes in which the line is modified via a Mutated or Moved modification, ignoring Added, Deleted and Ghost modifications. The set of Changes is then stored as a candidate Segment. An example of this procedure is shown in Figure 92, where Line History 3 is modified in Changes (1,2,4,6) which will result in the generation of a Segment containing these Changes. Other Segments in that example will include (1,4) generated from Line History 2, (1,5) generated from Line History 2 and (3,6) generated from Line History 4.



**Figure 92: Line History 3 is used as a basis to generate the Segment 1,2,4,6 consisting of the Changes connected to that Line History**

After all Line Histories have been visited a Change may appear in multiple Segments if multiple lines are modified in that Change. To ensure that each Change appears in exactly one Segment all Line Histories have been visited, a filtering process removes all but the largest Segment containing a specific Change.

When examining generated Segments and analysing which human-identified problem Segments were correctly identified or not identified, the LH generation algorithm's chief limitation becomes apparent. Since it focuses on individual Line Histories, it only detects problem Segments that consist of many modifications to a single line. Other problem Segments that consist of several lines being

modified may not appear in the largest machine-generated Segments if the number of modifications to individual lines is not large enough.

Segments are sorted in descending order of size. If selection of a specific number of Changes is desired, Segments are chosen for inclusion in the result until the addition of the next-largest Segment would lead to the number of selected Changes exceeding the *SegNrToSelect* value after which the selected Segments are returned. Alternatively all Segments can be returned.

### 7.6.5 Generation of Segments using Line History Graphs (LH-Graph Method)

The Line History Graph (LH-Graph) algorithm builds on the LH algorithm, but instead of mining Line Histories in isolation it produces a graph connecting Changes using Line Histories as junctions (see Figure 93). Like the LH algorithm, it also groups related Changes into Segments.

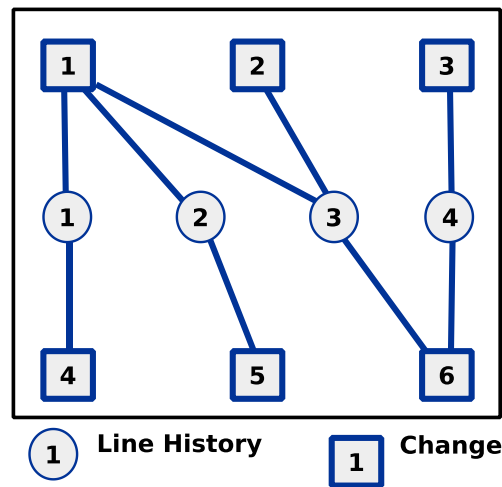
The LH-Graph algorithm is based on the software-engineering concept of co-change (Hassan & Holt, 2004) which involves the modification (addition, deletion or change) of different entities (such as methods or lines of source code) in the same project version (or in the terms used in this thesis, Change).

Most work based on co-change is entity-based (see Zimmermann & Weissgerber, 2004 for example), meaning that co-change is observed at the level of methods or files, whereas the LH-Graph utilises a line-based approach as pioneered by Canfora et al (Canfora et al., 2006, 2007; Canfora & Cerulo, 2006), tracking co-change at the line-level.

Most of the software-engineering approaches discussed in the literature review also use what has been termed a 'Detection of Related Entities' approach; co-change is analysed by identifying entities co-changed in different versions; when they are co-changed in the same temporal context (same version) they are assumed to be related. An example of this approach is used by Livshits and Zimmermann (2005) where co-addition of method calls is used to discover patterns of method calls that are added together in the same version.

The LH-Graph method utilises a more advanced '*Structuring of Project Histories*' approach also found in an approach proposed by Adams et al. (Adams et al., 2010). In addition to calculating co-change to identify related entities (in this case, lines of source code), these related entities are used to form a graph to find more distantly related entities. For example, if line A is co-changed in one version with line B and in one Change with line C, then the application of a structural approach can detect line B's indirect relationship to line C via its direct relationship to line A. The way in which the LH-Graph algorithm uses this method to generate Segments will now be described in detail.

The LH algorithm identifies Changes in which modifications to a particular line of source code occur and then forms Segments from those Changes. The LH-Graph algorithm expands this approach. It starts at a Change instead of at a Line History. At that initial Change, it then selects all Line Histories which have been modified (co-changed) in that Change and visits each Change in which that Line History was modified and adds them to the Segment. This is based on the fact that these Changes are co-modified with at least one of the source code lines from the initial Change. Then in each Change visited, the same algorithm is executed with each Line History being visited until a set depth of visitation is reached. The graph which is traversed by this approach is shown in Figure 93.



**Figure 93: Changes are connected by edges for which Line Histories form junctions**

The reasoning behind this approach is that a problem underlying a set of Changes may involve the modification of more than one line. For example, the implementation of an animation may involve the addition and modification of several OpenGL transformation calls. The lines implementing these calls are not necessarily going to be co-changed together every time. However, there is a good chance that they will be co-changed together on some occasions if they are conceptually related. By visiting all Line Histories of lines modified in those Changes where co-modification occurs, Changes involving the modification of not just one but many related lines can be detected and grouped to form Segments. These Segments are likely to include more of the Changes relevant to the underlying problem.

The first step is to generate the Line History Table containing Line Histories for all lines occurring in the Project History as described in Section 6.2.1.2. Instead of visiting every Line History in turn as was the case with the LH algorithm, every Change is visited in turn.

At every Change, a recursive algorithm determines all 'adjacent' Changes. The term 'adjacent' refers to Changes that are reachable from the current Change by using Line Histories modified in the

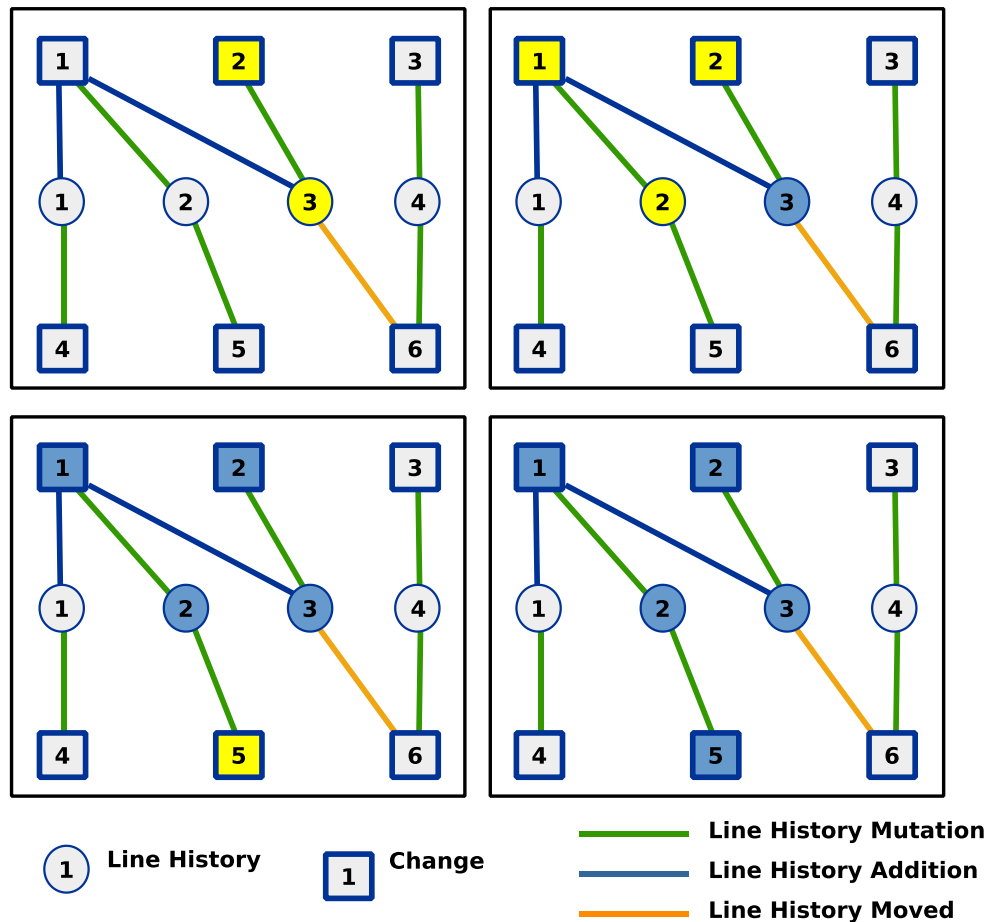
current Change as junctions. The recursive algorithm begins traversal at the root Change. It visits all Line Histories of lines modified in that Change if their modification type matches that of the modification types encoded by *ChangeLineFlags*.

Each of these Line Histories is used as a junction. All Changes in which modification to the line occurs are visited if they meet two conditions. First, their modification type (Added, Deleted, Mutated, Ghost, and Moved) must match a modification type encoded by *LineHistoryFlags*. Second, the Change must be within *MaxDistance* of the Change from which the Line History was visited. These Changes are added to the set of 'adjacent' Changes, and the process repeats, visiting all Line Histories in those Changes. The recursive algorithm continues until either no unvisited Changes are found or until a set depth specified via the *MaxDepth* parameter is reached. The set of all visited Changes is called the set of 'adjacent' Changes.

After 'adjacent' Changes have been identified, optional extension algorithms using the set of adjacent Changes as input can be executed to add additional related Changes to the set of identified Changes. Several such algorithms are presented and evaluated in Section 7.8 and 7.9. Optional filtering algorithms can then remove Changes detected to be unrelated from the set of identified Changes. Two filtering algorithms are presented in Sections 7.8.1 and 7.8.2. Additional extension or filtering algorithms can be implemented via a simple framework.

Again as with the LH algorithm only the largest Segments containing any given Change are kept. The remaining Segments (each Change being contained in exactly one Segment) are returned. If preferred, only Segments containing a total number of Changes not exceeding *SegNrToSelect* can be returned.

To better illustrate the LH Graph algorithm, an example based on the graph shown in Figure 94 will be discussed step by step. In this example, the algorithm is configured to only visit Line Histories that are connected to a Change via a Mutant modification, ignoring other modification types.



**Figure 94: Line Histories navigated as a graph, with navigation commencing from Change 2**

In the first step, Change 2 is visited and added to the Segment's Changes. Change 2 contains one line modification, a Mutant modification of Line History 3. Since the modification is a Mutant modification, it is visited.

In the next step, all Changes connected to Line History 3 (the Changes in which the Line History is modified) are selected for visitation. This includes Changes of modification types other than Mutant, since the modification restriction is only in place in deciding which Line Histories to visit from a Change, not when deciding which Changes to visit from a Line History. Therefore, Changes 1 and 6 are selected for visitation and added to the Segment's Changes. Change 2 is already marked as visited and is hence ignored. Two Line Histories are modified in Change 1. However, one modification is of type Added (the line is added in this Change) and hence the associated Line History 1 is ignored. The other modification is of type Mutated, so the associated Line History 2 is visited. For Change 6, the only associated Line History 4 is of type Mutated, so it is visited.

In the third step, Change 5 is visited from Line History 2 and Change 3 is visited from Line History 4. Since Change 3 and Change 5 both are not connected to any unvisited Line Histories the algorithm

terminates. The Segment generated consists of the Changes (1,2,3,5,6). The largest Segment that could have been generated by using individual Line Histories without graph navigation would have been the Segment (1,2,6) originating from Line History 3. In practice, the difference is often significantly larger, with large Segments identified by the Line History Graph algorithm being recognised as many small Segments by the Line History algorithm.

## 7.7 Evaluation of Machine-Segmenting and Change-Identification Algorithms

In this section, the different methods for the identification (and in the case of LH and LH-Graph, grouping) of ‘interesting’ Changes described in Section 7.6 will be evaluated using the evaluation methodology presented in 7.4. For a description of the format of data presented, see Section 7.4.5.

A detailed evaluation of each individual method is presented in Appendix Section 9.8.7. Below is the summary of this evaluation, giving only the best run out of multiple runs utilising different algorithm settings for each method.

Table 29 shows the best run for each of the evaluated algorithms (except for the trivial Random algorithm). The difference between the FileMetric and the LineMetric approach clearly demonstrates the advantage of using Line-level granularity in detecting ‘interesting’ Changes. The LineMetrics algorithm achieves significance in 9/10 assignments, showing that it is effective in identifying ‘interesting’ Changes. However, the LineMetric algorithm does not group these Changes.

**Table 29: Best run of FileMetrics, LineMetrics, LH and LH-Graph algorithms**

Method	Ratio	% of Max	Spread	Fit	spread* fit	p-val	<0.05	<0.01
FileMetrics	1.320	55.3%	-	-	-	0.123	6	2
LineMetrics	1.453	60.9%	-	-	-	0.018	9	6
LH	1.545	64.8%	0.738	0.789	0.582	0.010	9	7
LH-Graph	1.643	67.8%	0.681	0.756	0.515	0.003	10	9

The Line History algorithm improves on the LineMetrics algorithm in all categories. It produces the same number of significant results at  $p < 0.05$  but produces one extra significant result at  $p < 0.01$ . The average p-value is cut in half when compared to the LM algorithm. It also produces an increase in

the A/E ratio of ~10%. The LH algorithm also groups 'interesting' Changes. A spread of spread=0.738 and fit=0.789 indicates the approach is very successful in grouping related Changes.

The Line History Graph algorithm performs substantially better (according to p-values) than the LH algorithm in detection of 'interesting' Changes. It produces significant results at the  $p < 0.05$  level for all assignments, and at the  $p < 0.01$  level for 9/10 assignments. The average p-value is roughly 1/3 of that of the LH algorithm. The A/E ratio is improved by ~10% over that produced by the LH algorithm. However, the LH-Graph algorithm appears to perform more poorly at segmenting, since both its spread and fit value are slightly lower than those of the LH algorithm. This suggests that the additional grouping of Changes achieved by graph navigation of Changes may be grouping unrelated Changes leading to less precise Segments. However it is important to note that some Segments are in fact related to other such Segments. Such relationships are not explicitly described in Segments. It is possible that the LH-Graph algorithm is detecting such Changes belonging to related Segments and is hence assigning them to the same segment. This would lead to it achieving poorer fit and spread values. An example would be the assembly of the avatar and the animation of the avatar. Many students co-modify the source code belonging to assembly and animation when they are moving from one task to the next, leading to the entire stretch of programming being detected as one Segment.

While the LH algorithm does appear to produce slightly better segmenting in terms of fit/spread than the LH-Graph algorithm, the LH-Graph algorithm produces far superior results in identification of 'interesting' Changes, associated with a lower average p-value and a significantly better A/E ratio. The LH-Graph algorithm, utilising line-level histories and graph navigation of the Project History, produces the best results. This demonstrates its potential as a valuable part of a methodology for the source-code level analysis of Project Histories.

The LH-Graph algorithm also offers multiple extension points at which algorithms to improve results further can be applied. An evaluation of several extension algorithms that were developed to further improve the LH-Graph algorithm is presented in Sections 7.9 and 7.10.

A detailed evaluation of the LH-Graph using different 'distance', 'proximity' and 'modification' settings can be found in the appendix, Section 9.8.6.

## 7.8 Description of LH-Graph Extension Algorithms

This section describes sub-algorithms designed to improve the performance of the LH-Graph algorithm. As described in Section 7.6.5 these sub-algorithms receive Segments generated by the



core LH-Graph algorithm as input and then either filter the Segment to remove unrelated Changes or attempt to find additional related Changes to include in the Segment.

Each algorithm and its underlying rationale are described. In addition, the pseudo-code for each algorithm is described in the appendix; individual sections will contain a link to the algorithm's pseudo-code.

### 7.8.1 Compile Filter Algorithm

#### Algorithm Description

The compile-filter algorithm can operate in three modes. In Compile-Only mode it removes all non-compiled Changes from machine-identified Segments. In NonCompile-Only mode it removes all compiled Changes from machine-identified Segments. In Adaptive mode, it counts up the total number of compiled and non-compiled Changes in a Segment and removes the type of Change of which a smaller number is present in the Segment.

The reasoning behind this algorithm is that problems tend to revolve either around some syntax or semantic issue that is preventing code from compiling or around a logical issue that does not prevent the program from executing, but that makes it produce incorrect output. The compile filter makes it possible to focus only on one kind of problem (either syntax/semantic or logical problems) in Compile-Only or NonCompile-Only modes. In Adaptive mode, it tries to ensure that Segments only encapsulate Changes from syntax/semantic OR logical problems.

#### Setting Options

CompileType: *COMPILE\_ONLY* removes all non-compiling Changes from Segments. *NONCOMPILE\_ONLY* removes all non-compiling Changes from Segments. *ADAPTIVE* removes compiling Changes from Segments containing mainly non-compiling Changes and vice versa.

### 7.8.2 Small-Segment Filter algorithm

#### Algorithm Description

The small-segment filter algorithm divides a Segment into Sub-segments by splitting Segments whenever the distance between any two changes is greater than the *smallSegmentDivide* value. The algorithm then compares the number of changes in the Sub-segment to the boundary value of *smallSegmentDivide*. If the number of changes is smaller than the *outlierDensityAcceptBoundary*, it is removed from the Segment.

The rationale underlying the Small-Segment Filter algorithm is that small, isolated Sub-segments of Segments may not be related to the Segment after all since otherwise there would be more Changes

in the intervening gaps. The lack of such Changes indicates that the problem underlying the Segment was not worked on, and hence these outliers are more likely to represent a clean-up or formatting action rather than actual related work.

#### **Setting options**

`smallSegmentDivide`: The value based on which the parent Segment is divided into Sub-segments to be analysed. Gaps larger or equal to this boundary cause the Segment to be divided.

`smallSegmentAccept`: Sub-segments with sizes below this value are rejected and removed from the Segment.

### **7.8.3 Short-Lifespan Inclusion Algorithm**

The Short-Lifespan Inclusion algorithm examines Line Histories in Changes surrounding an input Segment's Changes. It compares the lifetime of each Line History (calculated by subtracting the position of the Change at which the line is deleted from the position of the Change at which it is added) to the *shortLivedBoundary* boundary value; any Line Histories whose lifetimes fall below that value have all Changes in which modifications to that Line History occur added to the Segment.

#### **Setting options:**

`shortLivedMaxDistance`: The number of changes surrounding each Segment member to examine.

`shortLivedBoundary`: The size below which line histories (and their associated changes) will be accepted and added to the Segment

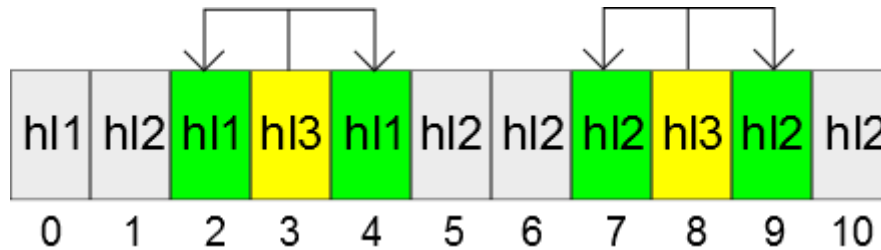
### **7.8.4 Line-History Friend Algorithm**

#### **Algorithm Description**

In the first step the friend algorithm collects all modifications occurring in Changes surrounding (within `friendLookDistance` of) the input Segment's Changes.

In the second step, for each Line History belonging to any of the collected modifications, the total number of collected modifications for that Line History is compared to the total number of modifications for that Line History. If the ratio of (collected / total) modifications is  $\leq$  *friendBoundary* then the Line History is accepted, and all Changes in which it is modified are added to the Segment. If the *addFriendAdj* option is activated, then all Changes adjacent to the Line History's Changes are also added to the Segment. Adjacent Changes are found using the core LH-Graph algorithm for finding adjacent Changes, described in Section 7.6.5.

In the example shown in Figure 95, the original Segment's two Changes (3,8) containing history line 3 are marked in yellow. A *friendLookDistance* value of 1 is used, meaning that Changes within 1 Change distance of Changes in the Segment are selected as friend candidates. In the example case, this means Changes (2,4,7,9) are selected as friend candidates.



**Figure 95: Changes surrounding the Segment's changes (marked 1) are marked by the algorithm**

In the next step, for each line history occurring in one of the friend candidate Changes a ratio calculating how many of the Line History's Changes are marked is calculated. In this example these are line history 1 (lh1) and line history 2 (lh2). Line History 1 occurs in Changes (0,2,4). Changes (2,4) were selected as friend candidates. This means that of all of the Line History's Changes,  $2/3 = 0.66$  of Changes were selected. Line History 2 occurs in Changes (1,5,6,7,9,10) of which (7,9) are selected, leading to a ratio of  $2/6 = 0.33$ .

If the *friendBoundary* is set to *friendBoundary*=0.5, then Line History 1 is accepted since 66% of its Changes are selected as friend candidates, whereas Line History 2 is rejected since only 33% < 50% of Changes are selected as friend candidates. As a result, all Changes involving Line History 1 are added to the Segment, whereas Changes involving Line History 2 are not.

If *addFriendAdj* is set to true, then Changes adjacent to Line History 1's Changes (as detected by the core algorithm) are also added to the Segment.

#### **Setting options:**

*friendLookDistance*: The number of changes surrounding each Segment member to mark.

*friendBoundary*: The percentage of total changes to a line history that must be marked in order for that line history (and the marked changes that contain it) to be accepted and added to the Segment.

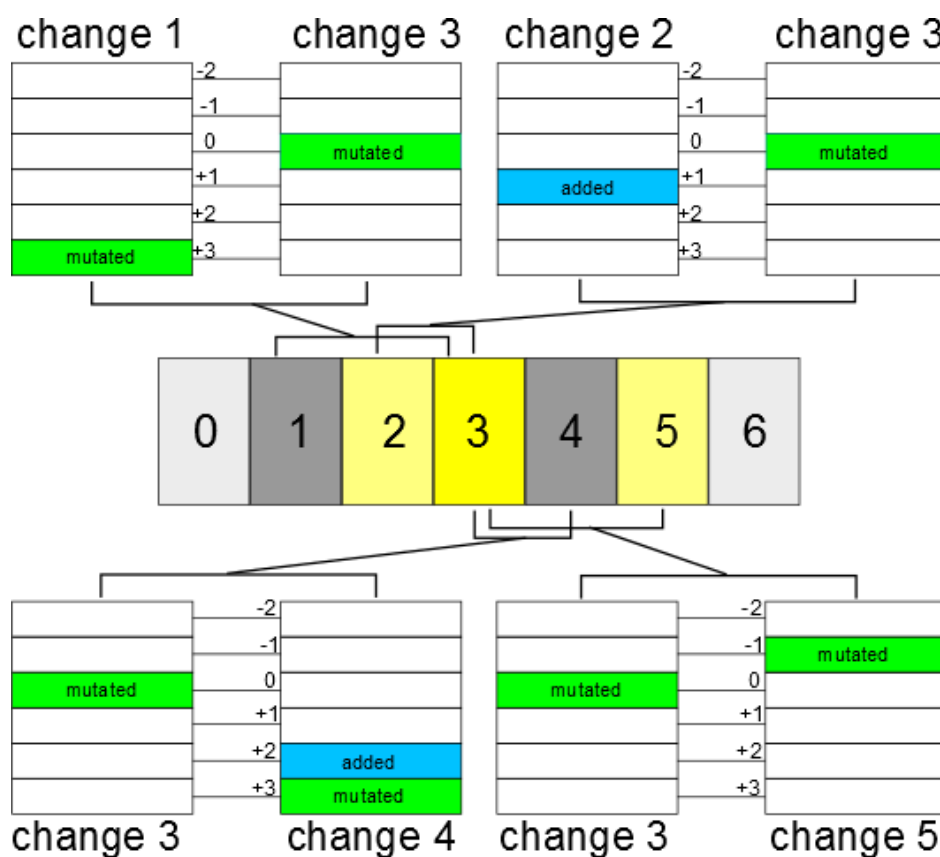
*addFriendAdj*: If true, adds nodes adjacent to any accepted friend nodes

### 7.8.5 Line Proximity Algorithm

The proximity algorithm selects Changes surrounding the input Segment's Changes that contain modifications to lines *near* lines modified in the Segment's Change.

For each Segment member Change, all Changes in *lookInProximityChanges* distance are examined. For each line modification in the member change, the line with that line number and the *lookInProximityLines* lines surrounding it are examined; if any contain a line modification, then the Change is included as part of the Segment.

An example run of the line proximity algorithm is shown in Figure 96. A *lookInProximityChanges* value of 2 is used. The original Segment contains the Change (3), meaning that Changes (1,2) and (4,5) will be evaluated by the algorithm.



**Figure 96: Lines surrounding a line modified in the Segment document are searched for modification in the adjacent document.**

In (3), the third line is mutated. In the main phase of the line proximity algorithm, each of the selected Changes is examined. Given a *lookInProximityLines* setting of 1, the algorithm tests to see whether any line within 1 line of lines modified in Change 3 has been modified in a candidate Change. In this case, since only the third line is modified in Change 3, this involves checking whether

the second or fourth line has been modified in any of the candidates. As Figure 96 shows, Changes 2 and 5 have a line modification within 1 line of the modification in Change 3. As a result, Changes 2 and 5 are added to the Segment, whereas Changes 1 and 4 are rejected.

#### **Setting options:**

`lookInProximityChanges`: The number of changes surrounding each Segment member to scan for proximity changes

`lookInProximityLines`: The number of lines surrounding a line modified in the member change to examine for changes in the candidate change.

### **7.8.6 Code Parsing Algorithm**

#### **Algorithm Description**

For each of a Segment's Changes, the Code Parsing algorithm visits Changes within *maxDistance* of the Segment's Change. The algorithm uses a C++ parser to produce an Abstract Syntax Tree (AST) for both the Segment Change's and the nearby Change's source code. It then gathers the names of function or class identifiers involved in function calls or function or class definitions. It compares the set of identifiers from the two Changes. If there is any overlap, the nearby Change is accepted and added to the Segment.

#### **Settings**

`maxParseDistance`: the distance from which (looking forward) Changes will be compared to the Segment's Change

### **7.8.7 Text Similarity Algorithm**

#### **Algorithm Description**

The Text Similarity algorithm compares the source code text of modifications of a Segment's Changes to the text of modifications in nearby Changes.

It visits each Change that is within *similarityDistance* distance of a Segment's Changes and then compares the text of each line in the Segment's Change to the text of each line in the nearby Change. If the difference between any pair of lines falls below the specified *similarityLevBoundary* value then the Change containing that line is added to the Segment.

The method utilised by the Text Similarity algorithm is essentially the same as the one used for Mutant and Ghost detection as described in Section 7.2.1; however, unlike Mutant or Ghost detection application of this algorithm does not link entries in Line Histories so it can be used with

more relaxed Levensthein distance parameters and a larger search distance (the search distance for Mutants is always 1) to augment Mutant/Ghost detection

### **Settings**

similarityDistance: the distance at which Changes are visited from any one of the Segment's Changes

similarityLevBoundary: the Levensthein distance below which the comparison of two lines will result in the Change being added to the Segment

## **7.8.8 SimProx Algorithm**

### **Algorithm Description**

Since both the Text Similarity and the Line Proximity algorithms produced significant improvements in A/E ratio, an algorithm combining both approaches was implemented. The SimProx algorithm analyses every Change within *simproxDistance*. For that Change, the text of all lines modified is compared to the text of the lines modified in the Segment's Change. In addition the algorithm determines whether any lines surrounding lines in the Change belonging to Line Histories modified in the Segment's Change have been modified. If either condition is met, then the Change is added to the Segment.

The algorithm also has a *keepAlive* mode, in which the algorithm keeps examining Changes until a Change is rejected. The algorithm can also include all Changes adjacent to a Change that is accepted using the core LH-Graph algorithm for finding adjacent Changes, described in Section 7.6.5.

### **Parameters**

simProxDistance: how many Changes surrounding the Segment's Changes are visited

maxLineDistance: the number of lines around lines modified in the Segment's Changes that are examined in the line proximity step of the algorithm

levDistBoundary: the boundary below which the text of two lines is considered matching, resulting in the associated Change being added to the Segment

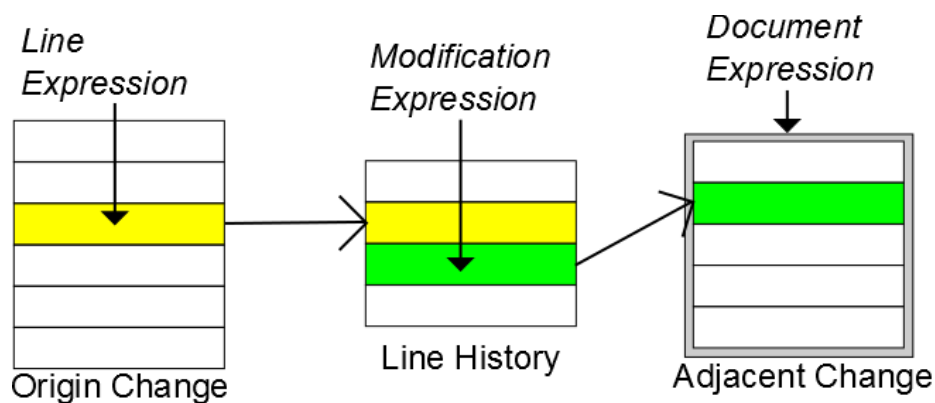
## **7.8.9 Visit Expressions**

### **Description**

Visit Expressions are not really an algorithm. Instead, they are an extension to the core algorithm for generation of Segments presented in Section 7.6.5. In the pseudo-code presented below, changes to that algorithm are presented and highlighted.

Where the core LH-graph algorithm utilised flags based on modification types to determine whether a Change's line or a Line History's modification should be visited, the Visit Expression extension to that algorithm allows the use of arbitrary conditional expressions to determine whether lines / modifications should be visited. In addition to using modification types in an expression, expressions also allow numeric conditionals to test whether the total number of modifications of a certain type fall above or below a set threshold.

Figure 97 shows the different parts of the adjacent-detection algorithm at which expressions take effect. At the origin Change for which adjacents are to be detected, the Line Expression is tested. Only if it evaluates as true is the Line History visited. In the Line History, each modification is tested against the Modification expression. For any modifications for which the Modification Expression evaluates to true, the associated Change is visited if and only if the Document Expression evaluates as true for that Change. Changes that are visited after all expressions being met are then accepted as adjacent Changes to the origin Change.



**Figure 97: The different parts of adjacent-Change detection at which expressions take effect**

Examples are presented below together with the *Parameters* description.

### **Settings**

*Document Expression:*

The Document Expression tests the number of total modifications, or number of modifications of certain types.

Example: The expression `(doc_all < 5 || doc_added < 10)` will accept documents that either have less than 5 modifications total OR less than 10 'Added' modifications.

*Line Expression:*

The Line Expression tests the type of modification to that line, or the number of modifications (total or of a certain type) to the line to which the Change belongs.

Example: the expression *(type == MUTATED || (doc\_all < 5))* will accept lines that have either been modified in the Change or lines that occur in a Change with less than 5 total modifications.

*Modification Expression:*

The Modification Expression tests the type of modification of a Line History's Modification, or the number of modifications (total or of a certain type) to that Line History.

Example: the expression *(type == MUTATED || (hl\_moved < 2))* will accept any modifications that are of type *Mutated* or any modifications from a line history that has less than two modifications of type *Moved*.

### 7.8.10 SimProx and Visit Expressions Combined

#### Description

Best results were achieved by the SimProx algorithm and the Visit Expressions extension to the LH-Graph algorithm. Improvements of ~10% and ~7% of A/E ratio respectively were achieved using these approaches.

Since these approaches were individually effective, both algorithms were combined to evaluate whether the combination would provide an even better improvement.

## 7.9 Evaluation of LH-Graph Extension Algorithms

This section presents the evaluation of the extension algorithms presented in the previous section. For each extension algorithm, the best run discovered experimentally through the use of different settings is presented. Detailed results for the evaluation of individual extension algorithms including all the runs using different settings can be found in the appendix, Section 9.8.9.

The result of the evaluation of the CompileFilter algorithm is shown in Table 30. Application of the CompileFilter algorithm using an adaptive setting produces a marked improvement in A/E ratio. It also produces a significant result at the 0.01 level for one additional Project History. Its application also causes a very small decrease in spread\*fit.



**Table 30: Evaluation results with and without the compile filter**

Setting	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.543	63.5%	0.529	0.007	10	8
CompileFilter	1.643	67.8%	0.515	0.003	10	9

The compile filter is clearly effective at identifying more ‘interesting’ Changes. Since it was one of the first algorithms to be developed and since it was shown to be so effective, it was applied together with all other algorithms evaluated. The results presented next all involve the use of the compile filter and an additional extension algorithm.

Table 31 shows results of the evaluation of the remaining extension algorithms. Algorithms which outperform the basic setting applying only the CompileFilter algorithm are shown in rows marked light blue, whereas algorithms that perform worse are shown in pink. Algorithms which produce a very similar result are shown in white.

The Small Segment Exclusion, Short-Lifespan Inclusion and Friend algorithms do not produce an improvement over the core LH-Graph algorithm even on their best run. The Code Parsing and Text Similarity algorithms produced only small improvements. The Proximity and Visit-Expressions algorithms both produced significant improvements, with the SimProx algorithm producing the largest improvement.

**Table 31: Evaluation results of application of extension algorithms. Algorithms in red produce no improvement.**

Setting	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
No Extension (CompileFilter only)	1.643	67.8	0.515	0.003	10	9
SmallSegments + CompileFilter	1.647	67.9	0.521	0.003	10	9
ShortLifespan + CompileFilter	1.647	67.9	0.517	0.003	10	9
Friend + CompileFilter	1.643	67.8	0.515	0.003	10	9
Proximity + CompileFilter	1.714	69.8	0.535	0.002	10	9
CodeParsing + CompileFilter	1.66	68.3	0.536	0.002	10	9
TextSimilarity + CompileFilter	1.654	68.2	0.531	0.002	10	9
SimProx + CompileFilter	1.748	71.6	0.531	0.001	10	10
Visit Expressions + CompileFilter	1.711	70.5	0.532	>0.001	10	10
SimProx + VisitExpressions + CompileFilter	1.756	71.7	0.54	>0.001	10	10

The two best-performing algorithms SimProx and Visit-Expressions were also applied together as shown in the last row of Table 31, producing a result that is slightly better than that produced by the SimProx algorithm alone.

Based on this evaluation the SimProx algorithm appears to be the most effective extension algorithm, with the SimProx, Visit-Expressions, Similar-Text, Code-Parsing and Proximity algorithms all producing small to significant improvement over the core LH-Graph algorithm. The next section will present evaluation of the core generation algorithms and extension algorithms on an independent data set.

## 7.10 Evaluation of Machine Segmentation, Change-Identification and Extension Algorithms on an independent data set

The evaluation of identification/generation algorithms presented in Section 7.7 and the evaluation of extension algorithms for the LH-Graph algorithm presented in Section 7.9 were both based on the data that was used during algorithm development. This means that the evaluation may be prone to an over-fitting of the algorithms to the data. For this reason, a second round of evaluation based on an independent data set is presented in this section.

In producing the independent data set, a shallower segmenting approach was used than for the Segments used for the implementation of algorithms. This is because the Segments used during the development of the machine algorithm were those analysed in depth earlier, and they had undergone a long process of analysis and refinement involving the breaking down of Segments according to classification categories.

In the shallower approach used for the production of the independent data set, segmenting was carried out based on grouping Changes according to the task the student appeared to be working on, without the additional step of then dividing such Segments into different Sub-segments relating to classification categories. One reason for the use of the shallower technique was that an in-depth approach would have been inordinately time consuming. However, the shallower approach is a more realistic application of Machine-Segmenting in any case. A machine method is incapable of doing the kind of classification and detailed analysis of content that led to the production of the more refined Segments evaluated in Chapter 6 which were used in the initial evaluation of algorithms.

Table 32 shows results for the comparison of the different algorithms described in Section 7.6 on an independent data set of six Project Histories, three from Assignment 1 and three from Assignment 3. These six Project Histories come from five different students (for one student, both the Assignment 1 and Assignment 3 Project Histories were used as they were both short). It should be noted that for one of the Project Histories there were so many 'interesting' Changes that the p-level of 0.01 was unreachable; even given 100% correctly identified Changes, the best achievable p-value was  $0.05 > p > 0.01$ . Therefore, algorithms can achieve significance at the  $p \leq 0.01$  level for only 5 out of the 6 Project Histories.

**Table 32: Evaluation of algorithms on the independent data set**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
FileMetrics	1.116	79.0	-	0.257	2	0
LineMetrics	1.135	80.5	-	0.218	1	0
LH	1.238	87.5	0.585	0.091	4	2
LHGraph	1.371	95.6	0.621	0.003	6	5

The FM and LM algorithms both perform poorly, achieving significance for only 2/6 and 1/6 Project Histories respectively at the  $p \leq 0.05$  level at for none at the  $p < 0.01$  level. The LH algorithm performs better, achieving significance for 4/6 assignments at the 0.05 level and for 2/5 Project Histories at

the 0.01 level. By far the best performance is produced by the LH-Graph algorithm which produces significance for all assignments at the  $p < 0.05$  level and for all possible assignments (5/5) at the  $p < 0.01$  level. The LH-Graph algorithm clearly performs well at identification and selection of large Segments and outperforms other algorithms.

In testing extension algorithms, the Compile Filter was again tested first as it had produced such large improvements in the previous evaluation. As Table 33 shows, the Compile-Filter produces a significant improvement in A/E ratio and in average p-value, leading to significance for 5/5 Project Histories at  $p \leq 0.01$  rather than 2/5 without its application. As a result, it was used in all LH-Graph evaluation.

**Table 33: Evaluation of the LH-Graph algorithm with and without compile filter**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.329	92.6%	0.635	0.016	6	2
CompileFilter	1.371	95.6%	0.621	0.003	6	5

Table 34 shows results for the evaluation of extension algorithms. All extension algorithms perform more poorly than the run using no extension algorithm, producing poorer ratios, lower average p-values and in several cases only 4/5 significant results at the  $p < 0.01$  level, though the differences in ratio are quite small. The algorithm that produces the worst A/E ratio produces the best spread\*fit value.

**Table 34: Evaluation of LH-Graph extension algorithms on the independent data set (including CompileFilter in all runs)**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
CompileFilter Only	1.371	95.6%	0.621	0.003	6	5
Prox + CompileFilter	1.356	94.7%	0.645	0.006	6	4
Parse + CompileFilter	1.364	95.2%	0.636	0.004	6	5
Similar + CompileFilter	1.367	95.4%	0.631	0.004	6	5
SimProx + CompileFilter	1.361	95.0%	0.641	0.005	6	4
Expressions	1.365	95.4%	0.63	0.006	6	4
SimProx+ Expressions + CompileFilter	1.365	95.2%	0.639	0.004	6	5

While the LH-Graph algorithm clearly performed well both in absolute terms and in comparison to other proposed algorithms, the extension algorithms failed to produce any improvement despite some of them being quite successful during the first phase of evaluation. One explanation is that the positive effect of these algorithms was entirely due to over-tuning of the algorithms.

However, it may also be that at least some of the performance difference is due to the different, shallower segmenting approach used to produce the independent data set. It may be possible to develop algorithms more effective in a shallow-segmenting scenario by explicitly designing them around this type of segmenting approach rather than the in-depth segmenting approach for which the proposed extension algorithms were designed.

## 7.11 Performance of LH-Graph for different types of Problems

While the LH-Graph algorithm has been shown to be generally effective in identifying related Changes that form problems, there are certain types of problems that are more reliably detected and some which are not reliably detected by the approach. Whether the LH-Graph algorithm is capable of identifying a problem or not depends on the way in which source code is produced while the problem is being worked on. This is due to the nature of the LH-Graph algorithm, which relies on co-change of lines in Changes to recognise the relatedness of those lines in order to form Segments from the set of Changes in which these lines are modified.

### 7.11.1 Well-identified Changes (true positives)

The core LH-Graph algorithm is good at identifying problems in which the student works on a set of 'core' lines, modifying them often throughout the solving of the problem. These 'core' lines will form the backbone of a Segment, and will help discover other related lines through co-change with these 'core' lines. Correctly identified Segments are *true positives* in Information Retrieval terminology.

One example is [IDA\\_A1.GL.1." Line stipple" \[31\]](#) in which the student works on developing a good pattern for stippling lines connecting UML elements in Assignment 1. As can be seen in Figure 99, many of the Changes involved in the solving of the problem modify the same line. This means that these associated Changes will be easy to navigate to from the gLineStipple line's Line History. The summary of machine-segment / human-segment overlap (Figure 98) shows that over eighty percent of the Changes related to that problem were correctly grouped in a single Segment.

Line stipple 1079-1109 ( : total = 31)	
1079-1090, 1096-1098, 1100-1109 ( : total = 25)	80.64516
-----	---
1092-1095 ( : total = 4)	12.90323
1099 ( : total = 1)	3.225806
1091 ( : total = 1)	3.225806
25/31	80.64516

**Figure 98: Summary of Machine-Segment /Human-Segment overlap**

```

1097, Linelcon.cpp -----
* (36:) glLineStipple(1, 0.5); /* dashed */
-> (36:) glLineStipple(8, 0.5); /* dashed */

1098, Linelcon.cpp -----
* (36:) glLineStipple(8, 0.5); /* dashed */
-> (36:) glLineStipple(8, 0xAAAA); /* dashed */

1099, Linelcon.cpp -----
* (36:) glLineStipple(8, 0xAAAA); /* dashed */
-> (36:) glLineStipple(8, 0xABDG); /* dashed */

1100, Linelcon.cpp -----
* (36:) glLineStipple(8, 0xABDG); /* dashed */
-> (36:) glLineStipple(8, 6); /* dashed */

```

**Figure 99: Excerpt from Ida.GL.1 implementing Line Stipple**

The problem Segment also includes other lines that are not directly connected to glLineStipple's Line History as shown in Figure 100. Here, Changes (1082-1085) do not have any lines that co-occur with the glLineStipple line. These lines are identified via the proximity part of the SimProx algorithm which detects that the modifications to Linelcon.cpp from (1082-1085) occur in lines that are close (in terms of line number) to the glLineStipple line. In this way, 'core' Line Histories can not only form the backbone of machine-identified Segments, they can also aid in the discovery of related Changes that they are not involved in co-change with.

```

1081, Linelcon.cpp -----
* (20:) glLineStipple(1, 0x0101);
-> (20:) glLineStipple(1, 3);
* (27:) glLineStipple(1, 0x00FF); /* dashed */
-> (27:) glLineStipple(1, 6); /* dashed */

1082, Linelcon.cpp -----
+ (18:) glEnable (GL_LINE_SMOOTH); /* Enable Antialiased lines */
+ (19:) glEnable (GL_LINE_STIPPLE);

1083, Linelcon.cpp -----
+ (21:) glEnable (GL_LINE_SMOOTH);
* (18:) glEnable (GL_LINE_SMOOTH); /* Enable Antialiased lines */
-> (18:) glEnable (GL_LINE_SMOOTH); /* Enable Antialiased lines */

1084, Linelcon.cpp -----
+ (37:) glDisable (GL_LINE_STIPPLE);
* (21:) glEnable (GL_LINE_SMOOTH);
-> (21:) glEnable (GL_LINE_SMOOTH);

1085, Linelcon.cpp -----
* (18:) glEnable (GL_LINE_SMOOTH); /* Enable Antialiased lines */
-> (18:) glEnable (GL_LINE_SMOOTH); /* Enable Antialiased lines */

1086, Linelcon.cpp -----
* (24:) glLineStipple(1, 3);
-> (24:) glLineStipple(1, 1);

```

**Figure 100: Excerpt 2 from Ida.GL.1 implementing Line Stipple**

Another example is the implementation of avatar assembly by Thomas ([THOMAS\\_A3.SP.5.1."Naive Assembly" \[39\]](#)). In this problem, Thomas assembles an avatar using OpenGL transformations. Unlike the previous example, there is no single 'core' line in this problem that the student continually works on throughout the problem. However, modifications to the transformation calls for different limbs overlap and this co-change allows the correct identification of 90% of the problem's Changes in a single machine-identified Segment as shown in Figure 101. However, even in such almost-ideal cases some Changes will not be correctly detected. Figure 102 shows a group of three Changes that were not included in the machine-generated Segment. The first Change shows the creation of the right leg via copy of the left leg. Since the action copied lines of code and hence created new lines, no connection to any other Line Histories exists. This Change was correctly linked to changes 52-53 which add a transform call in front of the leg but the Segment was isolated from the main problem Segment because no co-change with any of the main Segment's Line Histories occurred.

Naive Assembly 8-11, 13-17, 20-27, 29-46, 51-53, 122 ( : total = 39)	
8-11, 13-17, 20-27, 29-45, 51 ( : total = 35)	89.74359
-----	---
46, 52-53 ( : total = 3)	7.692308
122 ( : total = 1)	2.564103
35	89.74359

**Figure 101: Summary of Machine-Segment / Human-Segment Overlap for Thomas.Sp.1**

```

46, Main.cpp -----
+ (226:) glPushMatrix();
+ (227:) glTranslatef(0.0, -1.5, -0.5);
+ (228:) glObjects.at(objectAt)->getObjectsByName()["RightUpperLeg"]->draw();
+ (229:) glPushMatrix();
+ (230:) glTranslatef(0.0, -1.5, 0.0);
+ (231:) glObjects.at(objectAt)->getObjectsByName()["RightLowerLeg"]->draw();
+ (232:) glPushMatrix();
+ (233:) glTranslatef(0.0, -1.5, 0.0);
+ (234:) glObjects.at(objectAt)->getObjectsByName()["RightFoot"]->draw();
+ (235:) glPopMatrix();
+ (236:) glPopMatrix();
+ (237:) glPopMatrix();
- (213:) /*RightUpperLeg
- (214:) RightLowerLeg
- (215:) RightFoot*/
- (218:) //glRotatef(180,1.0,0.0,0.0);

52, Main.cpp -----
+ (227:) /**/
+ (228:) glPushMatrix();
+ (229:) glRotatef(90,0.0,1.0,0.0);
+ (230:) /**/
+ (245:) /**/
+ (246:) glPopMatrix();
+ (247:) /**/

53, Main.cpp -----
* (229:) glRotatef(90,0.0,1.0,0.0);
-> (229:) glRotatef(90,0.0,0.0,1.0);

```

**Figure 102: Excerpt from Thomas.Sp.1**

Most extension algorithms produce trade-offs. For example, utilising a larger search distance for the Text Similarity algorithm in the last example might have linked the isolated Segment to the main Segment since the first Change in the isolated Segment includes glVertex calls that are similar to calls used in the rest of the Segment.



However while this might increase the number of true positive results it would also increase the number of false positive results since more unrelated Segments would be linked. Discovering the best trade-off for such settings is necessary to improve the efficiency of the approaches proposed in this research project. This requires both a quantitative evaluation like the one presented in Sections 7.7, 7.9 and 7.10 as well as a detailed understanding of the operation of algorithms and the way in which they can create correct or incorrect linkages between Changes. To acquire a detailed understanding, examination of actual generated Segments such as is presented in this section is necessary.

### **7.11.2 Not identified (false negative)**

The LH-Graph algorithm performs well for problems which involve significant co-modification of lines. Based on the researcher's experiences, in the majority of cases problem-solving strategies do involve co-modification as the student modifies previous approaches in order to narrow in on the correct solution.

However, there are instances in which this assumption does not hold. These tend to be problems in which the student does not have a clear problem-solving strategy and is experimenting with different solution approaches, rapidly adding and removing lines of code. Since lines of code have short lifespans in this problem-solving approach, they do not serve well as 'core' lines to connect different related Changes. This would still not present a problem if there were recognisable transitions from one solution attempt to the next, with the student maintaining some lines from the old approach to work on at least initially. But if the student 'starts over' with each new solution attempt, it is unlikely that enough such connections exist to form large Segments. Such a problem-solving approach will result in multiple small machine-generated Segments whose member lines at first glance may not have much in common, but are in fact logically related to the same underlying problem.

Use of Line Proximity and/or Similar Text algorithms can alleviate this problem if newly created lines are textually similar or in proximity of lines modified in previous solution attempts, but this by itself tends not to be enough to form sufficiently large Segments to detect significance. In addition, some problems involve the addition of lines at different parts in the code. This is especially true for problems related to program flow. Debugging of such issues may involve the addition of different debug cout statements at various parts of the code, for example, and it is very difficult to detect the relatedness of these lines, given their non-proximity and non-textual-similarity.

Christopher.GP.8 is an example of a Segment of Changes that was poorly detected. Figure 103 shows the summary of machine-generated Segments containing Changes of the human-identified Segment. Most of the Segment was captured in small (size 2 or 1) Machine Segments, with only one machine-identified Segment containing 4 Changes. [CHRISTOPHER\\_A1.GP.8."Virtual keyword 2" \[15\]](#) involves the implementation of a ‘virtual’ function. The ‘virtual’ keyword frequently causes confusion for students since incorrect usage will result in the parent’s rather than the child’s method being called. The student encounters this problem during implementation of a virtual method. Since the student is unsure of where the problem lies, he modifies the code at many different points and across files (see Figure 104), changing the visibility of unrelated methods, changing the way in which the virtual method is called and adding and removing the virtual keyword to observe the effect. The different colours in the figure denote the way Changes are grouped into Segments, with those Changes sharing the same colour being in the same Segment. These different actions do not have similar text to the lines being modified, nor do they occur in line proximity (or even the same file).

Virtual keyword 1107-1121	
1108-1110, 1120 ( : total = 4)	26.66667
1116-1117 ( : total = 2)	13.33333
1111, 1115 ( : total = 2)	13.33333
1107, 1118 ( : total = 2)	13.33333
1113-1114 ( : total = 2)	13.33333
1112 ( : total = 1)	6.66667
1121 ( : total = 1)	6.66667
1119 ( : total = 1)	6.66667
0	0

**Figure 103: Summary of Machine-Segment / Human-Segment Overlap for Christopher.GP.8**

```

1107, DiagramObject.h -----
* (44:) virtual void move (int dx,int dy);
-> (44:) void move (int dx,int dy);
1108, Door.cpp -----
+ (64:) DiagramObject::select();
- (64:) if (!selected) {
- (65:) selected=true;
- (66:) } else {
- (67:) deselect();
- (68:) }

1109, Door.cpp -----
* (68:) selected=false;
-> (68:) DiagramObject::deselect();

1110, Door.cpp -----
+ (25:) */
* (12:) Door::Door(Point u, Point l)
-> (12:) /*Door::Door(Point u, Point l)
1111, Door.cpp -----
* (48:) if (selected) {
-> (48:) if (DiagramObject::selected) {

1115, Door.cpp -----
+ (48:) if (selected) {
- (48:) if (DiagramObject::selected) {
1116, DiagramObject.h -----
* (67:) private:
-> (67:) protected:

1117, DiagramObject.h -----
+ (67:) private:
- (67:) protected:
1118, DiagramObject.h -----
* (44:) void move (int dx,int dy);
-> (44:) virtual void move (int dx,int dy);
1119, Door.h -----
* (24:) /*          private:
-> (24:) private:
* (27:) Point lower;*/
-> (27:) Point lower;
1120, Door.cpp -----
+ (64:) if (!selected) {
+ (66:) } else {
+ (72:) selected=false;
+ (73:) }
- (25:) */
* (12:) /*Door::Door(Point u, Point l)
-> (12:) Door::Door(Point u, Point l)
* (65:) DiagramObject::select();
-> (65:) selected=true;
* (69:) DiagramObject::deselect();
-> (67:) deselect();

```

Figure 104: Excerpt from Christopher.GP.8. Changes belonging to the same machine-generated Segment are marked in the same colour.

Even for a human researcher it is at first difficult to establish the precise connection between these different lines and their modifications because of their apparent non-relatedness, and it is only through a reasoning process that the intent behind the modifications of these lines can be determined. Since a reasoning approach is not feasible for machine detection, such logically related Changes will probably continue to remain difficult to detect for machine approaches. In order to uncover such problems, the best strategy may be to utilise Machine-Segmenting as a way of uncovering the structure underlying an assignment and then going back to look at any parts of the Project History in which machine-generated Segments are very fragmented. Such periods of the programming process may indicate a break-down of problem-solving approach and may hence hide problems such as the one described.

### 7.11.3 Wrongly identified (false positives)

False negatives interfere with the identification and grouping of ‘interesting’ Changes because Changes that are assigned to small Segments will be at the bottom of the ranked list of Machine Segments and will not be deemed ‘interesting’. Whereas false negatives are cases in which related Changes are not grouped, false positives involve the grouping of unrelated Changes. If such groups are large enough, then they will achieve a high rank, and other groups which contain related Changes (potential true positives) may be pushed below the margin of significant size. In order to ensure that Segments containing related Changes are selected it is hence important to minimize the number of unrelated Changes that form false positive Segments.

Figure 105 shows a false positive from Christopher A1. The Machine Segment involves a period of syntax debugging. Two different lines (the glVertex2i line and the for loop line) are debugged. In the Changes from (104-106) they are debugged together, but then they are debugged separately at (107-108) and (109-110, 112, 122, 131-133). The debugging of the syntax error in the glVertex2i line could be seen as separate from the debugging of the syntax error for the loop line, and individually neither debugging action would have formed a Segment large enough to be included in the top 25% of ranked items. However, because of the initial co-change they were included in the same Segment and hence this Segment could be seen as a false positive.

```

103, Circlecon.cpp -----
+ (17:) r=_r
- (17:) xh=_xh;
- (18:) yh=_yh;
104, Circlecon.cpp -----
+ (24:) glBegin(GL_TRIANGLE_FAN)
+ (25:) glVertex2f(x1, y1);
+ (26:) for angle# = 0 to 360 step 5
+ (27:) glVertex2f(x1 + sind(angle#)*radius#, y1 + cosd(angle#)*radius#)
+ (28:) next

```

```

+ (29:) glEnd()
* (17:) r=_r
-> (17:) r=_r;

105, Circlelcon.cpp -----
- (28:) next
* (27:) glVertex2f(x1 + sind(angle#) * radius#, y1 + cosd(angle#) * radius#)
-> (27:) glVertex2f(x1 + sind(angle#) * radius#, y1 + cosd(angle#) * radius#);
* (26:) for angle# = 0 to 360 step 5
-> (26:) for (int angle = 0; i < 360 i++5) {

106, Circlelcon.cpp -----
* (26:) for (int angle = 0; i < 360 i++5) {
-> (26:) for (int angle = 0; angle < 360 i++5) {
* (27:) glVertex2f(x1 + sind(angle#) * radius#, y1 + cosd(angle#) * radius#);
-> (27:) glVertex2f(x1 + sind(angle) * radius#, y1 + cosd(angle#) * radius#);

107, Circlelcon.cpp -----
* (26:) for (int angle = 0; angle < 360 i++5) {
-> (26:) for (int angle = 0; angle < 360; angle++5) {

108, Circlelcon.cpp -----
* (26:) for (int angle = 0; angle < 360; angle++5) {
-> (26:) for (int angle = 0; angle < 360; angle+=5) {

109, Circlelcon.cpp -----
* (27:) glVertex2f(x1 + sind(angle) * radius#, y1 + cosd(angle#) * radius#);
-> (27:) glVertex2f(x1 + sind(angle) * r, y1 + cosd(angle) * r);

110, Circlelcon.cpp -----
* (25:) glVertex2f(x1, y1);
-> (25:) glVertex2i(x1, y1);

112, Circlelcon.cpp -----
* (27:) glVertex2f(x1 + sind(angle) * r, y1 + cosd(angle) * r);
-> (27:) glVertex2i(x1 + sind(angle) * r, y1 + cosd(angle) * r);

122, Circlelcon.cpp -----
* (27:) glVertex2i(x1 + sind(angle) * r, y1 + cosd(angle) * r);
-> (27:) glVertex2i(x + sind(angle) * r, y + cosd(angle) * r);

131, Circlelcon.cpp -----
* (31:) glVertex2i(x + sind(angle) * r, y + cosd(angle) * r);
-> (31:) glVertex2i(x + sin(angle) * r, y + cos(angle) * r);

132, Circlelcon.cpp -----
* (31:) glVertex2i(x + sin(angle) * r, y + cos(angle) * r);
-> (31:) glVertex2i(x + int(sin(angle)) * r, y + cos(angle) * r);

133, Circlelcon.cpp -----
* (31:) glVertex2i(x + int(sin(angle)) * r, y + cos(angle) * r);
-> (31:) glVertex2i(x + int(sin(angle)) * r, y + int(cos(angle)) * r);

```

**Figure 105: A false positive Machine Segment's Changes from Christopher A1.**

It was difficult to find a false positive to present in this section, and the one presented was in fact fairly small and low on the list of ranked Segments (though it was still in the top 25%). False positives did not appear to occur often. This suggests that efforts should mainly concentrate on reducing false negatives and to ensure that true positives do not contain too many non-related Changes.

Analysis of true positives, false positives and false negatives indicates that the LH-Graph algorithm's main limitation comes with dealing with problems that are marked by very diffuse programming actions that involve rapid adding and removing of solution attempts. This limitation seems difficult to overcome. To achieve any segmenting the LH-Graph algorithm must determine Segment boundaries at some point. Through the use of co-change that boundary is presumed to lie at the point where a new set of lines is modified. If this is not the case, then there is no apparent strategy for connecting Changes. Indeed, the problems that are difficult for the LH-Graph algorithm to detect are likely problems that will be difficult to detect manually as well since tell-tale signs of textual or line proximity are not present. One potential avenue to explore in the future would be the use of dynamic analysis to trace whether an added line might be executed in close program-flow proximity to a recently modified line; such an approach may be able to detect at least some of the program-flow issues that appear to underlie many false negatives.

However, in the end the machine identification and Segment grouping presented in this chapter should not be seen as stand-alone tools for the precise identification of all 'interesting' related Changes in a Project History. It is likely that for a certain number of problems not all related Changes can be grouped because their relationship is strictly logical rather than structural. Uncovering their relatedness then requires a reasoning process to determine intent behind individual modifications which is a task beyond the capabilities of any algorithm. Thus, these algorithms should be seen as tools which allow a researcher to quickly develop an understanding of the structure underlying a Project History. The researcher can then identify points of interest from which the programming process can be further broken down until a sufficient understanding of the student's actions as captured in the Project History is reached.

## **7.12 Limitations of the evaluation method**

To identify whether a given machine approach accurately identifies problem Segments, the Segments generated were compared to Segments as identified by a human researcher. These human-identified Segments were assumed to provide an accurate mapping of problems and tasks occurring in periods of the Project History to related sets of Changes.

The evaluation approach has two inherent limitations which are outlined next.

### **7.12.1 Subjectivity and errors in the manual segmenting of Project Histories**

The manual segmenting process is in itself partly subjective and is prone to errors. A different researcher may have produced a different set of Segments because they broke down tasks in a

different way or because they missed a relationship between a set of Changes that they would normally have classified as a Segment.

Care was taken to address human error in the manual analysis and to prevent problem Segments from being overlooked as far as possible. The examination of the assignments was very thorough, and involved over ten thousand hours of analysis. The data relating to the identified Segments as well as notes relating to Changes in the appendix allow for verification of the researcher's segmenting process. However, oversights and errors in human analysis were uncovered during the evaluation of machine-generated Segments. These errors were not corrected because doing so would have risked further undermining the results by editing the data to fit the method of analysis. Errors in analysis were only likely to be found in problem Segments identified by the algorithm but not human analysis or vice versa, whereas problem Segments overlooked by both would have remained hidden, and because of the bias in favour of making human-identified problem Segments fit the data in order to produce better-fitting results. However, these errors were not numerous and it is estimated that they comprised fewer than 2% of the entire corpus.

To address the issue of subjectivity of analysis as well as the possibility of problem Segments being missed by a single researcher, it would have been beneficial to have multiple researchers analyse assignments to determine problem Segments, and to then either compare the machine approach to each of the researcher's problem Segments or to reach a set of consensus problem Segments to use. However, given the time-consuming nature of manual analysis and the relative inaccessibility of the SCORE toolset due to its developmental nature during the research project and the limited resources of the primary researcher, it was not possible to involve more researchers.

It may be beneficial to involve students in future evaluation of segmenting algorithms. Since students produced the Project History, they are in the best position to judge the fit of a particular segmentation. For example, volunteer students could be asked to produce a manual segmentation of part of the Project History for their assignment project, and this segmentation could then be used to evaluate machine-generation methods. Due to time constraints and the fact that the SCORE Analyser was not ready for general use such evaluation could not be carried out in the scope of this project.

### **7.12.2 Non-generality of analysed Project Histories**

Second, the pool of assignments comprises only two different assignments (hence two assignment specifications / assignment designs) and five different students, so the data are not necessarily representative of assignment specifications in general or students in general.

Further evaluation of SCORE in different problem domains and with more students would indeed be desirable. Unfortunately, two factors prevented such evaluation from occurring within the scope of this research project. The first is time constraints. Manual examination is very time-intensive. The second factor is the inaccessibility of the SCORE tool-set to other researchers, due to the rapidly evolving nature of the code and the general lack of documentation. Such evaluation may determine that the current approaches with the settings used do not work well for a different problem domain or research question. However, they may also help inform whether it is possible to tailor the approaches to particular questions or domains through the modification of mechanism settings. It would be desirable to have researchers in different fields of Computer Science Education utilise the SCORE tool-set to provide additional data to test against in the future.

## 7.13 Proposed Extensions

Several extensions to the current algorithm for Segment generation are planned.

### 7.13.1 Reconsidering Small Segments

During initial segmenting, restrictions are placed on search distance, search depth and the line modification types that will be visited from a given Change. These restrictions are designed to prevent different Segments of unrelated Changes from merging, since otherwise most of an assignment's Changes would be part of a small set of very large Segments.

However, some of these Changes that would have been added to Segments will not form part of any Segment, or be part of very small Segments. An additional step of the LH-Graph algorithm could attempt to discover parent Segments for these isolated Changes by relaxing distance, depth and modification restrictions.

### 7.13.2 Segment Line History Overlap

As mentioned earlier, restrictions on the LH-Graph algorithm are designed to ensure that non-related Changes are not connected to Segments. It only takes two modifications to co-occur in one Change to link the Line Histories to which they belong, and this can link unrelated Changes, which in turn makes the restrictions necessary.

An additional post-generation step could compare the modified Line Histories of all Segments to detect whether any Segments share many overlapping modified Line Histories. Such Segments would then be merged even if the distance between them is larger than would be allowed by the LH-Graph's distance parameter. This approach would be useful for merging related Segments that are a large distance apart without incurring the penalty of generating overly large Segments due to an overlap of one or two Line Histories with another set of Changes.



### 7.13.3 Dynamic analysis

Dynamic analysis of source code could also be utilised to assist Change identification and segmenting. The Line Proximity algorithm can detect lines modified in close proximity to other lines in terms of the textual distance (number of new line characters) between those lines. However, it does not work across file boundaries, and certain program flow problems involve line modifications that may be at different parts of the code but that are in close *execution proximity*, that is to say the instructions associated with these lines of source code will be executed in close proximity in the program's flow. Changes where modifications are made in close *execution proximity* could be added to the Segment.

### 7.13.4 Student Annotation / Feedback

A planned extension to the SCORE Eclipse plug-in will prompt students to regularly provide feedback on the problem they are working on. This input can then be used to annotate the programming process data for detailed analysis. In addition, such data may also be used to improve segmenting algorithms. For example, students could be asked to provide feedback when they move on to a new problem, or times when students commit comments could be used to identify transition points.

## 7.14 Line History Generation and Machine-Segmenting Conclusion

This chapter presented the two algorithm-related software-engineering components of this thesis, the Line History generation algorithm and four different Change-Identification and Machine-Segmenting algorithms. Evaluation of the Line History generation algorithm showed that the algorithm produces good results. For the most difficult to detect modification type (Mutations) precision was  $\geq 0.95$  for all examined Project Histories and recall was  $> 0.9$ . For other modification types, the precision/recall values would be close to 1.0. Also in comparison to a previous approach of Line History generation (Canfora et al., 2007) the approach proposed in this thesis also detects so-called 'ghost' lines, lines that are deleted and then reintroduced in a following document. Also, based on the description of the evaluation method used by Canfora et al (2007) the evaluation involved calculating precision and recall for all mappings, whereas evaluation of the Line History algorithm presented in this thesis involved calculating recall and precision for the Mutant modification type only, which is the most difficult type to detect. Maintained modifications are both the most numerous modifications and the most accurately detected with precision and recall values which would be very close to 100%. Inclusion of Maintained modifications would hence likely have resulted in the algorithm substantially outperforming that presented by Canfora et al (2007).

Initial evaluation of Change-Identification and Machine-Segmenting algorithms presented in Section 7.7 showed that the LH-Graph Machine-Segmenting algorithm performed consistently better than a

random algorithm at segmenting Project Histories and produced better results than the other algorithms, with both Machine-Segmenting algorithms outperforming both Change-Identification algorithms. This result was confirmed through evaluation using an independent data set not used during the implementation of the algorithms in Section 7.10. The LH-Graph algorithm achieved a significantly better than random result at the  $p \leq 0.05$  and  $p \leq 0.01$  levels for all Project Histories. The LH and LH-Graph algorithms also produced good *spread* and *fit* values, indicating that the Machine-generated Segments described human-identified Segments with good accuracy.

Some extension algorithms (to the LH-Graph algorithm) produced significantly better results than the core algorithm during the initial evaluation on the set of Project Histories used during implementation (see Section 7.9) but did not produce improvement when tested against the independent data set (see Section 7.10). While some algorithms produced better *spread\*fit* values, they also produced worse Actual/Expected ratios. Differences were in all but one case minor. The CompileFilter algorithm performed very well across all students and both assignments on both the original and the independent data set. On this basis, only the CompileFilter algorithm appears to be effective. The difference in results between the first and second phase of evaluation may have to do with over-fitting of algorithms to the initial data, but it may also in part have to do with the shallower segmenting approach used to produce the second data set. Future work should aim at producing Machine-Segmenting algorithms for a shallower segmenting approach.

These results validate the LH-Graph method as an effective machine method for the generation of Segments. The application of Machine-Segmenting can serve to significantly decrease the time required to Segment a Project History, making the application of the Segment-Coding method described in Chapter 6 less time-intensive. The combination of the Machine-Segment generation method with the Segment-Coding method thus fulfils the research goal of implementing a viable method of source-code level Project History analysis.

The evaluation of Machine-Segmentation and Change-Identification algorithms involved two limitations. The first related to the involvement of only a single researcher, the second to the use of a relatively non-diverse set of Project Histories comprising only ten students and two different assignments. These limitations gives rise to potential validity concerns relating to the evaluation (Section 7.11), which due to the time constraints of this research project were unavoidable. However this sample was extensive in so far as it incorporated analysis of 13739 Changes, 9784 in the original set of Project Histories and 3955 in the independent data set. Also, the analytic approach was rigorous in terms of the conservative approach to calculating statistical significance by using simulation (see 7.4.2.4), due to the non-random distribution of 'interesting' Changes. A more naïve

approach using the hypergeometric distribution would have yielded extremely low p-values for all cases (see Section 7.4.2.3 for more details). To address these limitations in the future, production of segmenting data to test against could involve more researchers and more Project Histories produced to match different assignment specifications. Including data from non-Computer Graphics domains could be used to increase the generalizability of the analysis and results.

In addition to producing Machine-Segmenting algorithms, a framework for machine-generation and evaluation of machine-generated Segments was implemented. This will allow researchers to develop and evaluate new algorithms or different settings or extensions for current algorithms. Thus, the Machine-Segmenting approach may be utilised in different domains for which the current settings do not perform as well. The ‘Proposed Extensions’ section (Section 7.13) proposes several extensions which may improve the performance of the LH-Graph Machine-Segmenting algorithms whose implementation fell outside the scope of this research project due to time constraints.

# 8 Conclusion

---

## 8.1 Foreword

This thesis had two interrelated aims. The first was to develop a method of analysis of student source code as is stored in a version control system to answer the research questions regarding student Computer Graphics Education programming problems. The second was to apply this analysis method to Computer Graphics students' source code to determine what kinds of problems novice CG students face.

Given these research goals, three research questions were posed at the outset:

**RQ1) How can the analysis of students' coding as captured in a version control system be used to analyse student programming problem-solving?**

**RQ2) What kinds of problems do students learning Computer Graphics Programming experience?**

**RQ3) Is 'spatial programming' a difficult area/topic of Computer Graphics programming for students?**

The body of this thesis presented the gathering and evaluation of data to answer these research questions. This chapter will present the findings relating to these questions.

The first aim was addressed through the development of the Change-Coding and Segment-Coding methods. These new methods go beyond traditional methods such as the analysis of student perceptions, the analysis of data from strictly controlled experiments with limited generalizability or the analysis of errors final submissions of student work containing none of the problem-solving process involved in the solution of assignment tasks. The methods are intended to analyse the student programming process through analysis of individual student programming actions during their development of assessment tasks at a source code level. This is a novel approach providing an unprecedented level of detail in analysis, with existing methods of analysing the student programming process focusing on quality metrics or output data (compilation error messages). This produces a descriptive understanding of student errors rather than a deep understanding of the student problem-solving process. This research project evaluated the feasibility of such methods of analysis of Project Histories to determine whether the method is suitable for future Computer

Science Education research. The development of these methods also involved the development of software-engineering methods to facilitate the Project History analysis process. The development of these methods together addresses **RQ1**. The related contribution of providing a new method of source-code level Project History analysis for Computer Science Education is described in Chapters 5 and 6.

The second aim was addressed through the application of the developed methods to the analysis of CG student assignments. An educator's intuition suggested that students were struggling with Computer Graphics topics, some of which like spatial programming are unique to Computer Graphics Education, but there was little research in Computer Graphics Education which examines these student problems in any detail. The second aim of this thesis was to begin to fill this gap, analysing student programming to provide educators with insights on student Computer Graphics Education problems. This will enable the development of learning materials which can better scaffold and support student learning needs, refine the student learning experience, and hence optimise learning outcomes. In addition to identifying student issues (thereby providing answers to **RQ2**), analysis was also intended to determine whether such programming presented a special challenge to students (thereby answering **RQ3**). The related contribution of providing a better understanding of student Computer Graphics programming is outlined in Section 8.3.1.

## 8.2 Findings and Outcomes

### 8.2.1 A Grounded-Theory based method of source-code level Project History analysis

The first research question **RQ1** relates to the development of the analysis method: "How can source-code level Project History analysis provide detailed insight into student programming problem-solving?"

Based on the principles of the Grounded Theory methodology (Glaser & Strauss, 1967) (Glaser & Strauss, 1967) analysis of individual Changes within student programming projects led to a categorisation of student programming activities that included 'General Programming', 'Event-Driven', 'Animation', 'Spatial', 'View', 'Lighting', 'OpenGL' and 'Pipeline'. The developed data analysis method is called Change-Coding as elaborated in Chapter 5. Change-Coding produced a high-level view of the programming process. However, since it coded individual Changes the method did not capture the context in which Changes occurred as part of a larger solution attempt to a problem. Thus the Change Coding method did not provide a clear answer to **RQ1** relating to the development of an insightful source code analysis. As a result, it also could not provide a comprehensive answer

to **RQ2** and **RQ3** regarding student problems with CG programming and the special role of spatial programming.

The Segment-Coding method (presented in Chapter 6) addresses this limitation by coding Segments (sets of Changes pertaining to the same task) rather than individual Changes, thereby maintaining the context in which Changes occur. This allows for the identification of commonality in students' problem-solving processes, enabling the researcher to develop rich theory. As is demonstrated by the results presented in Section 6.5 and the analysis presented in Section 6.6, the Segment-Coding method has been shown to be a viable and effective way of analysing student programming.

The Segment-Coding method allows the researcher to develop a rich understanding of the underlying problems and breakdowns in student problem-solving through the identification of commonality in students' problem-solving processes, enabling the researcher to develop the properties of classification categories, thereby developing rich theory.

This means the Segment-Coding method provided a solid basis for answering **RQ1**: "How can the analysis of students' coding as captured in a version control system be used to analyse student programming problem-solving?". However, the Segment-Coding method did have its limitations, chief of which was the time-intensive nature of the application of the approach (see Section 6.9.2). The next finding discusses software-engineering-based approaches developed as part of this research project to address this limitation.

### 8.2.2 Software-Engineering-based approaches to Project History analysis

A 'Machine-Segmenting' software engineering method was developed to automatically identify related Changes within a Project History. This relied on the generation of Line Histories which are data structures containing all modifications to a particular line of code in a Project History. The Line History Generation algorithm that was developed produced a recall of >0.95 and precision >0.9 for all Project Histories for the detection of the most difficult to detect 'Mutation' modification type, and better results for other modification types.

Evaluation of four Machine-Segmenting algorithms (presented in Section 7.7 and 7.10) found that one of the developed algorithms, the LH-Graph algorithm, achieved significance at the  $p \leq 0.01$  for all Project Histories where possible<sup>46</sup> and achieved >95% of the maximum possible ratio of actual to

---

<sup>46</sup> For one project history, even a perfect segmentation could achieve significance only at the  $p \leq 0.05$  level since the project history contained too many potential correct positives compared to false positives. For that project history significance was achieved at the  $p \leq 0.05$  level.

expected precision on average (see Section 7.4 for details on the evaluation metrics). This shows the algorithm to be very effective at Machine-Segmenting Project Histories.

Together, Line Histories and Machine-Segmenting make the Segment-Coding method more time-efficient to apply. Segments (sets of Changes presumed to be related) can be identified based on co-change of lines of source code in a content-agnostic manner and without human intervention. Therefore, when taken together with the Segment-Coding method described in the last section, Machine-Segmenting and Segment-Coding provide an answer to **RQ3**: “How can source-code level Project History analysis provide detailed insight into student programming problem-solving”. To this extent this research project was successful in developing an effective method of source-code level Project History analysis.

### 8.2.3 Issues in student Computer Graphics programming

Analysis of results from the application of the Change-Coding method (presented in Chapter 5) showed that the majority of Changes in both of their assignments were related to work on ‘General Programming’ (47% in Assignment 1, 27% in Assignment 3), ‘Spatial’ (21% in Assignment 1, 27% in Assignment 3) and ‘Event-Driven’ (12% in Assignment 1, 7% in Assignment 3) tasks. The analysis also showed that ‘Lighting’ and ‘Event-driven’ programming produced fewer errors than expected (10% and 13% of ‘Lighting’ and ‘Event-Driven’ actions respectively resulted in errors, falling more than one standard deviation below the mean). Changes in the ‘View’ and ‘Pipeline’ categories produced more errors (52% and 48% of ‘View’ and ‘Pipeline’ actions respectively resulted in errors, falling more than one standard deviation above the mean). This may indicate that ‘View’ and ‘Pipeline’ problems are more difficult to resolve than other kinds of problems, or that students utilise a different problem-solving approach for these problems. While the Change-Coding method provided a high-level overview of computer graphics students’ programming practices, it was unable to provide detailed insight into student problem-solving (see Section 5.5). Thus, Change-Coding analysis only partially addressed the first and second research questions.

The Segment-Coding method that grouped and analysed Changes relating to the same student programming problem (presented in Chapter 6) led to the identification of a set of nine issues relating to student Computer Graphics programming, as outlined in Table 35 (see Section 6.6.6 for more detail). The ‘Conceptual’ issue was further broken down into a set of five concepts and misconceptions (as shown in section 6.6.6.1 and summarised later in this section).

**Table 35: Issues related to student problem-solving in Computer Graphics**

Conceptual	Issues related to concepts occurred when students did not grasp one of the concepts discussed in the previous section.
Difficult-to-Spatially Visualize	Issues related to student inability to visualize effects of spatial actions or mathematical programming, and hence to poor visual feedback of such programming actions. As Butler and Morgan's work (2007) has shown, topics involving low levels of visual feedback are considered more challenging by students than concepts with high levels of feedback.
Cognitive Difficulty of Spatial Programming	Issues related to a lack of student cognitive ability (spatial ability) hindering their development of spatial content, especially three-dimensional compound transformations.
OpenGL Pipeline Black-box	The 'OpenGL Pipeline Black-Box' issue relates to problems based on students' misunderstanding of the state of the OpenGL state machine, or an unawareness of the way in which an OpenGL action is processed in the graphics pipeline.
Program-Flow Understanding	Issues related to 'Program-Flow Understanding' occur when students do not understand the order in which their source code is executed when the program is run and what state variables will be in.
Programming Language Syntax and Semantic Concepts	This issue relates to student problems with the use of the programming language's syntax and semantics, such as the language's implementation of object-orientation.
Interplay of different problems	The 'Interplay of different problems' issue arises in any context in which different problems intersect, making individual problems harder to identify and solve.



Inconsistencies	Issues caused by inconsistencies between different APIs / libraries used by students during the coding of their assignments. An example that OpenGL uses degrees to measure angles whereas the standard C++ libraries utilise radians.
Tweaking	Issues related to students modifying coordinates by very small values in order to achieve pixel-perfect placement.

Issues relating to the development of specific concepts or incorrect misconceptions are captured as part of the ‘Conceptual’ issue. Other ‘issue’ categories are related to more generic student abilities, and to features underlying the design of the assessment task and the materials (such as the C++ programming language and the OpenGL API) utilised.

Five individual concepts and misconceptions, forming sub-categories of the ‘Conceptual’ issue, are presented in Table 36. Section 6.6.6.1 provides a more detailed description of these five concepts. The most important and hard-to-develop concepts were the two ‘Transformation’ sub-concepts which all students struggled with and two students did not properly develop.

**Table 36: Student Concepts and Misconceptions in Computer Graphics**

Transformation Concepts	Transformation concepts relate to the use of three-dimensional transformations. The ‘Hierarchical Transformations’ concept which involves understanding how to composite transformations and apply push/pop calls to create hierarchical models. The ‘Compositing of Transformations’ concept relates understanding how compositing of transformations changes the local coordinate system of a limb.
Mathematical Concepts	The category of mathematical concepts involves all incorrect applications of mathematical formulas.
View Concepts	View misconceptions relate to a lack of understanding regarding the way in which the conceptual View and Model components of the ModelView matrix interact.
Draw-Not-Store Misconception	The draw-not-store misconception involves the student immediately drawing out to screen the effect of actions without storing the effect of the action.

Time-based Behaviour Concepts	The time-based behaviour concept refers to the implementation of an algorithm for animations which can draw frames to screen in a way in which frames will not be overdrawn before they are visible.
-------------------------------	--

Potential solutions for each issue based on the nature of the problem-solving difficulty were also proposed as provided in Table 37. The application and evaluation of these solutions falls outside the scope of this research project., Future research could utilise the analysis methods developed in this thesis to evaluate these solution approaches for efficacy by recording and analysing student work when the suggested strategies were applied, with a special focus on those Segments identified to involve a candidate solution.

**Table 37: Potential solution approaches to addressing student problem-solving issues**

Conceptual	<ul style="list-style-type: none"> <li>• Provide students with learning materials for key concepts.</li> <li>• Structure assignments that relate the development of core tasks to individual concepts in a way that allows students to both apply the concept and test the correctness of their understanding.</li> </ul>
Difficult-to-Spatially Visualize	<ul style="list-style-type: none"> <li>• Teach students visio-spatial debugging techniques and/or provide them with visio-spatial debugging tools. For example, students can be taught to draw out the effect of every transformation in a series of transformations, showing the effect on the local coordinate system. In a Viewing context, students can be provided with methods for drawing the Viewing volume. This provides a higher level of feedback for students for such problems, which should support their learning since low levels of feedback make students perceive problems as more challenging (Butler &amp; Morgan, 2007).</li> </ul>

Cognitive Difficulty of Spatial Programming	<ul style="list-style-type: none"> <li>• Provide students with spatial ability training tailored towards Computer Graphics Education, much like has been done successfully in the engineering-Computer Graphics context (Alias et al., 2002; Blade &amp; Watson, 1955; Hsi et al., 1997; Leopold et al., 2001; Martín-Dorta et al., 2008).</li> </ul>
OpenGL Pipeline Black-box	<ul style="list-style-type: none"> <li>• Provide students with access to the OpenGL state machine by teaching them how to access states using the OpenGL API or by providing them with a library to do so.</li> <li>• Provide students with the ability to observe the execution of each step of the OpenGL pipeline and its state by using a breakpoint-style debugger.</li> </ul>
Program-Flow Understanding	<ul style="list-style-type: none"> <li>• Teach students better program-flow debugging techniques using breakpoints and stack traces.</li> </ul>
Programming Language Syntax and Semantic Concepts	<ul style="list-style-type: none"> <li>• Ensure students possess sufficient familiarity with the programming language utilised.</li> <li>• Avoid languages with confounding syntax.</li> <li>• Provide students with scaffolding and advice on any programming language syntax / semantics identified as problematic.</li> </ul>
Interplay of different problems	<ul style="list-style-type: none"> <li>• Structure assignments in a way that avoids the confluence of multiple problems in one task.</li> <li>• Teach students how to identify different types of problems, or to exclude certain types of problems, to allow them to better pinpoint different problems.</li> </ul>

Inconsistencies	<ul style="list-style-type: none"> <li>• Choose materials that have as few inconsistencies as possible, or work around inconsistencies by providing students with alternative libraries (e.g., replacing C++ math libraries with ones that use degrees).</li> <li>• Inform students of any inconsistencies and train them to identify whether any errors may be caused by underlying inconsistencies.</li> </ul>
Tweaking	<ul style="list-style-type: none"> <li>• Teach students techniques or provide them with tools to make precise pixel measurements on screen.</li> </ul>

Taken together, the findings from the Change-Coding Analysis and the Segment-Coding analysis provide answers to **RQ1**: “What kinds of problems do students learning Computer Graphics Programming experience?”.

Findings from Segment-Coding analysis were supported by analysis of student perceptions of Computer Graphics programming (see Section 3.6.4), with students indicating that they found three-dimensional spatial programming difficult, as well as implementation of camera views. While student perceptions validated several of the Segment-Coding findings, they could not provide the same level of detailed insight into student problem-solving processes as the Segment-Coding analysis.

#### 8.2.4 A model of Student Problem-Solving during Computer Graphics Programming

Moving beyond the identification of individual student problems, the issues uncovered during analysis shed light on the nature of the problem-solving process. The identified issues were used as a basis for a model of student programming problem-solving, as shown in Figure 106.

The model describes student problem-solving as consisting of four phases (see

Table 38), the *Identify* phase in which the source of the problem is identified, the *Understand* phase in which the student recalls or develops a correct solution approach, the *Apply* phase in which that approach is applied, and the *Perfect* phase in which the student perfects the solution. Identified issues (see Section 8.2.1) map to the model's phases as is shown in the 'Issues relating to phase' column in

Table 38 below. The phases are described in more detail in Section 6.6.7. Each identified issue maps to one of the phases of the problem-solving process during which it can cause the student to encounter difficulties in completing the phase as described in the 'Difficulty associated with failure to complete phase' column in

Table 38. The ability or skill required by the student to move on to the next phase of the model is shown in the 'Required Student Ability to move to the next phase' column in

Table 38. The model is described in more detail in Section 6.6.7.

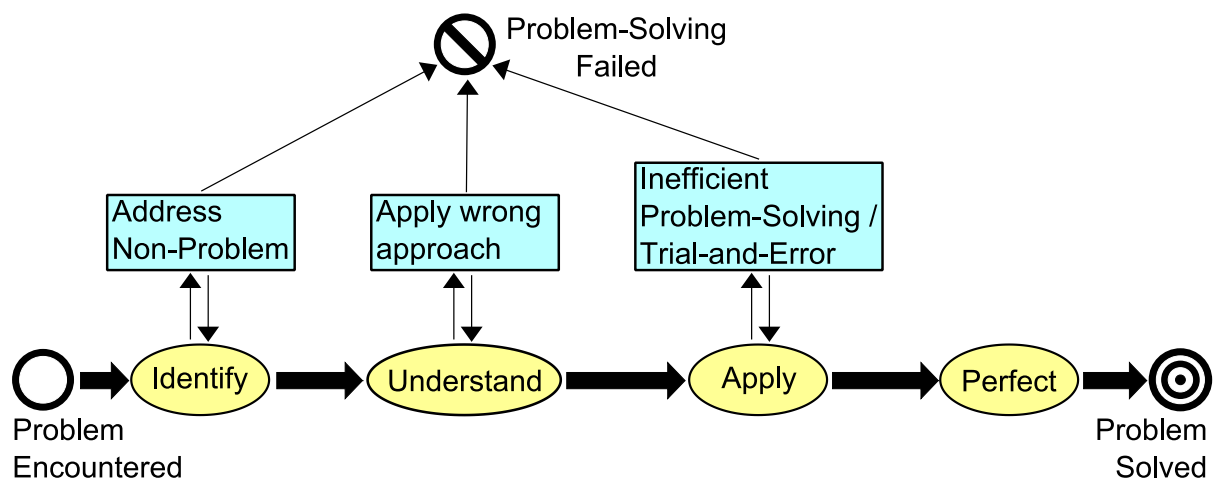


Figure 106: The four phases of the student Computer Graphics programming problem-solving process



**Table 38: Description of the model's phases of Computer Graphics student problem-solving**

Phase	Description	Required Student Ability to move to the next phase	Difficulty associated with failure to complete phase	Issues relating to Phase
Identify	Localise problem in source code	Problem-source identification	Misidentification of problem; student work does not address problem.	<ul style="list-style-type: none"> <li>• Difficult-to-Visualize</li> <li>• OpenGL Pipeline</li> <li>• Interplay of Problem Types</li> <li>• Inconsistencies</li> </ul>
Understand	Remember / Develop concept or solution approach to apply	Knowledge of concept / solution approach	Student uses incorrect solution approach; unable to solve problem.	<ul style="list-style-type: none"> <li>• Conceptual</li> <li>• Programming Language &amp; Syntax</li> </ul>
Apply	Apply the problem solving approach or concept to the problem	Skill in applying the concept / solution approach	Student cannot apply solution effectively; in worst case, must resort to trial-and-error.	<ul style="list-style-type: none"> <li>• Cognitive Difficulty of Spatial programming</li> <li>• Program-Flow Understanding</li> <li>• Algorithm Understanding</li> </ul>
Perfect	Improve solution in minor ways to make it more aesthetically pleasing	None, problem is already solved		<ul style="list-style-type: none"> <li>• Tweaking</li> </ul>

As the model was derived from Computer Graphics student problem-solving data, the extent to which it is transferable to other programming contexts is a matter for further research.

### 8.2.5 Spatial Programming is difficult

The third research question **RQ2** posed at the outset of this thesis is “Is ‘spatial programming’ a difficult area/topic of Computer Graphics programming for students”? It was hypothesised that spatial programming may present a significant challenge to students based on its connection to spatial ability.

Analysis of Segments of student programming presented in Section 6.5 showed that all students produced very large problem Segments relating to three-dimensional spatial programming which supports the veracity of the hypothesis that spatial programming is difficult. In-depth qualitative analysis of Segment contents (presented in 6.6.4) confirmed this, showing that students had significant problems with learning concepts related to spatial programming. The concepts students

had the most difficulty with were those concerned with the compositing of three-dimensional transformations. This aligns with findings from other fields that a lack of spatial ability has been implicated in poor performance in many fields which require visio-spatial thinking. Examples include physics education (Pallrand & Seeber, 1984), programming (Jones & Burnett, 2008), source-code navigation (Jones & Burnett, 2007), engineering education (Hsi et al., 1997) and chemistry education (Carter et al., 1987; Pribyl & Bodner, 1987).

A detailed quantitative examination of the animation task presented in Section 6.6.5 showed that students struggled with compound transformations. Students performed only a little better than chance (61% actual error rate versus 66% for random guessing) for non-compound transformations. Students performed worse than chance for compound transformations (94% error rate), almost never performing them correctly on the first attempt. This was despite students having previously worked on the assembly task, which required them to learn and apply spatial programming concepts involving the compositing of transformations. This demonstrates that spatial programming concepts are both hard to learn and to apply, even after students have a theoretical understanding of the concepts. The reason for this is a student spatial ability to visualize the effect of transformations. This finding corresponds with student perceptions of spatial visualization being difficult described by students in reflection questions. As one student described: “Transformations and viewing mainly because I was unable to visualise what the commands would achieve in my head” (see Section 3.6.4.1).

### **8.2.6 Students use different solution approaches for different three-dimensional spatial programming tasks**

The qualitative analysis of Segments presented in Section 6.7 suggests that students spent more time thinking about each Change during ‘View’ problems. However, assembly and animation tasks certainly were not easier to complete than view tasks. Students spent much more time in total and produced many more errors while working on assembly and animation tasks than on view tasks.

Based on observation and analysis of student programming actions (see Section 6.7 for further discussion or appendix Section 9.7.8.4 for a detailed analysis) the reason for this difference appears to be that students find it easier to use OpenGL to visualize their spatial programming actions when working on transformations on models than when working on transformations of the View. This is probably because it is difficult to view the view volume produced to implement a view, whereas it is relatively simple to view the outcome of a transformation applied to an object (in the case of Assignment 3 one of the avatar’s limbs). Since most students attempted to utilise OpenGL as a visualization tool in situations where it was straightforward to do so, assisting students in doing so

(for instance by providing students with methods for producing their own visualization of the viewing volume) may prove helpful to their learning. However given that the analysis was performed on a small set of Segments, further research should be conducted to explore this phenomenon in more detail.

### **8.2.7 Development of a source code data collection and analysis toolkit**

A final outcome of this project was the development of a source code collection and analysis toolset (the SCORE Toolkit described in Chapter 4) which was used to harvest student programming data and to enable the analysis of this data. The Toolkit consisted of the SCORE Plugin and the SCORE Analyser. The SCORE plugin captures and stores version data much like a version control system. The data is stored locally and submitted by the student along with the other files in the project.

The SCORE Analyser provides features aimed at facilitating source code analysis. It provides diff views which show changes from one version to the next as well as Line History views which provide an overview of all changes made by a student to a line of source code. It also provides a compiler and editor which allow the researcher to execute any version of the student's code, as well as modifying a copy of it for debugging purposes. It also provides access to the Machine-Segmenting functionality described in Chapter 7. Finally, it provides coding and note-taking facilities which support the application of the Grounded Theory coding and memoing process.

These source-code specific features are not present in other more general Qualitative Data Analysis tools, but were essential in enabling the application of the Grounded Theory-based Segment-Coding method. Features such as Line History views were also essential to gaining an understanding of the thousands of versions of source code which made up individual projects.

## **8.3 Significant Contributions**

### **8.3.1 A new method of source-code level Project History analysis for Computer Science Education research**

This thesis has made several significant contributions to the field, principally in two areas. The first area of contribution is a new, flexible method of source-code level Project History analysis that can be applied in any programming context and hence provides researchers with a new way of gaining insight into student programming. This contribution addresses **RQ1RQ3** on the development of an effective method for source-code level Project History analysis. The Segment-Coding data analysis method is based on GT methodology, and is applicable to any Computer Science domain. Its application is enabled by the SCORE toolkit, which provides data processing methods as well as

analysis aids. It also provides Machine-Segmenting methods which will reduce the researcher's workload and make the application of the method feasible in more research contexts.

The body of this thesis contains a brief introduction to the software and practical application of the analysis method. Both of these can be explored further through use of the SCORE Analyser included in the electronic appendix that accompanies this thesis. The package includes the Project History data which was analysed as part of this research project. A manual which describes the use of the SCORE Analyser is included in the appendix Section 9.5.3 and as part of the electronic appendix (Wittmann, 2012a). The electronic appendix also includes a video demonstration of the SCORE Analyser's main functionality (Wittmann, 2012b), which is also available online at [http://www.youtube.com/channel/UCOSYNGcycyPU\\_DhIEi1gyig/videos?view=1](http://www.youtube.com/channel/UCOSYNGcycyPU_DhIEi1gyig/videos?view=1).

### **8.3.2 A better understanding of student Computer Graphics programming and the role of spatial programming**

The second contribution is the detailed exploration of problems students face when engaging in Computer Graphics programming, an area that has seen little previous formal research. This contribution addresses **RQ1** regarding the nature of student problems in Computer Graphics programming and **RQ2** on whether spatial programming is difficult. Proposed solutions based on observed issues and misconceptions may help improve student learning and enable the crafting of more effective teaching materials. Analysis has also shown that students find spatial programming challenging, and identified spatial ability and problems with the visualization of spatial actions as topics worth addressing to improve student learning outcomes. It is hoped that this initial research can help provide a foothold for future Computer Graphics Education research to develop a more general theory of student computer graphics programming and spatial programming.

## **8.4 Future Directions**

There are several research and development possibilities as well as implications arising from the results of this research study. Firstly given the findings of this study a new topic, 'Visio-Spatial Programming and Debugging', is proposed in order to address issues related to students' spatial programming identified in Sections 8.2.1 and 8.2.5 (see Section 8.4.2.1 below for a discussion of how this topic could be implemented). Future research and development includes further analysis of student Computer Graphics programming, improvements to the SCORE Analyser and Plug-in as well as further development of Machine-Segmenting algorithms (as will be discussed in Section 8.4.2.2). Finally, efforts will be made to disseminate the SCORE Analyser Plug-in and Analyser as widely as possible so that other educators and students may benefit from its features, both in research

contexts similar to that underlying this research project as well as in other potential application areas (as will be discussed in Section 8.4.2.3).

#### **8.4.1 Proposal for a new topic of Computer Graphics Education: Visio-Spatial Programming and Debugging**

Analysis showed that students had significant problems with both learning and applying spatial concepts. For this reason, it is proposed that a new topic be added to the Computer Graphics Education syllabus: ‘Visio-Spatial Programming and Debugging’. Most Computer Graphics textbooks explain the mathematics behind transformations in detail. However, they pay scant heed to providing students with a spatial understanding of the underlying mathematics. As the analysis conducted in this thesis shows, this mathematical foundation does not generate a spatial understanding of transformations in students.

Material designed to convey a spatial understanding of such concepts would be presented and discussed alongside the mathematical material. Existing visualization aids (Andújar & Vázquez, 2006; Figueiredo et al., 2003; Görke, Hanisch, & Strasser, 2005) can serve as a basis for the development of visualization methods to better teach students visio-spatial concepts.

A tool to support student learning of visio-spatial programming concepts was developed as part of this research project, though it did not reach maturity. As a result was not formally evaluated. The SGLParser tool (see Appendix Section 9.9 for more details) was developed to address issues related to student challenges relating to Computer Graphics programming. Its main interface is shown in Figure 107.

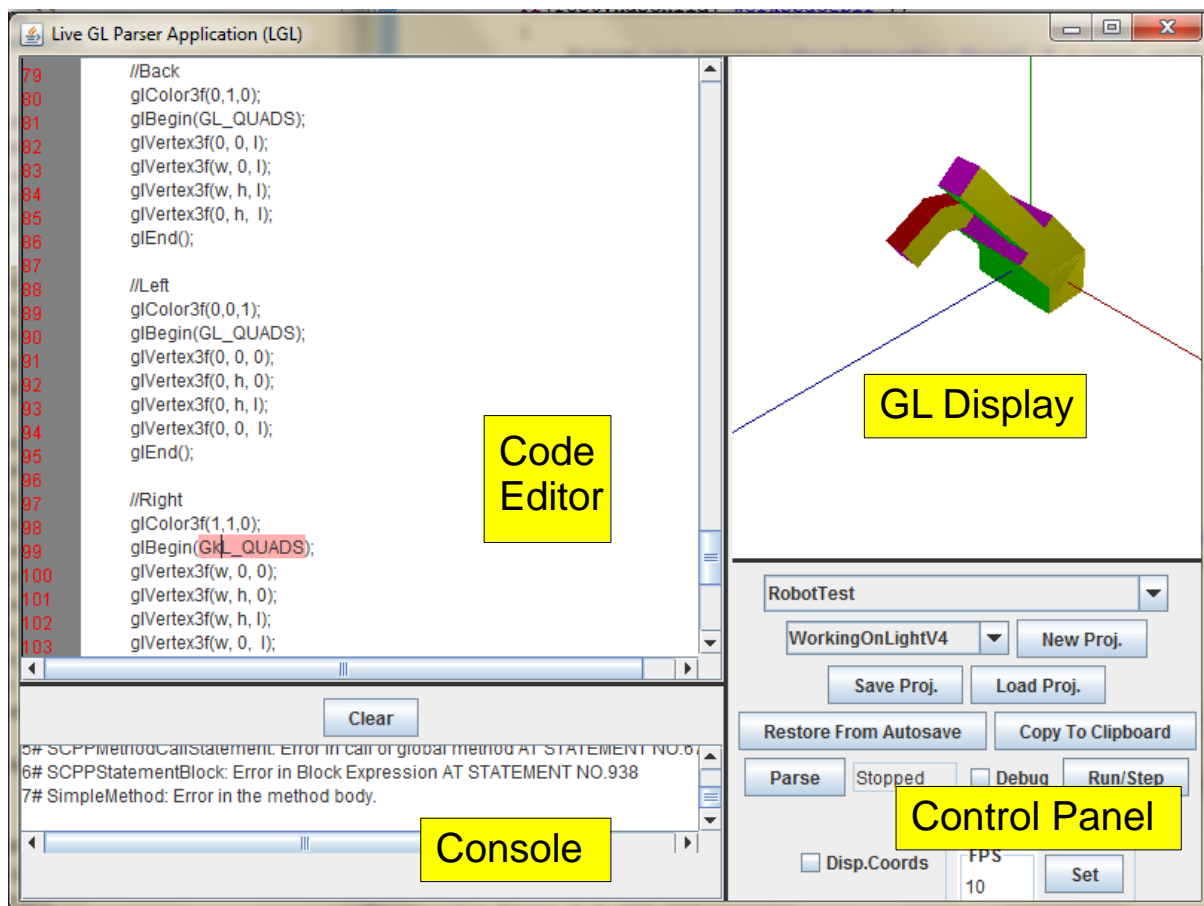
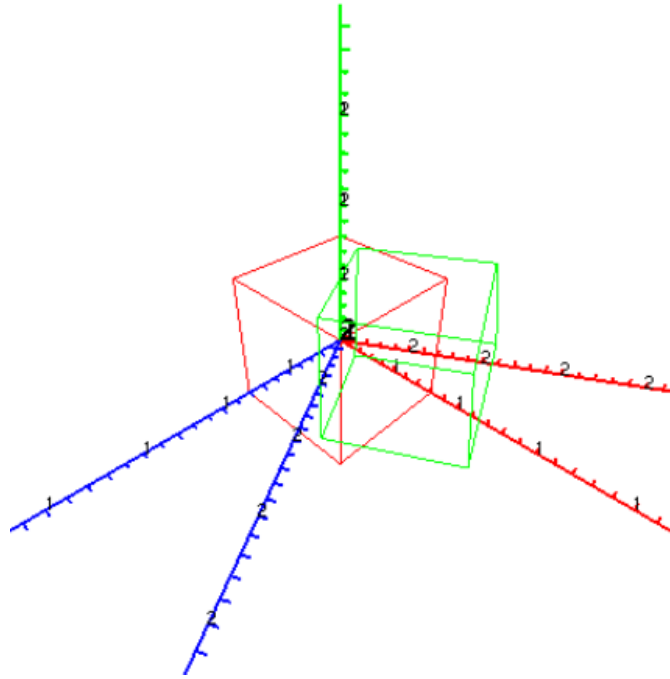


Figure 107: The SGLParser main interface

It has three main features:

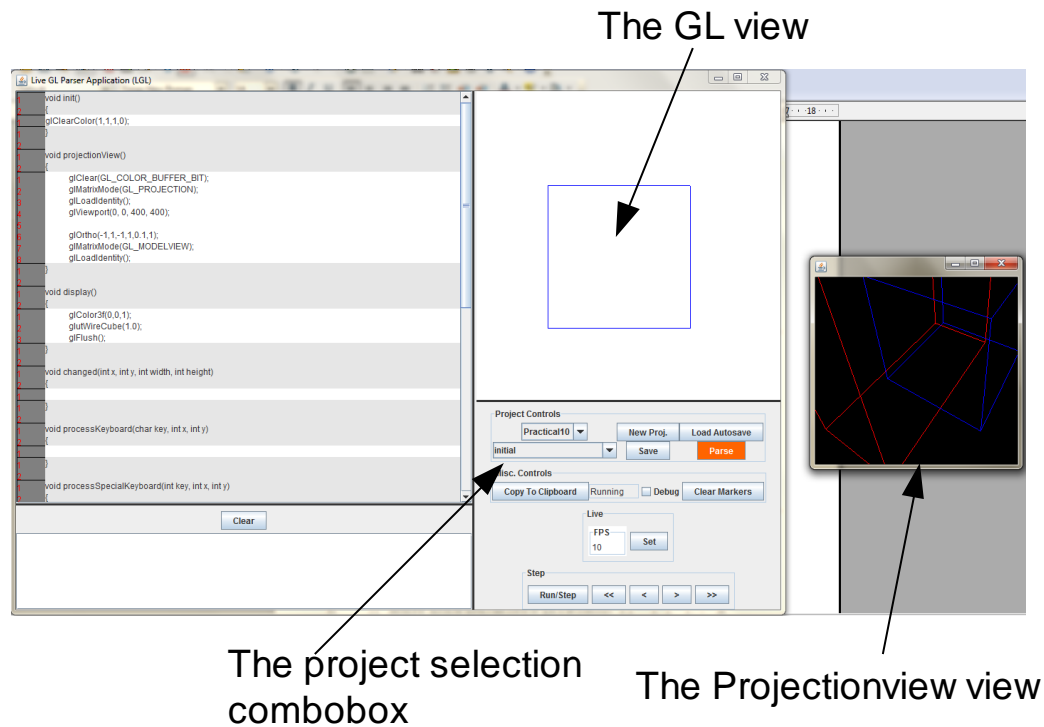
- Step-By-Step Execution:

The SGLParser allows the student to set breakpoints or to execute the program step-by-step to observe the effect of individual OpenGL commands. This could be useful in addressing the 'OpenGL Black-Box' and 'Event-driven program flow' issue.

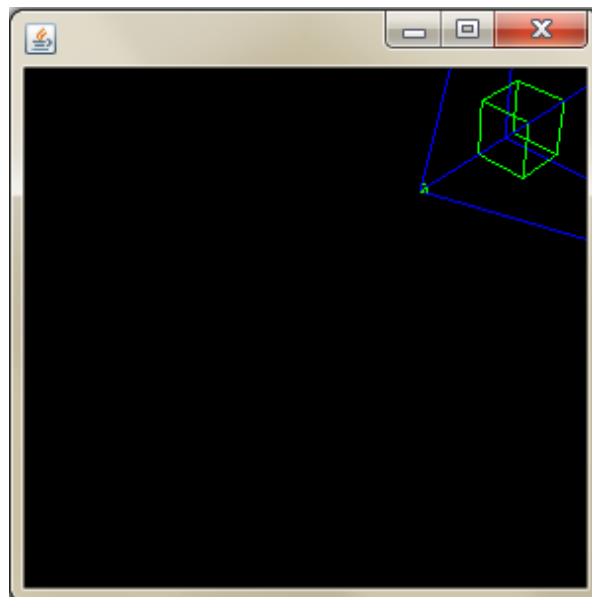


**Figure 108: SGLParser visualization of a transformation, showing the local coordinate systems associated with the individual transformations**

- **Transformation visualization:**  
The SGLParser can render the effect of transformations on the local coordinate system, including showing the effect of a compound transformation by showing the state of the local coordinate system after each step, as shown in Figure 108. This can help address issues related to 'Difficult-to-Visualize' issues to do with transformations.
- **Viewing visualization:**  
The SGLParser can render an external view of the viewing volume, while at the same time rendering the image produced by the view, as shown in Figure 109 and Figure 110. This addresses issues related to 'Difficult-to-Visualize' Views.



**Figure 109: The SGLParser with the projectionview window, showing the view volume**



**Figure 110: Example of a view volume shown in the projection view**

The tool was tested with students in three practical classes. The feedback was generally positive, but there was no opportunity to redesign and reuse the tool in the 2011 semester since the course was not offered in 2011. As a result no further development or formal evaluation was undertaken. Tools such as this could be provided to students as part of their training in visio-spatial debugging techniques, and to help them learn and develop concepts related to spatial programming.



Eventually, students should also be able to apply these strategies independent of the tool, so it would be best to eventually teach students the techniques utilised by the SGLParser to achieve these visualizations. While the SGLParser is not currently ready for unsupervised use, it may serve as a conceptual basis for the development of integrated tools or libraries providing functionality to address the issues identified in this research project.

When compared to the existing tools identified in the literature (Andújar & Vázquez, 2006; Figueiredo et al., 2003; Ullrich & Fellner, 2005), the SGLParser provides two additional features. The first is that it allows the student to type ‘real’ C++ OpenGL code (in fact the language used is SGL, based on a simplified C++ grammar developed by the author. This code is parsed as the student types, and if it compiles successfully the results are immediately shown on screen.

The second difference is the step-by-step execution facility provided by the SGLParser. It allows each command to be executed in turn, with the effect on objects being visible in the viewing window. For example, a student may observe the effect of a series of transformations by stepping through them one by one. The student can then observe changes to the object after each transformation is applied.

However, the topic of ‘Visio-spatial Programming and Debugging’ is not just about providing students with visualization aids or learning tools. It is about a fundamental shift in approach to teaching Computer Graphics, supplementing the traditional mathematical approach to teaching transformations and views with spatial learning materials involving the development of student spatial abilities. For example, most Computer Graphics textbooks do not spend much if any time on discussing the debugging of visio-spatial programs. Debugging is a key programming skill. Without a good grasp of how to debug programs, students will struggle with programming tasks. As analysis of student programming shows, these debugging techniques are insufficient for Computer Graphics programming, with `cout` statements of coordinates failing to provide students with the insight required to identify and correct problems. Analysis showed that students did not produce even simple visualization aids without assistance from an instructor but used such technique frequently once it was demonstrated. Hence students need to be taught spatial debugging techniques to debug spatial and mathematical problems; students should be able to apply these techniques in any context rather than having them available only in an external tool or learning aid. Such techniques may involve teaching students how to implement simple visual aids. These aids might include ways of drawing individual transformation steps or drawing out the circle of rotation for problems such as parent-child rotation. They might also include more advanced techniques that allow students to ‘draw out’ viewing volumes in a separate view. Where there is insufficient time to teach students

how to implement such techniques themselves, libraries or tools can be provided to fulfil these roles. Such techniques are especially important for students with poor spatial ability, since individuals with lower spatial ability benefit more from dynamic visualization tools (Höffler, 2010).

The topic of 'Visio-Spatial Programming and Debugging' would also include practical tasks to improve students' spatial ability since it is at the root of student difficulties with spatial programming, and is presumably also responsible for the difficulty students encountered when learning spatial concepts. Similar remedial courses and materials targeted at the improvement of spatial ability already exist for engineering Computer Graphics (see Section 2.3.4). Such courses and materials were introduced as a result of research identifying a lack of spatial ability as a causal factor in poor learning outcomes (see Section 2.3.3). These courses and materials can serve as a basis for the development of Computer Science-Computer Graphics specific spatial ability training materials. Computer Graphics instructors could adopt similar techniques, but these must be tailored to accurately reflect Computer Graphics programming tasks, as task-relevance is important in making spatial ability training effective (Baenninger & Newcombe, 1989). Materials produced for this new topic would then be evaluated based on whether they help address the 'Difficult-to-Visualize' and 'Cognitive Difficulty of Spatial Programming' issues discussed in Section 6.6.6.

## 8.4.2 Future Research and Development

### 8.4.2.1 Further Analysis of Computer Graphics Programming

There are several possibilities for future work emanating from this research study. The identification of issues provides answers to **RQ1**: "What kinds of problems do students learning Computer Graphics Programming experience?", with several different types of problems having been identified. However the analysis conducted as part of this research project may have not reached what is termed 'theoretical saturation' in GT parlance. The analysis of further data may produce further codes and categories to enrich the description of Computer Graphics problems. These categories can then be collected in a catalogue of 'Problems in Computer Graphics programming', and could include both a detailed description of the issue and actual examples like the example presented in Section 6.6.3. This would produce a catalogue similar to that presented by Johnson and Science (1983) providing a collection of common student issues in introductory Computer Graphics Education. In contrast to the original bug catalogue which examined individual versions of source code containing errors this bug catalogue would present student problems spanning many versions of source code, focusing on the problem-solving process underlying the production of errors.

This research focused on a small set of students and a subset of Computer Graphics topics thought to be at the core of Computer Graphics Education. Implementation of the method itself (including

implementation of software) was a time-consuming but integral task that took up around half the time spent on the project meaning that less time was available for actual analysis. A complete analysis of all problems faced by students in Computer Graphics programming was never a feasible goal for this research project. More complete analysis should involve multiple Computer Graphics educators. It should also span the whole range of different topics that can be covered under the mantle of Computer Graphics Education. As just one of many examples, such research might focus on the role on Shaders, which play an increasingly important role in modern Computer Graphics programming. It is hoped that this research together with the proposed Segmentation method of Project History analysis will be a starting point for the ongoing development of a theory of student Computer Graphics programming.

#### 8.4.2.2 Improvements to the SCORE plug-in and analyser

The SCORE Analyser software will continue to be developed. Several improvements to the SCORE analysis toolset were discussed in Section 4.5. These include extending the SCORE plug-in to allow a hybrid local / server Project History storage model. This would give researchers the option to collect Project History data as they program. Another future improvement would be to implement a comment entry form for the SCORE plug-in to be displayed in the main Eclipse interface. This would periodically request a comment from the student regarding the work they were performing. Storing these comments alongside the Project History data would make it easier for the researcher to understand the student's programming efforts, and would also yield some additional student perspectives on their programming.

#### 8.4.2.3 Further development of Machine-Segmenting

Work will also continue on the implementation of different Machine-Segmenting algorithms or extension algorithms. Also, existing algorithms will be evaluated with more data, including data coming from other areas besides Computer Graphics. Such evaluation could also shed light on whether different settings are more appropriate for different kinds of assignment specifications. Several potential extensions to the current Machine-Segmenting algorithms were discussed in Section 7.13.

#### 8.4.3 Broader applications of the SCORE analysis toolset

The analysis of Project Histories via the SCORE software package may also prove useful in other contexts:

- **Evaluation of assessment task effectiveness:** In a classroom setting, analysis of Project Histories could be used to evaluate the effectiveness of assessment task specifications. Analysis of Project

Histories reveals the amount of time students spent on different tasks, including those unrelated to the core topics related to the course in question. It can also reveal problems with an assignment specification which confound students and impede learning, such as the extraneous challenge caused by the interplay of different problems during students' implementation of child-parent rotation. Recognising such flaws in the design of learning tasks allows the educator to re-formulate the task in pedagogically sound ways.

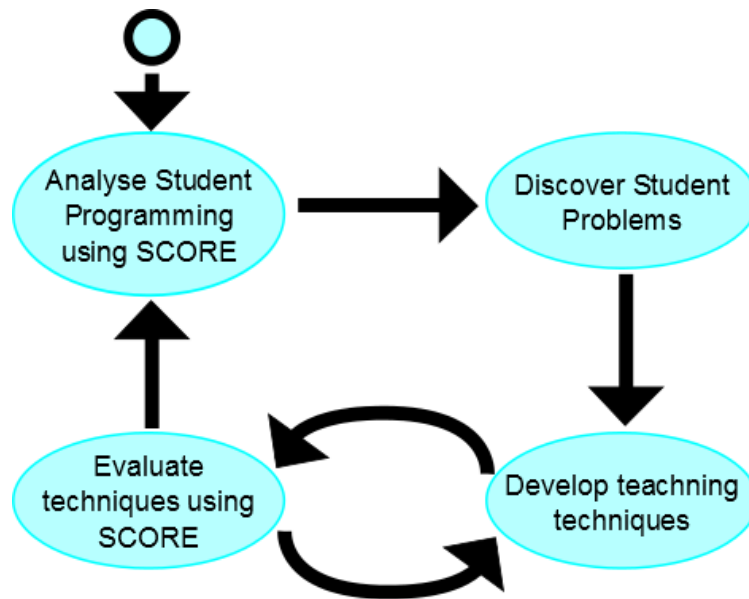
- **Evaluation of course materials:** Analysis of Project Histories can be utilised to analyse the effectiveness of different course materials such as the role played by a programming language or a library by examining Project Histories and focusing on those Segments with involve topic, technique or material in question. Careful analysis can also reveal the effect of interventions such as providing students with an additional tool or learning material by comparing Project Histories produced in a course iteration without and an iteration with the intervention in place.
- **Identification of student problems during the semester:** If educators conduct ongoing analysis of student work during the semester student problems can be identified immediately. This allows the educator to address individual student problems by providing assistance to students, and to identify problems shared by many students which can then be discussed in class. At the end of the semester, collected problems can then be contributed to a growing catalogue of student misconceptions and issues.
- **Marking assistance:** Analysis of Project Histories can help better assess the amount of effort put into an assignment by a student, which could lead to fairer marking of students and provision of better feedback. While analysis of student assignments may take more time than regular marking, the proposed machine approaches could significantly speed up this process and make it feasible in an educational context.
- **Plagiarism detection:** Analysis of fine-grained Project Histories could also be used to detect plagiarism. Large amounts of source code suddenly appearing would clearly indicate plagiarism. Students could not circumvent this by simply introducing the source code line by line, since then none of the versions until the final version would compile, which can also be easily detected during analysis. Other metrics derived from Project Histories measuring the similarity of two histories could exclude many attempts at circumventing such a plagiarism detection mechanism by copying an entire Project History.

These are areas of future work that may be conducted by the principal researcher, but that would also be possible for other educators to complete based on the methods and tools emanating from this study.

## 8.5 Concluding Remarks

This research project was commenced with the intention of discovering the types of problems students faced during Computer Graphics programming. The research uncovered several issues related to Computer Graphics problem solving. Some were expected, such as issues surrounding spatial programming, but in these cases analysis clarified how issues surrounding these topics arose. For the example of spatial programming, analysis showed that students struggle both with concepts related to spatial programming and with the application of these concepts. Other discovered issues were unexpected, such as the 'Interplay of different problems' issue involving multiple overlapping problems occurring at the same time. This particular issue represented a substantial problem for several students, but would not have been detected if not for a thorough analysis of student programming actions. Going forward the discovery and analysis of such issues will allow educators to better understand and support student learning. The topic 'Visio-Spatial Programming and Debugging' proposed in Section 8.4 which is based on lessons learned during the research presented in this thesis may provide a starting point for these efforts. Both the Project History analysis method proposed and the findings presented are early forays which may generate new research directions or hypotheses to be tested, it provides a new approach for fellow Computer Science Education researchers.

In addition to being utilised in the identification of student issues, the SCORE toolset and associated analytic methods can be utilised to evaluate new learning and scaffolding techniques. It can facilitate a continuous investigative cycle (see Figure 111) involving the analysis of student issues, development of teaching techniques, evaluation of teaching techniques followed by improvement until teaching techniques are successful in addressing student issues. In this way, the cycle of analysis of student problems can lead to continual improvement in the understanding of student problems. This in turn can lead to continual improvement of educational materials. As course contents change over time, continual analysis will also enable the educator to identify student issues arising from these new topics.



**Figure 111: Continual analysis of student programming, development and evaluation of teaching techniques**

The proposed method of analysis using the SCORE Analyser is not in any way specific to the analysis of Computer Graphics Education; it can be applied to any Computer Science programming domain to provide an unparalleled level of analysis detail of real-world student programming. The data gathering approach can be applied with virtually no effort and is almost completely transparent to students. With the continued development of Machine-Segmenting approaches such as those proposed in this thesis, the method of analysis will also become increasingly time-efficient for the researcher. This will allow for the analysis of larger sets of Project Histories spanning more assignments or more students. The SCORE analysis toolset and associated methods can also help individual educators teaching in these areas to analyse the impact of their own materials and strategies upon students in their classes.

The process analysing student programs was fascinating. The understanding of particularly tricky student problems often led to “Eureka!” moments which clearly revealed the nature of a student issue which was then found to correspond to similar issues in other students’ programming. The experience was one of forensic investigation of student programming actions, unearthing rich and complex nuggets of student problem-solving. When coded, compared and analysed little pieces of insight provided a step-by-step understanding of the student problem-solving process. The most vital ingredient in a recipe of successful teaching is a deep, well-grounded understanding of student problems. Moving forward, it is hoped that the analysis methods and tools presented in this thesis can help contribute to the development of such understanding, leading to better and more precisely tailored education methods and ultimately to better learning outcomes for students.

*"We shall not cease from exploration, and the end of all our exploring will be to arrive where we started and know the place for the first time." T.S. Eliot.*

The End.

## References

- Adams, B., Jiang, Z. M., & Hassan, A. E. (2010). Identifying crosscutting concerns using historical code changes. In *2010 ACM/IEEE 32nd International Conference on Software Engineering, 2-8 March* (pp. 305–314). Cape Town, South Africa: ACM.
- Alias, M., Black, T. R., & Gray, D. E. (2002). Effect of instruction on spatial visualization ability in civil engineering students. *International Education Journal*, 3(1), 1–12.
- Allevato, A., Thornton, M., Edwards, S., & Perez-Quinones, M. (2008). Mining data from an automated grading and testing system by adding rich reporting capabilities. In *Educational Data Mining 2008, 20-21 June* (pp. 167–176). Montreal, Canada.
- Anderson, J. R., & Jeffries, R. (1985). Novice LISP errors: Undetected losses of information from working memory. *Human-Computer Interaction*, 1(2), 107–131.
- Andújar, C., & Vázquez, P. P. (2006). A Programmable Tutor for OpenGL Transformations. In *Eurographics 2006 Proceedings, 4-8 September*. Vienna, Austria: Eurographics.
- Angel, E. (1997). Teaching a three-dimensional computer graphics class using OpenGL. In *Proceedings of ACM SIGGRAPH 97, 3-8 August 1997* (Vol. 31, pp. 54–55). Los Angeles, USA: ACM.
- Atkins, D., Ball, T., Graves, T., & Mockus, A. (1999). Using version control data to evaluate the impact of software tools. In *Proceedings of the 1999 International Conference on Software Engineering, 22 May* (pp. 324–333). Los Angeles, USA: IEEE.
- Baddeley, A. D., & Hitch, G. J. (1974). Working memory. *The psychology of learning and motivation*, 8, 47–89.
- Baenninger, M., & Newcombe, N. (1989). The role of experience in spatial test performance: A meta-analysis. *Sex Roles*, 20(5), 327–344.
- Bailey, M., & Cunningham, S. (2007). A hands-on environment for teaching GPU programming. *ACM SIGCSE Bulletin*, 39(1), 254–258.
- Beckhaus, S., & Blom, K. J. (2007). Teaching, Exploring, Learning—Developing Tutorials for In-Class Teaching and Self-Learning. *Computer Graphics Forum 2007*, 26(4), 725–736.



- Biggerstaff, T. J., Mitbender, B. G., & Webster, D. (1993). The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering, 17-21 May 1993* (pp. 482–498). St. Louis, USA: IEEE.
- Blade, M. F., & Watson, W. S. (1955). Increase in spatial visualization test scores during engineering study. *Psychological Monographs: General and Applied*, 69(12), 1.
- Bodner, G. M., & Guay, R. B. (1997). The Purdue visualization of rotations test. *The Chemical Educator*, 2(4), 1–17.
- Boutin, P. (2009). Browser makers announce support for built-in 3D graphics. *Venturebeat.com*. Retrieved from <http://venturebeat.com/2009/08/04/browser-makers-announce-support-for-built-in-3d-graphics/>
- Breu, S., & Zimmermann, T. (2006). Mining aspects from version history. In *21st IEEE/ACM International Automated Software Engineering Conference, 18-22 September 2006* (pp. 221–230). Tokyo, Japan: IEEE.
- Bryce, R. C., Cooley, A., Hansen, A., & Hayrapetyan, N. (2010). A one year empirical study of student programming bugs. In *Frontiers in Education Conference (FIE), 27-30 October 2010* (pp. F1G–1 – F1G–7). Washington, DC, USA: IEEE.
- Butler, M., & Morgan, M. (2007). Learning challenges faced by novice programming students studying high level and low feedback concepts. In *Ascilite Singapore 2007, 2-5 December* (pp. 99–107). Singapore: Ascilite.
- Canfora, G., & Cerulo, L. (2006). Fine grained indexing of software repositories to support impact analysis. In *Proceedings of the 2006 International Workshop on Mining Software Repositories, 22-23 May* (pp. 105–111). Shanghai, China: ACM.
- Canfora, G., Cerulo, L., & Di Penta, M. (2006). On the use of line co-change for identifying crosscutting concern code. In *22nd IEEE International Conference on Software Maintenance, 24-27 September 2006* (pp. 213–222). Dublin, Ireland: IEEE.

- Canfora, G., Cerulo, L., & Di Penta, M. (2007). Identifying changed source code lines from version repositories. In *Proceedings of the Fourth International Workshop on Mining Software Repositories, 19-20 May 2007*. Minneapolis, USA: IEEE.
- Carroll, J. B. (1993). *Human cognitive abilities: A survey of factor-analytic studies*. Cambridge Univ Pr.
- Carter, C. S., LaRussa, M. A., & Bodner, G. M. (1987). A study of two measures of spatial ability as predictors of success in different levels of general chemistry. *Journal of Research in Science Teaching*, 24(7), 645–657.
- Case, C. (1999). Three Collaborative Models for Computer Graphics Education. In *Computer Graphics and Visualization Education '99, 3-5 July 1999*. Coimbra, Portugal: ACM.
- Celes, W., & Corson-Rikert, J. (1997). Act: an easy-to-use and dynamically extensible 3D graphics library. In *Proceedings of the X Brazilian Symposium on Computer Graphics and Image Processing, 14-17 October 1997* (pp. 26–33). Campos Do Jordao, Brazil: IEEE.
- Chalmers, A., & Cunningham, S. (2002). New Media and Future Education. In *Proceedings of the East-West-Vision International Workshop & Project Festival Computer Vision, Computer Graphics* (pp. 67–74). Graz, Austria: New Media.
- Chalmers, A., & Dalton, C. (2002). Visual perception in computer graphics education. In *Eurographics-ACM SIGGRAPH Workshop on Computer Graphics Education, 6-7 July 2002*. Bristol, UK: ACM.
- Charmaz, K. (2006). *Constructing grounded theory: A practical guide through qualitative analysis*. Thousand Oaks, California, USA: Pine Forge Press.
- Cheng, N. (1999). Playing with digital media: enlivening computer graphics teaching. In *Proceedings of the Association for Computer Aided Design in Architecture (ACADIA)*. Snowbird, Utah: ACADIA.
- Clevenger, J., Chaddock, R., & Bendig, R. (1991). TUGS—a tool for teaching computer graphics. In *ACM SIGGRAPH 91, 28 July - 2 August 1991* (Vol. 25, pp. 158–164). Las Vegas, USA: ACM.

- Cliburn, D. (2006). Incorporating Virtual Reality Concepts into the Introductory Computer Graphics Course. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, 1-5 March 2006* (pp. 368–372). Houston, USA: ACM.
- Corbin, J., & Strauss, A. (1990). Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1), 3–21.
- Creswell, J. W. (2012). *Qualitative inquiry and research design: Choosing among five approaches* (2nd ed.). Thousand Oaks, USA: Sage Publications.
- Creswell, J. W. (2013). *Research design: Qualitative, quantitative, and mixed methods approaches* (2nd ed.). Thousand Oaks, USA: Sage Publications.
- Cunningham, S. (2000). Powers of 10: the case for changing the first course in computer graphics. *ACM SIGCSE Bulletin*, 32(1), 46–49.
- Cunningham, S. (2002). Graphical problem solving and visual communication in the beginning computer graphics course. *ACM SIGCSE Bulletin*, 34(1), 181–185.
- Cunningham, S., Hansmann, W., Laxer, C., & Shi, J. (2004). The beginning computer graphics course in computer science. In *ACM SIGGRAPH 04, 8-12 August 2004* (Vol. 38, pp. 24–25). Los Angeles, USA: ACM.
- Design World Staff. (2007, August 23). Submarine Design Comes Together In Virtual Reality. *Design World*. Retrieved from [http://www.designworldonline.com/submarine-design-comes-together-in-virtual-reality/#\\_](http://www.designworldonline.com/submarine-design-comes-together-in-virtual-reality/#_)
- Ebert, D. S., & Bailey, D. (1999). An Interdisciplinary Approach to Teaching Computer Animation to Artists and Computer Scientists. In *Proceedings of the Computer Graphics and Visualization Education Workshop, 3-5 July 1999* (pp. 99–104). Coimbra, Portugal: ACM.
- Edwards, S. H., & Perez-Quinones, M. A. (2008). Web-CAT: automatically grading programming assignments. *ACM SIGCSE Bulletin*, 40(3), 328.
- Edwards, S. H., Snyder, J., Pérez-Quinones, M. A., Allevato, A., Kim, D., & Tretola, B. (2009). Comparing effective and ineffective behaviors of student programmers. In *Proceedings of*

- the Fifth International Workshop on Computing Education Research Workshop, 10-11 August 2009 (pp. 3–14). Berkley, USA: ACM.
- Eisenstadt, M. (1993). Tales of debugging from the front lines. In *Empirical Studies of Programmers: Fifth Workshop, 3-15 December 1993* (pp. 86–112). Palo Alto, USA: Ablex Publishing Corporation.
- Fenwick Jr, J. B., Norris, C., Barry, F. E., Rountree, J., Spicer, C. J., & Cheek, S. D. (2009). Another look at the behaviors of novice programmers. *ACM SIGCSE Bulletin*, 41(1), 296–300.
- Ferrini-Mundy, J. (1987). Spatial training for calculus students: Sex differences in achievement and in visualization ability. *Journal for Research in Mathematics Education*, 18(2), 126–140.
- Figueiredo, F. C., Eber, D. E., & Jorge, J. A. (2003). CGEMS: Computer graphics educational materials server. In *ACM SIGGRAPH 2003 Educators Program, 27-31 July* (pp. 1–7). San Diego, USA: ACM.
- Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2), 93–116.
- Ganovelli, F., & Corsini, M. (2009). eNVyMyCar: A Multiplayer Car Racing Game for Teaching Computer Graphics. *Computer Graphics Forum*, 28(8), 2025–2032.
- George, S. E. (2002). Learning and the reflective journal in computer science. *Australian Computer Science Communications*, 24(1), 77–86.
- Given, L. M. (2008). *Qualitative Research Methods* (Vol. 2). Thousand Oaks, USA: Sage Publications.
- Glaser, B. (1978). *Theoretical sensitivity: Advances in the methodology of grounded theory* (Vol. 2). Sociology press Mill Valley, CA.
- Glaser, B. (1992). *Emergence vs forcing: Basics of grounded theory analysis*. Mill Valley, USA: Sociology Press.
- Glaser, B. (1998). *Doing grounded theory: Issues and discussions* (Vol. 254). Sociology Press Mill Valley, CA.

- Glaser, B. (2008). Conceptualization: On theory and theorizing using grounded theory. *International Journal of Qualitative Methods*, 1(2), 23–38.
- Glaser, B., & Strauss, A. (1967). *The discovery of grounded theory: Strategies for qualitative research*. New York, USA: Aldine de Gruyter.
- Google Maps 3d. (2012). Google. Retrieved from <http://www.google.com/mobile/maps/3d/>
- Görke, J., Hanisch, F., & Strasser, W. (2005). Live graphics gems as a way to raise repositories for computer graphics education. In *ACM SIGGRAPH 2005 Educators Program, 31 July - 4 August* (p. 37). Los Angeles, USA: ACM.
- Gould, D. L., Simpson, R. M., & Van Dam, A. (1999). Granularity in the design of interactive illustrations. *ACM SIGCSE Bulletin*, 31, 306–310.
- Graetz, J. M. (1981). The origin of Spacewar. *Creative Computing*, 56, 67.
- Grissom, S. (1996). uisGL: a C++ library to support graphics education. In *ACM SIGGRAPH '96, 4-9 August 1996* (Vol. 30, pp. 36–37). New Orleans, USA: ACM.
- Guttman, R., Epstein, E. E., Amir, M., & Guttman, L. (1990). A structural theory of spatial abilities. *Applied Psychological Measurement*, 14(3), 217–236.
- Guttman, R., & Greenbaum, C. W. (1998). Facet theory: Its development and current status. *European psychologist*, 3(1), 13.
- Hachman, M. (2012). Google Tips New 3D Buildings, Offline Access. *Pcmag.com*. Retrieved from <http://www.pcmag.com/article2/0,2817,2405407,00.asp>
- Haney, F. M. (1972). Module connection analysis: a tool for scheduling software debugging activities. In *Proceedings of the December 5-7, Fall Joint Computer Conference, Part I* (pp. 173–179). Presented at the AFIPS Fall Joint Computer Conference, Anaheim, USA: ACM.
- Hassan, A. E., & Holt, R. C. (2004). Predicting change propagation in software systems. In *Proceedings. 20th IEEE International Conference on Software Maintenance, 11-14 September 2004* (pp. 284–293). Edinburgh, Scotland: IEEE.

- Hays, R. T., Jacobs, J. W., Prince, C., & Salas, E. (1992). Flight simulator training effectiveness: A meta-analysis. *Military Psychology*, 4(2), 63–74.
- Hill, F. S., & Kelley, S. M. (2001). *Computer graphics: using OpenGL*. Upper Saddle River, USA: Prentice hall.
- Hillier, B. (2012, June 8). Star Wars 1313 dev predicts graphics will be “indistinguishable from reality” in ten years. VG247. Retrieved from <http://www.vg247.com/2012/08/06/star-wars-1313-dev-predicts-graphics-will-be-indistinguishable-from-reality-in-ten-years/>
- Hitchner, L. E., & Sowizral, H. A. (2000). Adapting computer graphics curricula to changes in graphics. *Computers & Graphics*, 24(2), 283–288.
- Höffler, T. N. (2010). Spatial ability: Its influence on learning with visualizations—a meta-analytic review. *Educational Psychology Review*, 22(3), 245–269.
- Howard, T., Hubbard, R., & Murta, A. (1999). Maverik: a virtual reality system for research and teaching. In *Proceedings GVE99 Workshop, 3-5 July 1999* (Vol. 2). Coimbra, Portugal: ACM.
- Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1), 153–156.
- Hsi, S., Linn, M. C., & Bell, J. E. (1997). The role of spatial reasoning in engineering and the design of spatial instruction. *Journal of Engineering Education*, 86(2), 151–158.
- Iglesias, A., & Gálvez, A. (2004). An Introductory Course on Computer Graphics for Engineers. In *The Proceedings of the 7th 3IA*. Limoges, France: Springer.
- Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education*, 15(1), 25–40.
- Jadud, M. C. (2006a). Methods and tools for exploring novice compilation behaviour. In *Proceedings of the Second International Workshop on Computing Education Research, 9-10 September 2006* (pp. 73–84). Canterbury, UK: ACM.

- Jadud, M. C. (2006b). *An exploration of novice compilation behaviour in BlueJ* (Doctoral Thesis).  
University of Kent, Kent, UK.
- Jadud, M. C., & Henriksen, P. (2009). Flexible, reusable tools for studying novice programmers. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop, 10-11 August 2009* (pp. 37–42). Berkley, USA: ACM.
- Johnson, W. L., & Science, Y. U. D. of C. (1983). *Bug catalogue I*. Yale University, Dept. of Computer Science.
- Jones, S. J., & Burnett, G. E. (2007). Spatial skills and navigation of source code. *ACM SIGCSE Bulletin*, 39(3), 231–235.
- Jones, S. J., & Burnett, G. E. (2008). Spatial ability and learning to program. *Human Technology*, 4(1), 47–61.
- Knuth, D. E. (1971). An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2), 105–133.
- Ko, A. J., & Myers, B. A. (2003). Development and evaluation of a model of programming errors. In *Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments, 31 October* (pp. 7–14). Auckland, New Zealand: IEEE.
- Kosslyn, S. M. (1988). Aspects of a cognitive neuroscience of mental imagery. *Science*, 240(4859), 1621–1626.
- Kozhevnikov, M., & Hegarty, M. (2001). A dissociation between object manipulation spatial ability and spatial orientation ability. *Memory & Cognition*, 29(5), 745–756.
- Leopold, C., Gorska, R. A., & Sorby, S. A. (2001). International experiences in developing the spatial visualization abilities of engineering students. *Journal for Geometry and Graphics*, 5(1), 81–91.
- Liu, D., Marcus, A., Poshyvanyk, D., & Rajlich, V. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the Twenty-second*

- IEEE/ACM International Conference on Automated Software Engineering, 5-9 November 2007* (pp. 234–243). Atlanta, USA: ACM.
- Liu, Y., Stroulia, E., Wong, K., & German, D. (2004). Using CVS historical information to understand how students develop software. In *1st International Workshop on Mining Software Repositories, 25 May 2004* (pp. 32–36). Edinburgh, Scotland: IEEE.
- Livshits, B., & Zimmermann, T. (2005). DynaMine: finding common error patterns by mining software revision histories. *ACM SIGSOFT Software Engineering Notes*, 30(5), 296–305.
- Lowther, J. L., & Shene, C. K. (2000). Rendering+ modeling+ animation+ postprocessing= computer graphics. *Journal of Computing Sciences in Colleges*, 16(4), 20–28.
- Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval* (Vol. 1). Cambridge, England: Cambridge University Press.
- Marcus, A., Sergeyev, A., Rajlich, V., & Maletic, J. I. (2004). An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering, 18-22 April 2004* (pp. 214–223). Delft, The Netherlands: IEEE.
- Martín-Dorta, N., Saorin, J. L., & Contero, M. (2008). Development of a fast remedial course to improve the spatial abilities of engineering students. *Journal of Engineering Education*, 97(4), 505–513.
- Martín-Gutiérrez, J., Gil, F. A., Contero, M., & Saorín, J. L. (2010). Dynamic three-dimensional illustrator for teaching descriptive geometry and training visualisation skills. *Computer Applications in Engineering Education*, 21(1).
- Marton, F., & Säljö, R. (1976). On qualitative differences in learning - Outcome as a function of the learner's conception of the task. *British Journal of educational Psychology*, 46(2), 115–127.
- McGee, M. G. (1979). Human spatial abilities: Psychometric studies and environmental, genetic, hormonal, and neurological influences. *Psychological bulletin*, 86(5), 889.
- McGrath, M. B., & Brown, J. R. (2005). Visual learning for science and engineering. *Computer Graphics and Applications*, 25(5), 56–63.



- McKeogh, J., & Exton, C. (2004). Eclipse plug-in to monitor the programmer behaviour. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange, 24-28 October* (pp. 93–97). Vancouver, Canada: ACM.
- Mengel, S. A., & Ulans, J. (1998). Using Verilog LOGISCOPE to analyze student programs. In *28th Annual Frontiers in Education Conference, 4-7 November 1998* (Vol. 3, pp. 1213–1218). Tempe, USA: IEEE.
- Mengel, S. A., & Yerramilli, V. (1999). A case study of the static analysis of the quality of novice student programs. *ACM SIGCSE Bulletin*, 31(1), 78–82.
- Mierle, K., Laven, K., Roweis, S., & Wilson, G. (2005). Mining student CVS repositories for performance indicators. *ACM SIGSOFT Software Engineering Notes*, 30(4), 1–5.
- Mohilner, P. R. (1975). Implementation of a simulated display processor for computer graphics education. In *Proceedings of the 2nd Annual Conference on Computer Graphics and Interactive Techniques, 25-27 June 1975* (pp. 222–231). Bowling Green, Ohio: ACM.
- Mohler, J. L. (2009). A Review of Spatial Ability Research. *Engineering Design Graphics Journal*, 72(2), 19–30.
- Murphy, C., Kaiser, G., Loveland, K., & Hasan, S. (2009). Retina: helping students and instructors based on observed programming activities. *ACM SIGCSE Bulletin*, 41(1), 178–182.
- Naiman, A. C. (1996). Interactive teaching modules for computer graphics. In *Proceedings of ACM SIGGRAPH 96, 4-9 August 1996* (Vol. 30, pp. 33–35). New Orleans, USA.
- Necaise, R. D. (2006). Interactive graphics using OpenGL and the Graphix Windowing Toolkit. *Journal of Computing Sciences in Colleges*, 22(2), 220–227.
- Norris, C., Barry, F., Fenwick Jr, J. B., Reid, K., & Rountree, J. (2008). ClockIt: collecting quantitative data on how beginning software developers really work. *ACM SIGCSE Bulletin*, 40(3), 37–41.
- Owen, G. S., Zhu, Y., Chastine, J., & Payne, B. R. (2005). Teaching programmable shaders: lightweight versus heavyweight approach. In *ACM SIGGRAPH 2005 Educators Program, 31 July - 4 August* (p. 40). Los Angeles, USA: ACM.

- Pallrand, G. J., & Seeber, F. (1984). Spatial ability and achievement in introductory physics. *Journal of Research in Science Teaching*, 21(5), 507–516.
- Paquette, E. (2004). CoGIP: a course on 2D computer graphics and image processing. In *ACM SIGGRAPH 2004 Educators Program, 8-12 August* (pp. 1–2). New York, USA: ACM.
- PatentlyApple Staff. (2012). Apple Introduces us to Patent Pending Safari 3D. *Patently Apple*. Retrieved from <http://www.patentlyapple.com/patently-apple/2012/05/apple-introduces-us-to-patent-pending-safari-3d.html>
- Paterson, D. G., Elliott, R. M., Anderson, L. D., Toops, H. A., & Heidbreder, E. (1930). *Minnesota mechanical ability tests*. Minneapolis, USA: University of Minnesota Press.
- Pellegrino, J. W., Alderton, D. L., & Shute, V. J. (1984). Understanding spatial ability. *Educational Psychologist*, 19(4), 239–253.
- Pereira, J. A. M., & Gomes, M. R. (1999). Learning 3D Visualisation with MALUDA. In *Computer Graphics and Visualization Education 1999, 3-5 July*. Coimbra, Portugal: ACM.
- Pribyl, J. R., & Bodner, G. M. (1987). Spatial ability and its role in organic chemistry: A study of four organic courses. *Journal of Research in Science Teaching*, 24(3), 229–240.
- Rodrigo, M. M. T., Tabanao, E., Lahoz, M. B. E., & Jadud, M. C. (2009). Analyzing online protocols to characterize novice java programmers. *Philippine Journal of Science*, 138(2), 177–190.
- Saldaña, J. (2009). *The coding manual for qualitative researchers*. Thousand Oaks, USA: Sage Publications.
- Schweitzer, D., & Appolloni, T. (1995). Integrating introductory courses in computer graphics and animation. *ACM SIGCSE Bulletin*, 27(1), 186–190.
- Sears, A., & Wolfe, R. (1995). Visual analysis: adding breadth to a computer graphics course. *ACM SIGCSE Bulletin*, 27(1), 195–198.
- Seymour, N. E., Gallagher, A. G., Roman, S. A., O'Brien, M. K., Bansal, V. K., Andersen, D. K., & Satava, R. M. (2002). Virtual reality training improves operating room performance: results of a randomized, double-blinded study. *Annals of surgery*, 236(4), 458.

- Shabo, A., Guzdial, M., & Stasko, J. (1996). Addressing student problems in learning computer graphics. In *Proceedings of ACM SIGGRAPH 96, 4-9 August 1996* (Vol. 30, pp. 38–40). New Orleans, USA: ACM.
- Shirley, P., Sung, K., Brunvand, E., Davis, A., Parker, S., & Boulos, S. (2007). Rethinking graphics and gaming courses because of fast ray tracing. In *ACM SIGGRAPH 2007 Educators Program, 5-9 August* (p. 15). San Diego, USA: ACM.
- Smith, M. (2005). An Interactive Graphics Tool to Support Student Learning of Matrix-based 2D Transformation Calculations. In *Proceedings of the Sixth Irish Workshop on Computer Graphics (2005)*. Dublin, Ireland: Eurographics.
- Spacco, J., Hovemeyer, D., & Pugh, W. (2004). An Eclipse-based course project snapshot and submission system. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange, 24-28 October* (pp. 52–56). Vancouver, British Columbia, Canada: ACM.
- Spacco, J., Hovemeyer, D., Pugh, W., Emad, F., Hollingsworth, J. K., & Padua-Perez, N. (2006). Experiences with marmoset: designing and using an advanced submission and testing system for programming courses. *ACM SIGCSE Bulletin*, 38(3), 13–17.
- Spacco, J., Strecker, J., Hovemeyer, D., & Pugh, W. (2005). Software repository mining with Marmoset: an automated programming project snapshot and testing system. *ACM SIGSOFT Software Engineering Notes*, 30(4), 1–5.
- Stern, P. N. (1980). Grounded theory methodology: Its uses and processes. *Journal of Nursing Scholarship*, 12(1), 20–23.
- Stone, R. (2001). Virtual reality for interactive training: an industrial practitioner's viewpoint. *International Journal of Human-Computer Studies*, 55(4), 699–711.
- Strauss, A., & Corbin, J. (1998). *Basics of qualitative research: Techniques and procedures for developing grounded theory* (2nd ed.). Thousand Oaks, USA: Sage Publications.
- Stumpf, H., & Eliot, J. (1999). A structural analysis of visual spatial ability in academically talented students. *Learning and Individual Differences*, 11(2), 137–151.

- Sung, K., & Shirley, P. (2004). A top-down approach to teaching introductory computer graphics. *Computers & Graphics*, 28(3), 383–391.
- Sung, K., Shirley, P., & Rosenberg, B. R. (2007). Experiencing aspects of games programming in an introductory computer graphics class. *ACM SIGCSE Bulletin*, 39(1), 249–253.
- Surma, D. R. (2006). The use of presentations and competition in an introductory computer graphics course. *Journal of Computing Sciences in Colleges*, 22(1), 7–14.
- Sutherland, I. E. (1964). Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE Design Automation Workshop* (pp. 6–329). New York, USA: ACM.
- Talton, J. O., & Fitzpatrick, D. (2007). Teaching graphics with the OpenGL shading language. *ACM SIGCSE Bulletin*, 39(1), 259–263.
- Taxén, G. (2004). Teaching computer graphics constructively. *Computers & Graphics*, 28(3), 393–399.
- Thomas, P. A. V. (1979). DRAW (IT) N-a computer graphics education package. *ACM SIGCSE Bulletin*, 11(1), 228–231.
- Tori, R., Bernardes Jr, J. L., & Nakamura, R. (2006). Teaching introductory computer graphics using java 3D, games and customized software: a Brazilian experience. In *ACM SIGGRAPH 2006 Educators Program, 1-3 August* (p. 12). Boston, USA: ACM.
- Truong, N., Roe, P., & Bancroft, P. (2004). Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education, January 2004* (Vol. 30, pp. 317–325). Dunedin, New Zealand: Australian Computer Society, Inc.
- Ullrich, T., & Fellner, D. (2005). Computer Graphics Courseware. In *Eurographics 2005* (pp. 11–17).
- Van Someren, M. W., Barnard, Y. F., & Sandberg, J. A. (1994). *The think aloud method: A practical guide to modelling cognitive processes*. Academic Press London.
- Vee, M. H. N. C., Meyer, B., & Mannock, K. . (2005). *Empirical study of novice errors and error paths* (Technical report). ETH Zurich.

- Wernert, E. (1997). A unified environment for presenting, developing and analyzing graphics algorithms. In *Proceedings of ACM SIGGRAPH 97, 3-8 August 1997* (Vol. 31, pp. 26–28). Los Angeles, USA: ACM.
- Wiese, K. C., Kyrilov, V., Nesbit, J. C., & Calvert, T. (2003). Interactive Media and Team-based Learning in a Computer Graphics Course blending Online and Face-to-face Delivery. In *Computers and Advanced Technology in Education 2003, 30 June - 2 July*. Rhodes, Greece: ACTA Press.
- Wilde, N., & Scully, M. C. (1995). Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1), 49–62.
- Williams, C. C., & Hollingsworth, J. K. (2005). Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6), 466–480.
- Wittmann, M. (2012a). *A short SCORE Manual* (Manual). Macquarie University. Retrieved from [https://sourceforge.net/projects/score-package/files/SCORE\\_Analyser\\_Manual.pdf/download](https://sourceforge.net/projects/score-package/files/SCORE_Analyser_Manual.pdf/download), Accessed on the 17/09/2012
- Wittmann, M. (2012b). *A Visual Introduction to SCORE*. Retrieved from [http://www.youtube.com/watch?v=cvF5ICZtYbQ&list=PL5A1E9556055F67E2&feature=plpp\\_play\\_all](http://www.youtube.com/watch?v=cvF5ICZtYbQ&list=PL5A1E9556055F67E2&feature=plpp_play_all), Accessed on the 17/09/2012
- Wittmann, M., Bower, M., & Kavakli-Thorne, M. (2011). Using the SCORE software package to analyse novice computer graphics programming. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education, 27-29 June 2011* (pp. 118–122). Darmstadt, Germany: ACM.
- Wittmann, M., Kavakli-Thorne, M., & Bower, M. (2009). *Review of Computer Graphics Education*. Unpublished Manuscript, Macquarie University, Sydney, Australia.
- Wolfe, R. (2000). Bringing the introductory computer graphics course into the 21st century. *Computers & Graphics*, 24(1), 151–155.

- Wolfe, R. (2002). Teaching visual aspects in an introductory computer graphics course. *Computers & Graphics*, 26(1), 163–168.
- Wolfe, R., & Sears, A. (1996). An effective tool for learning the visual effects of rendering algorithms. In *Proceedings of ACM SIGGRAPH 96, 4-9 August 1996* (Vol. 30, pp. 54–55). New Orleans, USA: ACM.
- Xiang, Z. (1994). A nontraditional computer graphics course for computer science students. *ACM SIGGRAPH Computer Graphics*, 28(3), 186–188.
- Ying, A. T. T., Murphy, G. C., Ng, R., & Chu-Carroll, M. C. (2004). Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9), 574–586.
- Yu, T. T., Lowther, J., & Shene, C. K. (2005). Photon mapping made easy. *ACM SIGCSE Bulletin*, 37(1), 201–205.
- Zhu, M. (2004). *Recall, precision and average precision* (Working Paper). Waterloo: University of Waterloo. Retrieved from <http://140.118.107.213/personal/master95/kid0310/document/Recall,%20Precision%20and%20Average%20Precision.pdf>
- Zimmermann, T., & Weissgerber, P. (2004). Preprocessing CVS data for fine-grained analysis. In *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, 25 May (pp. 2–6). Edinburgh, Scotland: IEEE.
- Zimmermann, T., Zeller, A., Weissgerber, P., & Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6), 429–445.

# 9 Appendix

---

## 9.1 Glossary of Terms

**Analysis of Segment Features (Segment Feature Analysis):** A quantitative analysis method which involves analysing features of Segments (see Segments), produced via Segmenting (see Segmenting). These features include the number of Changes in a Segment, the time taken for all the Changes in the Segment and other quantifiable Segment features.

**Change:** A Change is the sum total of all the changes from one version (see the term version) to the next. For example, if the statement “print(Hello)” is removed from a file and the statement “print(Bye)” is added to a file, then the Change consists of these two modifications. Therefore, a Change is always associated with two versions, the ‘before’ version and the ‘after’ version.

**Change-Coding:** Change-Coding involves the Coding (see Coding) of each individual Change in a Project History.

**Coding:** Coding in the context of this thesis refers to the coding of data. When coding data, codes which summarize the properties of a piece of data are applied.

**Line History:** A Line History for a given line of source code consists of all the Modifications which have been applied to that line in the Project History. This consists of one Addition (the line is initially created) and any number of Deletion, Move, Mutation, Maintained and Ghost modifications. By definition, a line can be modified only once in any given Change. Also, since a line is considered Maintained if it is unchanged, a line will always have a modification type if it exists in a particular Change. Note that Line Histories need not be contiguous, since a line can be deleted and then brought back via a Ghost modification.

**Line History Generation:** The process of generating a Line History for each unique line of source code in the Project History. This process involves creating new Line Histories whenever a new line is discovered (Added), and adding modifications to existing Line Histories whenever a Change contains a modification to a line which had been added previously.

**Machine-Segmenting:** Machine-Segmenting is the Segmenting of a Project History utilising algorithms. An algorithm for Machine-Segmenting is proposed in the body of this thesis. A

Segmentation produced via Machine-Segmenting should still be corrected manually by a human researcher.

**Modification:** A Modification is a change to a line of source code. A Modification can either be a line deletion (Deletion), a line addition (Addition), a modification of an existing line (Mutation), movement of the line to a different position (Move), re-adding of a previously deleted line (Ghost), or maintaining of a line unchanged (Maintained).

**Project History (also Version History):** A Project History is a view of a student's project as stored in a version control system. It consists of all the Changes the student has produced while working on a project.

**Quantitative Analysis of Spatial Programming in Animation Segments:** A quantitative analysis method which involves analysing the production of student animations Change-by-Change. Each modification of a transformation is examined for correctness, and the ratio of correct to incorrect modifications for the production of each of the animation's steps is calculated and compared to the number of modifications expected if the process were to occur by randomly choosing dimensions.

**SCORE Analyser:** The SCORE Analyser is a software application developed as part of this research project. It serves to facilitate Project History analysis, especially analysis utilising the Change-Coding and Segment-Coding data analysis methods developed as part of this research project.

**SCORE Analyser:** A software application developed as part of this research project. It provides features designed to facilitate **Project History** analysis. These include the ability to modify and compile source code, views which show **Line Histories**, as well as the ability to apply **Machine-Segmenting to Project Histories**. These features are discussed in more detail in Chapter 4.

**Spatial Programming:** Spatial Programming may have different meanings in different contexts. In the body of this dissertation, Spatial Programming is used to refer to programming actions which involve two or three-dimensional space. An example would be drawing a rectangle. Another example would be applying a three-dimensional rotation to an object.

**Segment:** A Segment consists of a set of Changes. In the context of this thesis, Segments are used to identify sets of Changes which deal with the same underlying programming problem. For example, if a student spends a certain number of Changes implementing an animation, these Changes will be grouped as a Segment. Segments are not necessarily contiguous. There may be gaps in the Segment, which means that during these Changes the student was working on a different problem. These gaps may be very large. For example, if the student starts working on a problem at the beginning of the



assignment and completes work on the problem at the end, utilising the same problem-solving approach, then these Changes are part of the same Segment.

**Segmenting:** Segmenting is the activity of assigning a Project History's Changes to Segments. A Change is uniquely assigned to a single Segment. The Segmenting process can either be carried out manually by a researcher examining all of a Project History's Changes, or it can be carried out by the Machine-Segmenting algorithms proposed in the body of this thesis.

**Segment-Coding:** Segment-Coding involves the Coding (see Coding) of each Segment (see Segments) in a Project History.

**Source Code (Code):** Source Code (sometimes referred to simply as Code) is the text which makes up a computer program. Not to be confused with the verb form (to code) which refers to coding in a qualitative analysis context (see Coding).

**Version:** A version in the context of this thesis refers to a version of a source code file. Each time a source code file is changed, a new version of that file is created.

## 9.2 Final Ethics Report



### Human Research Ethics Committee

#### **FINAL REPORT FORM FOR TEACHING OR RESEARCH INVOLVING HUMAN PARTICIPANTS**

##### **\*\*\* Submission Instructions \*\*\***

- If you are a student, this form must be either signed or submitted via email by your supervisor
- If your application was reviewed by a Human Ethics Faculty Sub-Committee or you have received an email reminder from a faculty sub-committee, then you can submit your completed final report form to the relevant faculty sub-Committee.
- For all other Final Reports please submit your completed form to [ethics.secretariat@mq.edu.au](mailto:ethics.secretariat@mq.edu.au) or to the Ethics Secretariat, Research Office, Level 3, Research HUB, Building C5C.

**Handwritten forms will not be accepted.**

*Once your report has been submitted it will be noted by the Committee. **Please note that you will NOT receive any correspondence from the HREC regarding your report.** However, the HREC may undertake an audit at any time without notification.*

Please answer all questions. Please do not delete questions or any part of a question.  
Use lay terms wherever possible.

1. **TITLE of research project or unit code and name:**

Investigation of Learning of Computer Graphics by  
undergraduate students

2. **REFERENCE NO.:**

HE27FEB2009-D06330

3. **CHIEF INVESTIGATOR:**

(If you are submitting a Final Report for an ethics application submitted after 1 January 2010 then the CI must be a staff member/supervisor)

Name:	Maximilian Wittmann
Title:	Mr
Staff No.:	40107884
Student No.:	40107884

Position held:	<b>PhD Candidate</b>
Department & Faculty	<b>Computing / Science</b>
Tel. No.: (work)	<b>02 9889 1305</b>
Email address:	<b>damxam@gmail.com</b>

4. **SUPERVISOR: (For Honours, Post-Graduate and HDR Students:** If you are submitting a Final Report for an application submitted **prior to 2010** please complete supervisor's details)

**\*\* FOR APPLICATIONS SUBMITTED PRIOR TO 2010 where Student is CI \*\***

Name:	<b>Manolya Kavakli</b>
Title:	<b>Associate Professor</b>
Staff No.:	<b>20034812</b>
Department & Faculty	<b>Computing / Science</b>
Tel. No.: (work)	<b>+61-2-9850-9572</b>
Email address:	<b>manolya.kavakli@mq.edu.au</b>

5. Please indicate the current status of the project:

- (a) *Completed on [01/11/2012] (dd/mm/yyyy)*
- (b) *Not completed but the project has run for 5 years from the original approval therefore this is a Final Report for the current ethical approval.*

*I will be submitting a new application for approval to enable the project to continue.*

☐ Yes ☒ No

- (c) *Not commenced or discontinued on [       ] (dd/mm/yyyy)*

Give a brief report below explaining why the project was not commenced or was discontinued:

6. During the course of the project, have you complied with the conditions of approval (i.e. any conditions imposed by the Committee and the standard conditions of approval outlined on your letter of final approval)?

☒ Yes ☐ No

If you have answered NO, explain what conditions have not been met and why:

7. Have any ethical concerns or difficulties arisen during the course of the project? ☐ Yes ☒ No

If you answered **YES**, describe the ethical concerns or difficulties and any adverse effects on participants, and steps taken to deal with these:

8. The following questions relate to the current and future storage arrangements of the research data and the maintenance of its confidentiality and security:

- (a) Will the data be securely stored as listed in the initial Application (Item 6.9)?

☒ Yes ☐ No

If NO, please provide details.

- (b) Will anyone else have access to the data besides those listed in the application (Item 6.10) or in any approved amendments? ☐ Yes ☒ No

If YES, please provide details

- (c) Will you be keeping the data for the minimum 5 year period from the date the research was completed or 5 years from the date of the last publication?

☒ Yes ☐ No

If NO, please provide details.

--

- (c) Are there plans to destroy the data which were not mentioned in the initial application?

☐ Yes ☒ No

If YES, please provide details.


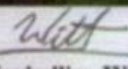
--

9. **CERTIFICATION:**

NB. If you are Honours, Postgraduate or HDR student and you submitted an ethics application prior to 2010, then your report needs to be signed by yourself and your supervisor. (Submission by your supervisor's email will be accepted in lieu of a signature).

I confirm that this project has been conducted in a manner that conforms in all respects with the *National Statement on Ethical Conduct in Human Research* (2007), all other relevant pieces of legislation, codes and guidelines and the procedures set out in the original protocol.

(Guidelines and National Statement available via  
[http://www.research.mq.edu.au/for/researchers/how\\_to\\_obtain\\_ethics\\_approval/human\\_research\\_ethics/policy](http://www.research.mq.edu.au/for/researchers/how_to_obtain_ethics_approval/human_research_ethics/policy))

Supervisor:	Student Investigator (If applicable):
Signed: 	Signed: 
Name: <b>Prof MANOLYA KAVAKLI</b>	Name: <b>Maximilian Wittmann</b>
Date: <b>3.11.2013</b>	Date: <b>30/10/2013</b>

Please note that you will NOT receive any correspondence from the HREC regarding your report.

NB. Students: Form must be signed by your supervisor (or submitted via email from your supervisor)

## 9.3 Methodology

### 9.3.1 Qualitative and Quantitative Research Approaches

There are two fundamental approaches to conducting research. The first is the quantitative research approach. When using a quantitative approach the researcher states hypotheses based on postpositive claims (Creswell, 2012). The researcher then seeks to verify these hypotheses through the application of statistical or mathematical methods to numerical data. Such data can be gathered through a variety of instruments such as through the use of questionnaires or via measurements taken during an experiment (Creswell, 2012; Given, 2008). Since statistical significance often requires a relatively large amount of data, quantitative researchers usually need to conduct their data-gathering on a large scale involving hundreds of participants.

The second is the qualitative research approach. Instead of relying on postpositive claims, the researcher bases knowledge claims on constructivist perspectives. Hypotheses are not validated in the same definitive involving statistical analysis of measurements. Rather, qualitative research is generally open-ended (Creswell, 2013). Rather than verifying an existing hypothesis it can serve to generate new hypotheses or even theory as part of the research process. Since the analysis does not depend on the application of statistical methods, the scale of data-gathering can be much smaller than that required for quantitative approaches. Some qualitative research methods focus on single individuals or small groups of individuals.

Initially, this research project adopted a mixed approach, and indeed the Change-Coding method described later has both qualitative and quantitative aspects. However, as the research project progressed it became clear that given the research questions and aims, a qualitative approach was better suited for the primary analysis method. One reason is that the research questions are general and difficult to answer quantitatively. This is because there is little research in the field to base solid hypotheses on. For this reason, an open-ended approach is better suited to addressing the research questions. As Stern writes, “qualitative methods can be used to explore substantive areas about which little is known or about which much is known to gain novel understandings (Stern, 1980)”. Secondly, it was found that it is exceedingly difficult to convert source code changes to quantitative data, especially when attempting to base this conversion on postpositive claims. Instead, analysis experiences confounded pre-set categorization schemes and the re-negotiating and re-coding of data took on a definite qualitative nature. For this reason, the primary method of source-code analysis, Segment-Coding, is based on a qualitative research method.

### 9.3.2 Comparison of Qualitative Research Methods

Having chosen to follow a qualitative paradigm after initial quantitative analysis using the Change-Coding method proved unable to explore the data to the desired depth, it became important to settle on a qualitative research method which was well-suited to the task of source-code analysis. The following comparison of five qualitative research methods is based on a comparison by Creswell (2012, pp. 53–80).

Narrative research focuses on exploring the ‘story’ of the life of an individual or a small group of individuals. It explores individual experiences, and then re-tells these in a coherent story after ‘re-storying’. This approach is ill-suited for this research project because the focus on exploring individual life stories does not provide a good way of answering the research questions.

Phenomenology focuses on understanding the essence of experience. Creswell summarises the problem type it is meant to address as “Describing essence of lived phenomenon” (Creswell, 2012). This approach is unsuited for two reasons. First, the research questions seek to do more than to describe the phenomenon of how students program Computer Graphics, they seek to uncover problems and challenges. Second and most importantly, phenomenology relies on lengthy interviews in which a skilled interviewer elicits rich data from the interviewee by asking follow-up questions and steering the interview in productive directions when required. It is hard to analogize this to the analysis of source-code.

Creswell (2012) describes ethnographic research as “describing and interpreting a cultural group”. From this definition, it is clear that ethnographic research is not suited as a research method for this project.

Case study research focuses on “Developing in-depth description and analysis of case or multiple cases” (Creswell, 2012). It can be applied to a single event or individual, or several events or individuals. Each student version history could be seen as a case. In applying the case study research method, descriptions of each of these ‘cases’ could be produced and compared for similarities and contrasts. However, regarding each project history as a ‘case’ would be cumbersome, as building up a description of a project history in its entirety is an exceedingly difficult task and requires significant analysis in its own right. In addition, case study research places emphasis on accurate description of cases, whereas this research project aims to do more by providing theoretical insight into the types of problems Computer Graphics students face. Therefore, while Case Study Research might have been a feasible method for this research project, it was not considered optimal, especially when compared to the method discussed next.

The Grounded Theory (GT) method's focus is on "Developing a theory grounded in data from the field" (Creswell, 2012). Its unit of analysis is at the level of studying a process, an action or an interaction involving many individuals. Its unit of analysis is ideally suited to this research project, as the aim is to analyse student programming over the course of an assignment, which is indeed a process. This process can be seen to be composed of many programming actions. GT can be applied at both of these levels. Furthermore, the focus on developing theory matches with one of the aims of this thesis, which is to develop a theoretical understanding of student problems during Computer Graphics programming. For these reasons, Grounded Theory was selected as the method underlying the Segment-Coding method, which is the primary data analysis method developed as part of this research project.

### **9.3.3 Defining Grounded Theory**

Grounded theory can be seen to be composed of four central processes or approaches. These are coding of data (an approach it shares with many other research methods), the Constant Comparative Method (CCM) of analysis, the process of memoing and theoretical sampling and saturation.

#### **9.3.3.1 Types of Grounded Theory analysis**

There are three popular types of Grounded Theory analysis. Two are defined by the authors of the original work on Grounded Theory (Glaser & Strauss, 1967), the method proposed by Strauss and Corbin (Corbin & Strauss, 1990; Strauss & Corbin, 1998) and the method proposed by Glaser (1992, 1998). Strauss' approach is marked by a rigorous application of a coding process consisting of well-defined steps, a rigour which Glaser thought went counter to the free-flowing nature of GT research (Glaser, 1992). Charmaz (2006) proposes a GT approach that is more similar to that of Glaser and is based firmly on constructivist principles, eschewing the formal application of the coding process in favour of a more free-flowing process in which the researcher is an active participant. In particular, Charmaz believes that the researcher's biases and perceptions necessarily shape the production of theory (Charmaz, 2006), whereas Strauss moves more towards a positivist interpretation of GT with his rigorous framework.

The Segment-Coding method proposed as part of this research project could be classed as a data analysis method which follows the GT research method. However, it is more accurate to say that it is a new research method which borrows heavily from the GT research method, but de-emphasizes certain parts of the analysis that were not deemed productive in the context of source-code analysis. The discussion of the Segment-Coding method in this chapter 3.4.3.2 will highlight the ways in which the Segment-Coding method applies a GT approach, and in which ways it differs from traditional GT.



Since the Segment-Coding method does not strictly adhere to any formulation of GT, I will not claim allegiance to any particular version of GT.

### **9.3.3.2 Aim of Grounded Theory**

Before beginning the discussion of the different steps involved in applying GT, it is important to discuss the aim of GT. The aim of GT is to produce theory grounded in data. According to Strauss and Corbin (1998, p. 25) “Theorizing is the act of constructing (we emphasize this verb as well) from data an explanatory scheme that systematically integrates various concepts through statements of relationship. A theory does more than provide understanding or paint a vivid picture. It enables users to explain and predict events, thereby providing guides to action.” This coincides with the aim of developing an understanding of Computer Graphics student problems, since the theoretical basis of such an understanding will involve explaining student programming actions and predicting the difficulties faced by Computer Graphics students in general.

Even Strauss and Corbin (champions of a more rigorous application of process) “emphasize strongly that techniques and procedures, however necessary, are only a means to an end. They are not meant to be used rigidly in a step-by-step fashion. Rather, their intent is to provide researchers with a set of tools that enable them to approach analysis with confidence and to enhance the creativity that is innate, but often undeveloped, in all of us (Strauss & Corbin, 1998, p. 14)”. So this work falls well into the tradition of utilising GT in a manner best applicable to a specific research project, rather than dogmatically following a certain framework to the letter. Strauss & Corbin relate the following as core aspects to any GT: “the procedures of making comparisons, asking questions, and sampling based on evolving theoretical concepts are essential features of the methodology.” (Strauss & Corbin, 1998, p. 46). As will be shown, the work presented in this thesis included the application of all these procedures.

The following paragraphs will discuss the four foundation themes of GT which are coding, the constant comparative method, memoing and theoretical sampling and saturation.

### **9.3.3.3 The Constant Comparative Method**

The Constant Comparative Method (CCM) is at the heart of Grounded Theory. Its application ensures that theory developed via application of GT methods remains ‘grounded’ in the data. The following description is taken from Glaser & Strauss’s work (1967, Chapter 5).

Before the development of CCM and GT, there were two main flavours of analysis for qualitative data. One was the conversion of qualitative data to quantitative data through coding of the data in order to test hypotheses quantitatively. The second was the inspection of data for properties of

theoretical categories, followed by memo-writing which captures the properties of these categories (Glaser & Strauss, 1967, pp. 101–102). The second method does not involve formal and thorough coding since such codes would be rendered insufficient after memo-writing, since the researcher would have found many inadequacies in these initial codes.

Grounded Theory as proposed by Glaser & Strauss (1967) combines the explicit coding step of the first method via the analytic procedure of constant comparison with the style of theory development of the second method (Glaser & Strauss, 1967, p. 102). This means that theory is generated more systematically than allowed by the second approach, since it uses explicit coding and analytic procedures. However, it does not quantify coded data as is the case when applying the first approach since the aim is not provisional testing as is the case with the first approach. Rather, the aim is to discover hypotheses and generate new theory (Glaser & Strauss, 1967, pp. 102–103). It “is concerned with generating and plausibly suggesting (but not provisionally testing) many categories, properties, and hypotheses about general problems” (Glaser & Strauss, 1967, p. 104). The CCM consists of four stages (Glaser & Strauss, 1967, p. 105): comparing incidents applicable to each category, integrating categories and their properties, delimiting the theory, and finally writing the theory. Each of these steps will now be described in detail. While analysis proceeds from one stage to the next, previous stages remain in operation even as the researcher proceeds to work on a later stage, meaning that development of theory occurs at all stages of the process (up to the stage at which the researcher is currently operating) during the entire research project (Glaser & Strauss, 1967, p. 105). This contrasts with the first method, in which codes are fixed early on during the analysis, and the theory hence quickly becomes moribund.

The first is comparing incidents applicable to each category. During coding, every coded incident is compared to other incidents coded in the same category (Glaser & Strauss, 1967, p. 106). This constant comparison starts to generate the theoretical properties of categories (Glaser & Strauss, 1967, p. 106). It will also generate conflicts between the researcher’s previous understanding of a category and newly analysed data. When such conflicts arise, the researcher should record a memo on her ideas, which will help direct the research and relieve the conflict (Glaser & Strauss, 1967, p. 107).

The second step is integrating. Categories and their properties are integrated. As this process occurs, the units of comparison change from incident-to-incident comparisons as incidents are coded to comparisons of incidents to the properties of the category to which the incident is assigned (Glaser & Strauss, 1967, p. 108). This forces the analyst to “make some related theoretical sense of each comparison (Glaser & Strauss, 1967, p. 109).”

The third step is delimiting. Delimiting curbs the voluminosity of analysis task (Glaser & Strauss, 1967, pp. 110–111) by moving from (many) more concrete categories to a (smaller) set of more abstract categories which share uniform properties (Glaser & Strauss, 1967, p. 110). As categories become theoretically saturated, analyst can quickly identify whether incident provides new aspect to theory or not, and can hence better filter out repeated data (Glaser & Strauss, 1967, p. 111).

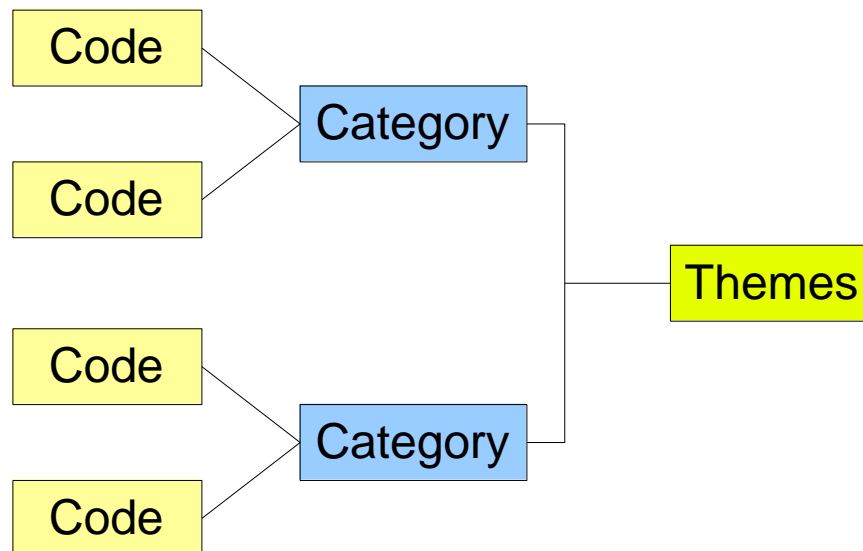
The fourth step is writing theory. At this stage of the analysis, the researcher is equipped with coded data, memos and an emergent theory (Glaser & Strauss, 1967, p. 113). When the analytic framework “forms a systematic substantive theory” (Glaser & Strauss, 1967, p. 113), the analyst collates memos on categories. The memos describe the content behind categories. From this content, the major themes of the theory are developed (Glaser & Strauss, 1967, p. 113).

Because of the application of CCM, the resulting complex theory should correspond closely to data. This is because the researcher was constantly comparing and hence dealing with diversity in data during analysis, recognizing similarities and differences between incidents and properties of categories (Glaser & Strauss, 1967, pp. 113–114).

The next sections describe the coding of data which lies at the heart of the constant comparative method.

#### **9.3.3.4 Coding in Qualitative Research**

A General approach to coding is described in detail by Saldaña (2009). The relationship between codes, categories and themes is shown in . At the lowest level of the coding process codes are applied to data. A code is a short phrase which captures the essential attributes of the underlying data, and this coding scheme is produced dynamically as data is analysed, with codes being added, merged or replaced as analysis proceeds. The coding and re-coding of data proceeds until a functional coding scheme which properly captures the underlying data is developed.



**Figure 112: Relationship between Grounded Theory entities; several codes form a category and analysis of categories leads to the discovery of themes underlying the data (adapted from Saldaña (2009))**

Once the codes reach a level of maturity they are collected in categories. The properties of codes and categories are compared, and relationships and attributes are elicited. Based on this, themes underlying the data are developed. These themes crystallize the theoretical content of codes and categories and the collection of themes is used to build an overarching theory.

### 9.3.3.5 Coding in Grounded Theory

Charmaz explains the role of coding in the application of GT: “Grounded theory coding generates the bones of your analysis. Theoretical integration will assemble these bones into a working skeleton. Thus, coding is more than a beginning; it shapes an analytic frame from which you build the analysis (Charmaz, 2006, pp. 45–46).” She goes on to explain how GT coding is used to generate theory. Coding is described as the link between the data and the emergent theory. As analysis progresses, the codes form the basis of the nascent theory and guide further data-gathering and analysis (Charmaz, 2006, pp. 45–46).

In both the versions of GT laid out by Strauss & Corbin (1998) and Glaser (1978) coding proceeds with Open Coding, followed by Selective Coding. Strauss and Corbin (1998) introduce a further phase of coding, called Axial Coding, which takes place after Open Coding, before and during Selective Coding. Glaser introduces a different style of coding called ‘theoretical coding’ (Glaser, 1978) which follows Selective Coding. According to Glaser theoretical coding makes axial coding unnecessary (Glaser, 1992).

The following paragraphs will describe these coding methods in detail. Open Coding is described as “The analytic process through which concepts are identified and their properties and dimensions are

discovered in data” (Strauss & Corbin, 1998, p. 101). During open coding, data are broken down, examined and constantly compared for similarities and differences. Conceptually similar actions are grouped as categories (Strauss & Corbin, 1998, pp. 102–103). These categories therefore operate at a higher level of abstraction than individual actions. Categorization enables the researcher to remember and think about categories better. It also allows the researcher to better develop the properties of categories. Properties are defined as “Characteristics of a category, the delineation of which defines and gives it meaning” (Strauss & Corbin, 1998, p. 101). Categorization also allows the researcher to further differentiate categories in order to break them down into subcategories (Strauss & Corbin, 1998, p. 114). Open coding can be carried out at different levels of detail, from a word-by-word analysis to a document-by-document analysis, depending on the research context (Strauss & Corbin, 1998, pp. 119–120). Initial codes produced during open coding are provisional, and the researcher should remain open to theoretical possibilities. This may mean creating new codes, dropping existing codes or sub-dividing existing codes. The researcher should follow up codes which fit the data and then gather more data to explore and fill out those codes (Charmaz, 2006, p. 47). Open coding (which Charmaz calls ‘Initial Coding’) helps the researcher to produce a GT analysis which both fits the empirical world and provides an analytic framework which makes relationships between processes and structures visible (Charmaz, 2006, p. 67).

Axial coding is a type of coding introduced by Strauss & Corbin. Glaser actively argues against applying it (Glaser, 1998). Axial coding is designed to reassemble data which was fractured by the open coding process. During axial coding, categories are related to subcategories to form more precise descriptions of observed phenomena (Strauss & Corbin, 1998, p. 124). Strauss & Corbin (1998, pp. 125–126) describe the process as involving the relating of categories to subcategories along the lines of their properties and dimensions, evaluating how “categories crosscut and link” (Strauss & Corbin, 1998, pp. 125–126). A subcategory is a category, but rather than representing phenomenon it describes features of phenomenon (when, where, why, who, how, and with what consequences). This is intended to give the concept greater explanatory power (Strauss & Corbin, 1998, pp. 125–126). Charmaz (2006, p. 61) notes she does not apply the formal axial coding procedures as proposed by Strauss & Corbin, but does develop subcategories of category and shows links between them. The resulting categories and subcategories and links reflect her interpretation of data.

Selective coding is part of both Strauss & Corbin’s and Glaser’s GT method. It is the final coding stage in Strauss & Corbin’s coding framework. During selective coding, major categories are integrated to form a larger theoretical scheme. The outcome of this process is the developed GT theory (Strauss &

Corbin, 1998, p. 143). Charmaz calls this phase “Focused coding” (Charmaz, 2006, p. 57). She suggests using it to explain larger segments of data, using the most significant / frequent codes identified during earlier phases to sift through large amounts of data. Charmaz notes (2006, p. 58) that coding is not a linear process, and hence some observations may involve studying new data or studying data from a new perspective by moving back to open coding.

Glaser (1978) proposes an alternate coding phase which he terms theoretical coding. During this phase, the researcher develops and applies theoretical codes to categories discovered during focused coding. These theoretical codes specify relationships between categories. Glaser (1992) argues this makes axial coding unnecessary because it serves the same function of producing a theory from fractured codes. Charmaz (2006, p. 63) suggests that theoretical codes can be used to “clarify the general context and specific conditions in which a particular phenomenon is evident” which enables the researcher to “learn its temporal and structural orderings and discover participants' strategies for dealing with them”.

GT coding is followed by integration. The researcher integrates findings from memos and diagrams, recognizing relationships between concepts and categories (Strauss & Corbin, 1998, p. 144). Those concepts which have been elevated to categories at this stage are abstractions which represent the experiences of many persons or groups. However, because of the constant comparison which occurs during the application of GT, these categories still have relevance to all cases / instances in the study (Strauss & Corbin, 1998, p. 145). Strauss & Corbin suggest that if the aim is theory generation, the result should be a set of interrelated concepts rather than a listing of general themes (Strauss & Corbin, 1998, p. 145).

#### **9.3.3.6 Theoretical Sampling and Saturation**

As was mentioned during the discussion of coding, coding proceeds through a process of Theoretical Sampling and continues until Theoretical Saturation is reached. Strauss & Corbin define theoretical sampling as “Data gathering driven by concepts derived from the evolving theory and based on the concept of “making comparisons,” whose purpose is to go to places, people, or events that will maximize opportunities to discover variations among concepts and to densify categories in terms of their properties and dimensions” (Strauss & Corbin, 1998, p. 207).

The aim is to “maximize opportunities to compare events, incidents, or happenings to determine how a category varies in terms of its properties and dimensions” (Strauss & Corbin, 1998, p. 202). This is important “when exploring new or uncharted areas because it enables the researcher to choose those avenues of sampling that can bring about the greatest theoretical return” (Strauss &

Corbin, 1998, p. 202). At its core, theoretical sampling is about seeking out data to answer questions which arise during analysis (Strauss & Corbin, 1998, p. 206).

Theoretical sampling is different to other forms of sampling because its aim is to drive theoretical development of the budding theory rather than attempting to represent a population or increasing the generalizability of the theory (Charmaz, 2006, pp. 100–101). Furthermore, because of its role in elaborating and refining theoretical categories, it is essential to have developed categories before commencing theoretical sampling; this sampling then drives the development of the categories (Charmaz, 2006, pp. 102–103). In some cases, the analysis of data produced by theoretical sampling may serve to further delineate and subdivide existing categories based on newly observed properties (Charmaz, 2006, p. 106).

During selective coding, theoretical sampling becomes more discriminate, aimed at integrating categories to form theory (Strauss & Corbin, 1998, p. 211).

However, as is the case with GT in general, it is important not to be dogmatic; Charmaz (2006, p. 107) describes it as a strategy rather than an explicit procedure, which can involve different methods depending on the study context.

The aim of theoretical sampling is to reach the point of theoretical saturation. Strauss & Corbin describe theoretical saturation as the point at which “(a) no new or relevant data seem to emerge regarding a category, (b) the category is well developed in terms of its properties and dimensions demonstrating variation, and (c) the relationships among categories are well established and validated” (Strauss & Corbin, 1998, p. 212). They consider it to be essential to the development of theory since otherwise the resulting theory will be “unevenly developed and lacking density and precision” (Strauss & Corbin, 1998, p. 212).

#### **9.3.3.7 Memoing**

The process of memoing is one of the central processes which occur during the application of GT. Strauss & Corbin describe the memo as a “very specialized types of written records—those that contain the products of analysis or directions for the analyst. They are meant to be analytical and conceptual rather than descriptive” (Strauss & Corbin, 1998, p. 217). The purpose of memo-writing is to keep the researcher grounded in data, and to help the researcher retain awareness of the emerging theory (Strauss & Corbin, 1998, p. 218). It also forces the researcher to engage with, review and update codes early in the research process (Charmaz, 2006, p. 72).

Charmaz (2006, p. 72) describes memos as “giv[ing] you a space and place for making comparisons between data and data, data and codes, codes of data and other codes, codes and category, and category and concept and for articulating conjectures about these comparisons.” Memoing should lead to raising relevant codes to theoretical categories (Charmaz, 2006, p. 92). According to Charmaz this process should eventually lead to narrative memos which describe categories. They define categories, the properties of categories, the conditions under which the category arises or changes, its consequences, and its relationship to other categories (Charmaz, 2006, p. 92).

### **9.3.4 Analysis of Student Perceptions**

#### **9.3.4.1 Introduction**

As part of their work on the third assignment in the 2010 semester, students were asked to complete several reflection questions which they then submitted along with their assignment project. The results of the analysis of student answers will be presented in this chapter.

The answers were analysed after the initial analysis of Project Histories produced in the 2009 semester (results of which are not discussed in detail in this thesis) but before analysis of the 2010 Project Histories on whose analysis the remainder of this thesis focuses.

The questions were designed to elicit students’ subjective experiences of Computer Graphics programming, specifically the problems students had encountered while working on their assignments. Since the reflection questions were designed after initial analysis of Project Histories from the 2009 semester had already been completed, some of the questions were designed to probe potential student problems identified in the 2009 analysis of Project Histories.

Student answers helped guide subsequent analysis as well allowing for a contrasting of perceived student problems with the actual problems that were discovered in students’ Project Histories.

#### **9.3.4.2 Description of Questions**

As part of their work on the third assignment, students were asked to complete a set of questions regarding their experiences with Computer Graphics programming. This section will present the summarised results.

Answers to the reflection question answers were submitted as part of student assignment submission. This section will describe the structure and content of the questions posed.

The first consists of six questions which were to be answered using a seven-point Likert scale. The questions are:



- I am often unsure of what OpenGL is doing in the background
- I would have liked to have more insight into the OpenGL “black box” through an application or set-up which allows me to understand what's going on behind the scenes.
- I often used trial and error to make things work.
- I found OpenGL programs harder to debug than other computer programs I’ve programmed.
- Working on OpenGL problems during practicals and assignments often required me to think spatially.
- I was not always able to completely understand all problems spatially.

The first two questions are designed to detect whether the student is uncomfortable with the OpenGL graphics pipeline and state machine model. The third and fourth questions probe how students engage with programming and debugging their Computer Graphics programs. The fifth and sixth questions query whether students thought they frequently performed spatial thinking during their completion of the assignment (Q5) and whether they felt limited by their insufficient spatial ability (Q6).

The second subsection (Q7-Q9) consists of three open-ended questions to allow students to express their thoughts about different aspects of working on the assignment.

- What do you find is different to programming OpenGL than other programming you have done? What is more difficult? Why?
- In the coding of your assignments, what did you find the most challenging area: mathematics, programming constructs, spatial thinking or understanding what OpenGL was doing behind the scenes? Why?
- What did you feel were the most difficult problems you completed (or failed to complete) while working on the assignment? Why were they difficult?

The first question focuses more on general programming aspects of OpenGL programming and the OpenGL API. The second question asks the student to identify what concepts encountered during the programming of the assignment were particularly hard to learn and apply. The third question asks which problems were the most difficult to complete while working on the assignment implementation.

The third subsection consists of a single rank order question. Students are asked to rank six problem types in order of difficulty, with 1 being the most difficult and 6 being the least difficult. The six problem types are:

- Transformations
- Modelling primitives

- Viewing and Projection
- Lighting and Shading
- Interaction (GUI, event-driven programming)
- Animation and Time-driven behaviour

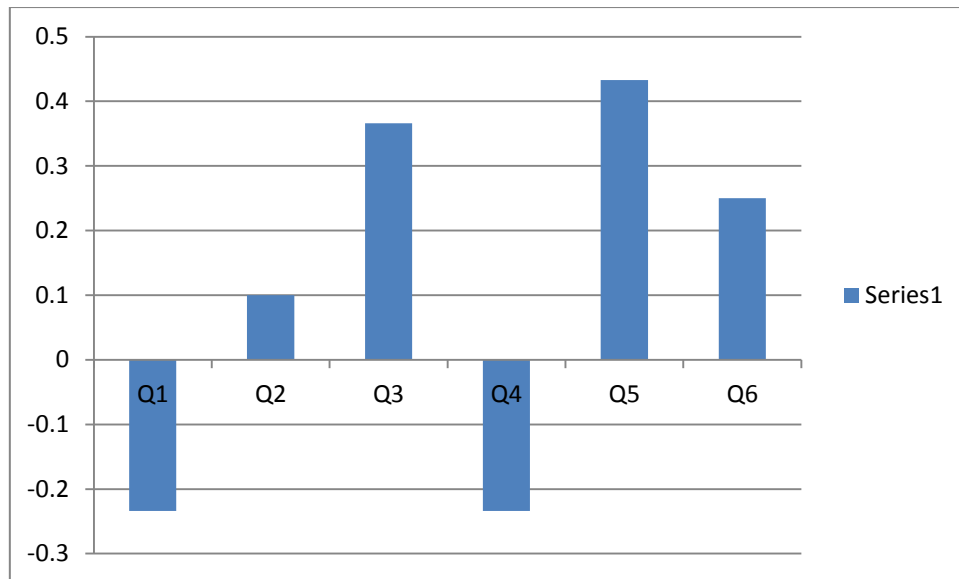
Fifteen students agreed to participate in the research project and also correctly answered the research questions. One student's answers to the rank order question had to be excluded as he answered the question incorrectly, utilising the same rank multiple times.

### 9.3.4.3 Answers to Likert Questions (Q1-Q6)

Table 39 presents the results for the Likert question Q1-Q6. The final row summarises the result across students by dividing the number of results of somewhat agree, agree and strongly agree by the total number of results. Figure 113 shows those same results as bar charts which show whether the majority of answers were generally in agreement or generally in disagreement.

**Table 39: Result of Likert questions; +++ agree strongly, ++ agree, + somewhat agree, / neutral, - somewhat disagree, -- disagree, --- strongly disagree**

Student	Q1	Q2	Q3	Q4	Q5	Q6
S8	+	+	+	++	+++	++
S9	/	+	+	--	++	+
Ida	-	++	+++	-	+	--
S10	-	-	+	-	+	/
Christopher	--	+++	---	+	-	---
S11	-	-	+++	+	+++	+
S12	+	-	++	++	+	+++
S13	+	++	/	-	++	-
S14	++	++	+++	-	++	+
S15	-	+	+	-	+	--
Thomas	--	--	+	--	+	+
John	--	--	+++	/	++	+
S16	--	--	+++	/	++	+
S7	/	+	++	--	+++	++
Michael	/	+	++	/	++	++
Agree / Total	4/15=0.266	9/15=0.6	13/15=0.866	4/15=0.266	14/15=0.933	10/15=0.75



**Figure 113: Graph of the Results**

The results show that most students do not agree with Q1 (I am often unsure of what OpenGL is doing in the background) and Q4 (I found OpenGL programs harder to debug than other computer programs I've programmed).

Responses are mixed on Q2 (I would have liked to have more insight into the OpenGL "black box" through an application or set-up which allows me to understand what's going on behind the scenes.).

Most students agree with Q6 (I was not always able to completely understand all problems spatially.) and almost all students agree with Q3 (I often used trial and error to make things work) and Q5 (Working on OpenGL problems during practicals and assignments often required me to think spatially).

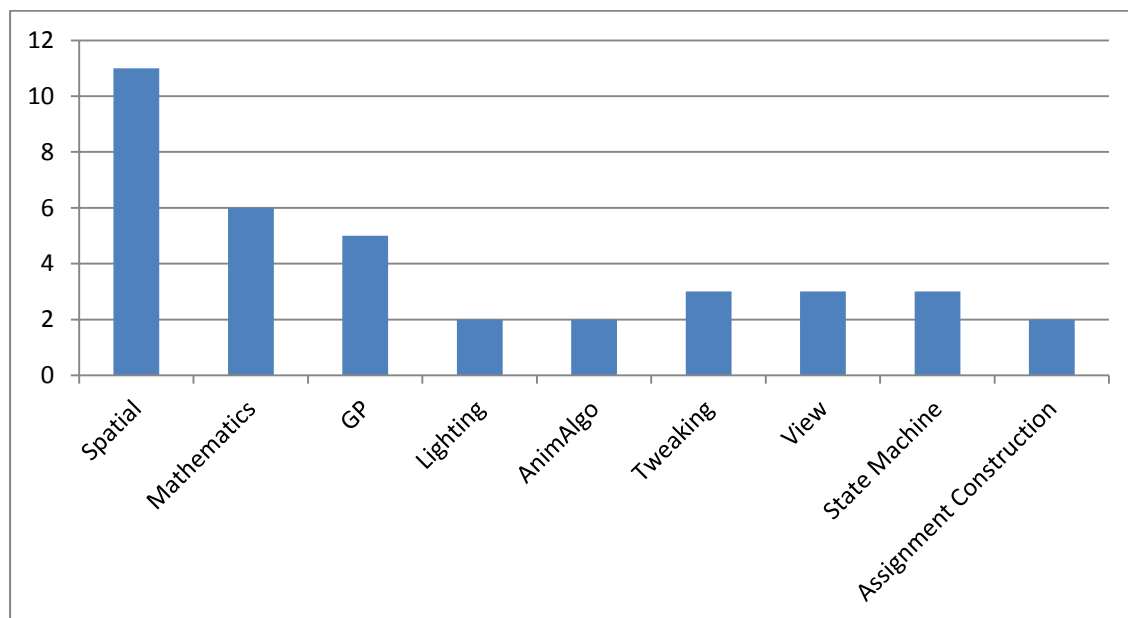
#### **9.3.4.4 Answers to Open-Ended Questions (Q7-Q9)**

To analyse open-ended questions, each answer is coded as relating to one of the following categories: Spatial, Mathematical, General Programming, State Machine, View, Spatial Tweaking, Animation Algorithm, Lighting or Assignment Construction. Most of these categories correspond to classification categories used in the coding of Project Histories. Answers that fall into more than one category were broken up into multiple answers, with each part being assigned to the appropriate category.

For example, the answer to Q8: "Transformations and viewing mainly because I was unable to visualise what the commands would achieve in my head. I also found debugging extremely difficult

due to the lack of feedback and debugging capabilities of eclipse/opengl/glut.” is broken down into two parts. The first part “Transformations and viewing mainly because I was unable to visualise what the commands would achieve in my head.” is categorised as referring to spatial ability and reasoning, whereas the second part “I also found debugging extremely difficult due to the lack of feedback and debugging capabilities of eclipse/opengl/glut.” is categorised as related to general programming (general Computer Science problems, problems with the OpenGL API, etc).

The total number of answers falling into each of the categories is shown in Figure 114. For each student, a category brought up in response to more than one question was counted only once in the final result count since students often repeated part of their answer to one question when answering the next question. The full set of answers can be found in the appendix, Section 9.3.5.1.



**Figure 114: Graph of the categorisation of open-ended questions**

Most students (11 of 15 in answers A1-A17) expressed some difficulty with spatial reasoning. The chief reason given was difficulty relating to spatial visualization (see A4, A5, A7, A10, A12, A13, A14, A15, A16 in the appendix, Section 9.3.5.1). For example, Ida (A4) wrote “Spatial thinking was the most challenging. I personally believe I am capable in terms of visualising, but it was still quite demanding of the imagination to create scenes without trial and error”. Student 14 (A7) indicates his difficulty with “Transformations and viewing mainly because I was unable to visualise what the commands would achieve in my head”. Student 7 (A14) suggested that his difficulty in visualization led to him utilising trial and error to implement transformations (A15): “I also still can not [sic] visualise transformations well and often resort to trail [sic] and error”.

Several students also indicated difficulty with developing a working model of hierarchical assembly utilising composite transformations (A6, A9, and A17). Student 15 (A9) writes: “I was struggling with manipulating the matrix stack to exactly how I wanted. To be honest the theory is explained in a straight forward manner and backed up by good examples in the OpenGL programming guide. However applying it proved problematic”. This suggests that while material explaining these concepts was available, he was unable to develop these into a correct spatial understanding of the concepts being explained.

Another point brought up by several students (A1, A2, A8) related to the difficulty of understanding spatial actions in terms of the mathematical constructs implementing them. Student 8 (A1) wrote: “Thinking in 3D space, especially trying to calculate a point in 3D space is very hard”. Two students point out the difficulty of understanding OpenGL space in terms of local and global coordinate systems (A3, A11). Student 9 (A3) said that: “Rotation and movement was also difficult due to the fact there were a number of things needed considering (moving the camera around GLGuy, moving the world, moving GLGuy)”. It should be noted that the students who produced the best assignment submissions all noted difficulty with spatial thinking (A3, A4, A10, A12) suggesting that problems related to spatial programming are not limited to students who performed poorly in the unit. In fact, the ‘best’ students produced the clearest and most verbose descriptions of problems relating to spatial thinking.

Many students (6/15) also indicated struggling with the mathematics involved in completing the assignment (A18-A25). Student 16 (A25) states that “Mathematics is the most challenging area” because he “calculated each viewing angle based on trigonometry equations”. Christopher also indirectly expressed difficulty with trigonometric calculations (A23) as he mentioned polar coordinates (based on trigonometric calculation) as “slightly challenging”. Student 8 (A20) brought up his difficulty with debugging problems with his mathematical model for camera movement: “Not sure why the maths didn't work out, it's almost there but fails to perfectly sync up the camera with the head, unfortunately I couldn't figure out what exactly was causing the weird rotation when turning with the head and/or torso tilted. Wasn't quite sure how to help diagnose the problem, tried heaps of trial and error tests but nothing improved it”.

The third-most common topic brought up by students is General-Programming -related issues, with (5/15) students expressing problems with non-Computer-Graphics related programming. Several students (A27, A29, and A30) bring up the use of C++ as problematic, with Student 12 (A27) writing that: “I used to program with Java a lot in past one year. So I think C++ is too old and not as effective as using Java”. Two students (A26, A33) found OpenGL difficult to learn. Student 7 (A33) writes that

“The most difficult aspect is learning OpenGL itself and it took me long time to study it”. Student 14 found himself limited by the OpenGL API and state machine model which he found too restrictive (A31) and the Eclipse IDE (A32) which he found did not offer sufficient debugging facilities.

Three students discussed problems with too much time spent on spatially tweaking transformations (A40-A42), three students mentioned the difficulty of implementing views (A37-A39), and three students indicated they had had difficulty dealing with some aspect of the OpenGL state machine model (A34-A36).

Two students mentioned issues relating to the implementation of Lighting (A45-A46), two students discussed difficulty in implementing a good animation algorithm (A43-A44) and two students discussed issues with the assignment specification (A47-A48), stating that the assignment was time-intensive to complete.

#### **9.3.4.5 Answers to the Rank Order Question (Q10)**

Results from rank order questions are presented in

Table 40. One student's (Student 13) answer had to be excluded because he assigned the same rank to multiple categories. Items were asked to rank items in the order of difficulty, with the first item (lowest-ranked) being the most difficult. Calculating the sum of rankings shows Transformations to be the lowest-rated category overall, with Viewing and Lighting receiving the next-lowest ratings. Modelling of primitives, event-driven and time-driven programming were all rated high (easy) overall. Unfortunately usage of the mean or sum of responses to ranked questions is not generally considered an acceptable analysis technique since the distance between different ranks is undefined and it is hence not possible to compare the distance (difference in difficulty) between the ranks 1 and 2 to the distance between the ranks 2 and 3, for example.

**Table 40: Summary of results for the rank order question**

Student	Transformations	Modelling Primitives	Viewing / Projection	Lighting / Shading	Interaction (GUI)	Animation / Time-Driven Behaviour
S8	5	6	2	1	4	3
S9	3	5	1	2	6	4
Ida	1	6	2	4	3	5
S10	1	2	3	6	4	5
Christopher	1	3	6	5	2	4
S11	2	1	3	4	5	6
S12	1	3	4	2	5	6
S14	3	6	2	1	5	4
S15	2	1	5	4	3	6
Thomas	1	2	6	5	4	3
John	2	5	3	4	1	6
S16	4	5	2	1	6	3
S7	4	6	1	2	5	3
Michael	2	6	4	5	3	1
Number of answers that rank 1/2/3/4/5/6 And ratio of (1/2) to (5/6)	5/4/2/2/1/0 (9:1)	2/2/2/0/3/5 (4:8)	2/4/3/2/1/2 (6 : 3)	3/3/0/4/3/1 (6:4)	1/1/2/3/4/2 (2:6)	1/0/4/3/2/4 (1:6)
Total	32 (1)	57 (5)	44 (2)	46 (3)	56 (4)	59 (6)

Instead, comparison will be based on the number of students that rank the problem as difficult (rank of 1 or 2) compared to the number of students who rank the problem as easy (rank 5 or 6). The hardest problem, rated as difficult by 9 students and as easy by only a single student, is *Transformations*. Viewing and Lighting are the next-most difficult problems; both are rated as difficult by 6 students. The *Modelling Primitives* category is considered easy by the largest number of students (8) but is also considered difficult by four students. User interface interaction and time-based animation are both considered easy by six students and difficult by 2 and 1 student respectively.



The results make it clear that students considered the implementation of Transformations to be the most challenging problem area, and that several students also struggled with the implementation of Views and Lighting.

#### 9.3.4.6 Discussion

Evaluation of Likert questions showed that most students agreed that Computer Graphics programming requires spatial thinking, that it was sometimes difficult to fully understand problems spatially, and that they sometimes used trial-and-error approaches to complete tasks.

In their answer to open-ended questions, most students discussed issues with spatial programming, much of it focussed on spatial thinking. Mathematics and General Programming were also discussed by several students.

The rank order question showed most students see transformation as very challenging, with many students also ranking lighting and view problems as challenging.

Students indicated difficulty with spatial programming on all question types. The open-ended questions in particular showed the extent to which many students including those who performed best on evaluation tasks struggled with spatial programming problems.

Viewing was also discussed by three students in the open-ended questions, and was ranked second-highest in difficulty. Several of the open-ended answers relating to 'Spatial' programming could also be seen to include the implementation of Views.

'Lighting' was ranked as the third-most difficult problem in the rank order question, and two students discussed issues around Lighting in the open-ended section. Given the simplicity of the Lighting task for Assignment 3 this result could be considered surprising.

A final issue identified by open-ended questions is the negative attitude three students had towards C++ as a programming language. Several students experienced issues in dealing with C++ syntax, especially in carrying out object-oriented programming for the first assignment.

#### 9.3.4.7 Conclusion

The evaluation of reflection questions students submitted along with their third assignment did help shed some light on student problems. Spatial programming in particular was identified as challenging by many students in all question types.

The results go some way toward confirming hypothesis **RQ2** "Spatial programming is difficult to perform and poses a significant challenge to students". The open-ended questions also provided

some answers to the research question **RQ1** “What kinds of problems do students learning Computer Graphics Programming experience?”, with students identifying problems with spatial programming, mathematics, and with the C++ programming language.

However, the responses do not provide detailed insight into the nature of student problems. Even though many students gave long answers, they did not describe the precise nature of problems. The answers do not show how students addressed these problems during the implementation of their assignments. Furthermore given that students most likely answered the questions after completing their assignments, they are unlikely to provide a detailed and accurate description of their problem-solving approach. Instead, their recollection is likely to be tainted by various cognitive distortions. For example, they are more likely to be recalling issues faced during the latter phases of the implementation than in the earlier phases.

To better address these questions instead of analysing student recollection of their problem-solving work, this thesis will examine actual student problem-solving by examining all source code modifications carried out by students during their work on assignment assessment tasks. The SCORE plug-in with which these modifications were stored as part of Project Histories as well as the tool used to analyse these Project Histories are discussed in the next chapter.

## 9.3.5 Reflection Questions

### 9.3.5.1 Student Open-Ended Question Answers

#### 9.3.5.1.1 Spatial

Question ID	Student / Question	Category	Statement
A1	Student8 Q7	Spatial / Mathematics	Thinking in 3D space, especially trying to calculate a point in 3D space is very hard!
A2	Student9 Q7	Spatial / Mathematics	I personally found understanding some of approaches OpenGL like view volumes and matrices difficult.
A3	Student9 Q9	Spatial	Rotation and movement was also difficult due to the fact there were a number of things needed considering (moving the camera around GLGuy, moving the world, moving GLGuy).
A4	Ida Q8	Spatial	Spatial thinking was the most challenging. I personally believe I am capable in terms of visualising, but it was still quite demanding of the imagination to create scenes without trial and error. With the second assignment in particular, there were lots of variables involved (i.e. rotations and translations etc. In a specific order.), so that it was not just purely a spatial problem, but also a logical one.
A5	Student12 Q8	GP / Spatial	Programming construct and spatial thinking. I think the range of these knowledge are too big for me. I can't catch easily up like everybody else does.
A6	Student13 Q9	Spatial	It was just lengthy hardest part i found was to pin point the pivot point of a limb.
A7	Student14 Q8	Spatial	Transformations and viewing mainly because I was unable to visualise what the commands would achieve in my head.
A8	Student15 Q8	Spatial	Manipulating the matrix stack was quite annoying. Although I had the theory in my head, when applying it I often encountered problems.
A9	Student15 Q9	Spatial	I was struggling with manipulating the matrix stack to exactly how I wanted. To be honest the theory is explained in a straight forward manner and backed up by good examples in the OpenGL programming guide. However applying it proved problematic.
A10	Thomas Q8	Spatial	Spatial thinking. It is simple to think in 3D for one or two rotations, but after stacking five or so, it becomes very difficult to envision which rotation would work, and which would not. The other issues seemed to fall into place after a few trials.

A11	Thomas Q9	Spatial	One of the most difficult problems was getting the guy to move freely. I used trigonometry to compute offsets and movement values based on direction, but I had to take care that these calculations were not stacking on top of each other. Effectively I had to move the guy to the origin, compute rotation, compute distance and then move him back.
A12	John Q8	Spatial	Spatial thinking is the hardest for me. While doing the animations I had many weird situations where I could not readily tell why some rotations were not working.
A13	John Q9	Spatial	Animations were certainly the hardest and most time-consuming. I find it very hard to imagine all the transformations being applied to the model simultaneously
A14	Student7 Q8	Spatial	Spatial Thinking was also hard and required a lot of trial and error to get right.
A15	Student7 Q9	Spatial	I also still can not visualise transformations well and often resort to trail and error.
A16	Michael Q8	Spatial	spatial thinking especially when you implement transformations, if there were many transformations applied on a certain object, the orders of instructions are really confusing.
A17	Michael Q9	Spatial	Making a change to be applied on the whole scene was hard. For example, to rotate a certain part of Guy such as UpperArm, We have to also rotate the LowerArm and the palm without losing their information(transformation applied on them before) and that was just too hard and complicated for me.

### 9.3.5.1.2 Mathematics

Question ID	Student / Question	Category	Statement
A18	Student8 Q7	Spatial / Mathematics	Thinking in 3D space, especially trying to calculate a point in 3D space is very hard!
A19	Student8 Q8	Mathematics	fine it was hard trying to calculate a point in space based on the orientation on a line of limbs, ie body-head, even though the maths should have been fine the result in OpenGL was a bit of a mixed bag and appears to be osolating when you turn.
A20	Student8 Q9	Mathematics	Not sure why the maths didn't work out, it's almost there but fails

			to perfectly sync up the camera with the head, unfortunately I couldn't figure out what exactly was causing the weird rotation when turning with the head and/or torso tilted. Wasn't quite sure how to help diagnose the problem, tried heaps of trial and error tests but nothing improved it.
A21	Student9 Q7	Spatial / Mathematics	I personally found understanding some of approaches OpenGL like view volumes and matrices difficult.
A22	Student10 Q8	Mathematics	Some of the maths.
A23	Christopher Q8	Mathematics	above the mathematics and computation of the polar coordinates to enable the first-person and third person views to change as the character rotated was slightly challenging.
A24	Student11 Q7	Mathematics	Somehow we need to include some calculation since our tasks are basically work with animation, resolution, pixel, etc.
A25	Student16 Q7	Mathematics	Mathematics is the most challenging area. When I was working on camera views, I calculated each viewing angle based on trigonometry equations.

#### 9.3.5.1.3 GP

Question ID	Student / Question	Category	Statement
A26	Student10 Q8	GP	Assignment 1 was the amount of code that had to be written and the steep learning curve with OpenGL.
A27	Student12 Q7	GP	I used to program with Java a lot in past one year. So I think C++ is too old and not as effective as using Java. Sometimes some actions like declaring object makes me confused. I'm also not familiar with the inheritance used in OpenGL.
A28	Student12 Q8	GP / Spatial	Programming construct and spatial thinking. I think the range of these knowledge are too big for me. I can't catch easily up like everybody else does.
A29	Student12 Q9	GP	I have troubles mostly on the C++ programming. I found OpenGL programming itself not too difficult compared to the C++ programming. I'm really really having trouble in solving and deciding what to do while programming in C++. I don't know why, I just can't get what the codes do.

A30	Student13 Q7	GP	I have programmed most of my acadmic year OOP languages and opengl is somwat procedural language just the combustion of two some time annoying.
A31	Student14 Q7	GP	The heavy reliance on the Display function and not being able to execute different Display function for different commands or scenarios.  The inability to encompass a Display function into a custom made structure or the compartmentalising or grouping of display objects in the display function.
A32	Student14 Q8	GP	I also found debugging extremely difficult due to the lack of feedback and debugging capabilities of eclipse/opengl/glut.
A33	Student7 Q8	GP	The most difficult aspect is learning OpenGL itself and it took me long time to study it.

#### 9.3.5.1.4 State Machine

Question ID	Student / Question	Category	Statement
A34	Student8 Q7	Lighting / State Machine	Also it's often unknown why something doesn't work in OpenGL eg global ambient lighting wouldn't work yet the code appeared to be fine and lights positioned in space seems to cling to the perspective camera yet stayed put in the orthogonal views.
A35	Student9 Q8	State Machine	Understanding what OpenGL was doing behind the scenes because I don't still fully understand what it's actually doing behind the scenes.
A36	Student10 Q9	State Machine	Even though double buffering was turned on the screen still flickered.

#### 9.3.5.1.5 View

Question ID	Student / Question	Category	Statement
A37	Ida Q9	View	The camera views were also conceptually challenging but rewarding, particularly the third person view, since we had to consider the position of the guy in the rotation.
A38	Christopher Q9	View	If I was to choose something it defiantly would be handling the rotation of objects with respect to camera views.
A39	Student9 Q7	Spatial Mathematics / View	I personally found understanding some of approaches OpenGL like view volumes and matrices difficult.

### 9.3.5.1.6 Tweaking

Question ID	Student / Question	Category	Statement
A40	Ida Q9	Tweaking	Implementing the animations was the most tedious as it involved a lot of tweaking and trial and error. Although it was no difficult conceptually, it was time consuming, but still enjoyable to create.
A41	Student10 Q9	Tweaking	The animations because they contained a lot of transformations to carry out one simple animation. The transformations need to be timed appropriately and was some what difficult and time consuming
A42	Student16 Q9	Tweaking	Animation is easy but boring, because I need to spend time to hardcode on each frame to achieve a smooth action.

### 9.3.5.1.7 Animation Algorithm

Question ID	Student / Question	Category	Statement
A43	Student9 Q9	Animation Algo	Animation was difficult because it was difficult to figure out a good implementation that is also elegant.
A44	Student14 Q9	Animation Algo	<p>Creation of animations into an easy to use object oriented format that could be reused for any rotational variation applied to the Guy object.</p> <p>Initially I attempted to build in a perform function which would accept an Animation object and perform it. However calling glutPostRedisplay from the Animation.cpp would only draw the animation on top of the existing pre-animation display. Leaving a trail of the whole animation drawn until the animation completed.</p> <p>I then moved the Animation handlers to the Main.cpp and made sure the Display function used the Animation's frames when Redisplaying</p>

### 9.3.5.1.8 Lighting

Question ID	Student / Question	Category	Statement
A45	Student8 Q7	Lighting / State Machine	Also it's often unknown why something doesn't work in OpenGL eg global ambient lighting wouldn't work yet the code appeared to be fine and lights positioned in space seems to cling to the perspective camera yet stayed put in the orthogonal views.
A46	Student7	Lighting	I still do not understand how OpenGL works with lighting. I found the

	Q9		text book too complicated and could not find any good examples
--	----	--	--

#### **9.3.5.1.9 Assignment Concerns**

Question ID	Student / Question	Category	Statement
A47	Christopher Q8	Assignment Construction	The most difficult aspect of this assignment for me was sitting in front of a computer screen for the 10 hours or so it took to write and test the application
A48	Student11 Q9	Assignment Construction	Additional tasks that require self practice and more logical thinking. With so many additional tasks given, I found it hard to complete the whole assignment.

### **9.3.6 Description of Assignment Specifications**

This section describes the assignment specifications that were used for Assignment 1 and Assignment 3 that were used for the Comp330 unit in 2010. Assignment data from that year meeting the goals of these assignment specifications were analysed in this research project.

#### **9.3.6.1 Assignment 1**

##### **9.3.6.1.1 Description**

Assignment 1 was intended to ensure that students were familiar with the OpenGL API and the OpenGL state machine model. It was also intended to familiarise students with event-driven programming utilising a graphical user interface, two-dimensional drawing using OpenGL and storage of dynamically created content in data structures.

The assignment involved the creation of a simple tool for the creation of architectural plans. Students were to implement user-interface functionality such as buttons and a drop-down menu; these user interface objects were to activate the different kinds of functionality to be implemented according to the assignment specification.

The tool was to allow the placing of simple objects (square ‘furniture’, round ‘flowerpots’ and doors) as well as rooms (consisting of an arbitrary number of corner points); these objects were to persist between redraw events, requiring students to produce data structures to store the objects for display each time the display function is called. Students were also required to implement



functionality allowing these objects to be moved or resized, which required event-driven actions to modify the data stored in object data structures.

The most complex task of the assignment was the implementation of a parenting mechanism which would parent an object to a different object and drawing a connector between the two objects. After this, moving the parent object was to also move the child object. A rotate function was to rotate a child object around its parent; this would require students to use trigonometric functions, and to consider spatial relationships between different objects.

### **9.3.6.1.2 Tasks**

The following is a list of the assignment specification's tasks, as well as a brief explanation of the intended role of the task in the teaching process:

#### **1. Draw a background grid (2 marks)**

Task: To draw a simple grid to allow the user to more precisely place objects.

Goal: To introduce students to the basics of drawing graphics primitives using OpenGL.

#### **2. Create functional buttons (4 marks)**

Task: To display buttons on the drawing surface and to produce simple icons indicating functionality for each button. Buttons should trigger actions if they receive a mouse click.

Goal: To familiarise students with the OpenGL input handling syntax and model

#### **3. Display Status window (2 marks)**

Task: To draw a status window displaying the name of the currently active action.

Goal: To familiarise students with GLUT facilities for displaying text.

#### **4. Create a drop-down menu (1 marks)**

Task: To implement a drop-down menu, either from scratch or using the built-in OpenGL drop-down menu.

Goal: To give the student more practice with event-driven programming.

#### **5. Change stroke style for lines (size/pattern) (2 marks)**

Task: To enable the user to change the stroke style and width of lines.

Goal: To familiarise students with the operation of the OpenGL state machine model, specifically the glEnable/glDisable methods activating and deactivating machine model settings.

## 6. User Interface Clipping (2 marks)

Task: To stop items from being drawn over the user interface.

Goal: To teach students to perform conditional tests while executing event-driven actions.

## b) Object Creation and Persistence (19 marks)

### 1. Object Persistence (6 marks)

Task: To create data structures for storage of objects created via the GUI to ensure that dynamically created objects persist between redraw events.

Goal: To teach students the difference between items being drawn on screen and items being programmatically stored, and how to create persistent objects through the use of data structures.

### 2. Adding Objects (8 marks)

Task: To enable the user to add objects to the application's drawing area.

Goal: To have students produce data structures for storage of object coordinate data.

### 3. Adding Rooms (2 marks)

Task: To allow the user to add rooms consisting of an arbitrary amount of 'corner' points between which walls are drawn.

Goal: To force students to consider different types of objects such as objects consisting of an arbitrary number of points, and to have students produce models for storing such objects.

To have students implement complex event-driven commands that do not complete during a single mouse drag or click action.

## c) Modifying Objects (23 marks)

### 1. Object Selection (5 marks)

Task: To enable programmatic event-driven selection of objects and display a bounding box to show the current selected object.

Goal: To have students implement an event-driven way of selecting objects.

### 2. Changing Object colour (3 marks)

Task: To provide functionality which allows the user to change the colour of objects.

Goal: To make students consider the different properties of objects that may need to be stored in graphical applications.

### 3. Moving Objects (3 marks)

Task: To enable the user to move the currently selected object using the mouse.

Goal: ED, modifying objects in space

### 4. Parenting Objects (8 marks)

Task: To provide functionality to declare one object the 'child' object of another object.

Goal: To have students implement hierarchical relationships between objects

### 5. Parent-Child Move (3 marks)

Task: To ensure that child objects retain their same relative position to the parent object when the parent object is moved.

Goal: To have students understand in what ways event-driven actions on one object may affect other objects.

### 6. Parent-Child Rotate (4 marks)

Task: To allow students to rotate a child object around its parent object using the mouse.

Goal: To have students utilise trigonometric functions to create simple spatial transformations.

To have students experiment with different ways of utilising mouse input to determine the amount of rotation (distance between mouse up and mouse down or similar) and to consider how to map mouse input to actions when the mapping is not direct (not moving from point A to point B).

## 9.3.6.2 Assignment 3

### 9.3.6.2.1 Description

Assignment 3 was the final assignment. It was intended to ensure that students developed an understanding of three-dimensional transformations and three-dimensional space, including the ability to composite multiple three-dimensional transformations to create hierarchical models and the ability to manipulate the OpenGL 'camera' to create different Views.

The assignment involved the assembly of an avatar using OpenGL transformation commands. The assembly was to use composite transforms to enable hierarchical movement of limbs. For example, a rotation of the lower limb should also rotate all other limbs attached to it. Furthermore, limbs should rotate about their natural 'joints', the point at which they connected to the 'parent' limb. Students were then expected to implement a time-driven animation algorithm which they would use to implement three animations. Students were also to implement three views: a top-down view, a third-person view and a first-person view. The third-person view was to be able to orbit around the

avatar, and to zoom away from and toward the avatar. The avatar was to be moveable via keystrokes, and the avatar was to move in the direction in which it was facing which required the use of polar coordinates. Students were also to implement simple OpenGL lighting.

#### **9.3.6.2.2 Tasks:**

The following tasks were set in the assignment specification:

##### **a. Avatar Tasks**

###### **1. Avatar Assembly (20 marks)**

**Task:** To assemble the avatar's limbs (which in the initial skeleton program were all piled at the 0,0,0 coordinate) into a working avatar using hierarchically composited transformation commands. Limbs were both to be affected by the movement of their parent limbs and to rotate around the point at which they connected to their parent limb.

**Goal:** To understand the compositing of three-dimensional transformations and how it can be used to produce hierarchical models.

###### **2. Avatar Animations (20 marks)**

**Task:** To produce a time-based animation algorithm (using either `glutIdleFunc` or `glutTimerFunc`), and to use this algorithm to create three simple animations for the avatar, one of which was to be a walking animation including movement of both arms and legs, one a pickup animation and the final animation was left up to the student.

**Goal:** To ensure students have an understanding of time-based behaviour in contrast to user-driven behaviour. To develop a deeper understanding of the compositing of transformations.

###### **3. Avatar Movement (5 marks)**

**Task:** To enable movement of the avatar in the direction it is currently facing.

**Goal:** To teach students about spherical coordinates, and about avatar movement relative to camera movement relative to world movement.

##### **b. Viewing/Camera**

###### **1. Viewing (15 marks)**

**Task:** To create a first-person, third-person and top-down camera View, all of which should follow the avatar. The third-person view should also be able to orbit around the avatar, and to zoom towards and away from the avatar.

**Goal:** To develop an understanding of the virtual camera and of OpenGL Viewing. To develop an understanding of the concepts of 'world' and 'camera' and 'global' and 'local' coordinate space.

### c. Quality

#### 1. Quality:

Task: To produce more complex animations.

Goal: To encourage students to produce more complex animations to further their experience with compound transformations.

### d. Peripheral Tasks

#### 1. Lighting (5 marks)

Task: To implement basic OpenGL lighting.

Goal: To familiarise students with the basics of OpenGL lighting and materials, including the state machine commands enabling lighting and the types of light (Ambient, Diffuse, Specular).

#### 2. Simple User Interface (5 marks)

Task: To implement a simple User Interface using buttons which allows animations to be executed

Goal: To remind students of event-driven concepts learned in the first assignment.

#### 3. (Bonus) Interactivity (5 Bonus marks)

Task: To enable the avatar to pick up an object as part of the pickup animation.

Goal: To have students develop an understanding of object interaction in Computer Graphics programs based on time-driven behaviour.

## 9.4 Literature Review

### 9.4.1 Factors and Facets of Spatial ability

Structural descriptions of spatial ability range from monolithic to structures comprised of hierarchies of sub-factors. For an overview of the development of structural explanations, see Mohler's review of spatial ability research (2009). A number of the most influential proposed structures will be presented to give some insight into the consensus and disagreement that exists within the spatial ability research community; however, this research is presented with the proviso that the literature does not lie within the researcher's area of expertise.

#### 9.4.1.1 Factor Analysis

Most of the research investigating the structure of spatial ability utilises *factor analysis*. Factor analysis is a statistical method which utilises variability of observed variables to discover underlying unobserved variables (factors). Observed variables can then be modelled as linear combinations of these factors. In practice, most studies aimed at describing the structure of spatial ability classify a large number of spatial ability tests according to several dimensions. These tests are then used in an experimental setting, and results are evaluated in terms of the spatial test classification scheme and

test performance. A factor analysis of such results then produces factors which are supposed to describe spatial ability.

#### 9.4.1.2 Visualization + Orientation

In 1979, M. G. Gee (McGee, 1979) reviewed the results from many factor-analytical studies and identified two factors that many of the proposed structures had in common. Spatial Ability was split into two sub-abilities. *Spatial Visualization* describes the ability to rotate or otherwise manipulate (three-dimensional) objects in place, whereas *Spatial Orientation* describes the ability to be able to compare spatial patterns with one another, to understand spatial relationships between objects and to be able to mentally take different perspectives and orientation. Kozhevnikov and Hegarty (Kozhevnikov & Hegarty, 2001) proposed a similar breakdown of spatial ability consisting of a factor called *object-manipulation spatial ability* (analogous to *spatial visualization*) and *spatial orientation ability*.

#### 9.4.1.3 Relations + Visualization

J.W Pellegrino et al (Pellegrino et al., 1984) broke spatial ability down along different dimensions. In their model, *Spatial Relations* involved the ability to recognise a relatively simple object after rotation/transformation (to be applied in tasks that involve pairs of stimuli to be compared); high ability in this factor is characterised by fast performance where the stimulus presented is comparatively simple. *Spatial Visualization* involves complex manipulation of complex objects such as folding of objects which involves the movement of internal parts of the stimulus presented; high ability in this factor is characterised by the ability to produce the correct answer for this more complex stimulus. In short, according to J.W Pellegrino et al spatial ability is decomposed into the ability to quickly process and manipulate simple stimulus (Spatial Relations) and the ability to correctly process and manipulate complex stimulus (Spatial Visualization).

#### 9.4.1.4 Relations + Visualization + Pattern

Probably one of the most influential proposed structures is that of Carroll (1993) because it was produced through the meta-analysis of a large number of factor-analytic studies, with factors in common to studies being identified and added to the structure. The structure maintains the factors concerning manipulation and transformation of objects as proposed by Pellegrino et al. (Pellegrino et al., 1984): *Visualization* (measuring the complexity of visual stimulus that can be successfully manipulated) and *Spatial Relations* (measuring the speed with which relatively simple visual stimulus can be successfully manipulated). The structure also includes factors involving the matching of visual patterns not found in that work: *Closure Speed* measures the speed with which previously unknown disguised or obscured patterns can be identified; *Flexibility of Closure* measures the speed with

which a known disguised or obscured pattern can be identified; and *Perceptual Speed*, the speed with which a known, non-disguised pattern can be found.

#### 9.4.1.5 Facet Theory

Facet theory is a bottom-up approach, with theory being developed to explain the unobserved variables (factors) uncovered through factor analysis. Facet theory on the other hand is more top-down; the technique utilises a mapping sentence which describes the proposed structure via a number of facets. Spatial ability tests are then classified according to these facets. Analysis of data obtained from experiments is then used to determine whether the facet structure contained in the mapping sentence is tenable or whether it requires modification.

Using the facet theory approach, Guttman et al (1990) proposed that spatial ability as measured by spatial ability tasks had at least three facets. The first facet is the *Type of Rule Task* specifies whether the task involves discovering the rule (rule inference) or applying a rule presented via a given relationship (rule application). The second facet is *dimensionality*, whether the problem is two-dimensional or three-dimensional. The third facet is *rotation*, whether the problem requires rotation of objects or not. Their evaluation of results validated the existence of the *Type of Rule Task* facet and the *Rotation* facet, with the *Dimensionality* facet showing an unclear relationship between problems and their dimensionality.

Stumpf and Eliot (Stumpf & Eliot, 1999) performed a facet analysis on academically talented students. In their study, they validated the *Dimensionality* facet proposed by Guttman et al. (Guttman et al., 1990) and also validated the *Rotation* facet; however, they discovered a further subdivision of the *Rotation* facet into separate sub-facets for two-dimensional and three-dimensional rotation. They also found a *Speededness* facet, which in their work replaces the *Type of Rule Task* facet from R.Guttman et al's work. Finally, they proposed an additional facet to separate between tasks that involve *internal features* versus tasks that involve *external construction*.

#### 9.4.1.6 Factor/Facet Summary

The preceding section provides an overview of different models of spatial ability from literature. Factors theories all include a spatial visualization factor which is implicated in the mental transformation of complex stimulus; most factor studies also include a *spatial relation* factor which is involved in recognising transformed simple stimulus.

The meta-review by Carroll (1993) also introduced three factors related to the recognition of patterns (*closure speed*, *flexibility of closure* and *perceptual speed*).

Another meta-review of factor studies showed strong support for a *spatial orientation* factor involved in perspective-taking and comprehension of relationships between objects.

The facet-theory studies also showed evidence for *dimensionality* aspect differentiating two-dimensional and three-dimensional tasks.

The factor of *spatial visualization* is clearly important in Computer Graphics Education. Learning the concept of a Computer Graphics *transformation* (rotation, translation, scale) on objects and their local coordinate axes would clearly benefit from better *spatial visualization* skills allowing for the effective mental transformation of objects; as will be shown later, students appear to have significant problems in performing such transformations effectively and reliably.

The factor of *spatial orientation* likely also plays a role in Computer Graphics Education. Implementation of views/cameras and the understanding of the relationship between local and global space is likely to involve *spatial orientation*. Such tasks are not straight-forward, as analysis of student implementation of views presented in Section 6.6.4.2 shows.

While research on the factors of *spatial visualization* and *spatial orientation* is relevant to Computer Graphics Education, even complex stimulus used in spatial ability research is still usually simple when compared to the complex transformations (transformations built on transformations) required in Computer Graphics Education. It may be productive to conduct formal experiments with stimulus of the kind students are exposed to when learning Computer Graphics, but this is outside the scope of this research project. It may also be interesting to investigate whether the level of ability in the respective factor would manifest via a student being able to more easily learn transformations compared to views or vice versa, though an experiment to test this hypothesis would be difficult to construct.

The role of dimensionality is also not entirely clear in existing research, with facet analysis suggesting a role for dimensionality whereas most factor research has not identified a separate factor for two-dimensional and three-dimensional stimulus. Clarification of this issue would be helpful, since it would clarify whether the learning of two-dimensional transformations may play a role in improving spatial ability in a way that will support three-dimensional transformations.

In reviewing spatial ability literature, the focus was on empirical studies of spatial ability and models of spatial ability based on the findings of such studies. There is also a significant body of literature relating to theoretical descriptions of underlying cognitive processes such as the research on *Mental Imagery* (Kosslyn, 1988) which falls outside the scope of this review.



### 9.4.2 Spatial Ability and Performance

The study “Spatial ability and achievement in introductory physics” (Pallrand & Seeber, 1984) found that exam performance in physics education is correlated to spatial ability. Physics education was also found to improve student spatial ability, and spatial ability was improved even further through a targeted intervention that was applied to a subset of students.

Jones and Burnett (2008) compared student assessment outcomes in computer programming and non-programming modules that were part of a computer science course. The results indicated a clear correlation between spatial ability and outcomes in the programming module, and no correlation between spatial ability and the non-programming module, suggesting that spatial ability plays a significant role in computer programming.

The same authors also conducted another study (Jones & Burnett, 2007) in which student spatial ability (measured via a spatial ability test) was correlated to student source code navigation behaviour. The study was carried out by having students complete a program navigation and comprehension exercise, with student actions being recorded via screen-recording software. The study found that high spatial ability correlated strongly with high performance. The authors also observed that time taken for program navigation and comprehension was significantly shorter for students with high spatial ability, and that students with high spatial ability displayed different navigation strategies. These results suggest that students with high spatial ability will be able to more effectively navigate and understand source code, thereby increasing their performance in programming-related activities.

Hsi, Linn and Bell (Hsi et al., 1997) found a significant correlation between spatial ability and performance in an engineering course. Their study found that a spatial ability training intervention (targeted at students with lower spatial skills, but open to all students) improved spatial ability and closed the gap (measured in the pre-test at the beginning of the semester, when males scored significantly better) between males and females.

The effect of spatial ability on Chemistry education was investigated by Carter et al (1987). Student spatial ability was found to correlate with test and exam performance. Exam questions were further broken down into sub-categories, and the correlation between performance in these question categories and spatial ability were determined. Correlation was found to be stronger on question types that require problem solving skills compared to questions types that elicit rote knowledge or the application of a learned algorithm.

A follow-on study at the same university investigating the link between spatial ability and chemistry education in four different chemistry courses (Pribyl & Bodner, 1987) confirmed the results presented by Carter et al. (1987). Spatial ability was shown to have a moderate but significant correlation with performance. In addition, the impact of spatial ability was highest for the course for which the exams had been found to contain more questions requiring high order cognitive skills or problem solving skills.

### **9.4.3 Improving Spatial Ability**

#### **9.4.3.1 Meta-Analysis on Spatial Skill Training**

A meta-analysis of several studies on the effect of experience and training on spatial ability (Baenninger & Newcombe, 1989) showed that experience with spatial tasks is related to spatial ability (though the meta-correlation is small; the authors hypothesize it may be due to the poor reliability of estimated experience). Also, meta-analysis of results on the impact of training of spatial ability showed that such training resulted in permanent improvement of spatial skills, with long-term training providing a higher level of improvement than short-term training and training specific to the type of spatial task being used in evaluation providing better improvement than non-specific training. However, non-specific training still provided significant improvement to spatial skills.

#### **9.4.3.2 Architecture**

Gutierrez et al (Martín-Gutiérrez, Gil, Contero, & Saorín, 2010) conducted a study on improving spatial abilities through the teaching of 'descriptive geometry' using both a traditional pencil-and-paper methodology and using a computer program of their own design. The result of pre and post spatial ability tests indicate that students engaging in a descriptive geometry intervention experienced a significantly higher increase in their spatial ability (as measured by standard spatial ability tests) compared to the control group; it also showed no significant difference between whether the course was taught using the pen-and-paper method compared to the computer-based method.

#### **9.4.3.3 Physics**

A study following students taking an introductory physics course (Pallrand & Seeber, 1984) found that student spatial abilities increased for students after completion of the course, and that students with lower spatial ability had higher drop-out rates despite their performance in mathematics courses not being statistically different to those students that did not drop out. Furthermore, a spatial ability training course involving a treatment, control and placebo group showed greater improvement for students in the treatment group (receiving additional spatial-ability focused training) than in either the control or the placebo group.

#### 9.4.3.4 Mathematics

Not all attempts at improving spatial ability to improve learning outcomes have been successful. A study exploring spatial training for mathematics students (Ferrini-Mundy, 1987) provided general (non-mathematics specific) spatial training to a treatment group enrolled in a calculus course. The study found that the spatial training had no effect mathematics ability as measured via course performance. The spatial training did have a significant effect on one measure of spatial ability designed for the study (and hence probably to some extent reflecting the material of the training course), while not having a significant effect on the other spatial ability test. While the spatial ability training's results on spatial ability were mixed, participation in the mathematics course (with practice effects controlled for) did improve spatial ability on both measures. This suggests that spatial ability training must be well-designed, and may also indicate a need for task-oriented spatial ability training.

#### 9.4.3.5 Engineering

Much of the literature on spatial ability training relating to a particular field of study comes from engineering education research, since many skills essential to engineers such as the use of CAD tools to produce three-dimensional models or the visualization of three-dimensional objects based on a set of side or top-down orthogonal views are exceedingly likely to be linked to spatial ability, especially the kind of ability described by the *spatial visualization* factor. Indeed, studies as early as 1955 (Blade & Watson, 1955) showed that the study of engineering improves spatial ability.

A study of engineering students (Hsi et al., 1997) showed a significant correlation between spatial ability (as measured via administered spatial ability tests) and course performance. That study also examined the effect of a remedial course targeted at improving the spatial ability of students that performed poorly on a spatial test at the beginning of the semester. While there was no evaluation of general performance gain, the intervention did close the spatial ability gap between males and females (females having scored significantly poorer on the pre-test). The intervention also significantly reduced drop-out rates for the students that had participated in the remedial course.

Martín-Dorta et al (Martín-Dorta et al., 2008) implemented a remedial course for engineering students using Google Sketchup. The remedial course led to a statistically significant improvement in spatial ability. The gender gap which had been statistically significant before the intervention was no longer significant after the intervention.

A study by Alias et al (Alias et al., 2002) involves a treatment / control group experimental design. The intervention involved pencil-and-paper sketches and physical artefacts. Spatial ability

measurement instruments were designed specifically for the study. These instruments involved mental rotation items, cube construction items and engineering drawing items to reflect actual engineering tasks. The study found a significant improvement for the treatment over the control group only for the engineering drawing task. This could be because of a relatively high degree of independence between different sub-abilities considered to form part of spatial ability. If this is the case, then it would suggest the need for spatial instruction tailored towards the area of study rather than generalised spatial ability improvement courses.

An international study (Leopold et al., 2001) analysed whether engineering courses offered by three different universities increased spatial ability, and whether spatial ability predicted success in the final examination. Of three spatial ability tests used in the study, two were significant predictors of success in the final examination at all universities, and one was a significant predictor of success at only two universities. When compared to control groups engaging in different courses thought to have less of a spatial component, a statistically significantly larger improvement was measured for students engaged in the courses being investigated at all three universities. This suggests both that spatial ability is an important factor to consider in students' engineering education, and that carrying out tasks requiring spatial reasoning leads to an improvement in spatial ability.

#### **9.4.3.6 Summary**

The research on spatial ability training and the role of experience in spatial ability clearly suggests that experience with tasks that require spatial ability (such as science or engineering courses) plays an important role in the level of spatial experience, and that training can increase spatial ability

Results from the study by Alias et al (Alias et al., 2002) suggest that it is important to create spatial ability training that mirrors real tasks in order to achieve positive outcomes. The failure of a study by Ferrini-Mundy (Ferrini-Mundy, 1987) to achieve significant improvement in student performance through general spatial ability training supports this notion. Given that much of the existing research on spatial ability training focuses on spatial ability in the context of engineering, this necessitates the development of new methods for spatial ability training in the Computer Graphics programming context which mirror actual Computer Graphics programming tasks.

#### **9.4.4 Snapshot Approaches**

The term 'Concept Assignment Problem' refers to the problem of reverse-engineering an understanding of human-oriented concepts embedded in programs and connecting these concepts to their actual implementation in the source code. It was coined by Biggerstaff et al (1993). Biggerstaff suggests the use of more 'fuzzy' / heuristic (plausible reasoning) approaches as opposed

to highly formal approaches due to the difficulty of creating machine approaches that generate human understanding. Wilde and Scully (1995) sought to solve a similar problem that they described as *feature location* (or feature detection, feature mapping). *Feature Location* involves finding the implementation of some program feature or functionality. While the concepts of *Concept Assignment* and *Feature Location* are perhaps not entirely equivalent (*Feature Location* may be seen to have a more narrow focus) the difference is relatively minor.

Several different approaches to solving the *Concept Assignment / Feature Location* problem exist. Biggerstaff et al (1993) utilised a static-analysis approach presenting call graph and program slicing data to aid a human researcher in discovering program structure and investigating areas of the source code related to an area of interest.

Wilde and Scully (1995) instead used dynamic analysis. A test case designed to elicit the execution of a certain feature is executed, with functions and methods called during execution being captured. The result of this execution is then compared to the execution of a test case not invoking the feature, with methods / functions that occur in the first but not the second set being presumed to be related to the feature in question.

Another more recent approach to feature location is the use of Information Retrieval (IR) techniques. Information Retrieval research focuses on the retrieval of information from documents, either by analysis of their textual content or associated metadata. In the approach proposed by Marcus et al (Marcus, Sergeyev, Rajlich, & Maletic, 2004), source code files are treated as documents and are broken down into individual identifiers and comments. The Information Retrieval technique applied to these data then retrieves sets of documents based on similarity with a search query. Evaluation of the method according to measures of precision and recall (measures commonly used in IR research) indicated that the method was successful in identifying relevant documents. The interesting difference of IR approaches compared to either standard static or dynamic approaches is that the IR algorithm treats the source code as text, instead of analysing properties of the underlying programming constructs as is the case with call graphs or program execution traces used in static and dynamic analysis. Such an approach is presented by Liu et al (D. Liu, Marcus, Poshyvanyk, & Rajlich, 2007).

## 9.4.5 Review of Computer Graphics Literature

### Review of Computer Graphics Education literature

by Maximilian Wittmann

Note: This draft review was produced during the early part of the candidate's doctoral work and has not been substantially revised. As a result, the standard of writing is inconsistent.

Rather than presenting material directly relevant to this thesis, the review serves to demonstrate the thoroughness with which the relevant literature was examined in the initial phases of this research project.

#### 9.4.5.1 Introduction

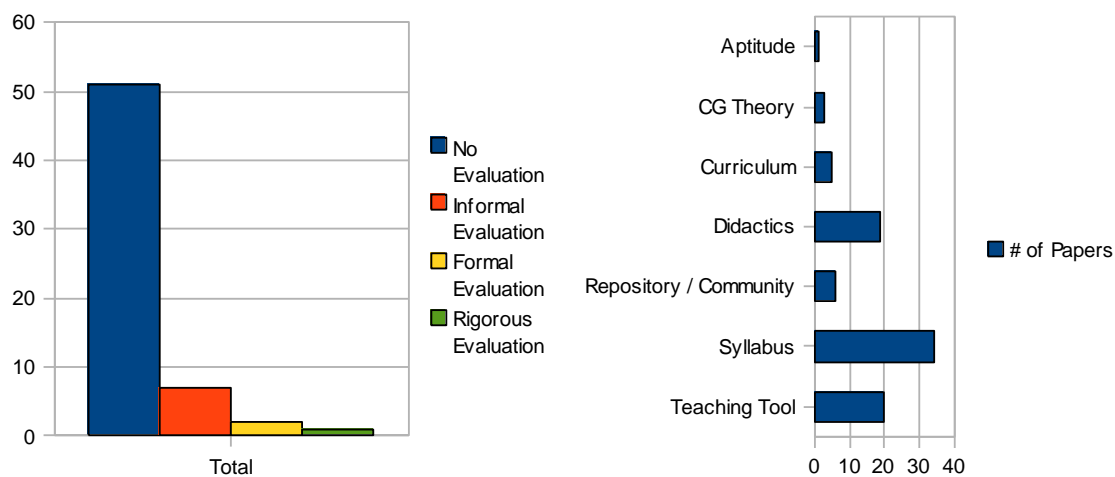
In conducting this review of Computer Graphics, I analysed 88 journal, conference and workshop papers. In reading these papers, my aim was a high-level overview rather than a detailed look at the issues explored in individual papers. I also excluded 66 papers from the review because they were irrelevant or not in fact full papers.

I collected papers from the ACM database and Google scholar which contained the key phrases "Computer Graphics Education" and "Introductory Computer Graphics". While this heuristic will not turn up every single paper relating to Computer Graphics education research, I hope to have gathered a substantial subset of the relevant literature.

While collecting and reading the material, I also used a simple categorization scheme to provide some empirical data on what areas in Computer Graphics education are most extensively researched, and which areas are under-researched. Where a paper fit into more than one category the author chose the category he thought was most appropriate. This occurred most often with papers which discussed both teaching tools and teaching methods. The categories underwent some changes to better fit the papers; the finalized categories are as follows: the *syllabus* category contains papers which describe what material is to be included or excluded from Computer Graphics courses; the *curriculum* category contains papers which describe the role of Computer Graphics in the broader curriculum; the *didactics* category contains papers which suggest novel teaching methods or approaches for teaching Computer Graphics; the *teaching tools* category includes all papers which deal with the design, implementation and/or use of a piece or pieces of software (visualization applets, graphics libraries and frameworks, game engines) to teach Computer Graphics. The *repository / community* category contains papers which propose or describe existing repositories for Computer Graphics teaching material. The *Computer Graphics theory* category includes papers which discuss an aspect of Computer Graphics theory and how to present it successfully in detail. It is different to the didactics category because the focus is on a specific theoretical aspect; the final category is *aptitude*. Papers in this category focus on determining the

aptitude of students for learning Computer Graphics and what effect it has on the success rate of Computer Graphics students.

As we can see in Figure 115 the syllabus, teaching tool and didactics categories make up the bulk of published work. 83% of papers fall into one of these categories. Considering that Computer Graphics education falls under the broader umbrella of Computer Science Education research, these numbers are hardly surprising and probably quite representative of Computer Science Education research in general. Didactics and syllabus are a natural focus for education researchers, and computer scientists have a natural tendency towards developing computer tools for solving problems, and this is no different in the education domain.



**Figure 115: On the left, a graph showing the evaluation method used in Computer Graphics Education papers; on the right, the categories into which reviewed papers fall.**

The next table shows how many of the papers performed any kind of proper evaluation in terms of effectiveness or student satisfaction aside from word of mouth. The category of informal evaluation includes papers that used very simple evaluations such as utilizing student data for two semesters or utilizing a questionnaire. Formal evaluations were more thorough and did not rely on a single source of data. Rigorous evaluation meant that the entire paper was structured around the evaluation method for the treatment or teaching tool.

Apparently scientific evaluation is not one of the strengths of Computer Graphics education research. Of course, a lack of rigorous evaluation of proposed methods is not uncommon in Computer Science Education research in general, but the subset of Computer Graphics education research forms such a small body of research to begin with that this spells an almost total absence of empirical research in the field.

#### 9.4.5.2 Review (in order of magnitude)

I will now review each of the categories in turn, going from the most to the least populated category.

**Syllabus** It is only natural that a field which is very technology-driven and prone to rapid change would foster a great deal of discussion about what topics should be included in an introductory course.

An additional issue in Computer Graphics Education is that it is usually taught as a third-year elective (occasionally as a second-year elective or as a postgraduate elective). This means that it is often not possible to offer extensive follow-up courses to undergraduate students. Hence there is only a single semester in which to teach all the most important core concepts. Computer Graphics is also a very young field, even compared to Computer Science in general.

Historically, the syllabus in Computer Graphics subjects can be divided into two parts; the early pioneering years (1980-1990) and the modern Computer Graphics syllabus (1990-2008)(Cunningham, 2000). That is not to say that there were no changes in the way that Computer Graphics was taught in these intervals, or that it was or is taught uniformly at all places of higher education. However, there was and is a popular 'standard' syllabus for Computer Graphics which has become more standardized through SIGSE recommendations in 2004 (Cunningham et al., 2004).

In the early period (1970-1990) Computer Graphics was still a very young field, having been inspired by such applications as SKETCHPAD (Sutherland, 1964) and Spacewar! (Graetz, 1981). Since computers were expensive and equipped with very limited graphics facilities by today's standards, it was very difficult to achieve three-dimensional graphics. Also, few software packages existed for creating graphics applications, and most of these were commercial and expensive (Hitchner & Sowizral, 2000). As a result, most educators created their own limited libraries for their students to use. This in turn meant that students had to start at the very bottom, learning primitive operations and algorithms. Computer Graphics was taught almost exclusively in 2D, as a synthesis of image generation and image manipulation.

The modern period (1990-2008) is marked by a steep fall in the price of computers and a very rapid increase in the power of dedicated graphics hardware exceeding Moore's law, as well as the availability of free, open-source 3D Computer Graphics API's such as OpenGL and DirectX. Students would be able to complete projects creating three-dimensional content in a single semester. As a result, it became possible to move from 2D applications of Computer Graphics to true-3D graphics. Such a course was proposed by Cunningham (2000). Such a syllabus, driven by a high-level relatively



easy to use API can be described being top-down, as opposed to the bottom-up approaches necessitated by lacking hardware and software facilities in the early days of Computer Graphics computing. The syllabus recommended by SIGSE after the latest iteration in 2004 (Cunningham et al., 2004) recommends knowledge of the following concepts as the chief learning goal for Computer Graphics units:

- Transformations
- Modeling: primitives, surfaces, and scene graphs
- Viewing and projection
- Perception and colour models
- Lighting and shading
- Interaction, both event-driven and using selection
- Animation and time-dependent behaviour
- Texture mapping

A survey of what is actually taught at universities is presented by Wolfe (2000) and shows that these recommendations are largely being followed. Even so, Paquette (2004) believes that Computer Graphics education should return to first presenting concepts in 2D and should reintroduce Image Processing into introductory units in order to give students a more solid understanding of underlying principles, as well as graduating from easier 2D mathematics to the more difficult 3D mathematics.

A topic which some educators (Chalmers & Dalton, 2002; Cunningham, 2002; McGrath & Brown, 2005; Sears & Wolfe, 1995; Wolfe, 2002) would like to see explored more in Computer Graphics syllabi is *Visual Analysis*. Visual Analysis is defined as the ability to analyse an image and thereby gain an understanding of algorithms used in its creation.

Chalmers and Cunningham (2002) argues that it is important to foster *Visual Communication and Thinking* skills. Computer Graphics problems are very different from Computer Science problems in general in that their output is visual instead of conceptual. It is not mere data to be evaluated as correct or incorrect, but rather complex visual stimuli that require interpretation and even aesthetic consideration. By fostering Visual Thinking skills students can gain a far better understanding of how to create and visually debug (Wolfe, 2002) Computer Graphics applications. And teaching Visual Communication skills enables students to exchange an understanding of visual elements of a Computer Graphics application to be designed, which is vital for those who decide to apply their Computer Graphics skills in their work life and hence must work to specifications and common understanding.

Another topic which many educators suggest including in the curricula is *Shading*. The topic of shading covers techniques for altering the colour of surfaces in a scene based on the lights present in the scene. This includes making surfaces shone on by a close light source brighter than those farther away or not being lighted. It also includes advanced techniques such as creating shadows. Because shading is vital to create realistic Computer Graphics applications it is a very important topic, but it is a complex topic and was until recently not well supported in the OpenGL API used by most Computer Graphics educators, so it was often covered in a cursory fashion only. Now that OpenGL offers the OpenGL Shading Language (GLSL) it is feasible to teach shading using the same tool that are used to teach other unit content, at a similarly high level; before, separate tools were often used or a low-level approach with students implementing their own shaders was selected. Several educators now propose including shading in the introductory Computer Graphics unit (Iglesias & Gálvez, 2004; Owen, Zhu, Chastine, & Payne, 2005; Talton & Fitzpatrick, 2007; Xiang, 1994).

Other educators suggest including *Modelling* (the creation of three-dimensional objects) or *Animation* (Lowther & Shene, 2000; Schweitzer & Appolloni, 1995) as core topics in the Computer Graphics syllabus.

Cliburn (2006) proposes including Virtual Reality concepts in introductory Computer Graphics units as a means of providing realistic application-area tasks.

In 'Adapting Computer Graphics Curricula to Changes in Graphics' (Hitchner & Sowizral, 2000) the author makes several suggestions for what directions Computer Graphics teaching should take, the most emphasized being that Computer Graphics courses should be taught in Java to make them more accessible to students who are taught Java in first-year classes, and to enable learning materials to be delivered more effectively over the internet.

Shirley et al (2007) point to the feasibility of real-time ray tracers to suggest that real-time raytracing may make current-generation rasterisation-based rendering methods obsolete, which would cause upheaval to Computer Graphics education since it would mean a fundamental shift in how Computer Graphics programming is carried out.

**Teaching Tool** Computer Graphics is a difficult field because it combines programming which most Computer Science students are good at with mathematics which many Computer Science students are less competent and/or confident in. In addition, Computer Graphics requires a unique spatial and visual understanding of algorithms and code blocks. It is no surprise that many Computer Graphics teachers have used the art of Computer Graphics to create visualisation tools to aid their

students in understanding difficult concepts. The papers which I will review can be categorized as describing APIs or software libraries, visualization tools, or game engines. The age of the internet has also made online delivery important, and papers on this topic were also reviewed.

Before OpenGL, educators attempted to develop their own implementations of the graphics pipeline ; an example is the TUGS library (Clevenger, Chaddock, & Bendig, 1991). Such non-standardized approaches were not conducive to the standardization of the Computer Graphics syllabus across institutions. Other very early examples are work by Mohilner (1975) and Thomas (1979).

The OpenGL API is perhaps the most influential teaching tool in Computer Graphics education, even though it wasn't created for that purpose. In fact, it has shaped syllabi ever since it was widely embraced as the most suited high-level API for Computer Graphics education and more broadly, academic research (Angel, 1997; Cunningham, 2000). The result was a shift from 2D to 3D in Computer Graphics education, and from bottom-up to top-down approaches.

OpenGL has some shortcomings as a pedagogical tool for teaching 3D Computer Graphics. One of these is that it is dependent on the Operating System for windowing functionality, and every OS has a different implementation which OpenGL needs to interface with. This problem can be alleviated with the widely-ported GLUT library which hides OS-specific windowing details underneath a simple, abstracted interface. Unfortunately neither OpenGL nor GLUT provides many other bells and whistles; there are no GUI widgets and only a very partial implementation of on-screen text. This is especially unfortunate since interactivity allows the student to explore her own application and dynamically making changes may aid the student in gaining a better understanding of complex transformations, for example. One option would be to have students implement their own UI elements, but that task is time-consuming and has little to do with Computer Graphics; furthermore, students will be unable to implement high-quality UI components in the limited time they have available.

The Graphix toolkit for C++ (Necaise, 2006) provides GUI components and wraps GLUT's functionality in object-oriented containers to make it easier to use for students with little procedural programming experience. Of course, using Java instead of C++ as Hitchner and Sowizral (Hitchner & Sowizral, 2000) suggests is a valid solution for these problems, as Java provides a portable windowing and GUI framework.

The uisGL library (Grissom, 1996) is another library that serves to eliminate some low-level programming on the part of students. UisGL is aimed at abstracting from OpenGL's data structures

and transformation operations. Libraries like uisGL have since been supplanted by high-level APIs built on top of OpenGL; the first of these was OpenInventor, and OpenSceneGraph is rapidly gaining popularity today due to its ease of use, extensibility and powerful underlying scene graph concept.

Some libraries like the Act library (Celes & Corson-Rikert, 1997) abstract away the underlying programming language by presenting a high-level custom-tailored Computer Graphics language. This is a trade-off of having a shallower learning curve versus using a real-world framework which can be used to do actual graphics work.

Interest in creating educational libraries to support students seems to have decreased in recent years. The most likely reason is that OpenGL, OpenInventor and OpenSceneGraph provide excellent frameworks at different levels of abstraction, and are available for free. The exception to this rule is shading; shading is a complex topic, and even though OpenGL now has its own shading language (GLSL) it is not trivial to use and does not offer functionality such as raytracing. MALUDA (Pereira & Gomes, 1999) is a simple tool for processing shading commands and allowing students to easily generate shaded scenes. RenderMonkey [ATI] may be a good current, open-source alternative. GraphicsMentor (Yu, Lowther, & Shene, 2005) is a C++ library that allows for very advanced shading techniques including photon mapping not usually available through OpenGL.

The next category of Graphics Engines is in a way a continuation of the libraries category. A Graphics Engine uses an API to provide high-level functionality which hides much of the low-level detail required for OpenGL programming. In an educational context, such engines can be designed as modular which provides an opportunity for scaffolding by requiring students to progressively implement modules that were provided for earlier assignments, or by asking them to extend the engine itself. As such, they can enhance the top-down approach favoured by many Computer Graphics educators.

The MAVERICK Graphics Engine (Howard, Hubbard, & Murta, 1999) is specialized for VR applications. It provides stereoscopic views, culling, and other important graphics functions. It also includes a Builder which allows interactive creation of 3D content. This engine may be of interest for universities that have decided to incorporate VR into their Computer Graphics syllabi, since using a high-level engine will free students from having to learn low-level basics, allowing them more time to create VR applications.



**Figure 116: The eNVyMyCar Graphics Engine (vers. 15766)**

The eNVyMyCar Graphics Engine (Ganovelli & Corsini, 2009) (see Figure 116) is specialized for the creation of multiplayer racing games. The server part of the game engine handles all the game rules and game logic, such as how fast cars drive. The client part of the game engine is handed to students as a partial scaffold, and it is their task during the semester to create a fully functional client that can be connected to the server to allow gameplay. The Computer Graphics part of the work comes from the fact that no graphical representation of the cars or environment is present in the game client, and it is up to students to implement the visual aspects of the game client-side. This idea builds on scaffolding and competition, as well as the intrinsic interest computer games generate in many students. The game engine can be used to implement a large range of basic and advanced topics, thus providing material for all assignments during an introductory course.

We now move on to Visualization tools. Here, unlike in the previous category, there are no industry-strength ready-made default options available. One could see all Computer Graphics programming as visualization, since students receive visual output in response to their code input. However, this visualization is often limited by the difficulties inherent to making very interactive applications. Unless an application is very interactive, it will not allow the user to visualize a large range of possible states and will lead to a limited exploration and a limited understanding. Creating very interactive applications takes time the students may not have. Furthermore, the students may make errors in their code, and the mismatch between their mental representation of what they coded and the image on-screen may frustrate or misinform. Visualization tools provided by the educator are meant to overcome these problems. They should try to generate well-aligned output from student input, be very interactive and be vouched to be working correctly.

One of the most important introductory topics in Computer Graphics is transformations. Understanding transformations in two and three-dimensional space allows the student to position and orient objects in space. The idea is simple enough; after all, we perceive and can express position and orientation of the objects we perceive in the real world. The chief difficulty is understanding the underlying mathematics that make up these translations. The LEG (Smith, 2005) and the Interactive Notes Module (Shabo, Guzdial, & Stasko, 1996) are both tools for teaching 2D transformation. The Interactive Notes Module is an early attempt at providing online content, and as such is very limited in terms of interactivity (see (Naiman, 1996) for a similar application). It is essentially a question-answer application. LEG is more advanced, allowing students to explore 2D animations through different levels of scaffolding which are provided by input of different underlying conceptual complexity. The Programmable Tutor (Andújar & Vázquez, 2006) is a yet more powerful tool, allowing students to explore three-dimensional transformations by specifying OpenGL transformation commands such as *rotatef* or *translate* interactively through the drag-and-drop GUI interface. This allows for the swapping transformation commands to observe what difference ordering makes to non-commutable rotations, for example.

Bailey (2007) and Wolfe and Sears (1996) created *shading visualizers*. TERA allows students to select rendering algorithms from a list to see scenes rendered with the selected algorithm, and to quiz themselves by having to identify the algorithm that was used to render a scene. It does not provide a high level of interactivity, and the tool does not actually perform any rendering. Rather, it is equipped with a library of pre-rendered images. Glman is far more interactive, allowing students to interactively create and transform scene objects and then specify rendering algorithms. This will allow students to create their own rendering algorithms. The functionality is similar (and perhaps superseded by) ATI's RenderMonkey.

The Explorer (Wernert, 1997) is a multi-purpose application which allows scenes to be composed in a visual data flow language by dragging and dropping modules which perform diverse functions such as raytracing, clipping or illumination calculations. Due to the modular structure, it can also be extended with further functionality.

One paper discussed general rules for design and delivery of educational tools in the context of visualization applets. Gould, Simpson and van Dam (1999) suggests that the scope of an applet can be described as either coarse-grained, demonstrating several different concepts, or fine grained if they present a single concept in great detail.

Gould's experience suggests that coarse-grained applets are more difficult to code well and take a significant development effort if they are to be useful. In addition, their implementation requires more experienced programmers. While they are good at demonstrating the final outcome of an algorithm, the broad nature of the scope means they are poor at presenting the intermediate steps in fine detail and are hence not suited for teaching algorithms in detail.

On the other hand fine-grained applets are designed to deal with atomic concepts and hence limit their scope as much as possible. This makes it easier to code them, and while it takes more fine-grained applets to explain a selection of topics than it would coarse-grained applets, Gould found that the development time for coarse and fine-grained applets covering a set of topics was roughly equal. Fine-grained applets can go into the full detail of an algorithm, thereby providing more explanatory power to the user. They are also easier to consume as they do not overwhelm the user with information, and they are easier to reuse as their smaller code-base means they are easier to understand. The main consideration when implementing fine-grained applets is to ensure that the user interface is very simple and easy to learn and as consistent across applets as possible, because students cannot be expected to spend a long time understanding the user interface for each fine-grained applet.

**Didactics** In this section I will review the papers that discussed methods for teaching Computer Graphics. In reviewing the papers related to teaching methods it was interesting to note that Computer Graphics does present opportunities that would be harder to realize in other Computer Science units, especially in terms of collaboration and inclusion of intrinsically interesting and motivating material, such as computer games or interactive media.

Several papers discussed teamwork, competition or collaboration. Surma (2006) set team-based assignments on which upon completion each team would have to give a short presentation. In addition to assessing team members individually for their presentation contribution, Surma awarded each group an overall mark that contributed to an inter-group tally. At the end of the semester, Surma awarded prizes to the best and the most improved groups. While Surma did not undertake any formal evaluation, he believed that the competition significantly increased the quality of the student programs. Problems included initially poor presentation skills and uneven contributions within groups that led to conflict and member changes during the semester.

Ebert and Bailey (1999) describe their experiences with a Computer Graphics course in which computer science and visual arts students are grouped to complete assignments, thereby gaining interdisciplinary experience. No formal evaluation was undertaken, but the ecological validity of the

approach holds out much promise for universities with both Computer Science and Visual Arts departments.

Case (1999) also discusses collaboration in Computer Graphics education. She believes collaboration is important because it is more likely to produce Computer Graphics programmers that can tackle real-world problems and keep pace with changing technologies. She approaches the problem from an organization perspective, suggesting that universities create interdisciplinary curricula in which different departments contribute students to tracks of units to ensure a variety of skills in each unit.

In this section, I'll cover the papers which involve incorporating games and media into Computer Graphics units. Sung et al. (2007) noticed that many students chose to create game-like applications for their cap-stone project. Sung wanted to make the assignments for his Computer Graphics unit game-like to utilize the motivational power that the game concept has for many students, but he did not want this to in any way affect the syllabus for his unit. He did not want to teach game programming. He therefore wanted to emphasize the interactivity and graphics components of games programming. Sung found that though he increased student workload, students did just as well academically and rated the course similarly as before, and created higher-quality assignments than before the move to game-styled assignments. Hence, he found the benefits of this teaching approach to be chiefly motivational, though he did not carry out any formal evaluation or comparison.

Tori et al (2006) summarize their experiences with using computer games as a guiding theme, a motivational tool and as scaffolding for their students. Tori et al took the same position as Sung et al (2007) in not including any computer game specific material in his course. He also provided students with a custom pedagogically designed game engine for development of their assignments. The driving factor for Tori's choice of computer games as a theme for assignments was to have a focal topic that would result in student projects of similar complexity and functionality. The custom game engine also supports this aim. There were initial problems when students focussed on the wrong aspects of assignments (from a Computer Graphics standpoint, such as AI). Tori did not want to impose strict guidelines on the design of assignments since that undermines the core strategy of allowing students to design something they enjoy, and thereby decreases motivation. Instead, Tori laid out some rules and included a project proposal presentation in the assessment. After this presentation, students were advised on what changes should be made to projects to make them more Computer Graphics-oriented, more achievable or more interesting. Evaluation was based on



the final application, the documentation and the presentation. Tori does not describe any formal evaluation of the unit, but has been running the unit with the games material for several semesters.

The previous papers described units that used computer games to provide motivation and guidance to students. Cheng (1999) also wishes to present interesting tasks to his students because playfulness increases the willingness to explore and experiment, which is a desirable state for students to be in. Cheng outlines four tasks that encourage playfulness. The first task is to produce geometric compositions, which are achieved by using boolean functions on objects to create complex shapes. Cheng describes this task as constrained-components, constrained-operations, since both the components (simple geometrical shapes) and operations (logical operations) that can be used are implemented by the assignment designer. 'Kit of parts assemblage and exchange' is a constrained-components unconstrained-operations task. In this task, students are given a set of components and are asked to create a complex environment from these components. In the 'transformation' exercise, students are required to create a cardboard model of a house, and are then required to 'translate' this model into a computerized three-dimensional model by measurement. This teaches students the relationship between real-world and computerized concepts. Chang did not conduct an evaluation of her teaching methods.

Sung and Shirley (2004) proposes using computer games (he also refers to them as “popular 2D interactive graphics application(s)”) to teach Computer Graphics using a top-down method in which students are introduced to Computer Graphics by analysing code and then designing 2D games of moderate complexity (compared to the usual standard for Computer Science assignments) instead of the usual approach of examining each concept individually, building up to combining techniques later in the course. It is interesting to note that the 'usual' approach which is the result of the SIGGRAPH recommendations was first proposed as a top-down approach using an abstracted API compared to the bottom-up approach of studying raw algorithms. This shows how much Computer Graphics has changed over the last ten years; the understanding of what constitutes a low-level and what a high-level concept has shifted significantly with the increasing power and increasing abstraction of modern graphics APIs.

Several papers also deal with the online delivering of Computer Graphics education. Teaching Computer Graphics online has a problem which is fairly unique to Computer Graphics inside the computing field. Computer Graphics education relies on full-blown 3D graphics rather than plain text and simple diagrams. Transmitting code and simple diagrams over the web statically or dynamically is simple, and methods and systems for doing so have been in use long enough to be very mature.

Technologies to transmit three-dimensional visuals over the internet are in their infancy; there are not many of them, they do not provide all the features or the rich interactivity that is available for text and 2D images in online teaching systems, and they are not as well integrated. Currently the most well-known and used technologies to transmit 3D content over the internet are probably VRML (Virtual Reality Modelling Language) and Java JOGL or Java Java3D. VRML is a markup language designed to allow the embedding of three-dimensional vector graphics in web content. While it did not reach large-scale uptake as was initially hoped, plugins for viewing VRML content are available for many web browsers. Unfortunately it is difficult to implement complex models in VRML, and global illumination models or other advanced Computer Graphics topics cannot be implemented using VRML. VRML is therefore not suited to provide demonstration material, at least not for an entire Computer Graphics unit. VRML's successor X3D improves on aspects of VRML but like its predecessor has not found wide-spread adaption. Java JOGL (Java Open-GL, an OpenGL binding for java) and Java3D (Sun's own 3D library) are far more powerful. In fact, since JOGL provides bindings to all core OpenGL functionality, it is theoretically possible to implement almost any OpenGL program in Java. Such an application can then be embedded as an applet, providing a powerful resource for online Computer Graphics education. The major limitation with creating complex applications will be the time required to load resources; large applications are most likely not good candidates for casual delivery online. As we will see, most online approaches to teaching make use of Java applets in some form or another.

Wiese et al (Wiese, Kyrylov, Nesbit, & Calvert, 2003) presents experiences of an online course. The goal was to create a system-independent and low-cost solution to teaching Computer Graphics at a time when many tools used in industry and education were neither. The course provided lecture notes, applets, instructions and explanations online. Since neither Java3D nor JOGL were available at that point, a custom 3D graphics engine was implemented, extending the 2D capabilities of Java to 3D. The course proved to be effective in conveying content as evidenced by student performance. In today's context, the most valuable lesson to take from this early experiment may be that the methods to enhance learning which were available only through the implementation of a custom graphics engine are available for free today through JOGL.

Wiese (Wiese et al., 2003) summarizes experiences with teaching a partially-online course using Java applets, including the teaching of advanced Computer Graphics topics. The unit was taught without lectures, with groups of five students forming teams to work on assignments and a weekly face-to-face meeting with an instructor. Each week an 'assignment' is due, alternating between group and individual assignments. Course content followed the SIGGRAPH guidelines, having basic rendering,

lighting, algorithm and mathematics components. In weeks with group assignments teams were required to give a brief presentation on their assignment topic, with each topic assigned to two groups to ensure that the topic was adequately presented to the class, thus offering learning content to the presenters as well as the audience. Wiese found that applets worked well at explaining even the most complex topics covered in the unit; the main issue with teaching Computer Graphics online turned out to be teaching the mathematical aspects of Computer Graphics because of the limitations of online teaching systems in displaying three-dimensional mathematics content or displaying mathematics content in general dynamically. No formal evaluation of the results was undertaken.

One paper did discuss using a specific learning theory. A study by Taxen (2004) describes Taxen's constructivist approach to Computer Graphics education. Constructivism postulates that knowledge is constructed internally rather than received from an outside source. A mental model may be revised if the learner realizes that it is not aligned and does not produce expectations that correspond with reality. In practice, this means that students should be exposed to problems with which they can build mental models as they work on them and solve them, rather than merely being exposed to solutions. Taxen developed an introduction to Computer Graphics session in which he and participants together came up with the graphics pipeline by reasoning about how such a system would have to be implemented to function. By directing participants and helping them when they got stuck, Taxen managed to produce the entire pipeline and many key concepts with the participants. He tried his approach twice, once with a group of volunteers in a one-session short introduction to Computer Graphics, and once with the students of his Computer Graphics unit. He did not perform an evaluation of the first one-session course. He performed evaluation via questionnaires of the session he conducted with his students, and found that 82% of his students preferred ordinary lectures to this technique. Taxen reasons that it may be that the technique is misaligned with student attitudes and with student assessment. Students are very busy, and hence there are many *strategic learners* (Marton & Säljö, 1976) who simply want to do enough to pass the class. They prefer answers they can learn and then be expected to be asked for in the final exam, rather than a process which causes them to think and understand areas surrounding the topic, since much of this understanding probably won't be in the exam. The assessment of students through assignments and exams encourages such strategic learning because deeper understanding and proficiency at skills that support being a good Computer Graphics programmer are not evaluated. Taxen states that it may require a combination of standardized tests, classroom tests, observation and self-evaluation. He believes that in order for constructivist techniques to become effective and popular with students more work needs to be done in adapting the academic setting to their use.

Beckhaus and Bloom's (2007) paper presents a teaching method in which students develop their own learning materials. While they does not mention constructivism, having students to create content for their tutorials fits in very well with students constructing knowledge instead of absorbing it passively. Instead of standard homework assignments, students developed Computer Graphics *teachlets*; teachlets include several slides of lecturing material, some questions for the audience and a small program to solve the assigned problem.

## 9.5 The SCORE Toolkit

### 9.5.1 Preparation of SCORE Data

#### 9.5.1.1 Basic Preparation

To prepare a Project History for use with the SCORE Analyser a configuration file pointing the SCORE Analyser to the Project History's location in the file system is created. A separate group configuration file can be created for sets of assignments, with individual assignment configuration files using relative paths based on the group configuration file.

Parameters such as the Levensthein distances used for Mutant and Ghost generation are configured by modifying the Config.cfg file.

When the assignment is first opened in the SCORE Analyser, the Analyser will automatically generate Line Histories for the project which will take around 1-5 minutes, depending on the size of the project (use of databases for storage would likely speed up the process significantly). Line Histories are then stored in .xml files and will be automatically loaded when the assignment is next opened, so the generation phase only needs to run the first time the assignment is opened.

With the Line History Generation phase complete, all non-execution-based functionality is available to the researcher.

#### 9.5.1.2 Compilation Configuration

In order to enable execution-based functionality (generating executable files and editing and compiling source code via the editor) the researcher must specify a path to a command-line compiler in either the individual or group level configuration file. Required include and library directories can also be specified.

File system paths can prove problematic to during compilation, since SCORE only copies .cpp and .h files for compilation, not data files. To resolve this problem, SCORE can utilise regular expressions to find and modify these paths before compilation. Regular Expressions for replacement can be defined either in the individual or group configuration files.

After compilation configuration is complete, the researcher can request the automatic compilation and screen capture of all of a Project History's Changes. This will cause a screen capture of the current Change to be displayed in the SCORE Analyser screen capture window, and will enable the researcher to execute the program at any Change. Compilation of all Changes is also necessary for some other functionality, such as the Compile Filter LH-Graph Extension algorithm described in Section 7.6.5.

## 9.5.2 SCORE Implementation Overview

This section is intended to provide an overview of the SCORE Analyser's implementation. In the interest of brevity and comprehensibility the description is often abstracted to some extent. In many cases, actual class or package names are replaced by more descriptive names. In those cases, the actual class / package name is provided in brackets.

### 9.5.2.1 SCORE Architecture Overview

#### 9.5.2.1.1 Project Statistics

The SCORE Analyser was implemented solely by the research project's primary researcher. It consists of the SCORE project providing the functionality described earlier in this chapter and the GeneralUtility project containing various reusable items of supporting functionality, including Java Swing components and layouts and database utility functions. Table 41 lists statistics relating to the implementation of the SCORE Analyser. Together, the SCORE Analyser and general utility packages contain 118054 lines of code in 312 packages comprising 1058 classes.

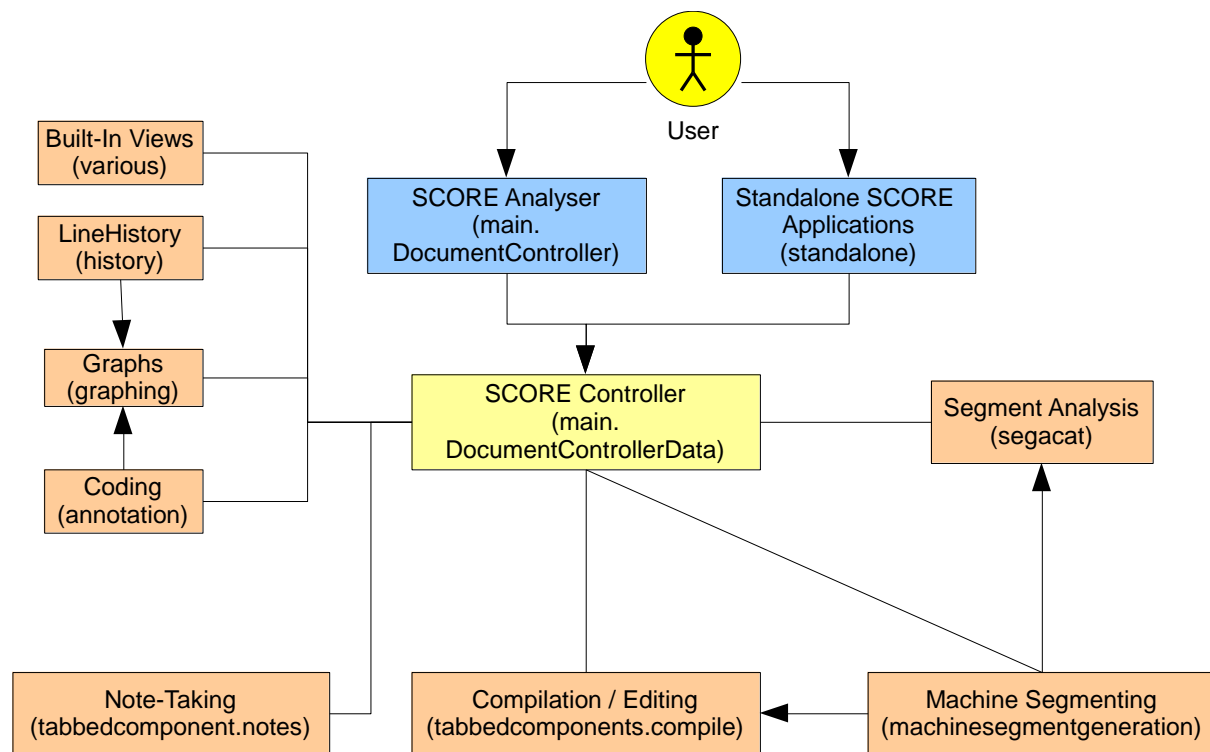
**Table 41: Implementation Statistics**

	SCORE	GeneralUtility	Total
Number of Methods	5756	848	6604
Number of Classes	1058	217	1275
Number of Packages	312	41	353
Total Lines of Code	118054	15428	133482
Method Lines of Code	78632	9640	88272

#### 9.5.2.1.2 Module / Component structure

Figure 117 provides an overview of the SCORE framework's major packages/modules. The user interacts with SCORE through (in blue) either the SCORE Analyser which provides the user interface

presented earlier in this chapter, or via a standalone application which draw on SCORE Analyser functionality.



**Figure 117: SCORE Architecture / Component Overview**

In either case, the application will communicate with (in yellow) the SCORE Controller which mediates between the application and the various SCORE Analyser modules. The SCORE Analyser or stand-alone application can request different modules from the controller and then utilise the modules' data to implement its functionality.

The Line History module provides access to Line Histories, as well as providing a View to examine those histories (see 6.2.2.2).

The Coding module (described in more detail in Section 5.3) provides Views to code individual Changes, as well as storing and loading coding data and providing a summary of that data on request.

Both the Coding and Line History modules can call on the Graphing module (which uses JfreeChart, an open-source Graphing and Charting library) to produce graphs of the occurrence frequency of modification types or coding categories over time.

The Note-Taking module provides the ability to store notes on file versions or files as described in Section 4.4.2.

The Compilation and Editing module provides functionality relating to the compilation and execution of different project versions, as well as editor views which allow for the editing and execution of project source code as described in Section 4.4.3.1.

The Segment analysis module provides the researcher with facilities for describing the Project History via Segment as described in Section 6.3.4; the Segment machine-generation module automatically generates candidate Segments for analysis as described in Section 7.4.5. It uses line history data and compilation data from the history and compile-control modules to generate Segments, and data from the Segment module to evaluate machine-generated Segments against if requested.

### 9.5.2.1.3 External dependencies

The SCORE Analysis framework requires several open-source libraries and packages:

- JFreeChart<sup>47</sup>: Used to draw Graphs in the **Graphing** module
- Apache Derby SQL Database<sup>48</sup>: Used to create and access databases for storing Segments in the **Segment** module
- CDT CPP Parser (part of the Eclipse project<sup>49</sup>): A part of the standard Eclipse C++ IDE distribution, it is used to parse source code for the Machine-Segmenting Parse extension algorithm (see Section 7.8.6) which is part of the **Segment Machine-Generation** module
- JExcelAPI<sup>50</sup>: A library for programmatic creation of Microsoft Excel files; used by various components which include write-to-excel-document functionality
- Regular Expressions Parser: A slightly modified version of this parser is used in the **Segment Machine-Generation** module to provide the expression functionality (see Section 7.8.9)
- RsyntaxPane<sup>51</sup>: A Swing component which provides Java syntax highlighting, used in the main Diff view as well as the **Compile Control**'s editor view
- TableLayout<sup>52</sup>: A Swing layout alternative to other popular layouts such as the GridBagLayout. Used in several modules as well as the SCORE Analyser's core UI.

---

<sup>47</sup> JFreeChart: An open-source java library used for creating charts (<http://www.jfree.org/jfreechart/>)

<sup>48</sup> Apache Derby: An open source relational database implemented in Java (<http://db.apache.org/derby/>)

<sup>49</sup> Eclipse: A popular open-source java-based IDE available for many programming languages ([http://en.wikipedia.org/wiki/Eclipse\\_\(software\)](http://en.wikipedia.org/wiki/Eclipse_(software)))

<sup>50</sup> JExcelAPI: An open-source library which can read, write and create Microsoft excel spreadsheets (<http://jexcelapi.sourceforge.net/>)

<sup>51</sup> RSyntaxPane: An open-source JEditorKit for the development of java swing source-code editors (<http://code.google.com/p/jsyntaxpane/>)

<sup>52</sup> TableLayout: A java layout manager (<http://java.sun.com/products/jfc/tsc/articles/tablelayout/>)

- Apache Commons IO<sup>53</sup>: Provides functionality to calculate the Levenshtein distance between two strings. Used in the **History** module and **Segment Machine-Generation** module for string similarity comparisons.

### 9.5.2.2 SCORE Data Model

This section provides an overview of how data is transferred between different SCORE modules. A visual representation is provided in Figure 118. Consumers (shown in red) like SCORE View managers or standalone applications require data to populate their views or carry out requested functionality such as machine generation of Segments. Such data is provided by the different modules discussed earlier. For example Machine Segment generation requires access to line history data from the history module, compile status data (for the compile filter, Section 7.8.1) from the compile control module and Segment data for evaluation purposes from the Segment module.

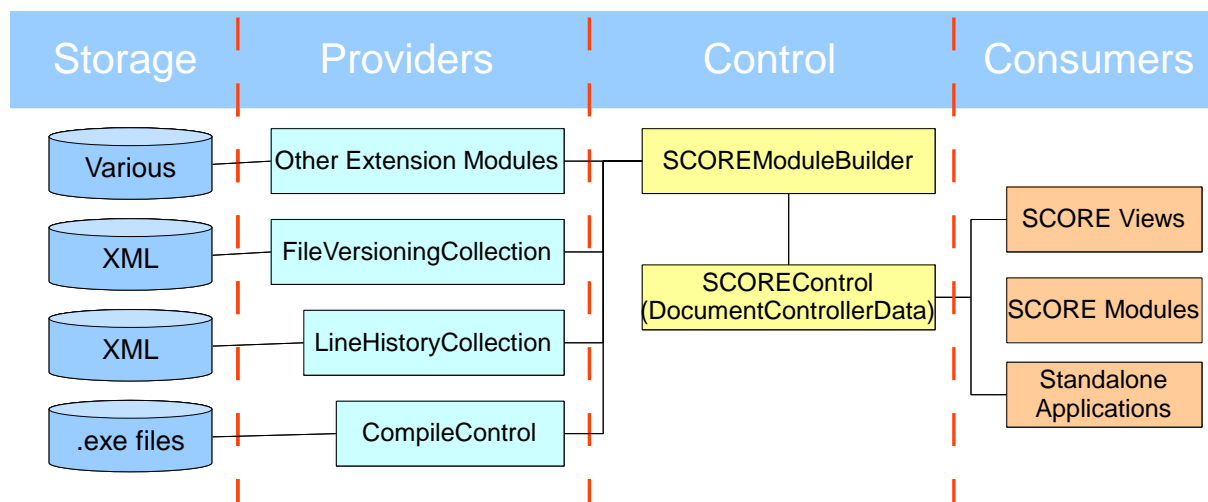


Figure 118: SCORE Data Model

To access data, the consumer requests access to the data controller from the SCORE control (shown in yellow). The SCORE control either returns the controller if it is available or uses the Module Builder to create an instance of the controller and then returns that instance. For each module, the module's author must ensure that it includes a method for the module's construction in the Module Builder, including any initialization tasks required. The consumer can then request data from the provider module. There is currently no standardised way of data transfer, so implementation of a consumer's data requests will require knowledge of the provider's data-getter methods. In the instance of the machine-segment generation module, it will request access to the history, compile control and Segment modules. It will then request data from these modules as needed; data from

<sup>53</sup> Apache Commons IO: A library which provides IO utility functions for the development of IO-related functionality (<http://commons.apache.org/io/>)



the history and compile modules will be used during Segment generation, while Segment module data will be used during the evaluation of generated Segments.

Providers (in light blue) provide their own mechanisms for storage and retrieval of data from storage sources (in dark blue). For example, while file versioning and history modules currently use XML documents to store data, the segment module uses an SQL database (via Apache Derby) to store segment-related data. Providers are also often consumers. For example, the **history** module provides Line History data, but to generate that data it requires (consumes) data from the **versioning** module. There is no fundamental difference between ‘core’ provider modules such as the **history** or **versioning** modules and other extension modules such as the **graphing** module.

### 9.5.2.3 SCORE User Interface Model

This section describes the mechanism by which views are implemented in the SCORE Analyser. All Views are part of the SCORE GUI (in light blue in Figure 119) which provides the user interface for the SCORE Analyser. The SCORE Analyser in turn relies on the SCORE Control to manage data transfer and event handling between different modules.

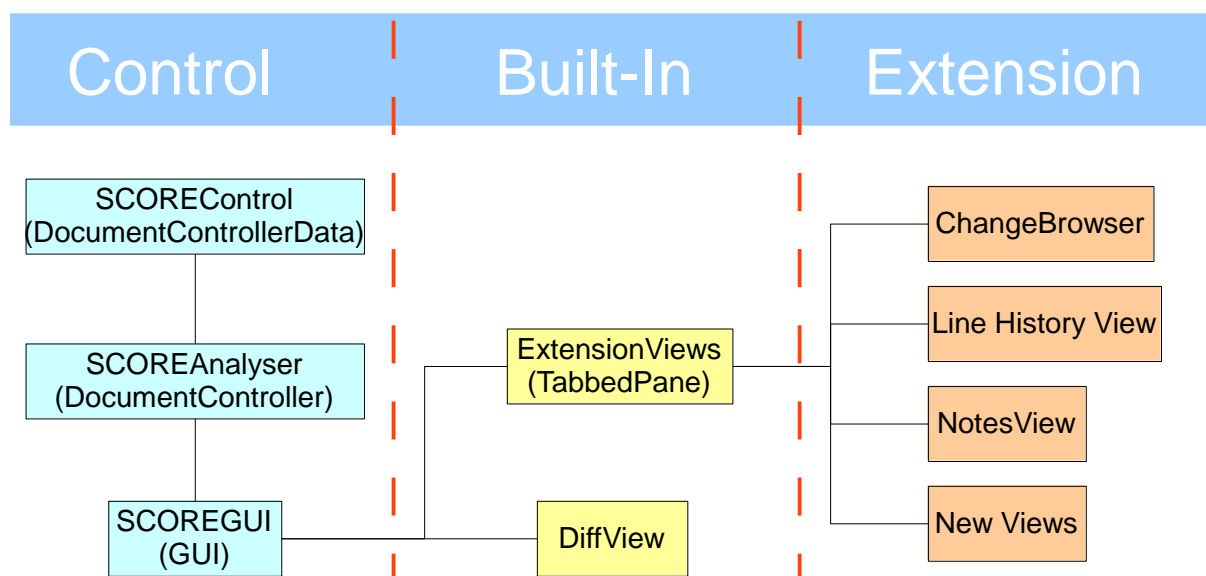


Figure 119: SCORE User Interface Model

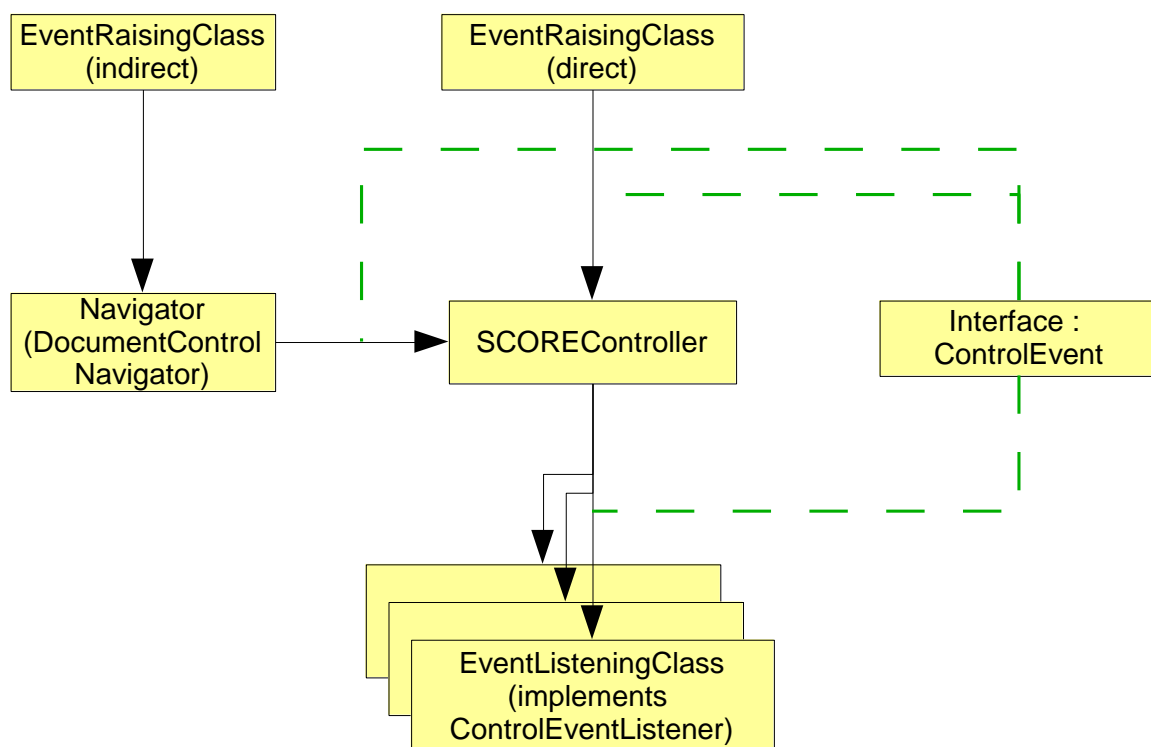
The SCORE GUI’s main view is the Diff View (Section 4.4.1, in yellow) which provides a diff-style comparison view between the current and previous version that are part of the currently selected Change. All other views are either shown in or launched via the ExtensionViews pane. The ExtensionViews manager allows modules to register Views which must be provided in form of a tabbed panel. This tabbed panel is then added to a tabbed pane on the right-hand side of the main

user interface. For example, the Notes module provides a tabbed view which is displayed in the main user interface's tabbed pane.

To implement Views that require more screen space a launch button (either as part of a separate tabbed panel view, or as part of the External View Opener View) to create a JFrame containing the View. In either case, the View's control can then access data from modules via the SCORE Control to populate the View. For example, the History module provides a View in a separate JFrame. The View can be accessed by pressing the "Open Mutant View" button in the *External View Opener View*; this opens a separate JFrame showing all Line Histories.

#### 9.5.2.4 SCORE Event Model

This section describes the event-handling mechanism which allows SCORE modules to communicate with one another (see Figure 120). The SCORE Control acts as the central distribution mechanism for events. Events must implement the ControlEvent interface which can be extended to package any data relevant to the event.



**Figure 120: SCORE Event-Dispatching and Handling Mechanism**

An event-raising class creates an event by implementing the ControlEvent interface. This event is then sent to the SCORE Control via a method. The SCORE Control sends the event to any registered listener. Objects can be register to receive events by calling the relevant SCORE Control method,

providing the object as the parameter, and having the object implement the `ControlEventListener` interface.

The SCORE Control does not itself analyse events with one exception. When a *NewChangeViewedEvent* (indicating that the Change being viewed has been changed) is processed, the SCORE Controller will update its own internal variable storing which Change is currently being displayed in the main SCORE Analyser Diff View. This data can be retrieved from the SCORE Control via a getter function.

In addition to directly sending events via the SCORE Controller, objects can also request access to a global utility object called the *navigator*. The *navigator* provides methods for moving the SCORE Analyser's interface to a different Change, either by moving forward or backward a number of Changes or by specifying the exact Change to move to. An object can also directly create an event to fulfil the same purpose; the navigator is merely a utility object which crafts the appropriate *NewChangeViewedEvent*.

Modules that register as event listeners are free to react to or ignore events as they please. However, there is currently no mechanism to prevent cycles in event dispatching from occurring if modules sent events in reaction to one another, so any event dispatching should be accompanied by safety measures to prevent cycles, such as disabling event-receiving functionality until the SCORE Control returns from the event-posting action.

An example of a use of the event mechanism is the selection of Line History modifications. If a line is clicked in the SCORE Analyser's main Diff View, the Diff View posts a *Line History Modification Selected* event. If the Line History View is active, it will have registered with the SCORE Analyser to receive events. Upon receiving the *LineHistoryModificationSelected* event, the Line History View will show the associated Line History, highlighting the row containing the selected Modification.

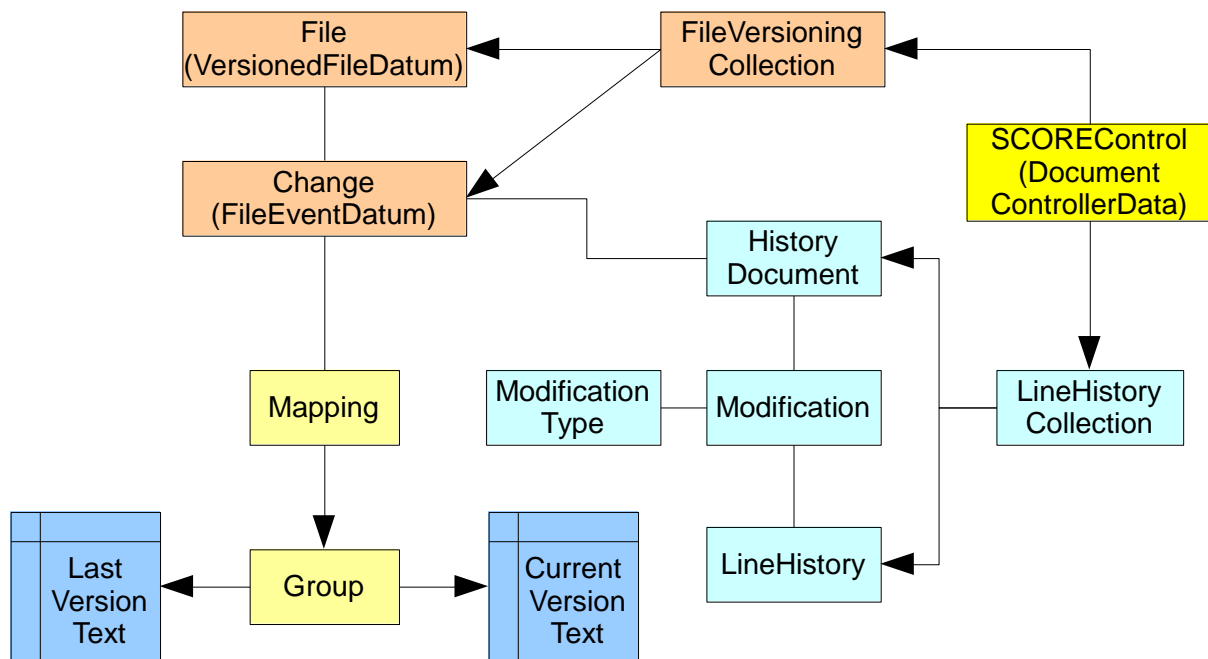
Implemented events:

- *NewChangeViewedEvent* (*FileEventChanged*): This event is sent by a module indicating that the Change displayed in the main Diff View has been changed. This event triggers store/load actions in modules which display information relating to the Change shown in the main Diff View, such as the Note module which when receiving the event will store the currently displayed notes for the previously displayed Change and load the notes for the currently displayed Change.

- **SaveTriggered:** This event is sent by a module to request that all modules save any modifications that have been made to their data and resolve any 'dirty' states. It can be triggered manually by pressing the 'Save' button.
- **ControlDeactivated:** This event is sent to controls to indicate they have lost focus. This may cause behaviour such as storage of currently loaded data.
- **ProgramClosing:** This event is sent by the SCORE Analyser before it terminates execution to allow modules to perform any necessary clean-up or saving actions.
- **MetaDirChanged:** This event is sent by the SCORE Analyser when a new project is loaded. Modules should ensure they load data related to the new project, while cleanly disposing of any data related to the old project.
- **LineHistoryModificationSelected:** This event indicates that a Line History modification has been selected. The Diff View and the Change Browser View send this event when a line is clicked. It causes modules such as the Line History View to display data relating to the selected modification.
- **Refresh:** The refresh event can include a set of reference objects to specify which modules/Views are intended to refresh themselves. For example, if a screen capture is taken, a Refresh event relating to the thumbnail viewer is sent. In response, the thumbnail viewer will update its displayed screen capture.
- **ChangeGroupSelected:** The Change Group Selected event includes a vector of Changes which modules interested in the event should display. For example, the Change Browser View will load the Changes' modifications.

### 9.5.2.5 SCORE File Versioning and Line History implementations

This section presents an overview of the implementation of classes to hold data relating to versioning and Line Histories. Figure 121 illustrates the relationship between different elements. The FileVersioningCollection and associated classes (in red) serve to store data for the versioning module. This data is generated in the first phase of Line History generation, and consists of the mapping of lines from one version to the next. Objects of the File class store data relating to files created and modified in the project's history. Each File object contains one Change object for each modification made to that file's text in the Project History. A Change object maps each maintained, mutated or moved line from the previous version's document to a line to the current version's document and keeps track of added and deleted lines. However, the data captures modifications only in the context of these two versions and does not know about a line's history outside these two versions.



**Figure 121: Versioning and History Data Model**

The Line History collection (storing data for the history module) extends the relation of lines stored in the File Versioning collection from between pairs of versions to the entire Project History as described in Section 6.2.1.2. After Line History generation, each Change will be associated with a HistoryDocument which stores all modifications made during that Change. Modifications are also stored on a line level, with Line Histories storing all modifications made to a single line. By following these links, it is possible to retrieve the Line Histories of all lines modified in a Change or all Changes in which a line is modified. A modification stores the HistoryDocument in which it occurs, the Line History to which it belongs, the line number at which it occurs in the History Document and the modification type (either *Added*, *Deleted*, *Moved*, *Mutated* or *Ghost*).

Both the FileVersioningCollection and the LineHistoryCollection can be accessed via the SCORE Control. The FileVersioningCollection is loaded when the SCORE Analyser is launched, as the Diff View requires it to display the diff view showing modification between successive versions. The history module and Line History Collection are loaded as needed, for example when the Line History View is opened.

### 9.5.3 A Short SCORE Manual

#### 9.5.3.1 Introduction

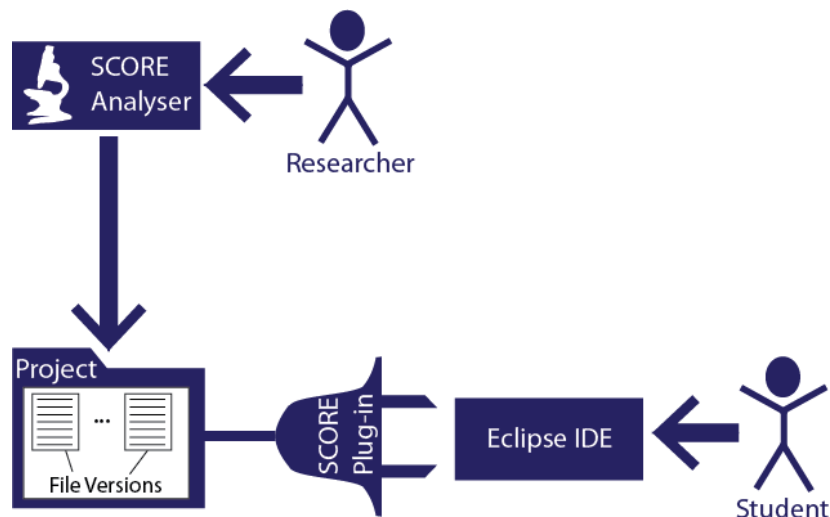
The SCORE toolset was developed as part of this doctoral dissertation for the purposes of identifying and analysing student problems with Computer Graphics programming in detail. It allows the researcher to explore Project Histories which store data much like a version control system except

that every modification is recorded automatically rather than storing only those modifications at which students perform a commit action. Through the analysis of all programming actions the researcher can develop a grounded understanding of student programming and identify student problems as well as illuminating the student problem-solving process.

The SCORE toolset consists of two components; the SCORE Eclipse plug-in which captures Project Histories as students work on their assignments, and the SCORE Analyser which is used to explore and code that data for evaluation.

The SCORE plug-in is used to recording student programming at a fine-grained level. It generates Project Histories containing one Change for every save or compilation action that modifies the text of a file that is part of the project. The SCORE plug-in is written for the Eclipse IDE. Currently the plug-in is configured to capture only C/C++ files but it could be extended for use with other programming languages which are compatible with the Eclipse platform. The SCORE plug-in uses a local storage model instead of communicating with a server, with the versioned code automatically being submitted by the students with their assignments since it is stored in a sub-directory of the main project directory.

This manual will discuss the SCORE Analyser functionality and provide instructions for utilising the SCORE Analyser for analysis. The SCORE Analyser is used to examine Project Histories stored by the data-gathering plug-in. Figure 122 shows the relationship between the Analyser and the SCORE plug-in. As the student works on her assignment the plug-in captures versions of the source code and stores them in the project directory. When the student submits the project as a zip file, the researcher can utilise the SCORE Analyser to explore the Project History. A demonstration video showcasing the SCORE Analyser functionality is also available as a YouTube playlist at: [http://www.youtube.com/watch?v=cvF5lCZtYbQ&list=PL5A1E9556055F67E2&feature=plpp\\_play\\_all](http://www.youtube.com/watch?v=cvF5lCZtYbQ&list=PL5A1E9556055F67E2&feature=plpp_play_all) (Wittmann, 2012b).



**Figure 122: Relationship between SCORE, the student and the researcher**

### 9.5.3.2 Installing and Launching

#### 9.5.3.2.1 Installing the SCORE Analyser from a zip file

If the SCORE Analyser is distributed as a zip file to install the SCORE Analyser, simply uncompress the zip file in the desired directory.

#### 9.5.3.2.2 Installing the SCORE Analyser from a directory

If the SCORE Analyser is distributed as a (non-compressed) directory, it can be installed by copying the root folder into the destination directory. The SCORE Analyser can also be run directly from the installation medium (such as a DVD) though any functionality requiring the saving of data will be unavailable and may cause the SCORE Analyser to terminate unexpectedly. The application may also run slower. For optimal performance, installation onto the local hard drive is recommended.

#### 9.5.3.2.3 SCORE Analyser Package Contents

The SCORE Analyser installation contains the following:

- SCORE\_XX\_XX\_XXXX.jar: SCORE Analyser as an executable jar file (bundled with all its dependencies) called,
- SCORE.bat: a batch file which launches the analyser,
- ./Comp330\_Workspace: ten Project Histories analysed in the thesis dissertation “[THESIS TITLE AND LINK HERE](#)”
- Config.cfg: The main configuration file
- ./ConfigurationFiles: The Project History configuration files

#### 9.5.3.2.4 Main Config File (Config.cfg)

The Config.cfg file contains the following options:

- Generation Settings

- useGhost: whether to use Ghost lines in Line History Generation
- useMoved: whether to use Moved lines in Line History Generation
- distMethodToUse: which distance measurement to use: LEVENSTHEIN or LCS (Longest Common Subsequence). Should be set to LEVENSTHEIN.
- ignoreWhitespace: whether to ignore whitespace during Line History Generation
- Ghost Generation settings
  - ghostLev: the Levenstein distance used in Ghost Generation
  - maxGhostAge: the size of the ghost window
- Autosave
  - autosaveOn: whether save reminders should be displayed at regular intervals
  - autosavePeriod: period (in milliseconds) at which autosave dialog will be displayed
- scoreOutputDir: the directory to which SCORE writes output
- General Options
  - showExperimentalControls: whether to show experimental (buggy) controls (recommended: false)
  - showObsoleteControls: whether to show obsolete controls (recommended: false)

#### 9.5.3.2.5 Project History Config Files

The SCORE Analyser can be configured at three different levels:

- Overall Configuration for all projects (OverallConfig.cfg)
- Configuration for one project (e.g. one assignment) (GroupConfig.cfg)
- Configuration for an individual Project History (\_StudentNameHere\_.cfg)

The following can be set at each configuration file level. At the overall level it will affect all Project Histories, at the project level it will affect all Project Histories which are part of that project (in the project's directory) and at the individual Project History level they will affect only that Project History:

- Setting up RegExps (Dependencies): Regular expressions which are applied to files when they are being compiled. Most commonly used to change paths.
- Setting up libraries and include files for compilation (CompileConstants)

In addition, the project configuration file specifies the path to the directory containing the project's Project Histories (rootdir) whereas the individual Project History configuration files specify the relative path from that directory to the individual Project History (projectdir / metadir).

Configuration files are correctly set up for the ten included Project Histories. No changes to these files should be necessary.

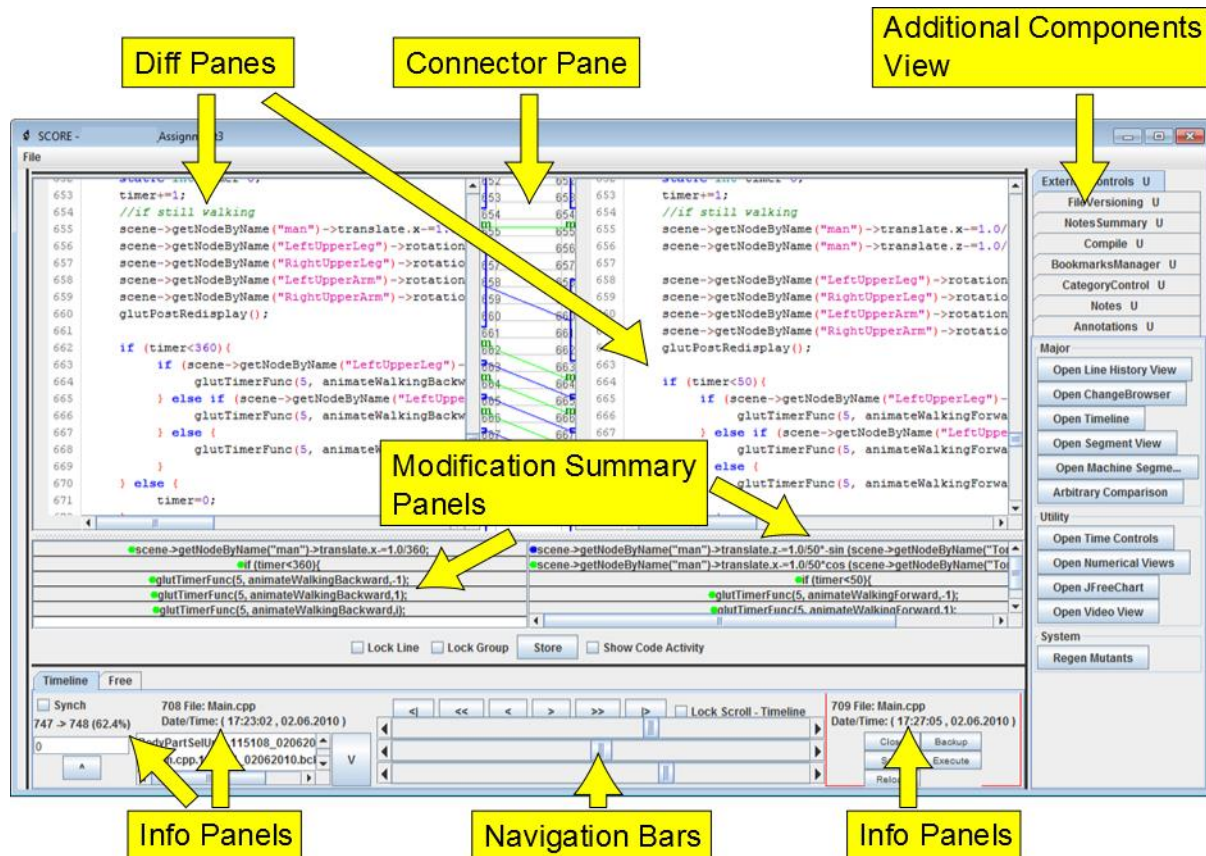
### 9.5.3.3 Main View

#### 9.5.3.3.1 Overview

The SCORE application is designed to allow researchers to better understand student programs as a sequence of file changes. At its core lies its ability to visualise these file changes, which in turn enables the researcher to understand, interpret and annotate sequences of changes. As can be seen



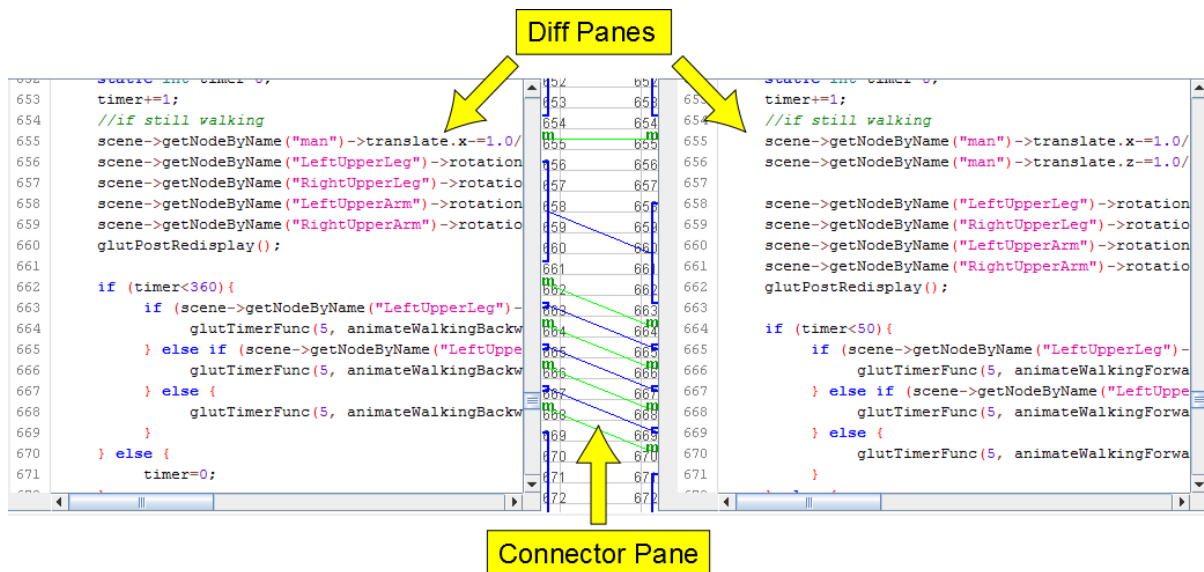
in Figure 123 the SCORE Analyser's main interface consists of the 'Diff Panes', 'Connector Pane' and 'Modification Summary Panels' which show the current and previous version of the source code, the 'Info panels' which show the position and timestamp of the current and previous version, the 'Navigation Bars' which allow the user to navigate to different versions of the project and the 'Additional Components View' which contains additional Views.



**Figure 123: The Main View includes the central Diff View and Navigation Bars and Info Panels as well as the "Additional Components View" which includes additional views and components**

### 9.5.3.3.2 Diff View + Connector Panel

The SCORE Analyser's main interface is a diff-style document version comparison view as shown in Figure 124 called the 'Diff View'. Through the application of *Line Histories* described in more detail in Section 6.2.1, the 'Diff view' matches not only lines maintained from the previous to the current version as would a regular Diff view but also moved lines, modified (Mutated) lines as well as lines that are re-introduced from a previous document in which they were deleted (Ghost).



**Figure 124: The Diff Panes and the Connector Pane which shows which lines from the old version lines from the new version map to**

The 'Diff View' displays a Change's previous and current version's source code text in two text areas, and the connector pane (in the centre) displays connectors which link groups of moved, maintained or mutated lines from the left (old) to the right (new) document. Maintained and moved groups of lines are linked by blue bracket connectors, while mutated groups are linked with lines marked with a green 'm' character. Ghost lines are marked with a purple 'g' character. Deleted lines (unmatched lines in the left document) and added lines (unmatched in the right) have no connectors.

### 9.5.3.3.3 Modification Summary View

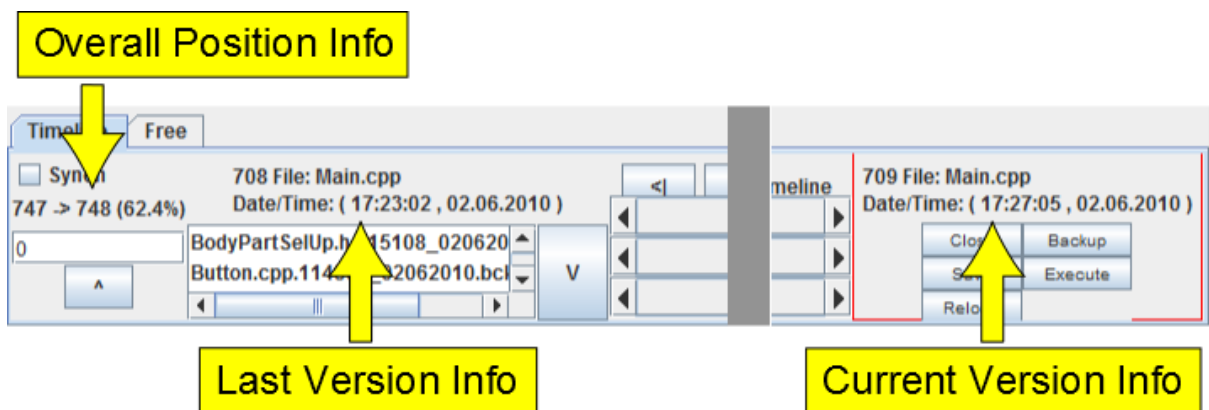
Below the difference view is the 'Modification Summary View' comprised of two summary panels which contain the list of modifications from the previous to the current version as shown in Figure 125. Clicking one of these changes scrolls the diff view to the line involved in that modification. Mutation modifications are marked with a green dot, additions or deletions with a blue dot (deletions appear in the left panel only, additions in the right panel only) and ghost modifications with a purple dot. Moved lines are not currently displayed in the summary view, since a 'Move' may involve many lines and would clutter the list. Clicking on a line also sends an event to all of the SCORE Analyser's components, requesting information for that line to be displayed in the component if appropriate. For example the 'Line History View' (see Section 9.5.3.7) will scroll to the Line History of a line when clicked in the 'Modification Summary View'.

●scene->getNodeByName("man")->translate.x=1.0/360;	●scene->getNodeByName("man")->translate.z=1.0/50*.sin (scen
●if (timer<360){	●scene->getNodeByName("man")->translate.x=1.0/50*cos (scen
●glutTimerFunc(5, animateWalkingBackward,-1);	●if (timer<50){
●glutTimerFunc(5, animateWalkingBackward,1);	●glutTimerFunc(5, animateWalk
●glutTimerFunc(5, animateWalkingBackward,i);	●glutTimerFunc(5, animateWalk

**Figure 125: The Modification Summary panel displays all mutation, addition, deletion and ghost modifications occurring between the two versions**

#### 9.5.3.3.4 Info Panels

The SCORE Analyser has three 'Info panels' as shown in Figure 126. The 'Overall Position Info' shows the overall position of the current version of the file, the previous version of the file and position of the version in terms of what percentage of versions precede it: *PreviousVersionOverallPosition -> CurrentVersionOverallPosition (PercentageVersionsBefore%)*

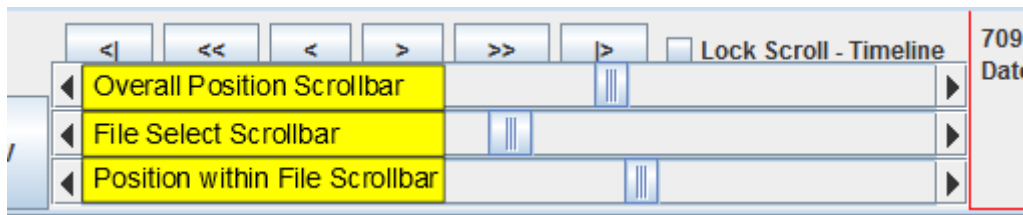


**Figure 126: The info panels which describe the current and previous version shown in the Main Diff view**

The 'Last Version Info' and 'Current Version Info' panels show the position of the version as part of the file's history (so the 54<sup>th</sup> modification to the file will be at position 54) as well as the file name and the timestamp associated with that version.

#### 9.5.3.3.5 Navigating from Change to Change

To navigate between a Project History's Changes SCORE provides navigation by overall position of the change or navigation through changes for a selected file via scrollbars shown in Figure 127. To move to a Change specified its overall position, use the top scrollbar. To select a file and then navigate through the versions of that file only first select the file via the second scrollbar then navigate through its versions with the third scrollbar.



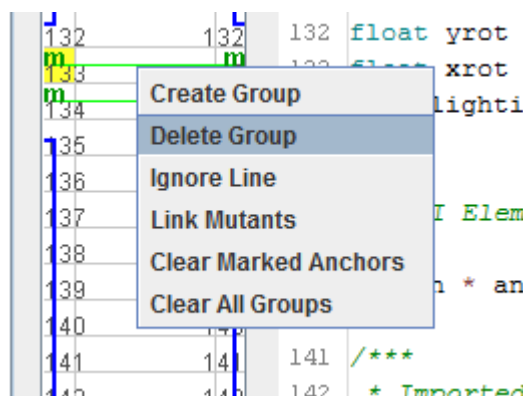
**Figure 127: The scrollbars are used to select the version to be displayed in the main Diff view**

#### 9.5.3.3.6 Correcting Line History Generation

Machine Generation of Line Histories is described in detail in Section 7.2. However as is discussed in the thesis the generation algorithm does not produce perfect line mappings in every instance (though results are very good).

If the researcher detects an error in the line mapping (two lines having been detected as mutants of one another when they are not, or two lines not having been detected as mutants when they are, for example) while analysing the Project History this error can be corrected in the “Connector Pane”.

To remove an existing mapping, select the line(s) to be unmapped by left clicking them; this will highlight the line’s anchor in yellow as shown in Figure 128. Right-click in the middle of the ‘Connector Pane’ which will open a context menu. Choose the “Delete Group” option. The line(s) will be unmapped, and the connector will disappear.



**Figure 128: Deleting a group**

To create a new mutant mapping between two lines left-click one of the line’s anchors and drag the mouse to the other line’s anchor and release the mouse. This will establish a mutation connection. While the mouse is being dragged a black connector line will be displayed as shown in Figure 129.



Figure 129: Creating a mutant connection

To create a “maintained” or “moved” mapping click the anchors of the first and last line of the maintained/moved group to be created on the left-hand and right-hand sides as shown in the left panel of Figure 130. Then right-click the “Connector Pane” and select “Create Group”. This will create the group as shown in the right-hand side panel in Figure 130.

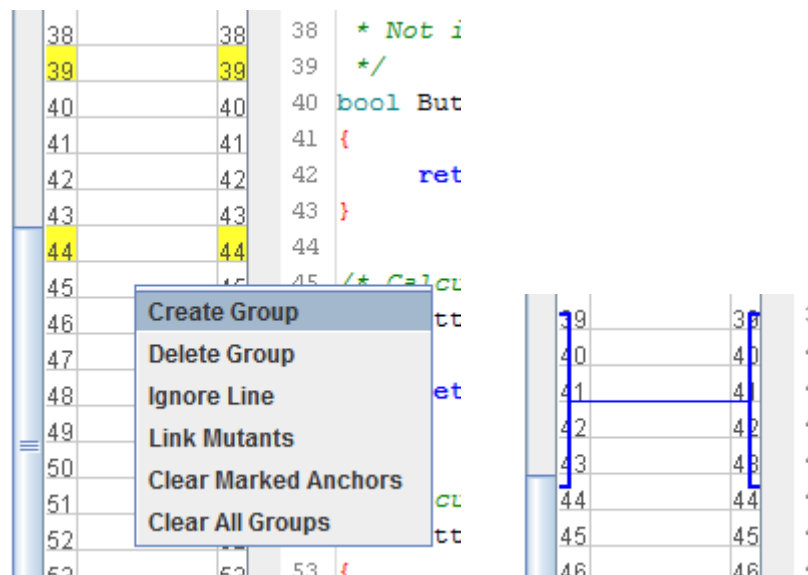


Figure 130: Creation of a group (Left); the resulting group (Right)

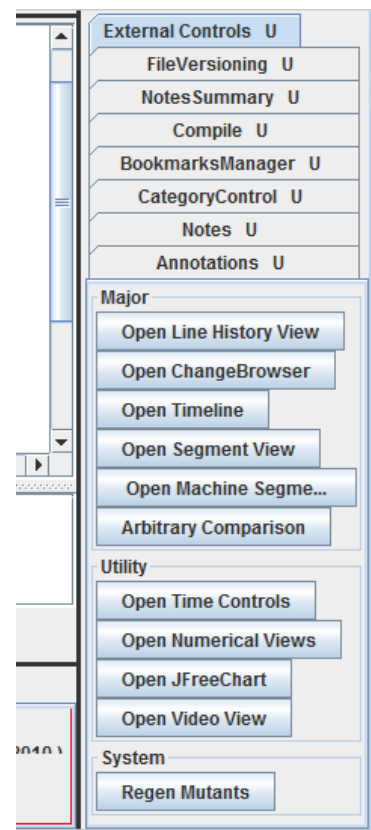
After completing the remapping of lines, press the “Save” button in the main interface to save the modifications. Modifying the mapping will require a regeneration of all Line Histories as the current Line Histories will be corrupted and this will lead to errors when attempting to access data of a Line History whose lines have been modified. To regenerate Line Histories press the “Regen Mutants” button in the ‘External Controls’ tab of the ‘Additional Components’ pane.

#### 9.5.3.3.7 Additional Components Tabbed Pane

The tabbed pane on the right-hand side of the main window (shown in Figure 131) contains additional tabbed views as well as buttons which launch additional views in their own windows.

The 'External Controls' tab pane is used to launch the 'Timeline View' (Section 9.5.3.4), 'Change Browser View' (Section 9.5.3.8), 'Timeline View' (Section 9.5.3.4), 'Segment View' (Section 9.5.3.10) and 'Machine Segmenting View' (Section 9.5.3.11).

The Notes tab pane shows the 'Notes View' (Section 9.5.3.5), the Compile tab pane shows the 'Compile Functionality' described in Section 9.5.3.6, the BookmarksManager tabbed pane shows the 'Bookmarks View' (Section 9.5.3.5.3), the CategoryControl tabbed pane shows the 'Categorising View' (Section 9.5.3.5.4) and the Annotation tabbed pane contains the 'Change-Coding View' (Section 9.5.3.9).

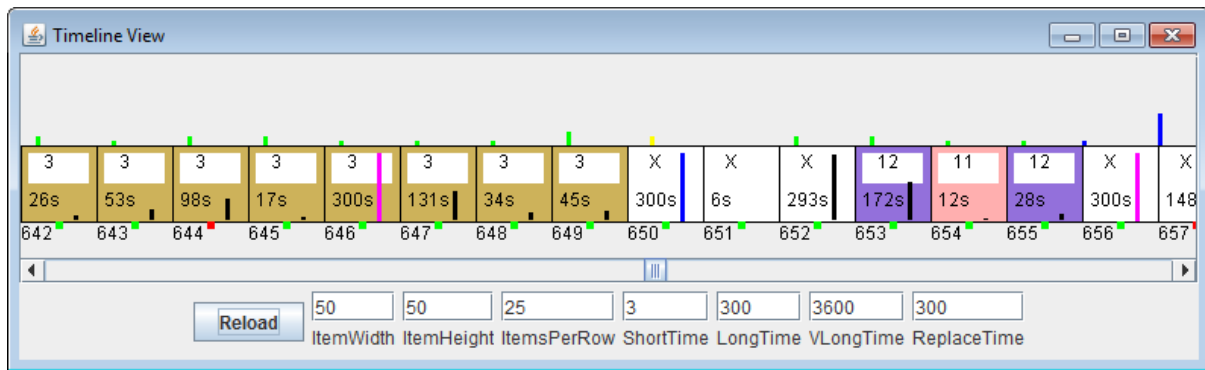


**Figure 131 The "Additional Components" pane**

## 9.5.3.4 Timeline View

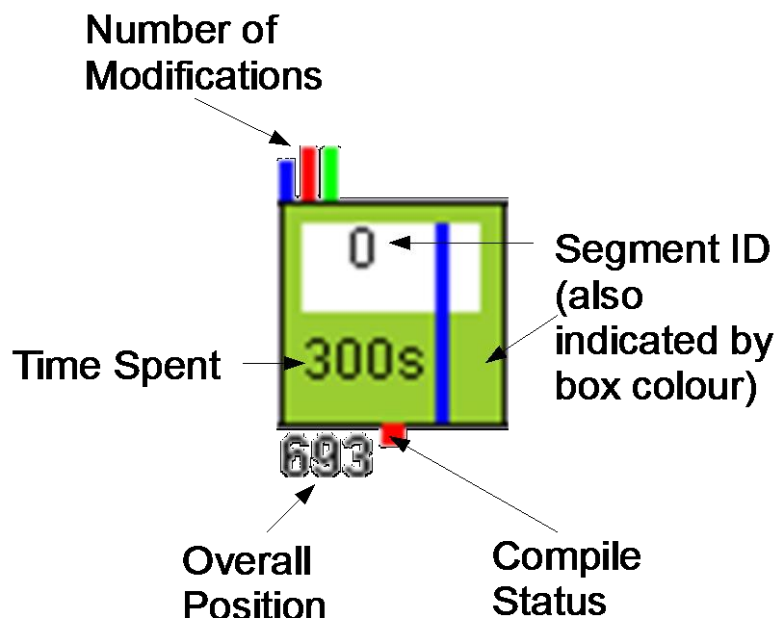
### 9.5.3.4.1 Timeline View

The Timeline View is launched via the 'External Controls' tab in the 'Additional Components' pane. When it is launched a popup combo box will request the user select one of the segmentation versions entered via the 'Segmenting View' described in more detail in Section 9.5.3.10. Segments of the segmentation version selected will be displayed in the timeline view via box colours and the id field described later. The 'Timeline View' provides a quick overview of how long students worked on Changes in a Project History and how many modifications students made Changes. As shown in Figure 132 the timeline consists of boxes each of which represents a Change.



**Figure 132: The Timeline View**

One such box is shown in Figure 133. The number below the box is the Change's overall position. Next to the overall position, the small coloured box indicates the Change's compile status (red for not compiled, green for compiled). Inside the box, the time taken by the Change is displayed as a number of seconds, with the bar next to the number providing a visual representation of time taken. The bars on top of the Change box show the number of additions (blue), deletions (red), mutations (green), moved (yellow) and ghost (purple) modifications occurring in that Change. The white box containing a number shows the id of the segment to which the Change is assigned. For example, in Figure 132 eight Changes are assigned to the same segment with id = 3. The segment id is also visually represented via the boxes brown colour, allowing for fast identification of 'related' Changes making up a segment.



**Figure 133: Representation of a Change in the Timeline**

#### 9.5.3.4.2 Timeline Details View

When a box in the timeline is clicked, the 'Timeline Details View' (shown in Figure 134) will be updated to show all of the lines modified in the associated Change, including the line's text, line number and modification type. This allows the researcher to quickly get an overview of the modifications made in that Change. When a box is clicked this also moves the main Diff View to the associated Change.

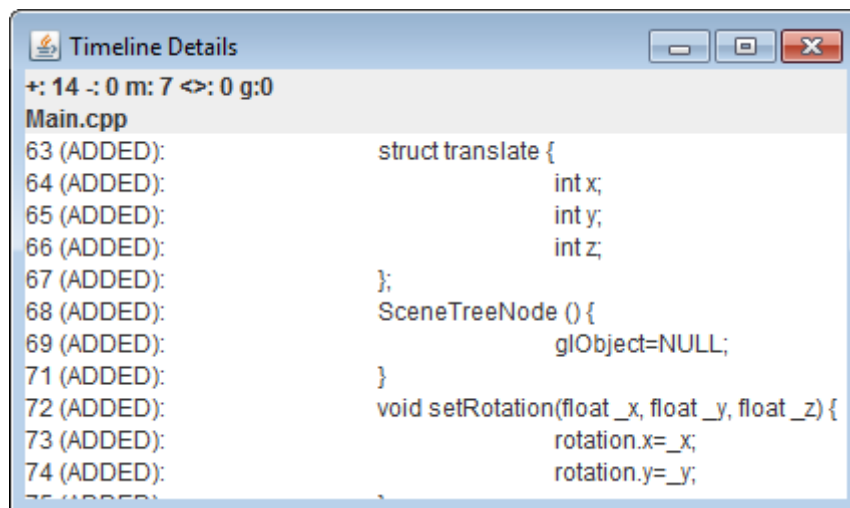


Figure 134: The Timeline Details view

### 9.5.3.5 Note-taking, Categorising and Bookmarking

#### 9.5.3.5.1 Note-Taking

The 'Note View' (Figure 135, left) is used to store text notes associated with individual Changes, with Versioned Files or with a project as a whole. The 'Note View' is updated every time a new Change is displayed in the main 'Diff View'. The Version field contains notes regarding the current Change, the File field contains notes containing the File to which the current version belongs, and the Overall field contains notes for the whole assignment (and is hence not changed when a different Change is viewed in the Diff Viewer).



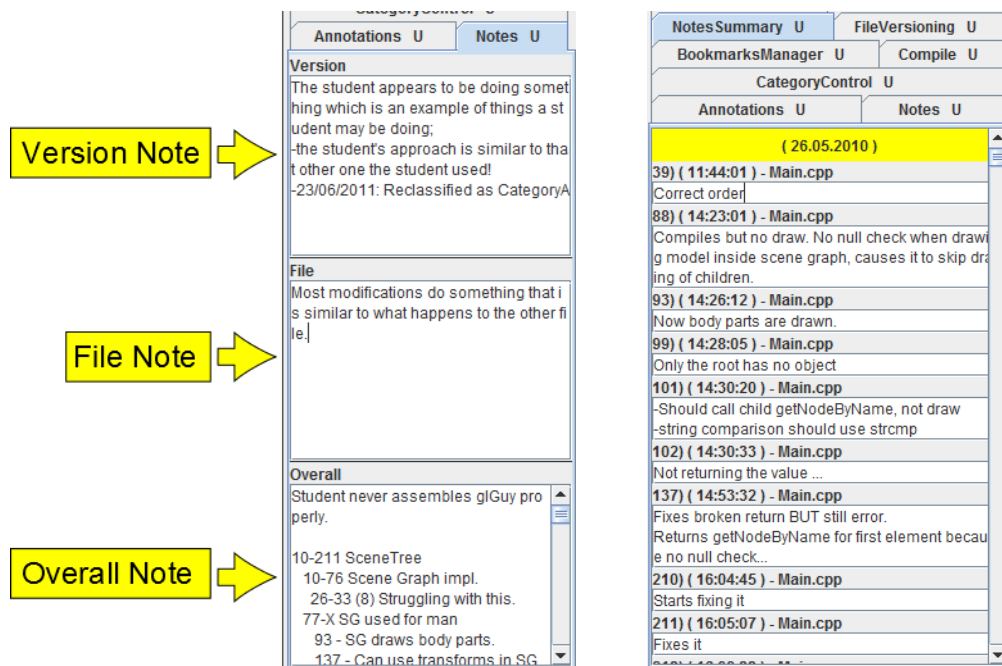


Figure 135: The Notes pane displaying notes for the version and file as well as the overall note (Left); the summary view displaying all of the version notes (Right).

The 'Note View' can be used in the GT memoing process, taking notes regarding individual Changes or the Project History as a whole during the analysis process and noting down any insights which occur, having them stored alongside the data being analysed.

The 'Note Summary View' (Figure 135, right) presents Version notes for Changes sequentially in a single window. Notes can be written to a summary text file. Notes are also displayed alongside Change data in some other components. Notes are stored as and loaded from text files.

#### 9.5.3.5.2 Other components that display notes (Change browser)

Some other components such as the ‘Change Browser’ (Figure 136) also display version notes. In the case of the ‘Change Browser’ these are displayed on the right-hand side of the View. Version notes in these views can be edited.



Figure 136: The Change Browser displays notes on the right-hand side

### 9.5.3.5.3 Bookmarking

The ‘Bookmark View’ (see Figure 137) allows the researcher to create bookmarks indexing individual Changes. A bookmark is created by entering the bookmark’s name and a short description in the ‘Bookmark Creation Fields’ at the bottom of the View. The bookmarked Change is the Change currently displayed in the main ‘Diff View’. When a bookmark is selected in the ‘Bookmarks List’, the main ‘Diff View’ displays the associated Change.

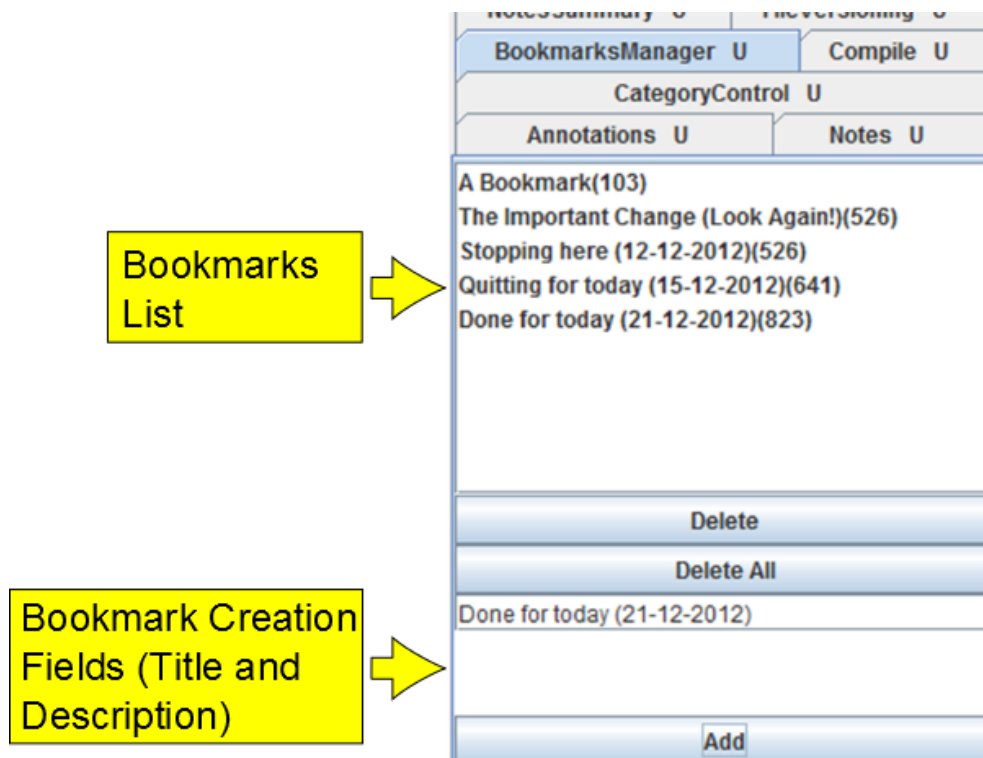


Figure 137: The Bookmark View

#### 9.5.3.5.4 Categorising

The 'Category View' (Figure 138) allows the researcher to create categories and then assign Changes to one or more categories. For example, categories might exist for user-interface implementation, drawing of icons and general programming related to creating data structures.

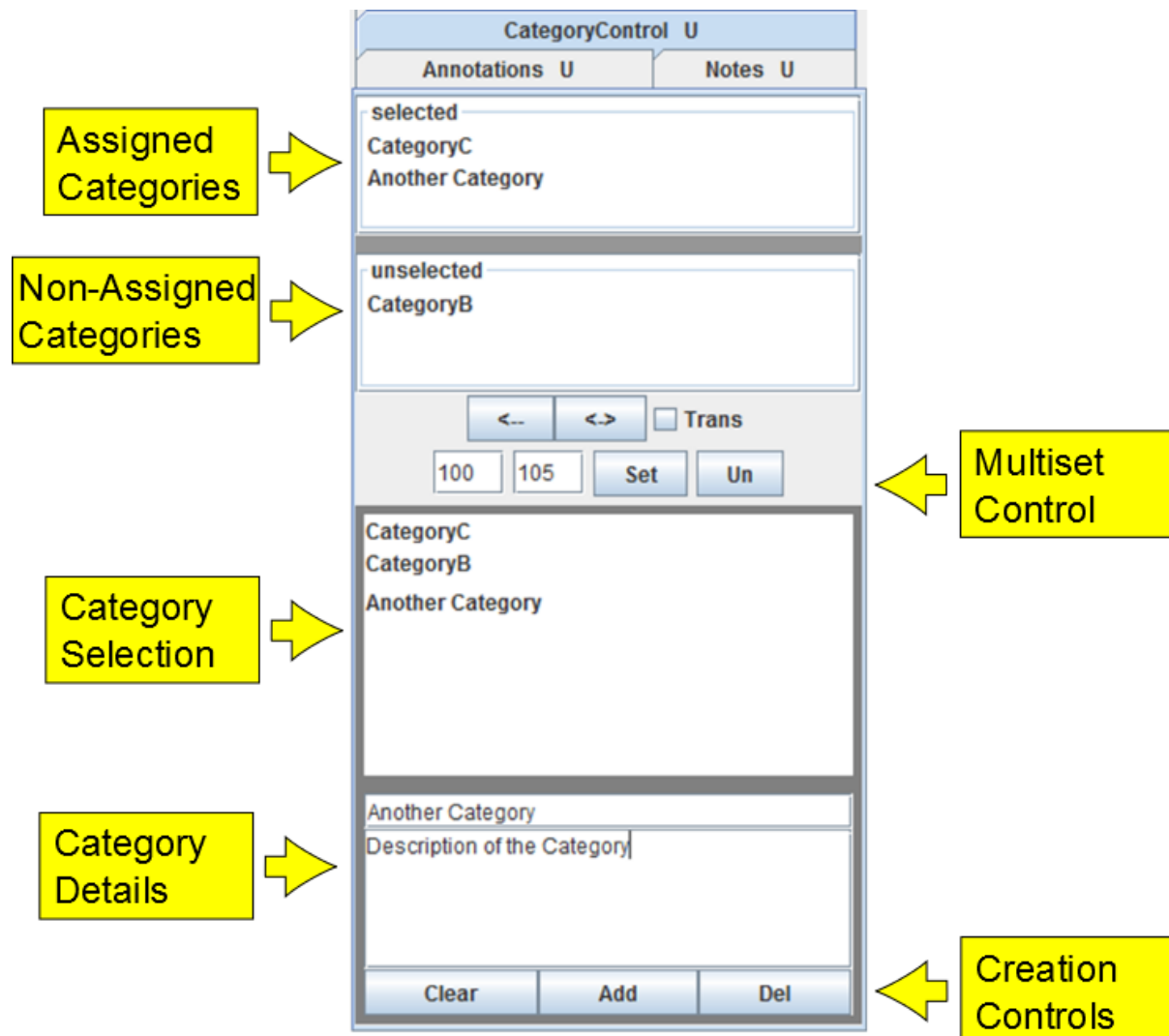


Figure 138: The Categorising View

To create a category enter a name and description in the 'Category Details' area (press clear before entering the details or else the modifications will be applied to an existing category) and press the "Add" button after which the category will appear in the 'Category Selection' area.

To assign a category to the Change currently visible in the main 'Diff View', select the category in the 'Non-Assigned Categories' list and double-click. To remove a category, select it in the 'Assigned Categories' list and double-click. The category will be moved from the assigned to non-assigned categories view or vice versa. To assign multiple Changes to a category, enter the range to be

assigned in the 'Multiset Control' and press "Set" to assign the category to the Changes or "Unset" to remove the category for the Changes.

### 9.5.3.6 Compile Functionality

#### 9.5.3.6.1 *Compilation and Execution of project versions*

Analysing a program by analysing its text only is difficult. Sometimes the effect of modifications is not apparent, or the researcher may not have confidence in her understanding of the effects of a particular modification. To resolve such issues the researcher should be able to execute the source code to observe the effect of modifications, and to debug the source code in cases where the nature of a modification still remains unclear after examination of source code and program output.

The SCORE Analyser can automatically compile each version of the project (producing one executable per project version), allowing the researcher to execute the program at any Change through the analyser interface. When a program version's execution is requested, SCORE will launch the executable if compilation was successful and will display any compile-time error messages or warning messages in a console window.

Compile functionality is found in the "Compile" tab in the 'Additional Components' tabbed pane as shown in Figure 139. To compile and execute a single Change, press the "Compile & Execute" button. To compile a range of Changes (or all Changes) enter the range in the 'Compiling a Range of Versions' fields and press "Compile From/To". Compilation is carried out in a separate thread so the SCORE Analyser remains functional during compilation. To interrupt compilation at any time, press the "Stop Compile" button. The Change currently being compiled is shown in the field below the buttons.

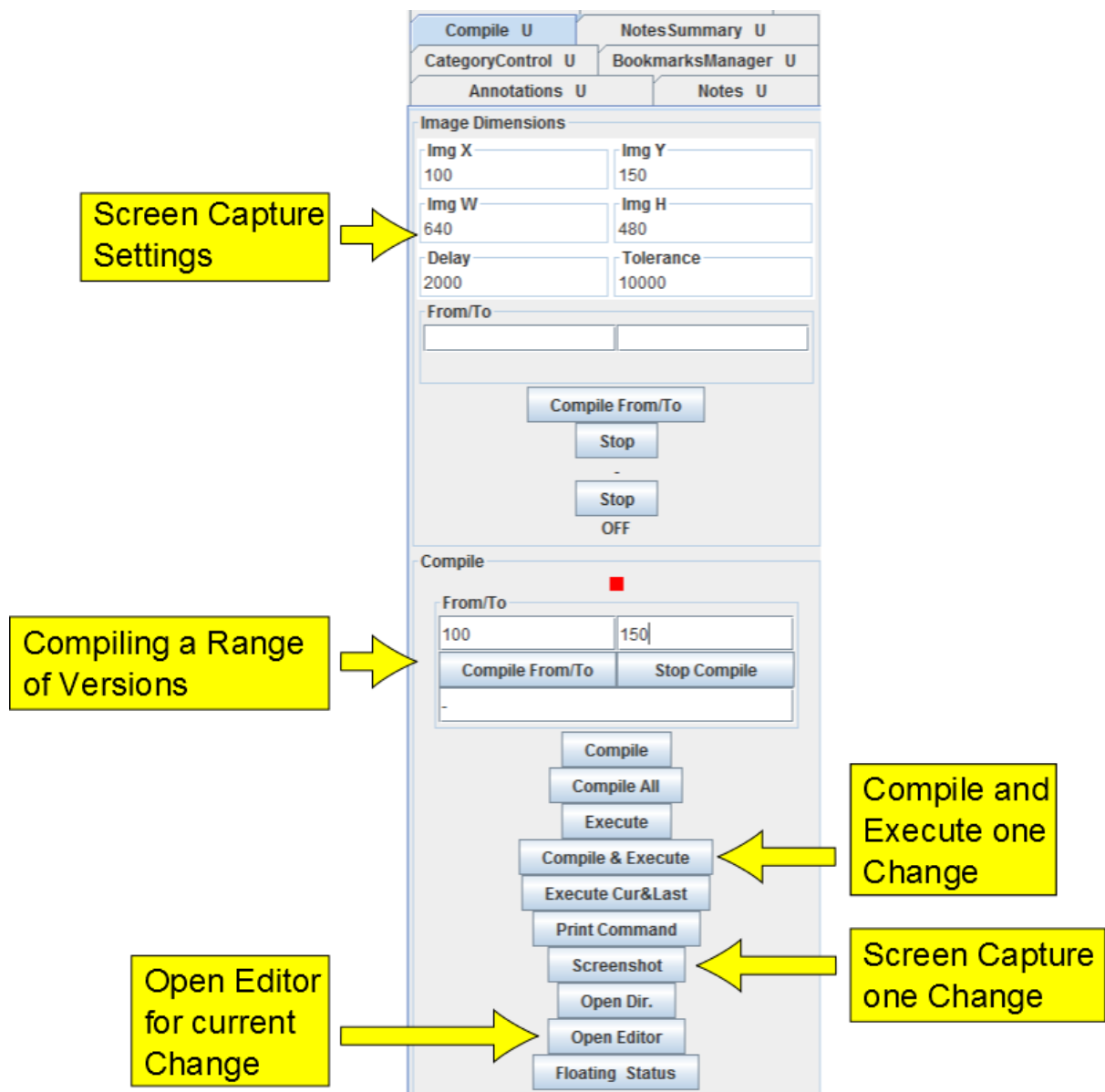


Figure 139: The "Compile" tab in the "Additional Components" pane contains all of the compilation, editing and screen capturing functionality

#### 9.5.3.6.2 Screen capturing project versions

For Computer Graphics programs, the main program output is visual. The SCORE Analyser can use compiled project version executables to automatically generate and store screen captures for every project version. The screen capture for the current Change is then displayed in a separate window (see Figure 140) and updated automatically when the currently selected project version is changed. This view is useful in giving the researcher a quick way to examine the program's output at that stage of the Project History; if this static view does not suffice the researcher can utilise the program execution functionality described earlier.

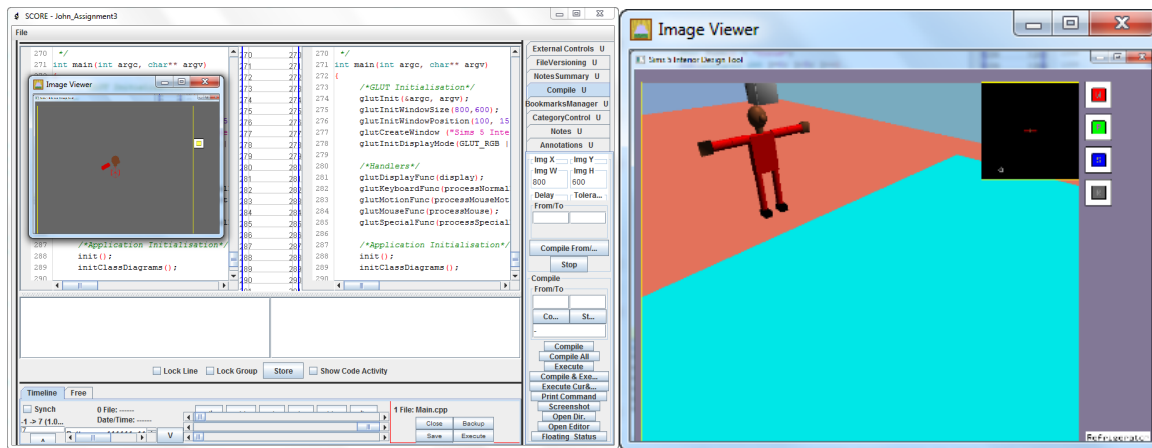


Figure 140: The Screen Capture window as part of the application (Left) and a close-up (Right)

For screen capturing, all Changes to be screen captured must first have been compiled. To screenshot a single Change press the “Screenshot” button in the “Compile” pane. To screen capture multiple Changes, enter the range of Changes in the “From/To” field of the “Screen Capture Settings” panel and press “Compile From/To”. To ensure successful screen capturing the settings must be updated to reflect the position at which the window is displayed, as well as the delay to apply before attempting the screenshot to allow the application to complete loading.

### 9.5.3.6.3 Editing versions with the Code Editor

When execution of the program is insufficient to debug and understand a Change, the SCORE Analyser offers a simple ‘Code Editor’ (see Figure 141) that allows the researcher to modify and execute a local copy of the project version’s code.

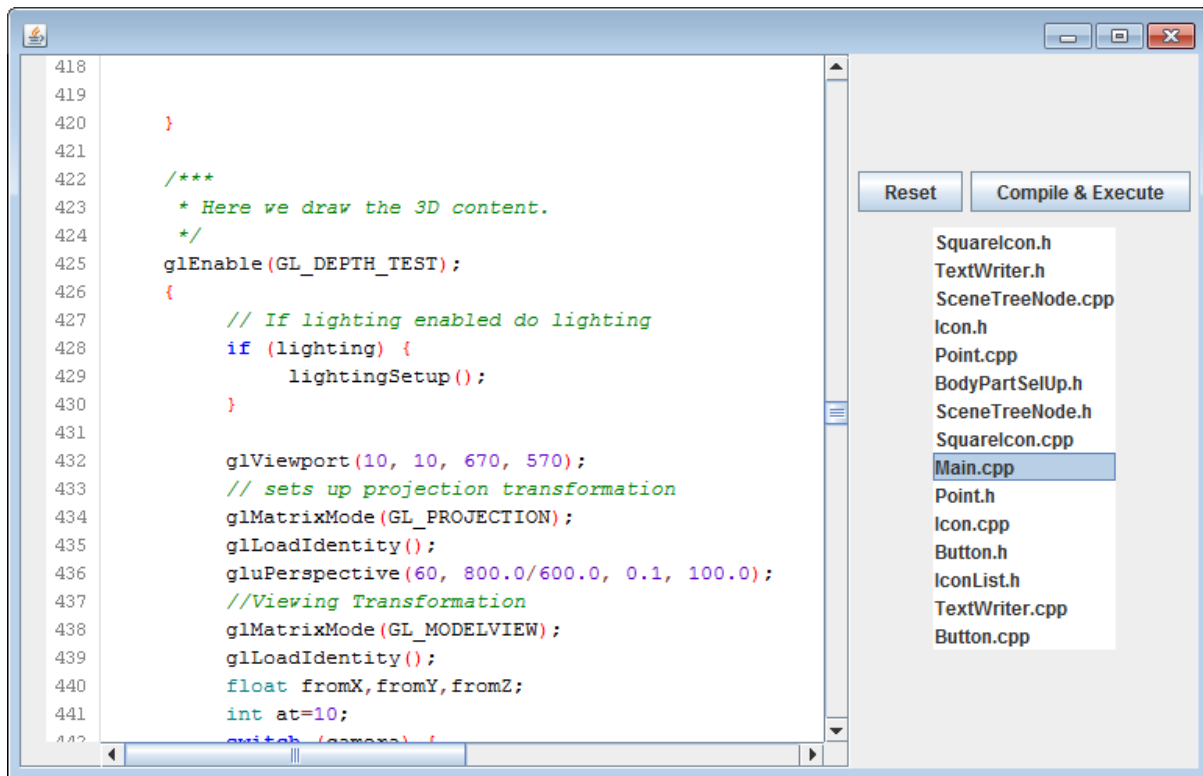


Figure 141: The Code Editor

Compilation is currently only enabled for C++ projects using the GNU g++ compiler. Compilation settings and dependencies such as include directories or library paths as well as regular expressions for the modification of any location-dependent source code the student may have included are configured via .xml files at the student and project level.

### 9.5.3.7 Line History View

The 'Line History View' (see Figure 142) allows the researcher to view all of a line's modifications occurring throughout the Project History as calculated by the Line History Generation algorithm described in Section 7.2.

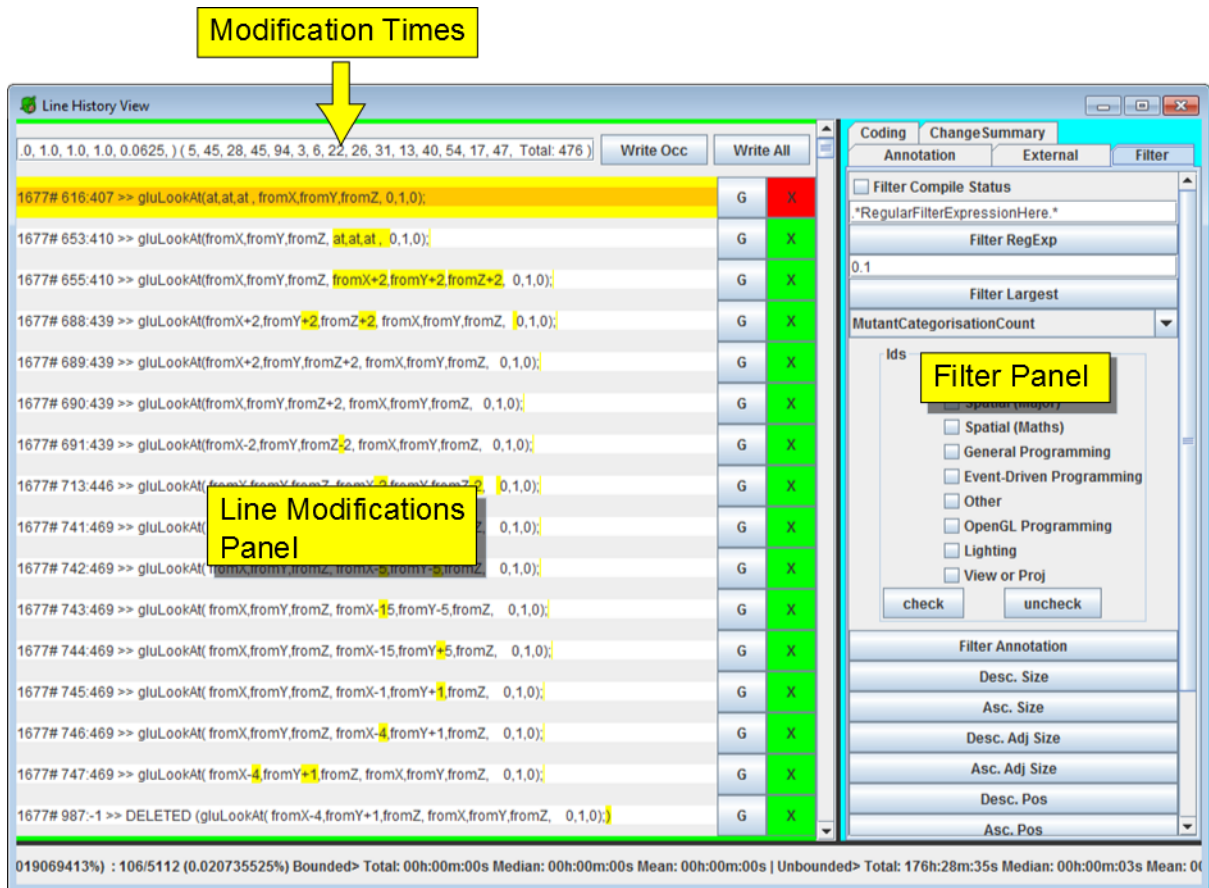


Figure 142: The Line History view

### 9.5.3.7.1 Line History Modifications

Modifications are presented as a list in order of occurrence as shown in Figure 143. For each line state, the line's text is presented along with the Change in which the modification occurs and the line number at which the line is located in that Change. Yellow parts of the line text are those modified since the last Change as detected via an algorithm which calculates the Longest Common Subsequence. The View also shows whether the Change belonging to that line version compiled or not (the box is red for non-compiled Changes, green for compiled Changes). Pressing the "G" button next to each line modification will make the 'Diff View' display the Change associated with that modification.



1582# 541:528 >> glutTimerFunc(10, animateHeadShake,i);	G	X
1582# 544:-1 >> DELETED (glutTimerFunc(10, animateHeadShake,i);	G	X
1582# 545:527 >> GHOST (glutTimerFunc(10, animateHeadShake,i+1);	G	X
1582# 547:521 >> MOVED (glutTimerFunc(10, animateHeadShake,1);	G	X
1582# 547:521 >> glutTimerFunc(10, animateHeadShake,1);	G	X
1582# 554:521 >> glutTimerFunc(10, animateWalking,1);	G	X
1582# 556:522 >> glutTimerFunc(10, animateWalking,1);	G	X

**Figure 143: A close-up of the modifications of a Line History as shown in the Line History view**

### 9.5.3.7.2 Line History Time data

Some additional information including the time taken for each modification in seconds is displayed in the text field above the list of modifications as shown in the top field in Figure 144.

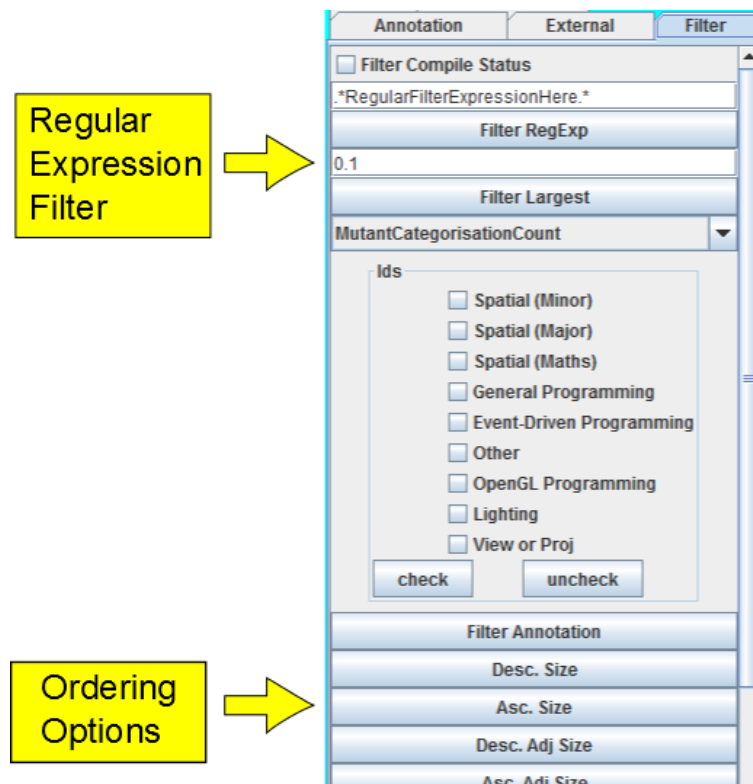
1333334, 0.16666667, 0.25, 0.33333334, ) ( 5, 49, 5, 12, 24, 36, 10, 7, 3, 18, 9, Total: 178 )			Write Occ	Write All
1582# 541:528 >> glutTimerFunc(10, animateHeadShake,i);	G	X		
1582# 544:-1 >> DELETED (glutTimerFunc(10, animateHeadShake,i);	G	X		
1582# 545:527 >> GHOST (glutTimerFunc(10, animateHeadShake,i+1);	G	X		

**Figure 144: Close-up of a Line History's time data, shown in a text field at the top of each Line History's description**

### 9.5.3.7.3 Filtering and Ordering Line Histories

The 'Line History View' also allows for Line Histories to be reordered or filtered using the panel shown in Figure 145. Line Histories can be filtered either by size to show only the Line Histories with the largest number of modifications, by time showing only the Line Histories whose modifications add up to the longest total time, or using a regular expression in which case only Line Histories containing at least one modification matching the regular expression are shown. The latter option is especially useful as it allows the researcher to identify Line Histories involving function calls the researcher is interested in. The View also allows the researcher to reorder the presentation of Line Histories, ordering them by size, time taken for all modifications or the Line History's position in the Project History.

To filter Line Histories, enter the regular expression in the top field in the ‘Regular Expression Filter’ and press “Filter RegExp”. To show only the largest x% of Line Histories enter the percentage in the field above the “Filter Largest” button and then press the button.



**Figure 145: The Filter panel (on the right of the Line History View) allows filtering and reordering of Line Histories**

To sort Line Histories in order of size or time, press the relevant button in the ‘Ordering Options’ area.

### 9.5.3.8 Change Browser

The core ‘Diff View’ (Section 9.5.3.3.2) is designed to allow for an easy visual comparison between one Change comprising two versions of source code. This is useful for comparing very complex changes in detail. However, most Changes involve the modification of a small number of lines (often a single line) and requiring the researcher to navigate each of these Changes in turn is unnecessarily cumbersome, both because it requires more time and because it requires the researcher to remember the last several Changes to be able to place the Change currently being examined into the larger context of the problem being worked on. The ‘Change Browser’ (Figure 146 and Figure 147) allows the researcher to examine several changes at once by showing only the lines that were added, deleted or mutated from one document to the next in the first column, Change-Coding data in the second and the Note for that Change in the third column. To set the range of Changes to be

displayed, enter it in the text field marked 'Range of Changes to display' and press the "Occs" button.

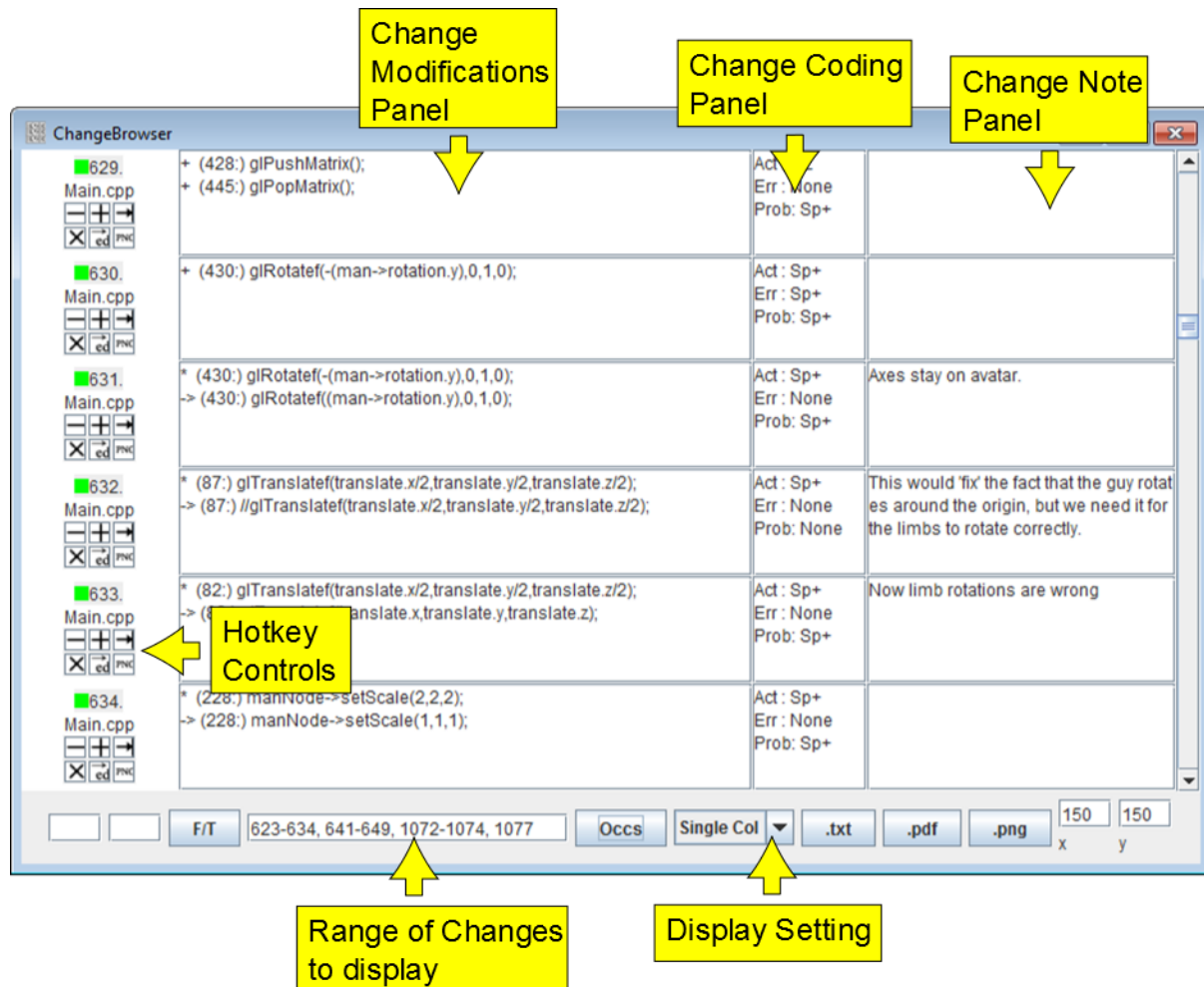


Figure 146: The Change Browser view

### 9.5.3.8.1 Change Browser View

+ (428:) glPushMatrix(); + (445:) glPopMatrix();	Act : GL Err : None Prob: Sp+	
+ (430:) glRotatef(-(man->rotation.y),0,1,0);	Act : Sp+ Err : Sp+ Prob: Sp+	
* (430:) glRotatef(-(man->rotation.y),0,1,0); -> (430:) glRotatef((man->rotation.y),0,1,0);	Act : Sp+ Err : None Prob: Sp+	Axes stay on avatar.
* (87:) glTranslatef(translate.x/2,translate.y/2,translate.z/2); -> (87:) //glTranslatef(translate.x/2,translate.y/2,translate.z/2);	Act : Sp+ Err : None Prob: None	This would 'fix' the fact that the guy rotates around the origin, but we need it for the limbs to rotate correctly.
* (82:) glTranslatef(translate.x/2,translate.y/2,translate.z/2); -> (82:) glTranslatef(translate.x,translate.y,translate.z);	Act : Sp+ Err : None Prob: Sp+	Now limb rotations are wrong
* (228:) manNode->setScale(2,2,2); -> (228:) manNode->setScale(1,1,1);	Act : Sp+ Err : None Prob: Sp+	

Figure 147: A close-up of modifications as summarised in the Line History view

### 9.5.3.8.2 Hotbuttons panel

Each Change also includes buttons (on the left) for navigating to that change in the Diff View, executing the change's executable or launching a source code editor for that Change as shown in Figure 148.

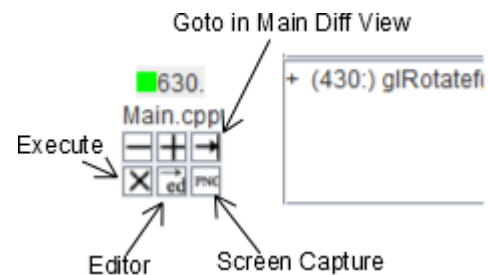
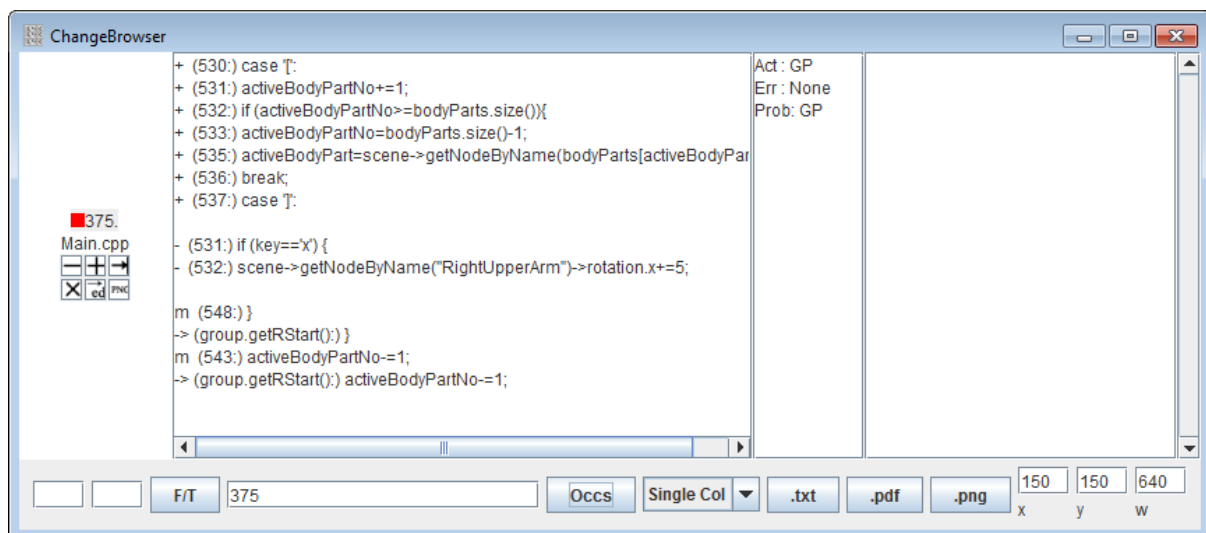
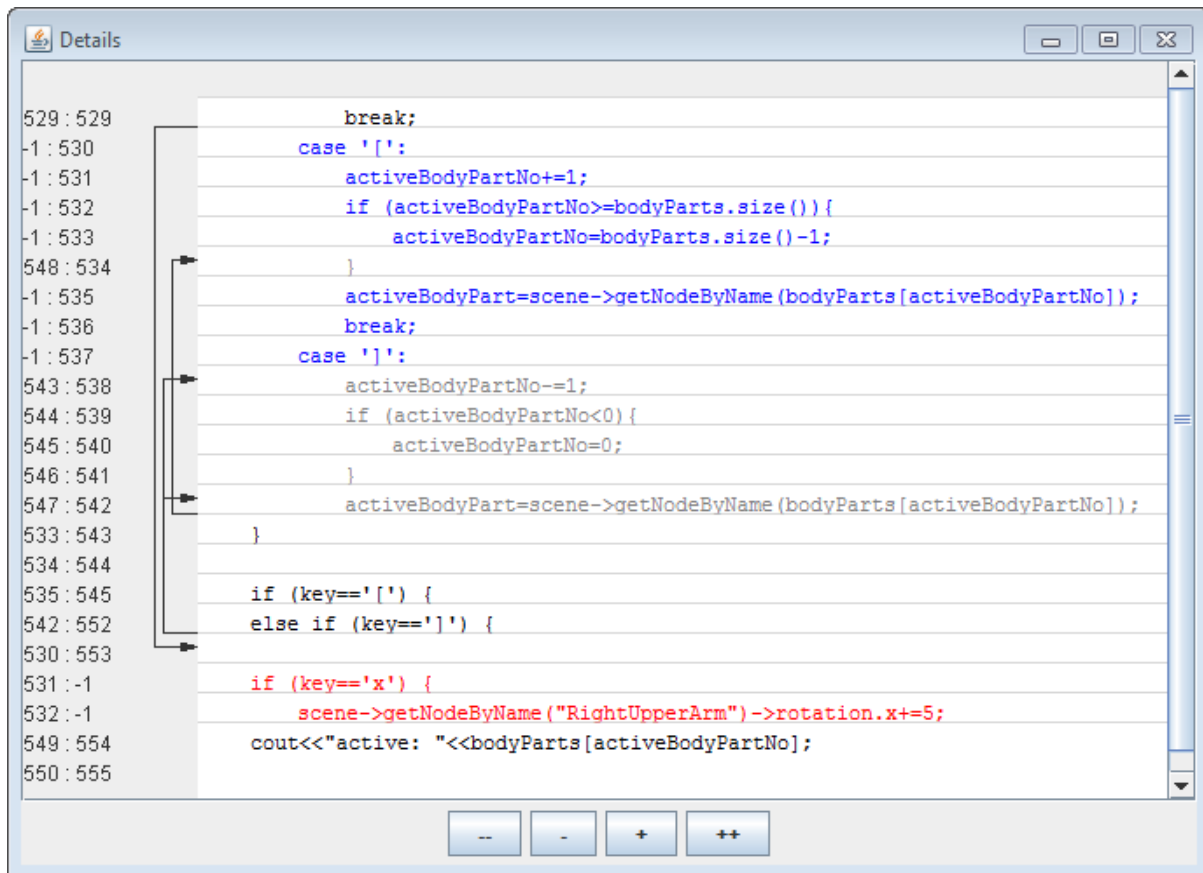


Figure 148: The hotbuttons panel

### 9.5.3.8.3 Change Browser Details View

In addition to the modifications displayed in the main 'Change Browser' panel, a 'Change Browser Details View' can be opened for each change by left-clicking the panel containing that Change's modifications. The 'Change Browser Details View' is shown in Figure 149. The View shows all of the version's source code lines along with visual representations of all line modifications that occurred during that Change. Moved lines are shown using arrows indicating the code's current and previous position. Additions and deletions are shown in blue and red respectively. Mutations are shown in the form of 'OldMutationText -> NewMutationText'. Compared to the standard Diff View, this view summarises all modifications to a Change in a single view, rather than showing them as mappings between two separate views.





**Figure 149: On top modifications as displayed in the main Change Browser view, on the bottom the same modifications as displayed using the Details view**

By setting the 'Display' checkbox to the 'Graphics' setting, the Change Browser's main view can display 'Detail Views' instead of displaying simple textual summaries as shown in Figure 150.

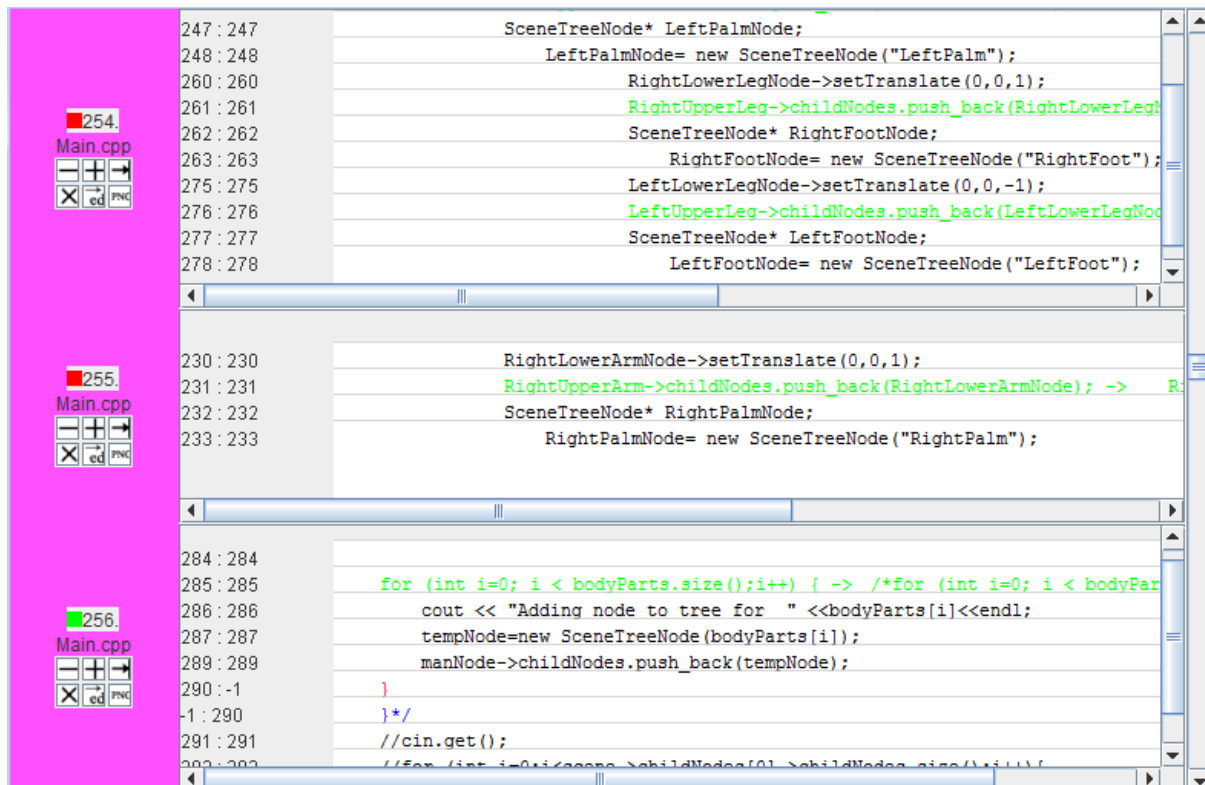


Figure 150: The Change Browser set to 'Graphic' display; it displays the 'Detail View' for each Change

#### 9.5.3.8.4 Exporting Details to a PDF document

All the “Detail Views” can also be printed to a PDF document an example of which is shown in Figure 151, providing a portable easy-to-read history of all modifications occurring in the Project History. These documents were found to be extremely useful in practice, either for use alongside the SCORE Analyser tool during analysis, or for ‘offline’ analysis of Project Histories.

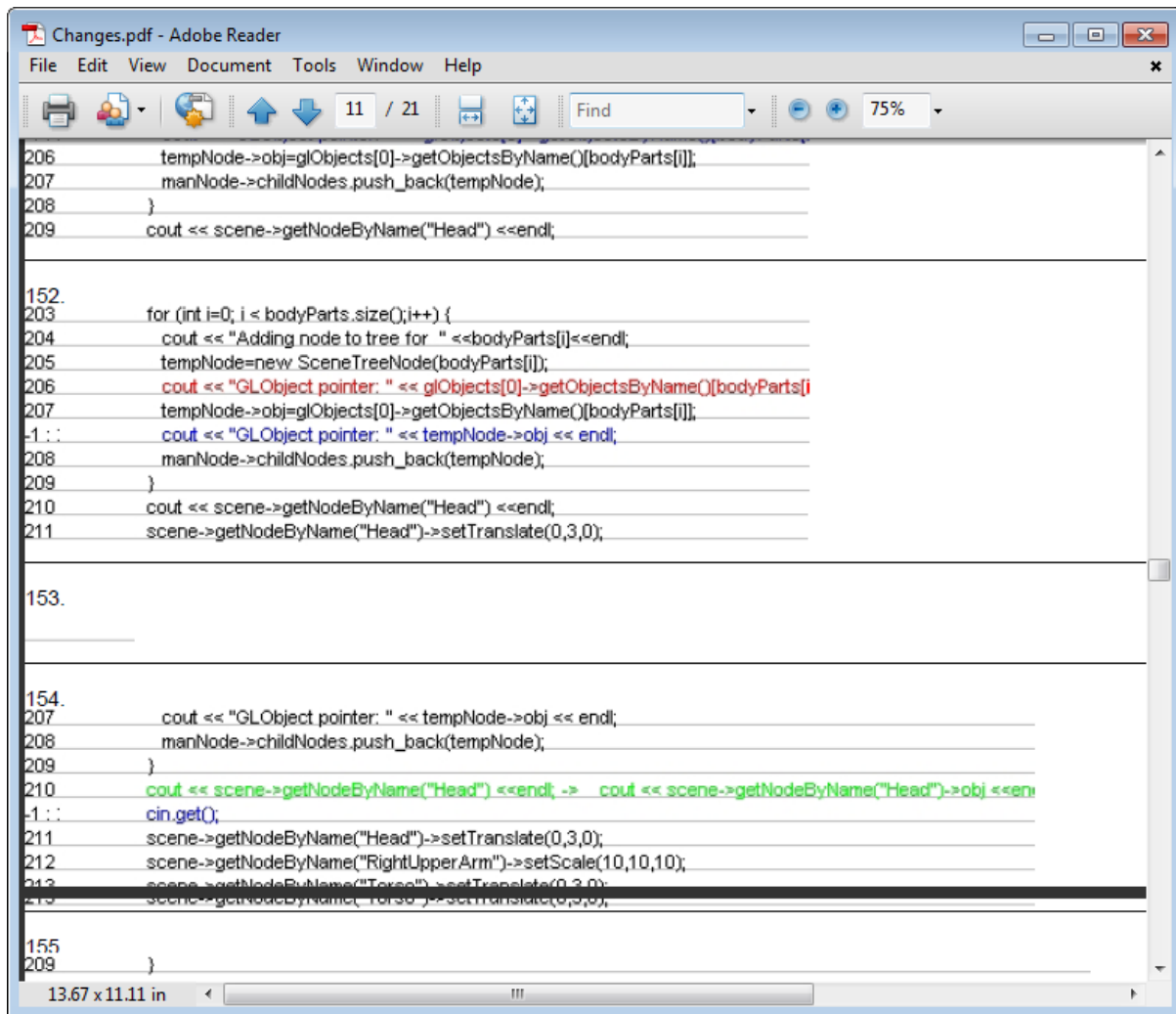


Figure 151: The generated PDF document containing the Details view for every Change

To produce a PDF, click the “.pdf” button. The PDF is saved to the output directory specified in the Config file.

### 9.5.3.9 Change-Coding Views

#### 9.5.3.9.1 Change-Coding Set-up (Config files)

To set up Change-Coding, modify the “AnnotationConfig.cfg” file which should be located in the project’s root directory. Several different components can be added to the Change-Coding panel:

- Static Labels

The following example creates a label titled “Task”, with white text on a green background:

```
<component>
  <name>TaskLabel</name>
  <title>Task</title>
  <type>LABEL</type>
  <default>
```

```

        <background>green</background>
        <text>white</text>
    </default>
</component>

```

- Dropboxes with multiple categories

Dropboxes containing an arbitrary number of user-defined categories. The following sets up a dropbox titled “Action” which includes two categories, “Major Spatial” and “Minor Spatial”. The item with id=1 (“MinorSpatial”) is selected as the default option to which fields are initialized before the user modifies them.

```

<component>
    <name>ActionDropbox</name>
    <title>Action</title>
    <type>DROPBOX</type>
    <options>
        <option>
            <text>Major Spatial</text>
            <abbrevName>Sp+</abbrevName>
            <id>0</id>
        </option>
        <option>
            <text>Minor Spatial</text>
            <abbrevName>Sp-</abbrevName>
            <id>1</id>
        </option>
    </options>
    <default>
        <val>1</val>
    </default>
</component>

```

- Single Checkboxes

A single checkbox. The example below creates a checkbox titled “Ignore” which by default is set to ‘false’ (unchecked):

```

<component>
    <name>Ignore</name>
    <title>Ignore</title>
    <type>CHECKBOX</type>
    <default>
        <val>false</val>
    </default>
</component>

```

- Multi-Checkboxes:

A component including multiple checkboxes, each of which can be individually checked or unchecked just like an individual checkbox. The example below creates a component titled



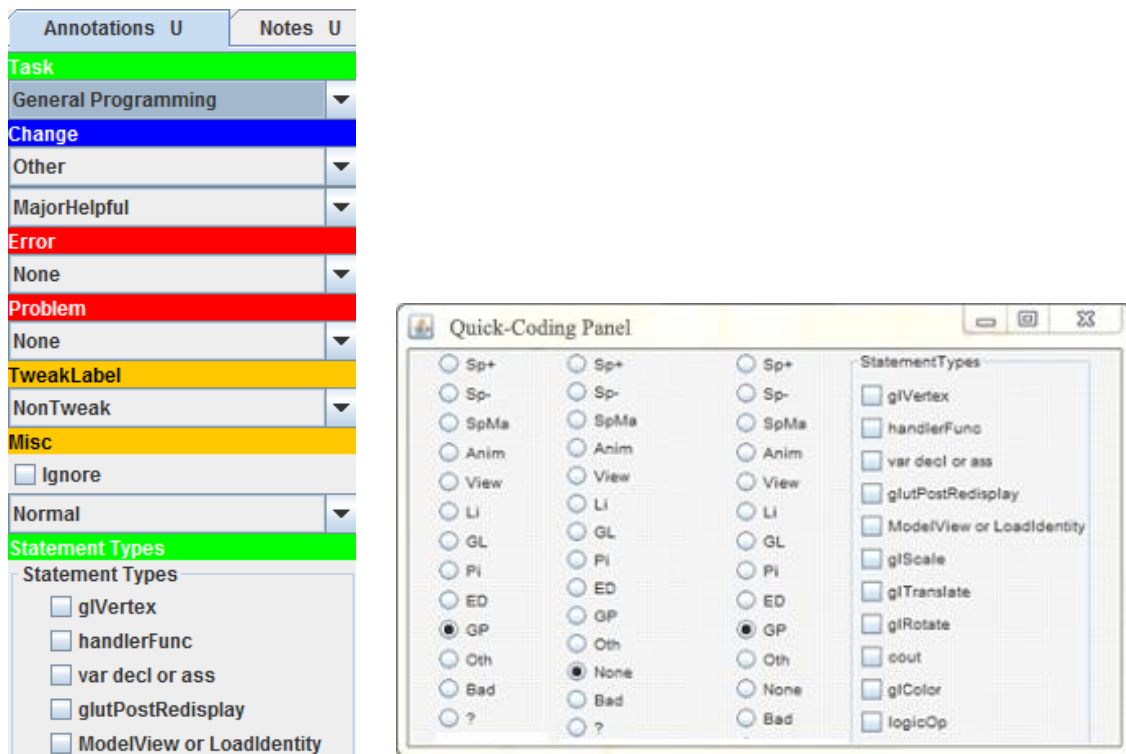
“Statement Types” which contains two items labelled “glVertex” and “glColor”. The first is unchecked by default, the second is checked by default.

```
<component>
  <name>StatementTypes</name>
  <title>Statement Types</title>
  <type>MULTICHECKBOX</type>
  <default>
    <checkbox>
      <name>glVertex</name>
      <title>glVertex</title>
      <val>false</val>
    </checkbox>
    <checkbox>
      <name>glColor</name>
      <title>glColor</title>
      <val>true</val>
    </checkbox>
  </default>
</component>
```

All components are positioned in the order in which they appear in the configuration xml file.

#### ***9.5.3.9.2 Change-Coding and Quick-Coding Panels***

The ‘Change-Coding View’ (shown in Figure 152, left) is found in the right-hand side tabbed pane under “Annotation”. It is used to code Changes using categories and labels defined via configuration files (consistent between all SCORE assignments that belong to the same project) which are represented using drop-down combo boxes, text fields and check boxes. Coding a Change involves modifying the components shown in the Change-Coding View. For quick coding for dimensions involving categories, a ‘Quick Coding panel’ (Figure 152, right) can also be utilised.



**Figure 152: The Change-Coding and Quick-Coding panels**

The Change-Coding Panel contains one component for every Change-Coding dimension specified in the Change-Coding setup xml file. Change-Coding data can be accessed by other SCORE components for further processing; for example, graphs showing the rolling average of changes in a certain category can be generated using the Graphing component as described in the next section.

To perform Change-Coding simply select the appropriate categories for that Change from the combo boxes and tick/untick checkboxes, or open the Quick-Coding view by pressing the “Open Quick Sel” button in the Change-Coding pane and utilise the radio buttons and check boxes. Modifications are automatically temporarily saved when navigating to a new Change. To permanently save Changes, press the “Save” button in the main interface.

### **9.5.3.9.3 Graphing Change-Coding Results**

Data produced through Change-Coding can be used to produce graphs showing the prevalence of different Change-Coding categories at different points of the Project History. The most common graphing option is to graph rolling averages for individual categories which produces peaks for periods where many Changes in the recent past fall into the same category. These periods can then be examined in more detail to see whether they make up a larger problem.

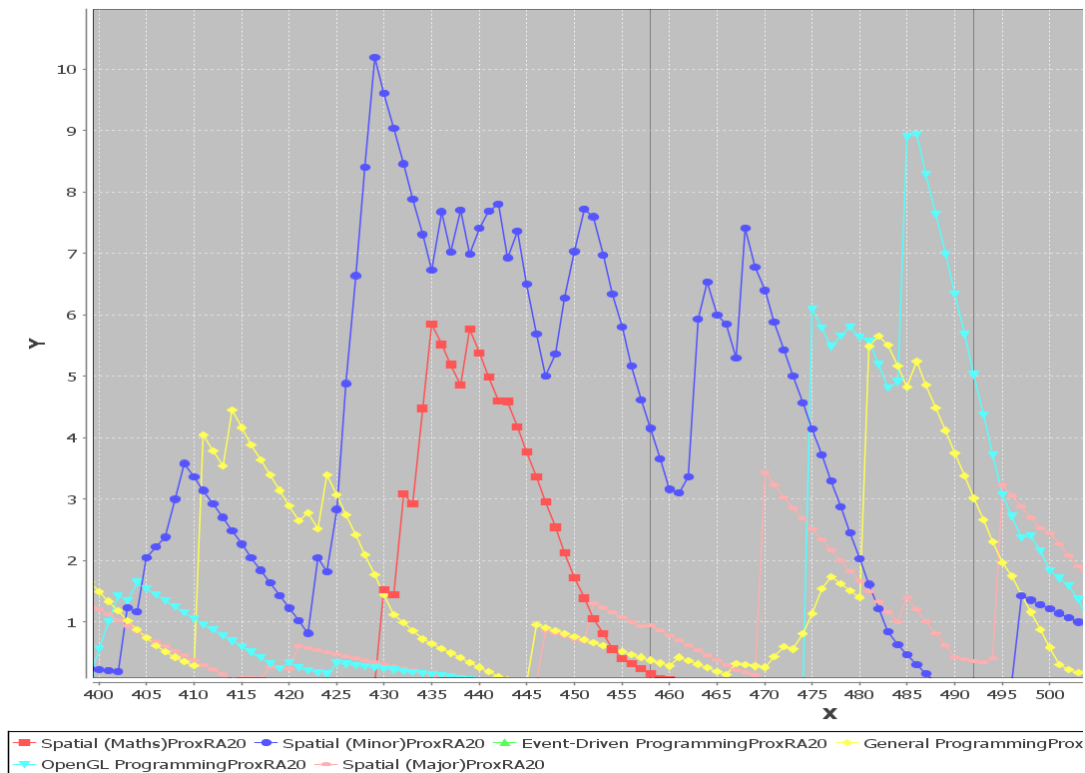
To produce a graph, first open the Graphing View by pressing the “Open JFreeChart” button in the “External Controls” pane of the “Additional Controls” panel. Then press the “Additional” button in

the Graphing View which will open the window shown in Figure 153. Tick any items to display (RA stands for rolling average) and press the “OK” button on the right to generate a graph like the one shown in Figure 154.

The image shows a 'Graph Settings Panel' window with a blue title bar and standard window controls. The panel is divided into three main sections: 'Error', 'Accum', and 'RA'. Each section contains a list of checkboxes for different data series and a vertical slider on the right.

Section	SpComb	Sp+	ED	Pi	Bad	Oth	Anim	View	GP	Li	SpMa	Sp-	GL	None	?
General (under Error)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10 (under Accum)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5 (under Accum)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
10 (under RA)	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
5 (under RA)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 153: The set-up panel for the graph



**Figure 154: A graph mapping rolling averages of the number of Changes belonging to different Change-Coding classification categories**

#### 9.5.3.9.4 Generating Quantitative Change-Coding Data

Graphing provides a simple visualization of Change-Coding data and allows the researcher to identify areas associated with high levels of activity for a certain type of Change.

The SCORE Analyser can also generate summaries of Change-Coding data as shown in Figure 155. To open this view, press “Open Numerical Views” in the ‘External Controls’ pane of the ‘Additional Controls’ pane on the right-hand side and press “Draw Chart”. The view displays per category the total number of Changes that were coded with that category, as well as the total amount of time spent on Changes of that category. When opening this view an excel spreadsheet containing the data is automatically written to the output directory specified in the Config.cfg file.

<b>ErrorDropbox</b>	
<b>None:</b>	884 (73.73%) 005h:59m:48s 00.00.0000 (71.98%) Tpc: 24.42
<b>General Programming:</b>	121 (10.09%) 000h:53m:36s 00.00.0000 (10.72%) Tpc: 26.58
<b>Bad:</b>	69 (5.75%) 000h:26m:48s 00.00.0000 (5.36%) Tpc: 23.30
<b>Major Spatial:</b>	49 (4.09%) 000h:26m:26s 00.00.0000 (5.29%) Tpc: 32.37
<b>Viewing or Projection:</b>	25 (2.09%) 000h:13m:25s 00.00.0000 (2.68%) Tpc: 32.20
<b>Unknown:</b>	12 (1.00%) 000h:03m:29s 00.00.0000 (0.70%) Tpc: 17.42
<b>Pipeline:</b>	8 (0.67%) 000h:04m:16s 00.00.0000 (0.85%) Tpc: 32.00
<b>Minor Spatial:</b>	8 (0.67%) 000h:03m:42s 00.00.0000 (0.74%) Tpc: 27.75
<b>Other:</b>	7 (0.58%) 000h:03m:07s 00.00.0000 (0.62%) Tpc: 26.71
<b>Event-Driven:</b>	5 (0.42%) 000h:00m:55s 00.00.0000 (0.18%) Tpc: 11.00
<b>Math Spatial:</b>	4 (0.33%) 000h:01m:39s 00.00.0000 (0.33%) Tpc: 24.75
<b>General OpenGL:</b>	4 (0.33%) 000h:01m:26s 00.00.0000 (0.29%) Tpc: 21.50
<b>Lighting:</b>	2 (0.17%) 000h:00m:45s 00.00.0000 (0.15%) Tpc: 22.50
<b>Animation:</b>	1 (0.08%) 000h:00m:29s 00.00.0000 (0.10%) Tpc: 29.00

Figure 155: Change-Coding Metrics View for a single dimension

### 9.5.3.10 Segment-Coding View

The 'Segment-Coding View' allows the researcher to create segments consisting of sets of Changes to describe the student programming process captured in the Project History. The Segment-Coding View is shown in Figure 156. To access the 'Segment-Coding View', press the "Open Segment-Coding View" button in the 'External Controls' tab in the 'Additional Controls' panel.

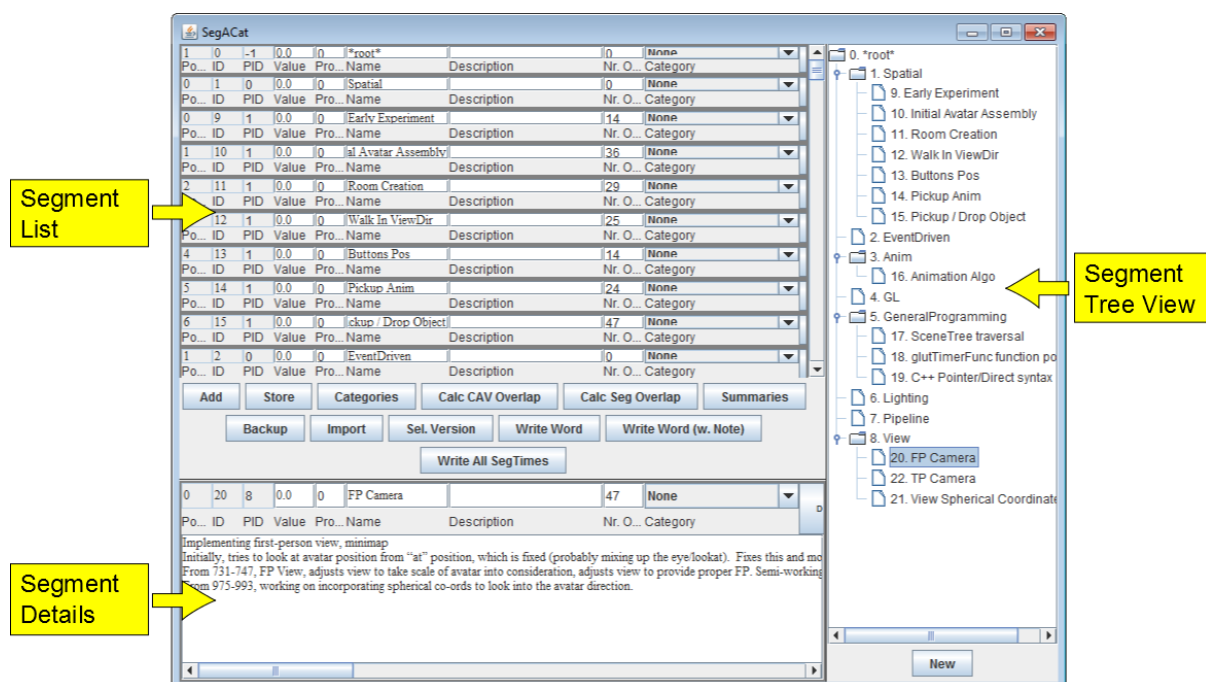
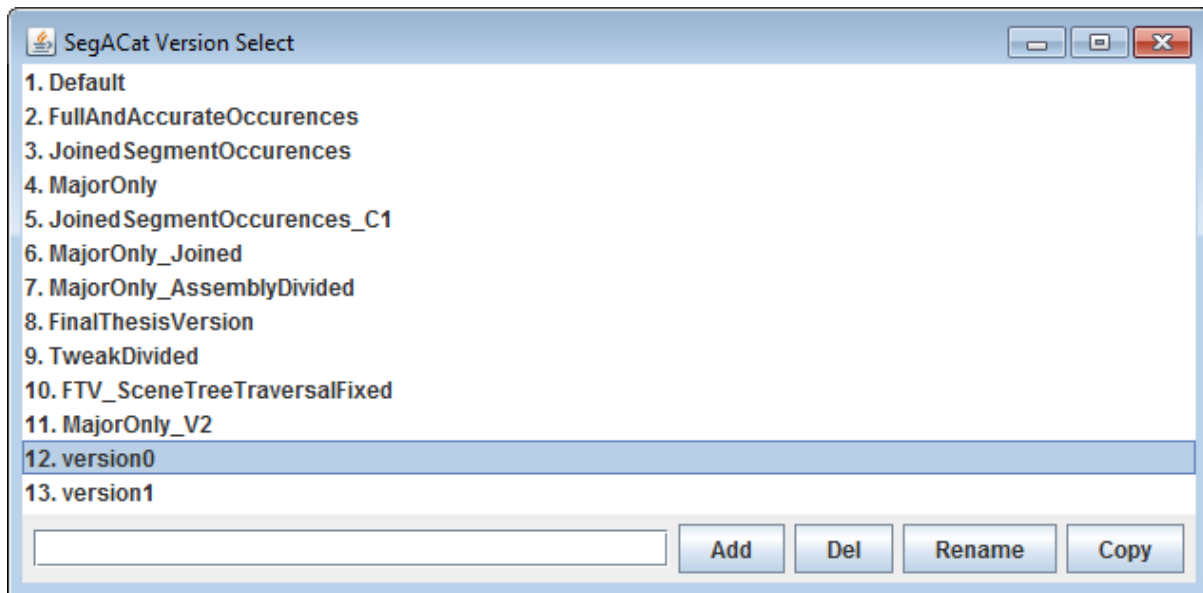


Figure 156: The Segment-Coding View

#### 9.5.3.10.1 Creating and selecting a Segmentation Version



**Figure 157: The Segment Version view**

The 'Segment-Coding View' allows for the creation of multiple Segmentation versions. For example, a researcher might create a new version every time she makes any considerable modifications to the previous Change-Coding method. Such different versions could also involve completely different Change-Coding schemes, developed to probe different research objectives.

To create a new Segment-Coding version, click the "Sel. Version" button. This will open the window shown in Figure 157. To create a new version, enter the name in the text field and press "Add". To select a version, press the version's name in the top window. To copy the contents of one version to another, first click on the version to be copied, then on the version to copy to and then press "Copy". Be careful not to select the version to be copied second or it will be overwritten.

#### **9.5.3.10.2 Adding, Deleting and Moving Segments**

The 'Segment Tree View' (see Figure 158) is used to add segments or to parent segments to other segments. The tree view also allows segments to be added to other segments as 'sub-segments'. The segment hierarchies produced in this way can be of an arbitrary depth. This functionality was used to assign individual segments to categories such as 'Spatial' or 'Lighting' by making them children of top-level category segment nodes. In addition to using the hierarchical structure for categorisation of segments, it can also be used to further break down segments into sub-segments. For example, the implementation of a pickup animation may involve movement of the avatar as well as movement of the object to be picked up. These two tasks can then be added to separate segments which are parented to the same "Pickup Animation" segment. To add a new segment, press the "New" button and a new segment parented to the root node will be created. To parent the Segment

to a different Segment select and drag the Segment onto its new parent or drag it onto the intended position between a parent's child Segments. When a Segment is selected, its details will be shown in the Details view described later.

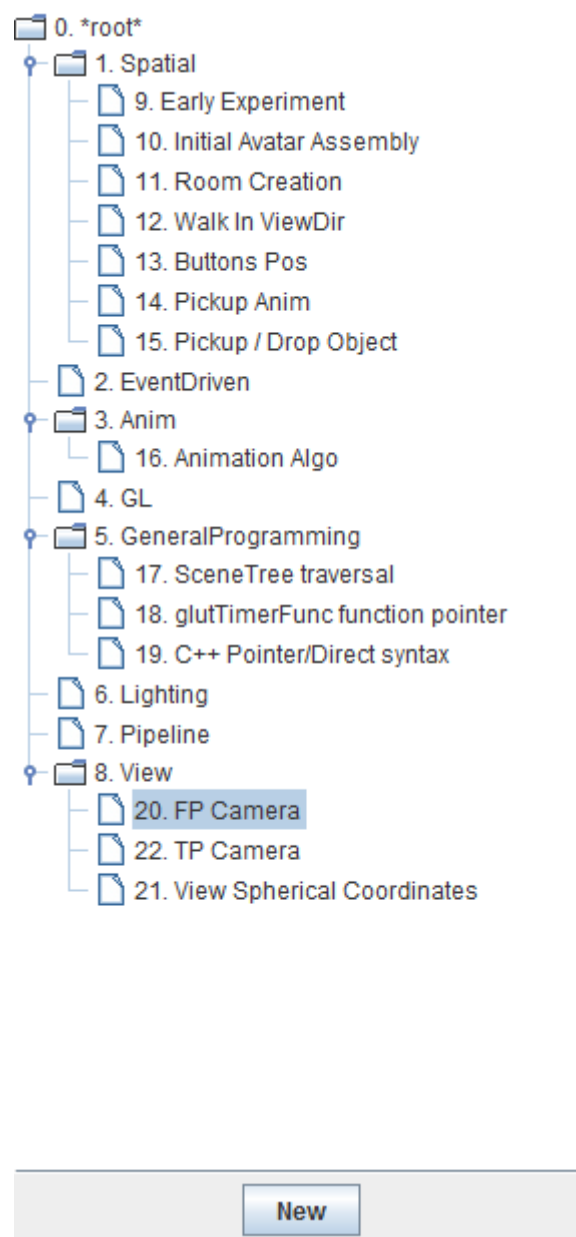


Figure 158: A close-up of the Segment Tree view

### 9.5.3.10.3 Segment List View

In addition to being displayed in the tree view they are also described in the 'Segment List' view shown in Figure 159. The list view shows each Segment's id as well as the Segment's name, description, occurrences and category. Pressing the "D" button will show the Segment's details in the details view, whereas the "X" button will delete the Segment (careful, there is no undo!).

1	0	-1	0.0	0	*root*			0	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				
0	1	0	0.0	0	Spatial			0	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				
0	9	1	0.0	0	Early Experiment		138-139, 146-147, 149, 155-157, 165-166	14	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				
1	10	1	0.0	0	Initial Avatar Asser		212-228, 258-260, 262-265, 267, 272, 274	36	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				
2	11	1	0.0	0	Room Creation		408-412, 414, 419, 422-426, 429, 435-440	29	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				
3	12	1	0.0	0	Walk In ViewDir		623-634, 641-649	21	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				
4	13	1	0.0	0	Buttons Pos			0	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				
5	14	1	0.0	0	Pickup Anim			0	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				
6	15	1	0.0	0	Pickup / Drop Obje			0	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				
1	2	0	0.0	0	EventDriven			0	None				
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category				

Figure 159: A close-up of the Segment List view

#### 9.5.3.10.4 Segment Details View

The lower-left 'Segment Details View' (see Figure 160) shows a segment's details. A segment is described by a unique ID, a name, a short description, a long comment, a set of Changes that make up the segment and an integer problem difficulty value that stores the researcher's judgement on how difficult the underlying problem was to resolve.

Segment Name		Segment Description			Segment Occurences		Segment Category				
0	20	8	0.0	0	FP Camera	608-615, 653-654, 712, 731-74	47	None			
Po...	ID	PID	Value	Pro...	Name	Description	Occurrences	Nr. O...	Category	d	x
Implementing first-person view, minimap Initially, tries to look at avatar position from "at" position, which is fixed (probably mixing up the eye/lookat). Fixes this and moves to an approach trying to look "ahead of" avatar (gluLookAt(fromX,fromY,fromZ, fromX+2,fromY+2,fromZ+2, 0,1,0)) which shows a misunderstanding of "ahead of". Also, the student's incorrect compositing of a scale transform when assembling the avatar means that the avatar's position as stored in the variables doesn't correspond to his coordinates on the screen. From 731-747, FP View, adjusts view to take scale of avatar into consideration, adjusts view to provide proper FP. Semi-working (w.out head rotation) at 740. Tries to tweak the eye/lookAt to provide a proper "look-ahead", but of course that won't work. From 975-993, working on incorporating spherical co-ords to look into the avatar direction.											
Segment Note											

Figure 160: A close-up of the Segment Details view

#### 9.5.3.10.5 Exporting a Segmentation to a text document

The 'Segment-Coding View' can be used to generate a human-readable text file containing all Segment data as shown in Figure 161 where each segment is listed according to its position in the hierarchy along with the detailed information associated with it.



```

Project: Assignment1_Project
Version: FinalThesisVersion

AStudent_Assignment1.SP."Spatial"
Name: Spatial

AStudent_Assignment1.SP.1."Door Impl."
Name: Door Impl.
Description: Spatial
Occurences: 68-75, 372-377 : total = 14 (+)
Detailed Description:
68-75, 372-377 (14 changes (+5 minutes, 7 changes), ~21 changes ++):
From 68-75, Implements door without error, one size tweak. (Basic shape
and borders)
From 372-377 fixes the door drawing, which had an error in the formula
for calculating the width of components.. First tries to fix by
tweaking, then realises an error in the offset formula.

...

```

Figure 161: An extract of a generated text file containing Segment descriptions

#### 9.5.3.10.6 Generating Quantitative Segmentation Time data

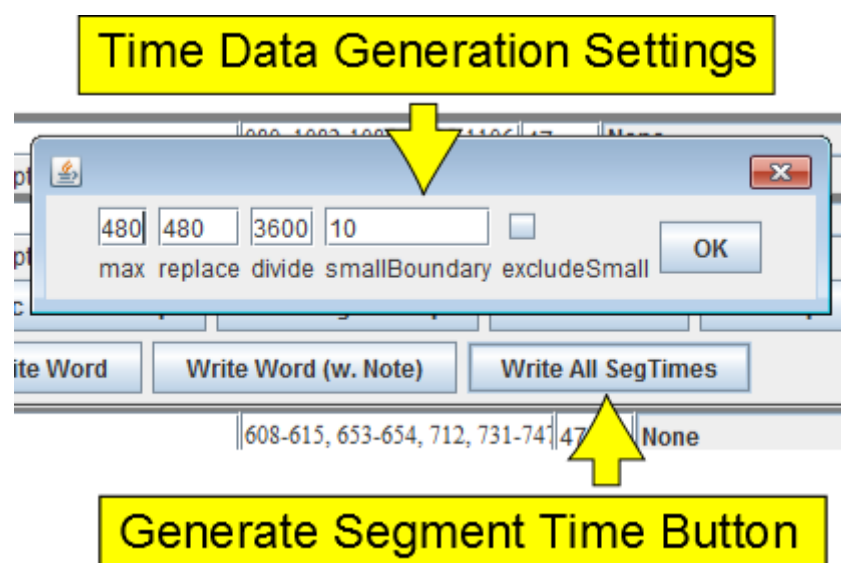
The 'Segment-Coding View' can also be used to generate time information for stored segments. However large gaps between Changes during work sessions may involve any number of activities such as looking up information online which is related to the Change's effort or chatting to a friend or browsing the internet which is not. The algorithm used to calculate time spent per segment heuristically circumvents these issues.

One type of interruption is when a student suspends work temporarily; for example, the student may stop work for the day and only commence working on the project several days later. The heuristic algorithm sets the time for such Changes to be equal to a user-defined period of time; in practice, the average time spent by students on a Change was used. Such Changes will be referred to as *Suspend Changes*.

Another situation may involve a smaller interruption of student work, such as the student chatting or going to the bathroom. While the time will be lower than that of a suspension of work, such small interruptions are likely to occur more often. To heuristically identify such periods, the algorithm can be set to cap time values smaller than the *Suspend* boundary but higher than another value which will be called the *Max* boundary. The value to which the time is capped can be defined by the user. In practice it was set to ten minutes, since such values are significant outliers to the distribution of Change times.

There is one further situation discovered during analysis in which there was an apparent inaccuracy in measuring time between Changes using timestamps. Changes with unreasonably small time values were discovered. These Changes always involve different files being modified. These Changes are most likely due to several files being modified at once. Once a save all/compile action is executed, each file is added to the Project History in turn, generating multiple Changes which should in fact only be a single Change. The small time values are entirely due to the speed of compilation. If such small Changes were included, they would significantly decrease averages measured for sets of related Changes. The heuristic algorithm can be set to exclude Changes whose time between versions falls below a certain threshold. In practice a value of 5 seconds was chosen as this is close enough to the time it would take for the student to compile the program that the student could not have done any useful work in the meanwhile. Such Changes are referred to as *Small Changes*.

The settings which the algorithm uses to determine whether Changes are normal, *Suspend*, *Max* or *Small* are entered using the ‘Segment Time Generation’ settings panel shown in Figure 162. The time calculated using this heuristic algorithm will not precisely calculate student time spent working on the Change (which is impossible without monitoring students constantly) but the results should reflect student work patterns well enough to enable qualitative analysis.



**Figure 162: The Time Data Generation Settings panel**

To generate Segment time data, press the “Write All SegTimes” button. This will cause the window shown as ‘Time Data Generation Settings’ to pop up. Enter the settings for max, replace, divide and smallBoundary (select “excludeSmall” to apply the smallBoundary exclusion) which were described earlier and press “OK”. An excel spreadsheet will be written to the SCORE output directory. The data is of the following format:

In addition to average time and median Change time per Segment as well as average deviation of Change Time from the Change time median for each segment, the algorithm also calculates the total number of modifications of different types (Added, Deleted, Moved, Ghost, Mutated) and presents those data along with data on the size of the segment, the problem difficulty (as assigned by the researcher through the 'Segment-Coding View' interface) and the category to which the segment was assigned. The output also includes the number of *Suspend*, *Max* and *Small* Changes contained in the Segment. An excerpt of the output of the Segment Time/Modification generation algorithm for one assignment is presented in Table 42. This data can then be used to perform quantitative analysis of Segments using statistical analysis software such as R.

**Table 42: Example of calculated Segment Time and Modification Data**

Seg ID	Name	Category	ProbDiff	Avg	Median	AvgDev	Small
1.1	Door Impl.	"Spatial"	1	126	79	108	0
1.2	Attempts GL transformations	"Spatial"	5	108	43	119	0
1.3	Arrow Icon	"Spatial"	1	119	80	92	0
2.1	Screen-Window convo	"ED"	2	174	139	113	3
2.3	GLUT Menu	"ED"	3	89	82	62	0
3.1	Virtual function and Constructor	"GP"	2	96	67	79	1
3.2	Data type for RGB	"GP"	2	77	49	59	0

Seg ID	Max	Susp	Added	Deleted	Moved	Ghost	Mutated	AllMod	size
1.1	2	0	7	1	0	0	7	70	14
1.2	3	1	5	4	0	0	9	75	15
1.3	1	1	4	1	0	0	9	65	13
2.1	3	2	6	4	0	0	11	75	12
2.3	1	0	11	5	0	0	12	115	23
3.1	0	0	5	1	0	1	7	55	10
3.2	0	0	3	0	0	0	8	50	10

### 9.5.3.11 Machine-Segmenting Views

Machine-Segmenting generates a Segmentation of a Project History much as a human researcher examining Changes and assigning them to Segments would. As opposed to the human researcher who would analyse the meaning of source code modifications Machine-Segmenting generates Segments based only on features of the source code. While this means that Machine-Segmenting cannot replace the human researcher as it cannot analyse Segments and determine their content, it can greatly speed up the analysis process by providing the researcher with sets of related Changes for analysis. Machine-Segmenting output is unlikely to perfectly match the way in which the researcher would segment the data and machine-generated Segments will usually require some degree of correction before in-depth analysis of Segments can begin.

#### 9.5.3.11.1 Entering settings and running the machine-generation algorithm

To machine-generate Segments access the Machine-Segmenting View from the 'External Controls' pane in the 'Additional Controls' panel. This will open the 'Machine Segmenting Settings' panel shown in Figure 163, the Segmentation controls panel shown in Figure 164, as well as two console windows and a link window and the method selection window. In the 'Machine-Segmenting Settings' window each of the rows represents one run of the algorithm, utilising the settings specified in that row. Settings for methods are based on the method's implementation and can be composed of an arbitrary number of Integer and Floating-Point numerical values, Boolean values and String values, each of which is identified via a unique name. Settings can also be imported from or exported as plain text; the import/export text area is located below the setting rows. It contains the name of each setting (only if it is set to a non-default value) and the value assigned to it. This makes it easy to re-run previously applied settings, or to slightly modify settings. Along with settings for the Machine-Segmenting algorithm, the settings window also allows the researcher to input the Segment cut-off level.

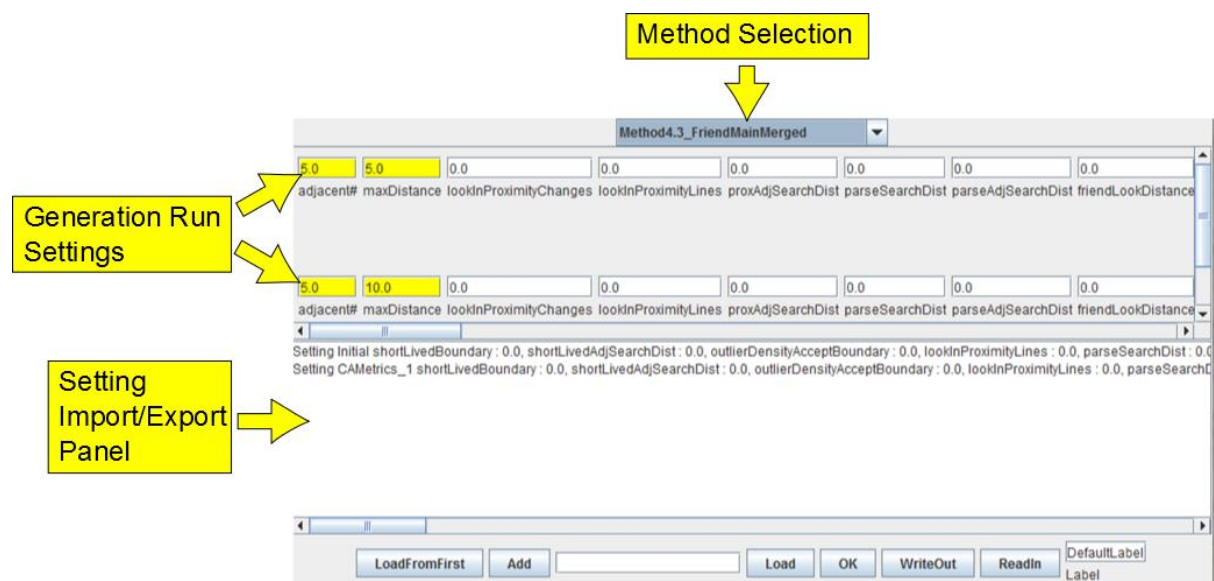
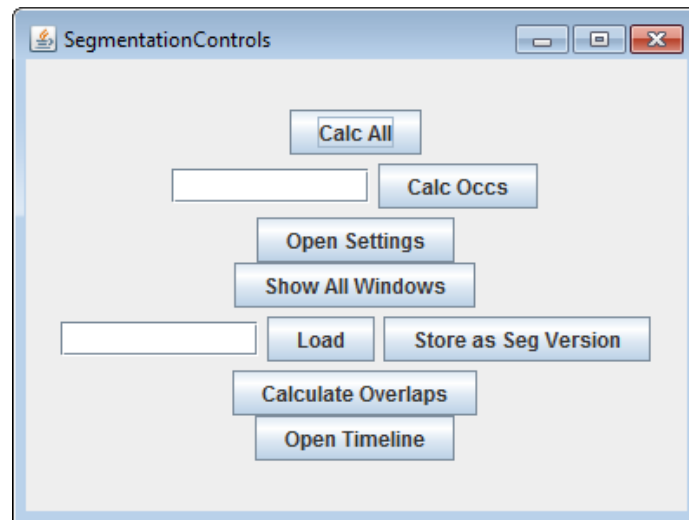


Figure 163: The Machine-Segmenting Settings panel



**Figure 164: the Segmentation Controls panel**

Since the LH-Graph method was found to be the most effective method some of its settings will be described here. These settings can be used to perform Machine-Segmenting.

First in the method selection box choose “Method5\_ExpressionMethod”. Now enter the following settings in the setting fields (note that all ‘best’ recommendations are based on evaluation occurring in Chapter 7 based on two assignment contexts. Other settings may perform better in different assignment contexts):

- Adjacents: The depth at which the algorithm will traverse the Line History graph; the best value was found to be 2.
- Distance: The farthest distance (in terms of overall position between two versions) at which a node will be visited in the Line History graph. The best setting found was 50.
- useMutant, useMoved, useAdded, useDeleted, useGhost: which modification types to visit. The best setting was found to utilise mutant, moved and ghost modifications.
- compileStatusFilter: whether the compile status filter should be activated; it was found to produce large improvements and hence should be activated
- hleExpression, hdocExpression, hdocLineExpression:expressions which are described in more detail in Section 7.8.9; to override these settings, simply enter “true == true” into all three fields

Another way of entering the settings is to enter a text string version of the settings into the ‘Setting Import / Export Panel’ and pressing “Load”. The following string contains all the ‘best’ settings detailed above:

```
adjacent#      :      2,maxDistance      :      50.0,hleExpression      :  
true==true,hdocLineExpression      :      true==true,hdocExpression      :  
true==true,proxChangeExpression : true==true,useMutated : true,useGhost :  
true,useDeleted : true,useAdded : true,useMoved : true,compileStatusFilter  
: true,
```

After settings have been entered, press the “OK” button. The settings window will disappear. To enter different settings, press the “Open Settings” button in the Segmentation Controls panel to reopen the settings window. Once the settings are entered, the Machine-Segmenting mechanism can be run by pressing “Calc All” upon which the entire Project History will be machine-segmented. To only machine-segment a subset of the Project History enter the range in the text field next to the “Calc Occs” button and then press the button.

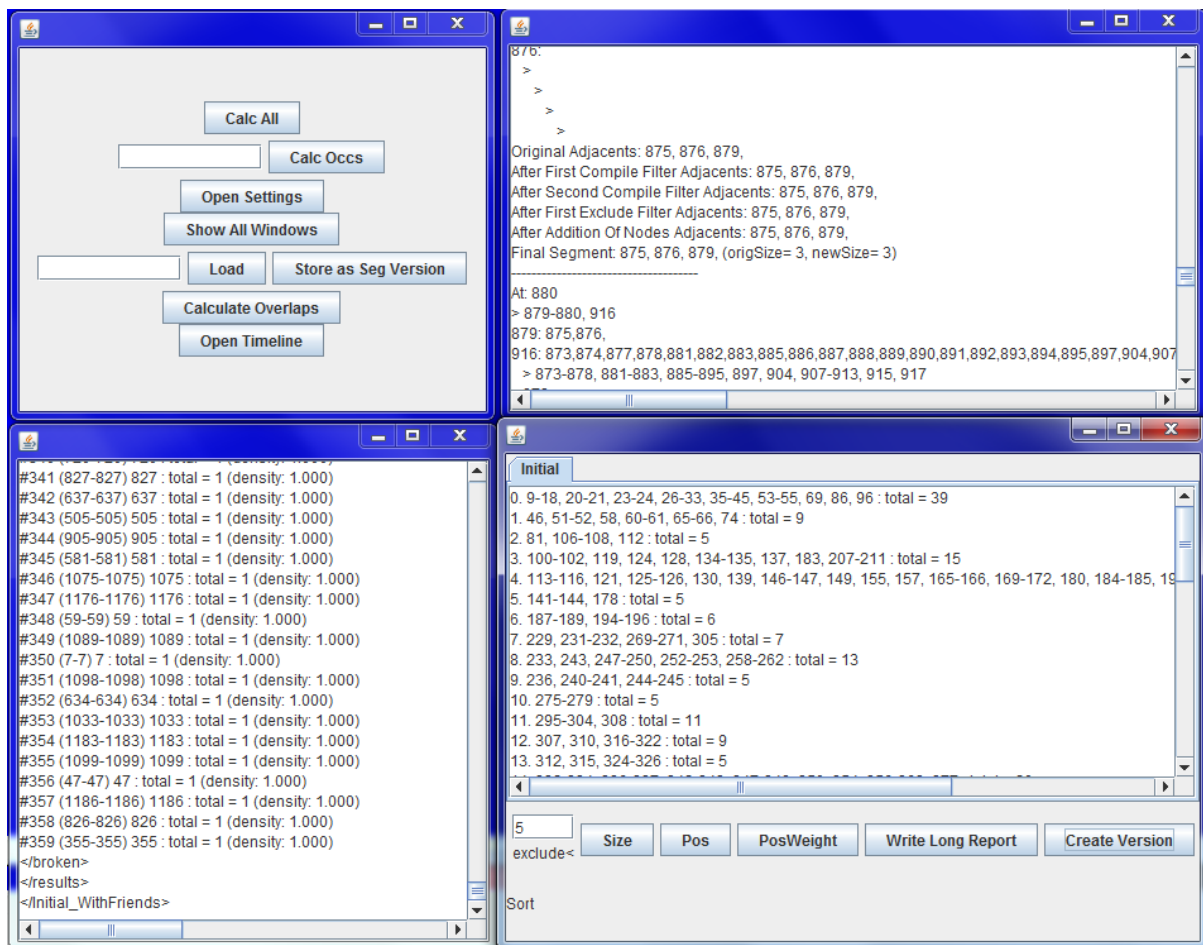
The generation of Segments will populate the ‘Machine-Segmenting View’ console and link window as described in the next section.

Pressing the “Calculate Overlaps” button generates data on the overlap of Machine Segments with human segments and stores this data as an HTML file. The format of this file is described in detail in Section 9.5.3.11.5.

### **9.5.3.11.2**    *Generating Segments*

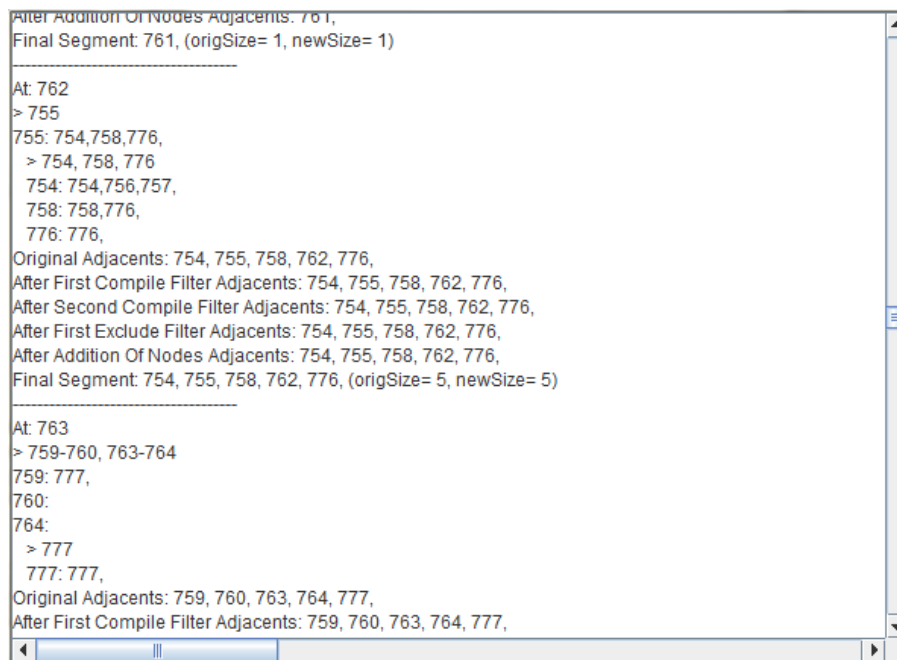
#### **9.5.3.11.2.1 SCORE Analyser Machine-Segmenting View**

When Segments are generated the full list of generated Segments is displayed in the ‘Machine-Generation Output Console’ which is part of the SCORE Analyser ‘Machine-Segment Generation View’ (Figure 165, bottom-right).



**Figure 165: SCORE Analyser Machine-Segment Generation Views; Control Panel (upper-left), Debug Console (upper-right), Output console (lower-left), Links View (lower-right)**

Machine-Segmenting via the SCORE Analyser also provides debugging output shown in the Machine-‘Machine-Generation Debug Console’ (Figure 166). This output is generated by machine generation segments writing to a special Debug object, and can be used to analyse the inner workings of a Machine-Segmenting algorithm, such as how Changes are identified or filtered out by different extension algorithms. Depending on the researcher’s use of the Debug object these statements can be very verbose, in which case the debug functionality should best be used with a limited number of Machine-Segmenting runs lest the text content of the window place too large a burden on available memory.



**Figure 166: Machine-Segment Generation Debug Console**

### **9.5.3.11.3 Navigating Segments using the Machine-Generation Links View and Change Browser**

When Segments are generated the ‘Machine-Generation Links View’ (Figure 167) is populated with a list of all the Segments (initially ordered in descending order of size, though pressing the “Pos” button will sort Segments in order of overall position of the Segment’s Changes). When one of the Segments is clicked in the ‘Machine-Generation Links View’ the Segment’s Changes are loaded into the ‘Change Browser’ (it must be open for this functionality to work) which allows the researcher to view all the modifications occurring in the Changes belonging to the Machine Segment, which allows the researcher to efficiently gain an understanding of the content of a machine-generated segment, as well as (through in-depth study of these modifications) an understanding of whether the segment is actually describing related Changes.



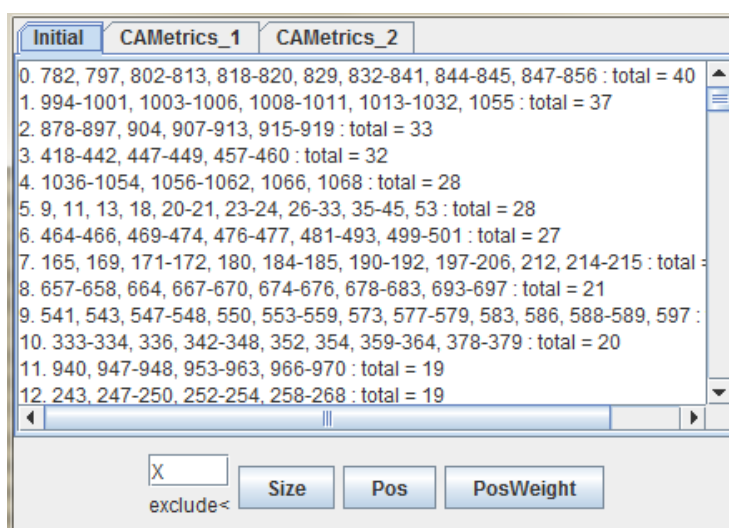


Figure 167: Machine-Generation Links View

Generated Segments can be imported into the 'Segmenting View' as a new Segmentation version in the 'Segment-Coding View' (see Section 9.5.3.10.1) by pressing the "Store as Seg Version" button in the 'Machine-Segmenting Control Panel'.

#### 9.5.3.11.4 Timeline View showing Machine-Segmenting Results

Machine-Generated Segments can also be visualised via the 'Timeline View' (see Figure 168) discussed in Section 9.5.3.4. To view Segments in the 'Timeline View', press the "Open Timeline" button in the 'Segment Controls View'. The 'Timeline View' provides a visual comparison between human-identified Segments and Segments from an arbitrary number of machine-generation runs.

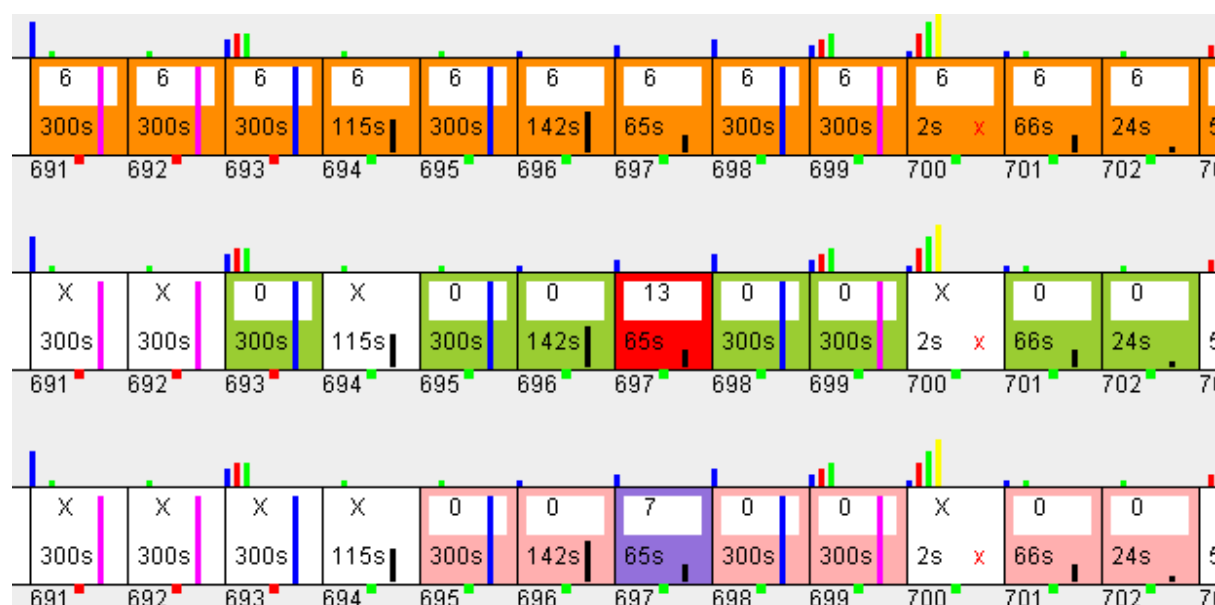


Figure 168: The Timeline view; the top row is shows human-identified Segments, the bottom two rows show the Segmenting created by different runs of a machine algorithm

The white box containing a number shows the id of the segment to which the Change is assigned. For example in Figure 168 for the first Machine-Segmentation run (displayed in the second row) six Changes are assigned to the same segment with id = 0. The segment id is also visually represented via the box colour, allowing for fast identification of ‘related’ Changes making up a segment.

The top row in the ‘Timeline View’ shows the human-identified segments as stored via the ‘Segment-Coding View’, whereas each row below that shows segments produced by a machine-generation run carried out during the previous machine-generation of segment candidates. This allows the researcher to quickly visually compare how the segments produced by different machine-generation runs match up with the ‘correct’ segments identified by the human researcher.

#### **9.5.3.11.5 Machine-Segmenting HTML Output**

Generation of Machine Segments (whether from inside the SCORE Analyser or via the stand-alone application) produces files which store the ranked list of segments produced by different runs during generation as well as several metrics calculated to enable evaluation of Machine-Segmenting algorithms and settings according to the quality of segments produced and the number of ‘interesting’ Changes identified.

For each assignment processed, two files are produced. One is a text file which contains the ranked list of segments for each run. This file can be loaded into the SCORE Analyser to populate the various segment generation views presented in the last section. The second is an .HTML file which contains metrics used for evaluation purposes, as well as a human-readable list of machine-generated segments. When a set of assignments is processed via the stand-alone Machine-Segmenting application, a summary HTML document containing averages for several of the metrics to allow for easy comparison for groups of assignments is also generated. If no comparison set of human-identified segments is provided then these files simply serve to list machine-generated segments. If a set of pre-identified (usually via manual human analysis) segments is provided during segment generation, then the files also include data on how well the machine-generated segments match/overlap the pre/human-identified segments. Human-identified segments are entered via the *Segment View* described in Section 9.5.3.10 and can then be accessed by the machine-generation algorithm.

##### **9.5.3.11.5.1 Individual-Assignment HTML Data**

Individual-Assignment data files include a ranked list of machine-identified segments as shown in Figure 169 and Figure 170. Each segment is listed and described in descending order of segment size. The description includes the Changes which make up the segment (in sequential order), as well as the size of the segment (number of Changes that make up the segment). If a set of pre-identified



195, 227, 234, 242, 244, 246, 250-252, 274-276, 296, 298-299, 301-302, 304-305, 308-315, 317-327, 330-331, 335-338, 346-348, 384-385, 387-388 ( : total = 51)

8	Furniture adding	227, 242, 244, 252 ( : total = 4)	18.181818
9	Logic Ops	304-305, 309, 315, 317-319, 322-327, 346-348 ( : total = 16)	43.243244

Candidate-HuId Overlap: 39.215687

1079-1090, 1095-1098, 1100-1109, 1112, 1118-1125 ( : total = 35)

10	Line stipple	1079-1090, 1095-1098, 1100-1109, 1112, 1118-1125 ( : total = 35)	72.91667
----	--------------	--	----------

Candidate-HuId Overlap: 100.0

526, 529-531, 536, 546, 551, 590-593, 596, 599, 609-615, 622, 627, 630-632 ( : total = 25)

Candidate-HuId Overlap: 0.0

421-432, 434-444, 454-455 ( : total = 25)

2	Rotate Icon	421-432, 434-444, 454-455 ( : total = 25)	60.975613
---	-------------	---	-----------

Candidate-HuId Overlap: 100.0

758 761 763 765-770 772-777 780-788 ( : total = 24)

Figure 170: Close-up of machine-generated segment descriptions from the HTML Evaluation data document

#### 9.5.3.11.5.2 Human -> Machine (Human Segments)

*Human to Machine Data (Human Segments)* The *Human to Machine Data* section presents all human-identified segments, and shows which machine-identified segments contain Changes from that human-identified segment. An example is shown in Figure 171. The first line shows the name of the human-identified segment and its ID as well as the Changes associated with it. The table below the human segment's data lists all Machine Segments that contain at least one of the human segment's Changes. For each Machine Segment it shows the overlapped Changes, as well as the percentage total Changes of the human-identified segment overlapped by the machine-generated segment. This allows the researcher to develop an understanding of how a human-identified segment's Changes are distributed, and how it may be possible to link the machine-generated segments containing the human-identified segment's Changes.

#### 7. Child object rotate 1140-1152, 1168-1210 ( : total = 56)(score: 0.023286872)

6	1176, 1184, 1190-1192, 1200, 1202, 1208-1209 ( : total = 9)	16.071428
13	1204, 1210 ( : total = 2)	3.5714288
31	1181, 1185, 1188 ( : total = 3)	5.357143
32	1178, 1189 ( : total = 2)	3.5714288
34	1145, 1170 ( : total = 2)	3.5714288
Total	18	32.142857

Identified : 1145, 1170, 1176, 1178, 1181, 1184, 1185, 1188, 1189, 1190, 1191, 1192, 1200, 1202, 1204, 1208, 1209, 1210,

Not Identified : 1140, 1141, 1142, 1143, 1144, 1146, 1147, 1148, 1149, 1150, 1151, 1152, 1168, 1169, 1171, 1172, 1173, 1174, 1175, 1177, 1179, 1180, 1182, 1183, 1186, 1187, 1193, 1194, 1195, 1196, 1197, 1198, 1199, 1201, 1203, 1205, 1206, 1207,

Figure 171: Human-Identified Segment to Machine-Identified Segments

#### 9.5.3.11.5.3 Selected -> AccPrec (AccPrecSoFar)

The *Selected to Recall/Precision* section shows how recall and precision as well as p-value and average segment size change with each additional segment selected from the ranked list (see Figure 172 for a description of these metrics). The rows in the table correspond to segments in the ranked list shown in the *Candidate Segments* section. Note that the value referred to by 'Acc' or 'Accuracy' is in fact the *recall* value. The Overlap% value shows what percentage of that segment's Changes are 'interesting'. This view of the data allows an overview of how recall and precision develop with each additional segment added. For example, in the table above it can be seen that after segment #6, several segments in succession (7-10) each decrease precision significantly. The number of trials used to calculate p-values (see Section 7.4.2 for calculation of p-values) is smaller than that used for the final calculated p-value, hence the p-values are less accurate and should serve as approximations only. They should not be reported.

Seg#	S.Size	S.OvL	Overlap%	AccSoFar	PrecSoFar	A. P-Mean	P-Exp	Occurrences
0	14	14	1.000	0.046	1.000	0.250	0.050 (1.000)	, (203-204, 207-218)
1	13	13	1.000	0.088	1.000	0.250	0.090 (1.000)	, (1097-1109)
2	12	1	0.083	0.092	0.718	0.250	0.070 (0.730)	65-66, 78-80, 83-87, 89, (104)
3	10	10	1.000	0.124	0.776	0.442	0.060 (0.620)	, (424, 430, 432, 435-436, 438-442)
4	7	0	0.000	0.124	0.679	0.442	0.030 (0.690)	55, 68-72, 94
5	6	6	1.000	0.144	0.710	0.058	0.040 (0.690)	, (1086, 1089-1090, 1118-1119, 1124)
6	5	4	0.800	0.157	0.716	0.058	0.010 (0.650)	, (97, 105, 112-113), 116
7	4	0	0.000	0.157	0.676	0.150	0.030 (0.690)	1230-1233
8	4	0	0.000	0.157	0.640	0.150	0.050 (0.710)	989-991, 1001
9	4	0	0.000	0.157	0.608	0.150	0.050 (0.860)	767-768, 775-776
10	4	1	0.250	0.160	0.590	0.150	0.060 (0.850)	673, 677-678, (723)
11	4	4	1.000	0.173	0.609	0.150	0.040 (0.710)	(400-402, 404)

Figure 172: Percentage of Selected Changes to Recall and Precision (RecPrecSoFar)

#### 9.5.3.11.5.4 Acc -> Prec (AccToPrec)

*Recall to Precision (AccToPrec)* The *Recall to Precision* section shows precision, A/E ratio (Act/Prec), the number of segments selected, average segment size and p-value (P-Exp) for all segments producing recall of 0.1, 0.2 and so on with an interval of 0.1. Since selection of segments will usually not produce exact accuracies, the number of segments selected will be the smallest number of segments for which the total number of 'interesting' Changes identified *exceed* the accuracy level.

For example, in Figure 173 sixty-one segments are required to produce recall of 30%, with the actual recall produced by including these 61 segments being 30.4%. When segments producing a recall of 30% are selected, a precision of 43.9% is achieved, producing an A/E ratio of 1.769. Average segment size for the 61 segments is 3.475 (very small), and the p-value associated with the selection of these

61 segments is  $p=0.03$ . As can be seen in the table, A/E ratio decreases rapidly over different recall levels, demonstrating the trade-off between recall and precision.

>Acc	Acc	Exp Acc	Prec	Exp Prec	Act/Prec	seg#	avgSegSize	P-Mean	P-Exp
0.100	0.124	0.040	0.776	0.248	3.127	4	12.250	0.442	0.040 (0.610)
0.200	0.209	0.093	0.557	0.248	2.244	21	5.476	0.256	0.050 (0.860)
0.300	0.304	0.172	0.439	0.248	1.769	61	3.475	0.313	0.030 (0.840)
0.400	0.402	0.252	0.395	0.248	1.595	125	2.488	0.267	0.060 (0.900)
0.500	0.503	0.296	0.422	0.248	1.701	179	2.039	0.046	0.000 (0.720)
0.600	0.601	0.558	0.267	0.248	1.079	502	1.371	0.434	0.310 (0.990)
0.700	0.703	0.751	0.232	0.248	0.935	741	1.251	0.733	0.610 (1.000)
0.800	0.801	0.801	0.248	0.248	1.000	802	1.232	0.687	0.440 (0.990)
0.900	0.902	0.881	0.254	0.248	1.024	901	1.206	0.633	0.340 (1.000)

**Figure 173: Recall to Precision (RecToPrec)**

The *Recall to Precision* view can be used to gain an understanding of recall-precision trade-offs, as well as an understanding of segment sizes required to achieve a certain level of recall. Small segments tend to be much poorer sources of ‘interesting’ Changes, and hence low average segment sizes (as can be observed even at the 20% accuracy level) indicate that the segmenting algorithm is not producing sufficient linking of related Changes.

#### 9.5.3.11.5.5 AccPrec per Segment (MachineOccsToAccPrec)

The *Total Selected to Recall-Precision* view (see Figure 174) shows the same data as the *AccToPrec* view, but instead of calculating data for the  $x$  top segments whose recall falls above a certain level, it shows data for the  $y$  top segments for which the sum of all the segment’s sizes equals or exceeds a percentage of the total Changes in the Project History. The levels start at 0.1 and are incremented in 0.1 intervals. By analysing A/E ratios (Act/Prec) in the above table, it becomes clear that somewhere between the 0.2 and 0.3 level the size of selected segments becomes 1, at which further selection is in fact useless since there is no way to identify ‘interesting’ segments from the set of segments of size 1 (since segments are selected based on their rank which is determined by size). Hence for the data and segmenting algorithm used, segments totalling no more than 20-30% of total Changes should be selected, since any additional segments will be selected essentially at random.



>%Total	%Total	Acc	Exp Acc	Prec	Exp Prec	Act/Prec	seg#	avgSegSize	P-Mean	P-Exp
0.100	0.100	0.209	0.100	0.516	0.248	2.081	24	5.167	0.256	0.060 (0.860)
0.200	0.201	0.317	0.201	0.391	0.248	1.577	79	3.139	0.399	0.080 (0.940)
0.300	0.301	0.523	0.301	0.431	0.248	1.739	185	2.005	0.046	0.010 (0.760)
0.400	0.400	0.588	0.400	0.364	0.248	1.469	308	1.604	0.238	0.020 (0.860)
0.500	0.501	0.588	0.501	0.291	0.248	1.175	432	1.431	0.500	0.210 (0.990)
0.600	0.600	0.621	0.600	0.256	0.248	1.034	555	1.335	0.576	0.380 (1.000)
0.700	0.700	0.621	0.700	0.220	0.248	0.887	678	1.274	0.831	0.690 (1.000)
0.800	0.801	0.801	0.801	0.248	0.248	1.000	802	1.232	0.687	0.480 (1.000)
0.900	0.900	0.951	0.900	0.262	0.248	1.056	925	1.201	0.744	0.190 (0.960)

**Figure 174: Recall and Precision by % of total Changes selected (MachineOccsToRecPrec)**

#### 9.5.3.11.5.6 Statistics (Statistics)

The *Statistics* section (see Figure 175) shows statistics for each run of Line History Generation. These statistics are calculated using the largest segments for which the total sum of all their Changes equals the percentage of total Changes specified by the cut-off value entered through the Machine Segmentation Settings window discussed in Section 9.5.3.11.1. The statistics are described in more detail in Section 7.4.

Setting	Candidate	Overlap	Human	Total	ActualPrec	ExpectedPrec	ActualAcc	ExpectedAcc	Diff	Max. Ratio	% of Max. Ratio	Exp. P-Value
Initial	247	101	306	1234	0.40890688	0.24797407	0.33006537	0.20016207	1.6489905	4.0326796	0.4089069	0.057
CAMetrics_1	247	111	306	1234	0.4493927	0.24797407	0.3627451	0.20016207	1.8122568	4.0326796	0.4493927	0.024
CAMetrics_2	247	121	306	1234	0.48987854	0.24797407	0.39542484	0.20016207	1.9755232	4.0326796	0.48987857	0.009
CAMetrics_3	247	133	306	1234	0.53846157	0.24797407	0.43464053	0.20016207	2.171443	4.0326796	0.53846157	0.003
CAMetrics_4	247	102	306	1234	0.41295546	0.24797407	0.33333334	0.20016207	1.6653172	4.0326796	0.4129555	0.051

**Figure 175: Evaluation Summary Statistics**

*Group Summary* When segments are generated for a group of assignments using the stand-alone segment generation application, the application also produces a summary HTML document. The summary document contains a table (see Figure 176) that is similar to the *Statistics* table shown in individual reports, but instead of reporting on metrics for different runs for individual students, it reports the average of these metrics calculated across all of the assignments which were machine-segmented, as well as the total number of assignments which achieved significance at a p-level of  $p < 0.05$  and  $p < 0.01$ . This summary allows the researcher to understand how well an algorithm is working for a set of assignments since performance among individual assignments can vary substantially.

Setting	Prec	Acc	Ratio	PVal	<0.05	<0.01
Initial	0.30728123	0.31023434	1.547746	0.132028	1	1
CAMetrics_1	0.35603946	0.3637398	1.8145363	0.069504	3	1
CAMetrics_2	0.40465602	0.42990452	2.1444259	0.018578	5	3
CAMetrics_3	0.41697535	0.4437878	2.213614	0.019982003	4	3
CAMetrics_4	0.3628372	0.39853558	1.9876773	0.037088	3	2

**Figure 176: Project Summary Evaluation Statistics**

### 9.5.3.12 Conclusion

In its current form the SCORE Analyser falls somewhere between an alpha and a beta application; because of its continual development throughout the research project for which it was developed and the lack of development resources (a single developer who also had to use it to conduct analysis and write up a thesis) the SCORE Analyser still includes many undocumented bugs and pitfalls, lacks many desirable features and is not well-designed from an architectural standpoint (as it grew organically from a prototype). It is hoped that these limitations can be addressed in future revisions of the SCORE Analyser software. The main purpose of the manual is to provide interested parties with the ability to utilise the core functionality to understand the research approach developed in this doctoral dissertation and its potential as a research method in Computer Science Education.

Also as this manual is intended to serve as an introduction to the SCORE Analyser's main functionality many options are not described in detail or omitted altogether to keep this manual as concise as possible as they are not part of the core functionality.

## 9.6 Change-Coding Method

### 9.6.1 Description of the Initial Classification Scheme

Action Dimension Categories	
Assembly	Using transform commands such as glTranslate and glRotate to assemble an object such as the avatar.
Placement	Placing an object using transformation commands.
Drawing	Producing an OpenGL primitive (two-dimensional or three-dimensional) through the use of glBegin/glEnd and glVertex commands.
Animations	Producing an animation via time-based modification of transformation commands.
Projection	Creation of a projection through the use of glFrustum, glOrtho, gluOrtho2D or gluPerspective.



Viewing	Implementation of a Viewing model, producing automatic or manual manipulation of the virtual camera.
Data Structure	Implementation of a data structure or class.
Lighting	Implementation of OpenGL lighting.
GUI	Implementation of user-interface functionality such as buttons or menus.
Debug	Actions such as commenting out of source code or addition of debug cout statements.
Cleanup	Removal of dead code, change of variable names or other cosmetic changes to source code.
Other	Any action not falling into one of the other categories.

Error Dimension Categories	
General Syntax	A syntax error such as omission of a semi-colon.
General Semantics	A semantic error, such as incorrect placement of a closing bracket including unintended statements.
OpenGL Syntax	A syntax error applying to an OpenGL command
OpenGL Semantics	A semantic error applying to an OpenGL command
2D Coordinate	An error involving an incorrect 2D coordinate for the glVertex2x command
3D Coordinate	An error involving an incorrect 3D coordinate for the glVertex3x command
2D Transform	An error involving an incorrect value for a 2D transformation such as glRotate2x
3D Transform	An error involving an incorrect value for a 3D transformation such as glRotate3x
Projection	An error involving an incorrect projection, usually involving a projection call such as glOrtho or gluPerspective
Viewing	An error occurring for a view command such as gluLookAt or a transformation such as glRotate intended to modify a view
Lighting	An error in a command relating to OpenGL lighting
State Machine or Pipeline	An error caused by the incorrect placement of a command in the program's flow such as the placement of a glVertex command without a glBegin command
Timing or Iteration	An error caused by incorrect time-based behaviour such as an animation being executed to quickly.
Cleanup	An error occurring during clean-up, such as the accidental removal of a necessary command

Other	An error not fitting into any of the above categories
None	No error occurred.

Output Dimension Categories	
No Compile	The application failed to compile.
Crash	The application compiles but crashes at some point during its execution.
No Display	The application executes but produces no visual output, for example when the student fails to bind the display function.
Skewed	The application produces skewed output. For example, the unintentional application of two projections will cause a skewed view.
Missing Items	Some items do not appear on the drawing surface.
Misplaced Items	Some items do not appear at the location intended by the student.
Missized items	Some items are not of the size intended by the student.
Wrong Animation	An animation does not look correct.
Wrong Lighting	Incorrect lighting, such as a scene being completely dark.
Wrong View	An incorrect View not showing the expected part of the scene.
Wrong Visual	A problem with visual output not falling into one of the more specific categories.
Other	A problem not falling into one of the other categories.
Correct	No problem exists in the Change's output.

## 9.6.2 Raw Coding Data

### 9.6.2.1 Totals

**Table 43: Assignment 1 Overall Raw Coding Data**

	Action Time	Action Changes	Error Time	Error Changes	Problem Time	Problem Changes
CombinedSpatial	63241	1356	20734	412	32767	678
Major Spatial	23457	494	10911	226	16496	356
Minor Spatial	29712	698	5571	128	7213	166
Math Spatial	10072	164	4252	58	9058	156
General	132652	3082	39901	762	96757	2085

Programming						
Event-Driven	51089	812	7255	112	17243	293
General OpenGL	17532	403	6380	119	6521	139
Pipeline	14383	300	7088	149	15336	340
Other	10411	247	2772	63	1612	46
Bad	10235	345	10722	348	10126	342
None	-	-	204543	4583	118322	2609
	299784	6556	299545	6553	298742	6535

**Table 44: Assignment 3 Overall Raw Coding Data**

	Action Time	Action Changes	Error Time	Error Changes	Problem Time	Problem Changes
CombinedSpatial	76353	1412	24701	433	33124	654
Major Spatial	48551	850	20971	369	28502	561
Minor Spatial	24225	506	2028	40	1702	42
Math Spatial	3577	56	1702	24	2920	51
General Programming	54087	1223	16432	275	40261	856
Event-Driven	14796	281	1072	19	2030	43
Viewing or Projection	11234	194	5802	92	6697	118
Animation	11111	188	2006	32	7356	121
General OpenGL	4638	101	2450	44	2891	42
Pipeline	3403	60	1369	23	4321	61
Lighting	5352	136	544	15	901	25
Unknown	240	11	245	12	245	12
Other	8091	191	1231	20	1129	15
Bad	7985	219	7914	217	7914	217
None	0	0	133429	2832	90362	1851
	197290	4016	197195	4014	197231	4015

## 9.7 Segment-Coding Method

### 9.7.1 Generation Algorithm Pseudo-Code

```
//Calculate Line History for each file in turn
foreach (FileHistory : ProjectHistory)
{
    //Calculate Line History by matching lines at each Change
    foreach (FileChange : FileHistory)
    {
        prevDocumentLines = Change.prevDocumentLines;
        curDocumentLines = Change.curDocumentLines;

        //Calculate MAINTAINED lines
        LCS = calculateLCS();
        foreach (Pair <LineOld, LineCur> : LCS)
        {
            matchMaintainedLine(LineOld, LineNew);
            prevDocumentLines.remove(LineOld);
            curDocumentLines.remove(LineCur);
        }

        //Calculate MOVED lines
        foreach(LineOld : prevDocumentLines)
        {
            foreach(LineCur : curDocumentLines)
            {
                if(LineOld == LineCur)
                {
                    matchMovedLine(LineOld, LineNew);
                    prevDocumentLines.remove(LineOld);
                    curDocumentLines.remove(LineCur);
                    break;
                }
            }
        }

        //Calculate MUTANT lines
        foreach(LineOld : prevDocumentLines)
        {
            foreach(LineCur : curDocumentLines)
            {
```

```

        distance = calculateDistance(LineOld, LineCur);
        distances.add(Pair <Pair <LineOld, LineCur>, distance>);
    }
}
sort(distances);
foreach(Pair <Pair <LineOld, LineCur>, distance> : distances)
{
    if(distance > mutantLevenstheinBoundary)
    {
        break;
    }
    matchMutantLine(LineOld, LineNew);
    prevDocumentLines.remove(LineOld);
    curDocumentLines.remove(LineCur);
}

//Calculate GHOST lines
foreach(LineGhost : ghostDocumentLines)
{
    foreach(LineCur : curDocumentLines)
    {
        distance = calculateDistance(LineGhost, LineCur);
        distances.add(Pair <Pair <LineGhost, LineCur>,
distance>);
    }
}
sort(distances);
foreach(Pair <Pair <LineGhost, LineCur>, distance> : distances)
{
    if(distance > ghostLevenstheinBoundary)
    {
        break;
    }
    matchMutantLine(LineGhost, LineNew);
    ghostDocumentLines.remove(LineOld);
    curDocumentLines.remove(LineCur);
}

//Calculate ADDED lines
foreach(LineCur : curDocumentLines)
{

```

```

        addAddedLine (LineCur);
        curDocumentLines.remove (LineCur);
    }

    //Calculate DELETED lines
    foreach (LineOld : prevDocumentLines)
    {
        addDeletedLine (LineOld);
        prevDocumentLines.remove (LineOld);
    }
}
}

```

### 9.7.2 Performance of Line History Generation algorithm using different settings with LH-Graph generation

The algorithm for the generation of Line Histories is described in Section 7.2.1. The algorithm's implementation takes three parameters.

The *MutantLevenstheinLimit* parameter specifies the limit below which two lines will be matched as Mutants.

The *GhostLevenstheinLimit* parameter specifies the limit below which two lines will be matched as Ghosts, while the *GhostDistance* specifies the size of the window in which previously deleted lines are stored and used as ghost candidates.

Lowering the Limit parameters loosens the similarity requirement between lines to be matched, meaning that less similar lines will still be accepted as mutants/ghosts. Decreasing the *GhostDistance* parameter means that fewer previous Changes will be included in the ghost generation window, hence the number of ghost candidates (which is the set of lines deleted in recent Changes) is restricted to lines that were more recently deleted.

It would be possible to evaluate the accuracy and recall for the algorithm using different settings in order to find the optimal combination of settings for producing accurate mutant and ghost mappings. However, this evaluation would be extremely time-consuming. Instead, the Line History Generation algorithm is first evaluated by how well the different settings perform in machine-generation of segments with the LH-Graph algorithm which is described in more detail in Section 7.6.5. The Line History Table generated by the best-performing setting is then evaluated in terms of precision and recall in the next section.

A detailed explanation of the different performance measurements utilised in the evaluation of the LH-Graph algorithm is given in Section 7.4. For now, suffice it to say that the Ratio performance measure shows how many significant Changes are identified compared to how many significant Changes a random algorithm would on average identify. For example, if an algorithm identified twice as many Changes as a random algorithm would on average, it would have a ratio of 2.0. Therefore, high ratios indicate better performance.

Summarised results for the best-performing LH-Graph settings are presented above. Full results utilising a wide range of LH-Graph settings can be found in the appendix (Appendix Section 9.7.2).

Table 45 shows the results of mutant-only generation (no ghost generation) using a maximum Levensthein distance of 0.1, 0.25 and 0.5. The best ratio is produced by the most lenient MutantLevenstheinLimit of m=0.5.

**Table 45: Data for LH-Graph Generation with different line history generation settings**

Gen. Setting	LH-Graph Setting	Ratio	% of Max Ratio	Fit*Spread	Ratio P-Value	p > 0.05	p > 0.01
m=0.1	50/5	1.529	0.657	0.45	0.012	9	6
m=0.25	150/2	1.558	0.663	0.431	0.005	10	7
m=0.5	50/3	1.573	0.675	0.47	0.008	9	9

Table 46 shows the results of combined Mutant and Ghost generation. The MutantLevenstheinLimit m=0.5 is used for all different trials, as it was the best-performing setting of the mutant-only evaluation stage. Evaluation is performed with GhostLevenstheinLimit values of (0.1, 0.25, 0.5) for a GhostDistance of 5 and a GhostDistance of 10. The best-performing runs are those using (g=0.1, gd=5) and (g=0.1, gd=10), so it appears that a relatively restrictive Levensthein limit of g=0.1 provides highest performance, while the choice of 5 or 10 as maximum distance seems relatively insignificant. The setting (g=0.1, gd=10) was chosen as the ‘best’ setting. It will be evaluated for precision and recall in the next section, and it is also the setting that will be used to generate Line History Tables for the evaluation of machine algorithms presented in Chapter 7.

**Table 46: Data for LH-Graph Generation with different line history generation settings**

Gen. Setting	LH-Graph Setting	Ratio	% of Max Ratio	Fit*Spread	Ratio P-Value	p > 0.05	p > 0.01
gd=5							
g=0.1	75/2	1.576	68.9%	0.452	0.004	10	8
g=0.25	75/2	1.447	77.7%	0.510	0.004	10	8
g=0.5	150/2	1.562	66.9%	0.423	0.007	10	7
gd=10							
g=0.1	50/3	1.573	68.6%	0.46	0.006	10	8
g=0.25	150/2	1.571	67.3%	0.419	0.005	10	8
g=0.5	150/2	1.565	67.6%	0.426	0.005	10	9

### 9.7.3 Generation Algorithm Full Evaluation Data

#### 9.7.3.1 mutant=0.1 ghost=X

**Table 47: Generation results for Assignment 1 using the settings mutant=0.1 ghost=X**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread* Fit	PVal	<0.05	<0.01
25/2	0.520	0.398	1.59	3.126	52.0%	0.511	0.823	0.420	0.016	4	3
50/2	0.509	0.39	1.556	3.126	50.9%	0.509	0.789	0.402	0.017	4	3
75/2	0.505	0.387	1.544	3.126	50.5%	0.511	0.744	0.380	0.021	4	3
150/2	0.509	0.391	1.560	3.126	50.9%	0.513	0.726	0.373	0.016	4	3



25/3	0.525	0.40 4	1.61 4	3.126	52.5%	0.564	0.7 58	0.427	0.0 08	5	4
50/3	0.534	0.41	1.63 8	3.126	53.4%	0.568	0.7 35	0.417	0.0 08	5	4
75/3	0.527	0.40 4	1.61 4	3.126	52.7%	0.569	0.7 21	0.411	0.0 1	5	4
150/3	0.546	0.42	1.67 6	3.126	54.6%	0.574	0.6 44	0.369	0.0 05	5	3
25/5	0.552	0.42 4	1.69 2	3.126	55.2%	0.586	0.7 39	0.433	0.0 05	5	4
50/5	0.56	0.42 9	1.71 3	3.126	56.0%	0.587	0.7	0.411	0.0 04	5	4
75/5	0.545	0.41 8	1.67	3.126	54.5%	0.586	0.6 74	0.395	0.0 06	5	4
150/5	0.536	0.41 4	1.65 4	3.126	53.6%	0.593	0.6 24	0.370	0.0 11	5	3

**Table 48: Generation results for Assignment 3 using the settings mutant=0.1 ghost=X**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread*	PVal	<0.05	<0.01
25/2	0.789	0.361	1.438	1.919	78.9%	0.677	0.79	0.535	0.006	5	4
50/2	0.77	0.353	1.409	1.919	77.0%	0.69	0.752	0.518	0.007	5	3
75/2	0.773	0.355	1.416	1.919	77.3%	0.699	0.71	0.496	0.007	5	4
150/2	0.749	0.344	1.370	1.919	74.9%	0.703	0.667	0.469	0.012	5	3
25/3	0.745	0.340	1.357	1.919	74.5%	0.732	0.753	0.551	0.016	4	4
50/3	0.734	0.337	1.344	1.919	73.4%	0.750	0.678	0.509	0.021	4	3

75/3	0.70 6	0.32 4	1.29 2	1.919	70.6%	0.761	0.63 4	0.482	0.03 3	3	1
150/3	0.71 0	0.32 7	1.30 4	1.919	71.0%	0.769	0.56	0.43	0.04 5	3	1
25/5	0.72 1	0.33 5	1.33 5	1.919	72.1%	0.767	0.69 7	0.534	0.01 4	5	1
50/5	0.72 8	0.33 7	1.34 5	1.919	72.8%	0.79	0.61 9	0.489	0.02 07	4	2
75/5	0.71 6	0.32 9	1.31	1.919	71.6%	0.798	0.60 1	0.479	0.02 6	3	2
150/5	0.70 7	0.32 9	1.31 3	1.919	70.7%	0.809	0.51 1	0.414	0.06 5	3	2

**Table 49: Generation results using the settings mutant=0.1 ghost=X averaged across Assignment 1 and Assignment 3**

Setting	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
25/2	1.514	0.655	0.478	0.011	9	7
50/2	1.482	0.639	0.46	0.012	9	6
75/2	1.48	0.639	0.438	0.014	9	7
150/2	1.465	0.629	0.421	0.014	9	6
25/3	1.486	0.635	0.489	0.012	9	8
50/3	1.49	0.634	0.463	0.015	9	7
75/3	1.453	0.617	0.446	0.021	8	5
150/3	1.49	0.628	0.4	0.025	8	4
25/5	1.514	0.637	0.484	0.01	10	5
50/5	1.529	0.644	0.45	0.012	9	6
75/5	1.49	0.631	0.437	0.016	8	6
150/5	1.483	0.621	0.392	0.038	8	5

### 9.7.3.2 mutant=0.25 ghost=X

**Table 50: Generation results for Assignment 1 using the settings mutant=0.25 ghost=X**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread*	PVal	<0.05	<0.01
25/2	0.539	0.413	1.649	3.126	0.539	0.541	0.793	0.429	0.012	5	3
50/2	0.537	0.412	1.645	3.126	0.537	0.545	0.766	0.418	0.009	5	4
75/2	0.534	0.417	1.637	3.126	0.534	0.544	0.751	0.409	0.01	5	4
150/2	0.547	0.421	1.680	3.126	0.547	0.549	0.697	0.382	0.006	5	3
25/3	0.555	0.426	1.701	3.126	0.555	0.579	0.757	0.439	0.004	5	4
50/3	0.556	0.427	1.704	3.126	0.556	0.587	0.729	0.428	0.003	5	4
75/3	0.553	0.425	1.697	3.126	0.553	0.586	0.694	0.407	0.003	5	5
150/3	0.552	0.427	1.706	3.126	0.552	0.589	0.633	0.373	0.003	5	5
25/5	0.56	0.431	1.721	3.126	0.56	0.596	0.736	0.438	0.003	5	4
50/5	0.56	0.431	1.723	3.126	0.56	0.6	0.682	0.409	0.003	5	4
75/5	0.551	0.425	1.698	3.126	0.551	0.603	0.672	0.405	0.003	5	5
150/5	0.54	0.418	1.67	3.126	0.54	0.61	0.594	0.362	0.009	5	3

**Table 51: Generation results for Assignment 3 using the settings mutant=0.25 ghost=X  
using the settings mutant=0.25 ghost=X**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread*	PVal	<0.05	<0.01
---------	------	-----	-------	-----------	-------------	--------	-----	---------	------	-------	-------

25/2	0.78 2	0.35 9	1.43	1.919	78.2%	0.706	0.79 5	0.561	0.00 3	5	5
50/2	0.77 1	0.35 6	1.41 7	1.919	77.1%	0.728	0.72 4	0.528	0.00 3	5	5
75/2	0.79 6	0.36 6	1.46 0	1.919	79.6%	0.741	0.68 2	0.505	0.00 2	5	5
150/ 2	0.77 8	0.36	1.43 6	1.919	77.8%	0.745	0.64 5	0.481	0.00 4	5	4
25/3	0.71 8	0.33 0	1.31 7	1.919	71.8%	0.753	0.72 3	0.545	0.02	5	2
50/3	0.72 2	0.33 3	1.32 9	1.919	72.2%	0.779	0.65 4	0.509	0.01 48	5	1
75/3	0.71 8	0.33 1	1.31 8	1.919	71.8%	0.79	0.60 1	0.475	0.03 2	4	2
150/ 3	0.69 2	0.31 9	1.27 4	1.919	69.2%	0.795	0.54 5	0.433	0.05 3	4	1
25/5	0.71 1	0.32 9	1.31	1.919	71.1%	0.783	0.69 4	0.544	0.01 9	5	0
50/5	0.72 8	0.34 2	1.36 3	1.919	72.8%	0.802	0.62 1	0.498	0.02 4	4	2
75/5	0.69 3	0.31 9	1.27 1	1.919	69.3%	0.814	0.56 6	0.461	0.04 13	3	1
150/ 5	0.67 7	0.31 6	1.26 1	1.919	67.7%	0.824	0.51 8	0.427	0.12 6	2	2

**Table 52: Generation results using the settings mutant=0.25 ghost=X averaged across Assignment 1 and Assignment 3**

Setting	Ratio	% of Max	spread*fit	avg p- val	<0.05	<0.01
25/2	1.539	0.66	0.495	0.007	10	8
50/2	1.531	0.654	0.473	0.006	10	9
75/2	1.549	0.665	0.457	0.006	10	9
150/2	1.558	0.662	0.432	0.005	10	7

25/3	1.509	0.636	0.492	0.012	10	6
50/3	1.517	0.639	0.469	0.009	10	5
75/3	1.507	0.636	0.441	0.0175	9	7
150/3	1.49	0.622	0.403	0.028	9	6
25/5	1.515	0.636	0.491	0.011	10	4
50/5	1.543	0.644	0.454	0.013	9	6
75/5	1.485	0.622	0.433	0.022	8	6
150/5	1.466	0.609	0.395	0.067	7	5

### 9.7.3.3 mutant=0.5 ghost=X

Table 53: Generation results for Assignment 1 using the settings mutant=0.5 ghost=X

Setting	Pre c	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread* Fit	PVal	<0.05	<0.01
25/2	0.51	0.423	1.688	3.126	55.1%	0.555	0.785	0.436	0.008	5	4
50/2	0.57	0.427	1.7057	3.126	55.7%	0.559	0.747	0.418	0.006	5	4
75/2	0.546	0.419	1.672	3.126	54.6%	0.56	0.705	0.395	0.007	5	4
150/2	0.553	0.425	1.698	3.126	55.3%	0.563	0.668	0.376	0.005	5	4
25/3	0.556	0.427	1.705	3.126	55.6%	0.594	0.748	0.444	0.004	5	4
50/3	0.568	0.437	1.746	3.126	56.8%	0.597	0.717	0.428	0.002	5	5
75/3	0.558	0.43	1.715	3.126	55.8%	0.594	0.686	0.407	0.003	5	5
150/3	0.564	0.437	1.744	3.126	56.4%	0.6	0.615	0.369	0.003	5	5
25/5	0.568	0.438	1.748	3.126	56.8%	0.603	0.729	0.44	0.003	5	4

50/5	0.5 65	0.4 36	1.741	3.126	56.5%	0.607	0.67 4	0.409	0.0 02	5	5
75/5	0.5 62	0.4 34	1.733	3.126	56.2%	0.611	0.64 9	0.396	0.0 02	5	5
150/ 5	0.5 43	0.4 21	1.679	3.126	54.3%	0.616	0.58 43	0.36	0.0 07	5	3

**Table 54: Generation results for Assignment 3 using the settings mutant=0.5 ghost=X**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread*	PVal	<0.05	<0.01
25/2	0.78	0.358	1.427	1.91	78.0%	0.717	0.777	0.557	0.003	5	5
50/2	0.768	0.355	1.414	1.91	76.8%	0.743	0.704	0.523	0.002	5	5
75/2	0.787	0.366	1.46	1.91	78.7%	0.757	0.669	0.506	0.002	5	5
150/2	0.766	0.357	1.421	1.91	76.6%	0.76	0.617	0.469	0.006	5	3
25/3	0.72	0.332	1.324	1.91	72.0%	0.761	0.704	0.536	0.015	5	1
50/3	0.753	0.351	1.4	1.91	75.3%	0.788	0.65	0.512	0.014	4	4
75/3	0.711	0.331	1.321	1.91	71.1%	0.797	0.583	0.465	0.047	3	2
150/3	0.682	0.32	1.276	1.91	68.2%	0.807	0.522	0.421	0.116	2	2
25/5	0.7	0.323	1.289	1.91	70.0%	0.787	0.697	0.549	0.028	4	0
50/5	0.716	0.336	1.340	1.91	71.6%	0.807	0.595	0.48	0.04	4	2
75/5	0.699	0.324	1.290	1.91	69.9%	0.818	0.549	0.449	0.065	4	2

150/5	0.674	0.314	1.253	1.91	67.4%	0.828	0.471	0.39	0.132	2	2
-------	-------	-------	-------	------	-------	-------	-------	------	-------	---	---

**Table 55: Generation results using the settings mutant=0.5 ghost=X averaged across Assignment 1 and Assignment 3**

Setting	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
25/2	1.557	0.665	0.497	0.006	10	9
50/2	1.56	0.663	0.471	0.004	10	9
75/2	1.566	0.667	0.451	0.004	10	9
150/2	1.56	0.659	0.423	0.005	10	7
25/3	1.515	0.638	0.49	0.01	10	5
50/3	1.573	0.661	0.47	0.008	9	9
75/3	1.518	0.634	0.436	0.024	8	7
150/3	1.51	0.623	0.395	0.06	7	7
25/5	1.519	0.634	0.494	0.015	9	4
50/5	1.541	0.641	0.444	0.021	9	7
75/5	1.512	0.63	0.423	0.033	9	7
150/5	1.466	0.608	0.375	0.07	7	5

#### 9.7.3.4 mutant=0.5 ghost=0.1 dist=10

**Table 56: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.1 dist=10**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread* Fit	PVal	<0.05	<0.01
25/2	0.552	0.424	1.691	3.126	55.2%	0.578	0.774	0.447	0.007	5	4
50/2	0.548	0.428	1.678	3.126	54.8%	0.582	0.721	0.42	0.006	5	4
75/2	0.542	0.416	1.661	3.126	54.2%	0.583	0.697	0.406	0.006	5	3

150/ 2	0.56 3	0.43 4	1.73 2	3.126		56.3%	0.587	0.64 9	0.381	0.00 3	5	5
25/3	0.55 1	0.42 4	1.69 3	3.126		55.1%	0.613	0.72 5	0.444	0.00 3	5	4
50/3	0.56 7	0.43 8	1.74 9	3.126		56.7%	0.617	0.69 7	0.43	0.00 1	5	5
75/3	0.54 9	0.42 5	1.69 5	3.126		54.9%	0.611	0.63 4	0.388	0.00 2	5	5
150/ 3	0.56 4	0.43 7	1.74 5	3.126		56.4%	0.622	0.58 5	0.364	0.00 1	5	5
25/5	0.56 9	0.43 9	1.75 3	3.126		56.9%	0.62	0.69 8	0.433	0.00 2	5	4
50/5	0.56 5	0.43 4	1.73 1	3.126		56.0%	0.622	0.65 6	0.408	0.00 1	5	5
75/5	0.55 3	0.42 8	1.70 8	3.126		55.3%	0.628	0.64 1	0.403	0.00 2	5	5
150/ 5	0.54 3	0.42	1.67 7	3.126		54.3%	0.629	0.58 3	0.367	0.00 5	5	3

**Table 57: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.1 dist=10**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread* Fit	PVal	<0.05	<0.01
25/2	0.78 2	0.36	1.43 5	1.919	78.2%	0.721	0.75 6	0.546	0.00 2	5	5
50/2	0.75 7	0.34 8	1.38 8	1.919	75.7%	0.75	0.7	0.525	0.00 5	5	4
75/2	0.79	0.36 8	1.46 6	1.919	79.0%	0.767	0.66 2	0.508	0.00 3	5	4
150/ 2	0.75 8	0.35 3	1.40 7	1.919	75.8%	0.769	0.59 2	0.455	0.00 9	5	3
25/3	0.72 1	0.33 3	1.32 6	1.919	72.1%	0.768	0.69 1	0.531	0.01 6	5	1



50/3	0.75 4	0.35 1	1.39 8	1.919	75.4%	0.795	0.61 6	0.49	0.01 1	5	3
75/3	0.72 1	0.33 6	1.33 8	1.919	72.1%	0.807	0.56 7	0.457	0.02 8	4	3
150/ 3	0.73 2	0.34 1	1.35 9	1.919	73.2%	0.818	0.51 9	0.424	0.02 1	4	3
25/5	0.68 4	0.31 7	1.26 3	1.919	68.4%	0.795	0.67	0.533	0.06 3	2	1
50/5	0.72 5	0.34	1.35 6	1.919	72.5%	0.817	0.57 2	0.468	0.05 6	3	3
75/5	0.69 4	0.32 2	1.28 2	1.919	69.4%	0.829	0.51 5	0.428	0.09 1	3	2
150/ 5	0.67 5	0.31 5	1.25 6	1.919	67.5%	0.835	0.42 9	0.358	0.12 7	2	2

**Table 58: Generation results using the settings mutant=0.5 ghost=0.1 dist=10 averaged across Assignment 1 and Assignment 3**

	Ratio	% of Max	spread*fit	avg p- val	<0.05	<0.01
25/2	1.563	66.7%	0.496	0.004	10	9
50/2	1.533	65.2%	0.472	0.005	10	8
75/2	1.563	66.6%	0.457	0.005	10	7
150/2	1.57	66.1%	0.418	0.006	10	8
25/3	1.51	63.6%	0.488	0.01	10	5
50/3	1.573	66.1%	0.46	0.006	10	8
75/3	1.516	63.5%	0.423	0.015	9	8
150/3	1.552	64.8%	0.394	0.011	9	8
25/5	1.508	62.7%	0.483	0.033	7	5
50/5	1.544	64.2%	0.438	0.029	8	8
75/5	1.495	62.4%	0.415	0.047	8	7
150/5	1.467	60.9%	0.363	0.065	7	5

### 9.7.3.5 mutant=0.5 ghost=0.25 dist=10

**Table 59: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.25 dist=10**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread*	PVal	<0.05	<0.01
25/2	0.554	0.425	1.696	3.126	55.4%	0.583	0.777	0.453	0.007	5	4
50/2	0.548	0.42	1.678	3.126	54.8%	0.588	0.721	0.424	0.005	5	4
75/2	0.549	0.421	1.679	3.126	54.9%	0.589	0.683	0.403	0.005	5	4
150/2	0.569	0.437	1.746	3.126	56.9%	0.593	0.651	0.386	0.002	5	5
25/3	0.559	0.43	1.716	3.126	55.9%	0.619	0.723	0.448	0.003	5	4
50/3	0.568697	0.44	1.755	3.126	56.9%	0.623	0.677	0.422	0.001	5	5
75/3	0.555	0.429	1.711	3.126	55.5%	0.618	0.653	0.404	0.002	5	5
150/3	0.562	0.436	1.741	3.126	56.2%	0.629	0.606	0.382	0.001	5	5
25/5	0.570	0.44	1.755	3.126	57.0%	0.628	0.703	0.442	0.002	5	4
50/5	0.57	0.441	1.759	3.126	57.0%	0.63	0.657	0.414	0.001	5	5
75/5	0.564	0.435	1.737	3.126	56.4%	0.635	0.627	0.398	0.002	5	5
150/5	0.538	0.416	1.662	3.126	53.8%	0.634	0.563	0.357	0.005	5	3

**Table 60: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.25 dist=10**

Setting		Acc		Max	% Max		Fit		PVal		
---------	--	-----	--	-----	-------	--	-----	--	------	--	--

ng	Prec		Rati o	Ratio	Ratio	Sprea d		Spread* Fit		<0.0 5	<0.0 1
25/2	0.77 5	0.35 6	1.42 1	1.919	77.5%	0.726	0.73 3	0.532	0.00 3	5	5
50/2	0.75 1	0.34 5	1.37 7	1.919	75.1%	0.755	0.68 5	0.517	0.00 6	5	3
75/2	0.78 1	0.36 2	1.44 2	1.919	78.1%	0.772	0.65 4	0.505	0.00 3	5	4
150/ 2	0.75 3	0.35	1.39 6	1.919	75.3%	0.774	0.58 3	0.451	0.00 8	5	3
25/3	0.71 2	0.32 9	1.31 1	1.919	71.2%	0.774	0.67 9	0.525	0.02 23	5	1
50/3	0.74	0.34 3	1.36 9	1.919	74.0%	0.801	0.61 3	0.491	0.01 01	5	3
75/3	0.69 6	0.32 4	1.29 2	1.919	69.6%	0.815	0.55 3	0.451	0.04 26	3	1
150/ 3	0.70 3	0.32 5	1.29 6	1.919	70.3%	0.824	0.5 0.5	0.412	0.03 66	4	1
25/5	0.68 5	0.31 8	1.26 8	1.919	68.5%	0.799	0.65 3	0.522	0.06 48	2	1
50/5	0.69 2	0.32 4	1.29 2	1.919	69.2%	0.824	0.55 8	0.46	0.08 65	2	2
75/5	0.67 1	0.30 9	1.23 3	1.919	67.1%	0.834	0.53 3	0.445	0.11 0.11	2	1
150/ 5	0.65	0.30 1	1.20 2	1.919	65.0%	0.846	0.44 7	0.379	0.18 5	1	2

**Table 61: Generation results using the settings mutant=0.5 ghost=0.25 dist=10 averaged across Assignment 1 and Assignment 3**

	Ratio	% of Max	spread*fit	avg p- val	<0.05	<0.01
25/2	1.558	66.4%	0.492	0.005	10	9
50/2	1.528	65.0%	0.471	0.006	10	7

75/2	1.561	66.5%	0.454	0.004	10	8
150/2	1.571	66.1%	0.419	0.005	10	8
25/3	1.514	63.5%	0.486	0.013	10	5
50/3	1.562	65.4%	0.457	0.006	10	8
75/3	1.501	62.5%	0.427	0.022	8	6
150/3	1.519	63.3%	0.397	0.019	9	6
25/5	1.512	62.8%	0.482	0.034	7	5
50/5	1.525	63.1%	0.437	0.044	7	7
75/5	1.485	61.7%	0.421	0.056	7	6
150/5	1.432	59.4%	0.368	0.095	6	4

### 9.7.3.6 mutant=0.5 ghost=0.5 dist=10

Table 62: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.5 dist=10

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread* Fit	PVal	<0.05	<0.01
25/2	0.547	0.42	1.676	3.126	54.7%	0.586	0.756	0.443	0.007	5	4
50/2	0.547	0.419	1.673	3.126	54.7%	0.591	0.721	0.426	0.006	5	4
75/2	0.549	0.42	1.676	3.126	54.9%	0.591	0.669	0.395	0.006	5	4
150/2	0.563	0.433	1.728	3.126	56.3%	0.597	0.655	0.391	0.002	5	5
25/3	0.56	0.431	1.72	3.126	56.0%	0.623	0.712	0.443	0.003	5	4
50/3	0.569	0.44	1.755	3.126	56.9%	0.63	0.677	0.426	0.001	5	5
75/3	0.561	0.433	1.729	3.126	56.1%	0.627	0.636	0.398	0.001	5	5
150/3	0.565	0.437	1.746	3.126	56.5%	0.635	0.594	0.377	0.002	5	5

25/5	0.58 1	0.44 6	1.78 1	3.126	58.1%	0.633	0.69 4	0.44	0.00 3	5	4
50/5	0.55 5	0.42 9	1.71 3	3.126	55.5%	0.64	0.65 4	0.418	0.00 2	5	5
75/5	0.56 5	0.43 5	1.73 5	3.126	56.5%	0.641	0.60 2	0.385	0.00 2	5	5
150/ 5	0.54 7	0.42 3	1.68 9	3.126	54.7%	0.625	0.53	0.331	0.00 3	5	5

**Table 63: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.5 dist=10**

Setti ng	Prec	Acc	Rati o	Max Ratio	% Max Ratio	Sprea d	Fit	Spread*	PVal	<0.0 5	<0.0 1
25/2	0.76 4	0.35 2	1.40 5	1.918	0.764	0.737	0.74 5	0.549	0.00 5	5	4
50/2	0.76	0.35 2	1.40 2	1.918	0.76	0.765	0.68 5	0.524	0.00 4	5	4
75/2	0.78	0.36 2	1.44 3	1.918	0.78	0.782	0.65 2	0.51	0.00 4	5	4
150/ 2	0.75 4	0.35 1	1.40 1	1.918	0.754	0.783	0.58 9	0.461	0.00 8	5	4
25/3	0.72 4	0.33 8	1.34 7	1.918	0.724	0.783	0.66 7	0.522	0.01 9	5	2
50/3	0.73	0.33 9	1.35 1	1.918	0.73	0.811	0.60 8	0.493	0.01 56	5	3
75/3	0.69 3	0.32 3	1.28 8	1.918	0.693	0.824	0.56 6	0.466	0.04 6	3	1
150/ 3	0.68 6	0.31 7	1.26 4	1.918	0.686	0.835	0.47 7	0.4	0.08 2	4	1
25/5	0.7	0.33	1.31 5	1.918	0.7	0.805	0.63 8	0.513	0.07 2	2	2
50/5	0.69 7	0.32 7	1.30 5	1.918	0.697	0.83	0.57	0.473	0.08 2	2	2

75/5	0.69	0.32 1	1.28	1.918	0.69	0.839	0.51 4	0.431	0.08 5	3	2
150/ 5	0.66 7	0.31 3	1.24 6	1.918	0.667	0.851	0.41 7	0.355	0.16 3	2	2

**Table 64: Generation results using the settings mutant=0.5 ghost=0.25 dist=10 averaged across Assignment 1 and Assignment 3**

	Ratio	% of Max	spread*fit	avg p- val	<0.05	<0.01
25/2	1.54	0.656	0.496	0.006	10	8
50/2	1.537	0.653	0.475	0.005	10	8
75/2	1.559	0.664	0.452	0.005	10	8
150/2	1.565	0.658	0.426	0.005	10	9
25/3	1.534	0.642	0.483	0.011	10	6
50/3	1.553	0.649	0.46	0.008	10	8
75/3	1.508	0.627	0.432	0.023	8	6
150/3	1.505	0.625	0.388	0.041	9	6
25/5	1.548	0.640	0.477	0.038	7	6
50/5	1.509	0.626	0.446	0.042	7	7
75/5	1.508	0.628	0.408	0.044	8	7
150/5	1.468	0.607	0.343	0.083	7	7

### 9.7.3.7 mutant=0.5 ghost=0.1 dist=5

**Table 65: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.1 dist=5**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread* Fit	PVal	<0.0 5	<0.0 1
25/2	0.54 9	0.42 2	1.68 3	3.126	54.9%	0.578	0.78 1	0.451	0.00 6	5	4
50/2	0.54 8	0.42 1	1.68	3.126	54.8%	0.583	0.72 6	0.423	0.00 5	5	4

75/2	0.53 9	0.41 4	1.65 5	3.126	53.9%	0.584	0.70 4	0.411	0.00 5	5	4
150/ 2	0.55 9	0.43 1	1.71 9	3.126	55.7%	0.587	0.65 5	0.384	0.00 3	5	5
25/3	0.54 9	0.42 2	1.68 6	3.126	54.9%	0.613	0.75 0	0.46	0.00 4	5	4
50/3	0.56 3	0.43 5	1.73 6	3.126	56.3%	0.619	0.70 2	0.434	0.00 2	5	5
75/3	0.54 3	0.42	1.67 5	3.126	54.3%	0.612	0.67 1	0.411	0.00 3	5	5
150/ 3	0.56 2	0.43 6	1.74	3.126	56.2%	0.624	0.60 1	0.375	0.00 1	5	5
25/5	0.57 3	0.44 1	1.76 3	3.126	57.3%	0.621	0.70 9	0.44	0.00 2	5	4
50/5	0.56 3	0.43 6	1.73 9	3.126	56.3%	0.624	0.67 7	0.422	0.00 1	5	5
75/5	0.55 2	0.42 6	1.70 3	3.126	55.2%	0.629	0.63 3	0.398	0.00 2	5	5
150/ 5	0.54 6	0.42 2	1.68 5	3.126	54.6%	0.628	0.59 2	0.372	0.00 5	5	4

**Table 66: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.1 dist=5**

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread* Fit	PVal	<0.05	<0.01
25/2	0.77 6	0.35 6	1.41 9	1.919	77.6%	0.719	0.75 6	0.544	0.00 4	5	4
50/2	0.76 6	0.35 2	1.40 3	1.919	76.6%	0.746	0.71 1	0.531	0.00 4	5	4
75/2	0.78 7	0.36 6	1.46 1	1.919	78.7%	0.763	0.65 4	0.499	0.00 4	5	4
150/ 2	0.76 2	0.35 5	1.41 4	1.919	76.2%	0.765	0.59 8	0.457	0.00 9	5	3

25/3	0.70 5	0.32 6	1.30 1	1.919	70.5%	0.766	0.70 1	0.537	0.02 6	4	0
50/3	0.73 9	0.34 4	1.37 2	1.919	73.9%	0.793	0.63 2	0.501	0.01 4	5	2
75/3	0.70 1	0.32 7	1.30 4	1.919	70.1%	0.804	0.59	0.474	0.04 3	3	2
150/ 3	0.69 7	0.32 6	1.30 1	1.919	69.7%	0.812	0.51	0.414	0.08	3	2
25/5	0.67 8	0.31 4	1.25 2	1.919	67.8%	0.792	0.66	0.523	0.05 8	2	0
50/5	0.72 6	0.34	1.35 7	1.919	72.6%	0.814	0.60 5	0.493	0.05 7	3	3
75/5	0.68	0.31 6	1.25 9	1.919	68.0%	0.826	0.53 1	0.439	0.10 1	3	1
150/ 5	0.66 8	0.31 2	1.24 4	1.919	66.8%	0.838	0.45 5	0.381	0.14 8	2	2

**Table 67: Generation results using the settings mutant=0.5 ghost=0.1 dist=5 averaged across Assignment 1 and Assignment 3**

	Ratio	% of Max	spread*fit	avg p- val	<0.05	<0.01
25/2	1.551	0.663	0.497	0.005	10	8
50/2	1.542	0.657	0.477	0.005	10	8
75/2	1.558	0.663	0.455	0.004	10	8
150/2	1.567	0.66	0.421	0.006	10	8
25/3	1.493	0.627	0.498	0.015	9	4
50/3	1.554	0.651	0.467	0.008	10	7
75/3	1.49	0.622	0.442	0.023	8	7
150/3	1.52	0.629	0.395	0.041	8	7
25/5	1.507	0.626	0.482	0.030	7	4
50/5	1.548	0.644	0.458	0.029	8	8
75/5	1.481	0.616	0.419	0.051	8	6
150/5	1.465	0.607	0.376	0.077	7	6



### 9.7.3.8 mutant=0.5 ghost=0.25 dist=5

Table 68: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.25 dist=5

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread*	PVal	<0.05	<0.01
25/2	0.551	0.423	1.689	3.126	55.1%	0.58	0.768	0.445	0.007	5	4
50/2	0.542	0.417	1.665	3.126	54.2%	0.586	0.741	0.434	0.005	5	4
75/2	0.539	0.414	1.654	3.126	53.9%	0.586	0.695	0.408	0.005	5	4
150/2	0.559	0.431	1.72	3.126	55.9%	0.59	0.654	0.386	0.003	5	5
25/3	0.549	0.422	1.687	3.126	54.9%	0.615	0.73	0.449	0.004	5	4
50/3	0.565	0.437	1.743	3.126	56.5%	0.62	0.689	0.427	0.002	5	5
75/3	0.547	0.422	1.686	3.126	54.7%	0.613	0.672	0.412	0.003	5	5
150/3	0.565	0.438	1.749	3.126	56.5%	0.625	0.618	0.387	0.001	5	5
25/5	0.573	0.441	1.762	3.126	57.3%	0.624	0.708	0.442	0.002	5	4
50/5	0.564	0.437	1.744	3.126	56.4%	0.626	0.683	0.428	0.001	5	5
75/5	0.556	0.429	1.714	3.126	55.6%	0.631	0.64	0.404	0.002	5	5
150/5	0.548	0.423	1.691	3.126	54.8%	0.631	0.591	0.373	0.004	5	4

Table 69: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.25 dist=5

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread*	PVal	<0.05	<0.01
25/2	0.772	0.354	1.41	1.919	0.771	0.722	0.764	0.552	0.005	5	4
50/2	0.763	0.351	1.399	1.919	0.763	0.75	0.715	0.536	0.004	5	4
75/2	0.782	0.362	1.443	1.919	0.782	0.768	0.661	0.508	0.004	5	4
150/2	0.743	0.347	1.38	1.919	0.743	0.771	0.599	0.461	0.011	5	2
25/3	0.703	0.325	1.298	1.919	0.703	0.77	0.695	0.535	0.028	4	1
50/3	0.741	0.343	1.369402	1.919	0.741	0.797	0.632	0.504	0.013	5	3
75/3	0.699	0.326	1.29811	1.919	0.699	0.809	0.58	0.47	0.048	3	2
150/3	0.68	0.316	1.260143	1.919	0.68	0.816	0.527	0.43	0.105	3	1
25/5	0.679	0.315	1.257262	1.919	0.679	0.795	0.685	0.544	0.061	2	0
50/5	0.718	0.336	1.338296	1.919	0.718	0.818	0.593	0.485	0.063	3	3
75/5	0.673	0.313	1.237232	1.919	0.673	0.830	0.541	0.449	0.104	2	1
150/5	0.656	0.304	1.211565	1.919	0.656	0.84	0.491	0.413	0.166	1	1

**Table 70: Generation results using the settings mutant=0.5 ghost=0.25 dist=5 averaged across Assignment 1 and Assignment 3**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
25/2	1.549	0.661	0.499	0.006	10	8

50/2	1.532	0.653	0.485	0.005	10	8
75/2	1.548	0.66	0.458	0.004	10	8
150/2	1.55	0.651	0.424	0.007	10	7
25/3	1.492	0.626	0.492	0.016	9	5
50/3	1.556	0.653	0.466	0.007	10	8
75/3	1.492	0.623	0.441	0.025	8	7
150/3	1.505	0.623	0.408	0.053	8	6
25/5	1.51	0.626	0.493	0.0314	7	4
50/5	1.541	0.641	0.457	0.032	8	8
75/5	1.475	0.615	0.427	0.053	7	6
150/5	1.451	0.602	0.393	0.085	6	5

### 9.7.3.9 mutant=0.5 ghost=0.5 dist=5

Table 71: Generation results for Assignment 1 using the settings mutant=0.5 ghost=0.5 dist=5

Setting	Prec	Acc	Ratio	Max Ratio	% Max Ratio	Spread	Fit	Spread*	PVal	<0.05	<0.01
25/2	0.549	0.421	1.681	3.126	0.549	0.584	0.781	0.456	0.007	5	4
50/2	0.549	0.421	1.681	3.126	0.549	0.589	0.734	0.433	0.005	5	4
75/2	0.554	0.424	1.692	3.126	0.554	0.59	0.688	0.406	0.005	5	4
150/2	0.568	0.437	1.743	3.126	0.568	0.594	0.645	0.383	0.003	5	5
25/3	0.557	0.427	1.705	3.126	0.557	0.62	0.732	0.454	0.004	5	4
50/3	0.569	0.439	1.751	3.126	0.569	0.627	0.693	0.435	0.002	5	5
75/3	0.551	0.425	1.698	3.126	0.551	0.624	0.67	0.418	0.002	5	5
150/3	0.568	0.44	1.754	3.126	0.568	0.631	0.603	0.38	0.002	5	5

25/5	0.57 7	0.44 4	1.77 4	3.126	0.577	0.630	0.71 5	0.451	0.00 2	5	4
50/5	0.55 8	0.43 2	1.72 4	3.126	0.558	0.633	0.65 7	0.416	0.00 2	5	5
75/5	0.56 5	0.43 5	1.73 7	3.126	0.565	0.637	0.63	0.402	0.00 2	5	5
150/ 5	0.55 8	0.43 1	1.72 2	3.126	0.558	0.621	0.59 3	0.368	0.00 2	5	5

**Table 72: Generation results for Assignment 3 using the settings mutant=0.5 ghost=0.5 dist=5**

Setti ng	Prec	Acc	Rati o	Max Ratio	% Max Ratio	Sprea d	Fit	Spread* Fit	PVal	<0.0 5	<0.0 1
25/2	0.75 5	0.34 7	1.38 2	1.919	0.755	0.729	0.75 8	0.553	0.00 7	5	3
50/2	0.75 5	0.34 7	1.38 5	1.919	0.755	0.757	0.71 1	0.538	0.00 4	5	4
75/2	0.78 4	0.36 4	1.45 1	1.919	0.784	0.772	0.66 5	0.513	0.00 4	5	4
150/ 2	0.74 4	0.34 6	1.38 2	1.919	0.744	0.776	0.59 7	0.463	0.01 1	5	2
25/3	0.71 6	0.33 4	1.33	1.919	0.716	0.772	0.68	0.525	0.02 4	5	2
50/3	0.73 9	0.34 3	1.36 8	1.919	0.739	0.802	0.63 6	0.51	0.02 4	4	3
75/3	0.69 6	0.32 4	1.29 3	1.919	0.696	0.814	0.59 3	0.483	0.05 3	2	2
150/ 3	0.68 3	0.31 7	1.26 4	1.919	0.683	0.821	0.5	0.411	0.09 5	2	1
25/5	0.68 5	0.31 9	1.27 1	1.919	0.685	0.798	0.64 9	0.518	0.07 3	2	2
50/5	0.72 6	0.33 9	1.35 2	1.919	0.726	0.823	0.57 8	0.476	0.05 4	3	3

75/5	0.67 1	0.30 9	1.23 3	1.919	0.671	0.836	0.53 4	0.446	0.10 2	1	1
150/5	0.66 4	0.31	1.23 7	1.919	0.664	0.846	0.44 7	0.378	0.16 4	2	2

**Table 73: Generation results using the settings mutant=0.5 ghost=0.5 dist=5 averaged across Assignment 1 and Assignment 3**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
25/2	1.532	0.652	0.504	0.007	10	7
50/2	1.533	0.652	0.485	0.005	10	8
75/2	1.57	0.669	0.46	0.004	10	8
150/2	1.562	0.656	0.423	0.006	10	7
25/3	1.517	0.637	0.49	0.014	10	6
50/3	1.56	0.654	0.472	0.013	9	8
75/3	1.496	0.624	0.45	0.028	7	7
150/3	1.509	0.626	0.396	0.048	7	6
25/5	1.522	0.631	0.485	0.037	7	6
50/5	1.538	0.642	0.446	0.028	8	8
75/5	1.485	0.618	0.424	0.052	6	6
150/5	1.479	0.611	0.373	0.083	7	7

#### 9.7.4 Example of Image/PDF Change browser output

Change 119:

```

407      glRotatef(guy.xrot, 1, 0, 0);
408      glRotatef(guy.yrot, 0, 1, 0);
409      glRotatef(guy.zrot, 0, 0, 1);
410
411      glPushMatrix(); {
413          glRotatef(guy.partsRot[0][0], 1, 0, 0);
414          glRotatef(guy.partsRot[0][1], 0, 1, 0);
415          glRotatef(guy.partsRot[0][2], 0, 0, 1);
412      glTranslatef(0.0, 8.0, 0.0);
416      guy.pointer->getObjectsByName()["Head"]->draw();
417      } glPopMatrix();
418
419      glPushMatrix(); {
420          glTranslatef(0.0, 6.0, 0.0);
421      guy.pointer->getObjectsByName()["Torso"]->draw();

```

Change 120:

```

407         glRotatef(guy.xrot, 1, 0, 0);
408         glRotatef(guy.yrot, 0, 1, 0);
409         glRotatef(guy.zrot, 0, 0, 1);
410
411         glPushMatrix(); {
415             glTranslatef(0.0, 8.0, 0.0);
-1 : .   →   glPushMatrix(); {
-1 : .       glTranslatef(0.0, 1.0, 0.0);
412             glRotatef(guy.partsRot[0][0], 1, 0, 0);
413             glRotatef(guy.partsRot[0][1], 0, 1, 0);
414             glRotatef(guy.partsRot[0][2], 0, 0, 1);
416             guy.pointer->getObjectsByName()["Head"]->draw();
417         } glPopMatrix();
-1 : .   } glPopMatrix();
418
419         glPushMatrix(); {
420             glTranslatef(0.0, 6.0, 0.0);
421             guy.pointer->getObjectsByName()["Torso"]->draw();
422         } glPopMatrix();

```

Change 121:

```

409         glRotatef(guy.zrot, 0, 0, 1);
410
411         glPushMatrix(); {
412             glTranslatef(0.0, 8.0, 0.0);
413             glPushMatrix(); {
415                 glRotatef(guy.partsRot[0][0], 1, 0, 0);
416                 glRotatef(guy.partsRot[0][1], 0, 1, 0);
417                 glRotatef(guy.partsRot[0][2], 0, 0, 1);
414                 glTranslatef(0.0, 1.0, 0.0); -> glTranslatef(0.0, -1, 0.0);
418                 guy.pointer->getObjectsByName()["Head"]->draw();
419             } glPopMatrix();
420         } glPopMatrix();
421
422         glPushMatrix(); {
423             glTranslatef(0.0, 6.0, 0.0);

```

Change 122:

```

407         glRotatef(guy.xrot, 1, 0, 0);
408         glRotatef(guy.yrot, 0, 1, 0);
409         glRotatef(guy.zrot, 0, 0, 1);
410
411         glPushMatrix(); {
414             glRotatef(guy.partsRot[0][0], 1, 0, 0);
415             glRotatef(guy.partsRot[0][1], 0, 1, 0);
416             glRotatef(guy.partsRot[0][2], 0, 0, 1);
417             glTranslatef(0.0, -1, 0.0);
413             glPushMatrix(); {
412                 glTranslatef(0.0, 8.0, 0.0);
418                 guy.pointer->getObjectsByName()["Head"]->draw();
419             } glPopMatrix();
420         } glPopMatrix();
421
422         glPushMatrix(); {
423             glTranslatef(0.0, 6.0, 0.0);
424             guy.pointer->getObjectsByName()["Torso"]->draw();

```

Change 123:

```

407         glRotatef(guy.xrot, 1, 0, 0);
408         glRotatef(guy.yrot, 0, 1, 0);
409         glRotatef(guy.zrot, 0, 0, 1);
410
411         glPushMatrix(); {
416             glTranslatef(0.0, 8.0, 0.0);
415             glPushMatrix(); {
-1 : .             glTranslatef(0.0, -1.0, 0);
412                 glRotatef(guy.partsRot[0][0], 1, 0, 0);
413                 glRotatef(guy.partsRot[0][1], 0, 1, 0);
414                 glRotatef(guy.partsRot[0][2], 0, 0, 1);
-1 : .             glTranslatef(0.0, 1.0, 0);
417                 guy.pointer->getObjectsByName()["Head"]->draw();
418             } glPopMatrix();
419         } glPopMatrix();
420
421         glPushMatrix(); {
422             glTranslatef(0.0, 6.0, 0.0);

```

Change 126:

```

420         } glPopMatrix();
421     } glPopMatrix();
422
423     glPushMatrix(); {
424         glTranslatef(0.0, 6.0, 0.0);
-1 : .         glPushMatrix(); {
-1 : .             glTranslatef(0.0, -3.0, 0);
-1 : .             glRotatef(guy.partsRot[1][0], 1, 0, 0);
-1 : .             glRotatef(guy.partsRot[1][1], 0, 1, 0);
-1 : .             glRotatef(guy.partsRot[1][2], 0, 0, 1);
-1 : .             glTranslatef(0.0, 3.0, 0);
425             guy.pointer->getObjectsByName()["Torso"]->draw();
426         } glPopMatrix();
-1 : .     } glPopMatrix();
427
428     glPushMatrix(); {
429         glTranslatef(0.0, 7.0, -1.5);
430         guy.pointer->getObjectsByName()["RightUpperArm"]->draw();
431     } glPopMatrix();

```

Change 130:

```

407         glRotatef(guy.xrot, 1, 0, 0);
408         glRotatef(guy.yrot, 0, 1, 0);
409         glRotatef(guy.zrot, 0, 0, 1);
410
411         glPushMatrix(); {
423             glTranslatef(0.0, 6.0, 0.0);
424
425             glTranslatef(0.0, -3.0, 0);
426             glRotatef(guy.partsRot[1][0], 1, 0, 0);
427             glRotatef(guy.partsRot[1][1], 0, 1, 0);
428             glRotatef(guy.partsRot[1][2], 0, 0, 1);
429             glTranslatef(0.0, 3.0, 0);
430             guy.pointer->getObjectsByName()["Torso"]->draw();
41: .
422         glPushMatrix(); {
412             glTranslatef(0.0, 8.0, 0.0); ->             glTranslatef(0.0, 2.0, 0.0);
413
414             glTranslatef(0.0, -1.0, 0);
415             glRotatef(guy.partsRot[0][0], 1, 0, 0);
416             glRotatef(guy.partsRot[0][1], 0, 1, 0);
417             glRotatef(guy.partsRot[0][2], 0, 0, 1);
418             glTranslatef(0.0, 1.0, 0);
419             guy.pointer->getObjectsByName()["Head"]->draw();
420         } glPopMatrix();
421
431     } glPopMatrix();
432
433     glPushMatrix(); {
434         glTranslatef(0.0, 7.0, -1.5);
435         guy.pointer->getObjectsByName()["RightUpperArm"]->draw();
436     } glPopMatrix();

```

Change 131:



```

410
411         glPushMatrix(); {
412             glTranslatef(0.0, 6.0, 0.0);
413
414             glTranslatef(0.0, -3.0, 0);
425         ┌─> glRotatef(guy.partsRot[0][0], 1,0,0);
426         │   glRotatef(guy.partsRot[0][1], 0,1,0);
427         │   glRotatef(guy.partsRot[0][2], 0,0,1);
418         │   glTranslatef(0.0, 3.0, 0);
419         │   guy.pointer->getObjectsByName()["Torso"]->draw();
420
421         glPushMatrix(); {
422             glTranslatef(0.0, 2.0, 0.0);
423
424             glTranslatef(0.0, -1.0, 0);
415         ┌─> glRotatef(guy.partsRot[1][0], 1,0,0);
416         │   glRotatef(guy.partsRot[1][1], 0,1,0);
417         │   glRotatef(guy.partsRot[1][2], 0,0,1);
428         │   glTranslatef(0.0, 1.0, 0);
429         │   guy.pointer->getObjectsByName()["Head"]->draw();
430         } glPopMatrix();
431
434         glPushMatrix(); {
435         ┌─> glTranslatef(0.0, 7.0, -1.5); -> glTranslatef(0.0, 1.0, -1.5);
-1 : │
-1 : │   glTranslatef(-1.5, 0.0, 0);
-1 : │   glRotatef(guy.partsRot[2][0], 1,0,0);
-1 : │   glRotatef(guy.partsRot[2][1], 0,1,0);
-1 : │   glRotatef(guy.partsRot[2][2], 0,0,1);
-1 : │   glTranslatef(1.5, 0.0, 0);
436         │   guy.pointer->getObjectsByName()["RightUpperArm"]->draw();
437         } glPopMatrix();
438
432         } glPopMatrix();
433
-1 : │
-1 : │
439         glPushMatrix(); {
440             glTranslatef(0.0, 7.0, -2.8);
441             guy.pointer->getObjectsByName()["RightLowerArm"]->draw();
442         } glPopMatrix();
443
444         glPushMatrix(); {

```

Change 132:

```

428         glTranslatef(0.0, 1.0, 0);
429         guy.pointer->getObjectsByName()["Head"]->draw();
430     } glPopMatrix();
431
432     glPushMatrix(); {
433         glTranslatef(0.0, 1.0, -1.5); ->         glTranslatef(-1.5, 1.0, 0.0);
434
435         glTranslatef(1.5, 0.0, 0);
436         glRotatef(guy.partsRot[2][0], 1, 0, 0);
437         glRotatef(guy.partsRot[2][1], 0, 1, 0);
438         glRotatef(guy.partsRot[2][2], 0, 0, 1);
439         glTranslatef(-1.5, 0.0, 0);
440         guy.pointer->getObjectsByName()["RightUpperArm"]->draw();
441     } glPopMatrix();
442
443 } glPopMatrix();
444
445
446
447 -1 : : /*
448     glPushMatrix(); {
449         glTranslatef(0.0, 7.0, -2.8);
450         guy.pointer->getObjectsByName()["RightLowerArm"]->draw();
451     } glPopMatrix();
452
453     glPushMatrix(); {
454         glPushMatrix(); {
455             glTranslatef(0.2, 2.0, 0.7);
456             guy.pointer->getObjectsByName()["LeftFoot"]->draw();
457         } glPopMatrix();
458     } glPopMatrix();
459 -1 : : */
460 } glPopMatrix();
461 }

```

Change 133:

```

428         glTranslatef(0.0, 1.0, 0);
429         guy.pointer->getObjectsByName()["Head"]->draw();
430     } glPopMatrix();
431
432     glPushMatrix(); {
433         glTranslatef(-1.5, 1.0, 0.0); ->         glTranslatef(0.0, 1.0, -1.5);
434
435         glTranslatef(1.5, 0.0, 0); ->         glTranslatef(0.0, 0.0, 1.5);
436         glRotatef(guy.partsRot[2][0], 1, 0, 0);
437         glRotatef(guy.partsRot[2][1], 0, 1, 0);
438         glRotatef(guy.partsRot[2][2], 0, 0, 1);
439         glTranslatef(-1.5, 0.0, 0); ->         glTranslatef(0.0, 0.0, -1.5);
440         guy.pointer->getObjectsByName()["RightUpperArm"]->draw();
441     } glPopMatrix();
442
443 } glPopMatrix();

```

Change 134:

```

430         } glPopMatrix();
431
432         glPushMatrix(); {
433             glTranslatef(0.0, 1.0, -1.5);
434
435             glTranslatef(0.0, 0.0, 1.5); -> glTranslatef(0.0, 0.0, 1.4);
436             glRotatef(guy.partsRot[2][0], 1,0,0);
437             glRotatef(guy.partsRot[2][1], 0,1,0);
438             glRotatef(guy.partsRot[2][2], 0,0,1);
439             glTranslatef(0.0, 0.0, -1.5); -> glTranslatef(0.0, 0.0, -1.4);
440             guy.pointer->getObjectsByName()["RightUpperArm"]->draw();
441
442             glPushMatrix(); {
443                 glTranslatef(0.0, 7.0, -2.8); -> glTranslatef(0.0, 0.0, -1.3);
444
445                 glTranslatef(0.0, 0.0, 1.3);
446                 glRotatef(guy.partsRot[3][0], 1,0,0);
447                 glRotatef(guy.partsRot[3][1], 0,1,0);
448                 glRotatef(guy.partsRot[3][2], 0,0,1);
449                 glTranslatef(0.0, 0.0, -1.3);
450                 guy.pointer->getObjectsByName()["RightLowerArm"]->draw();
451             } glPopMatrix();
452         } glPopMatrix();
453
454         glPushMatrix(); {
455             glTranslatef(0.0, 7.0, -3.8);
456             guy.pointer->getObjectsByName()["RightPalm"]->draw();
457         } glPopMatrix();
458
459         glPushMatrix(); {
460             glTranslatef(0.0, 4.1, -0.7);
461             guy.pointer->getObjectsByName()["RightUpperLeg"]->draw();

```

Change 135:

```

440      guy.pointer->getObjectsByName()["RightUpperArm"]->draw();
441
442      glPushMatrix(); {
443          glTranslatef(0.0, 0.0, -1.3);
444
445          glTranslatef(0.0, 0.0, 1.3); ->          glTranslatef(0.0, 0.0, 1.1);
446          glRotatef(guy.partsRot[3][0], 1, 0, 0);
447          glRotatef(guy.partsRot[3][1], 0, 1, 0);
448          glRotatef(guy.partsRot[3][2], 0, 0, 1);
449          glTranslatef(0.0, 0.0, -1.3); ->          glTranslatef(0.0, 0.0, -1.1);
450      guy.pointer->getObjectsByName()["RightLowerArm"]->draw();
451
452      glPushMatrix(); {
453      -1 : .
454      -1 : .
455      -1 : .
456      -1 : .
457      -1 : .
458      glTranslatef(0.0, 7.0, -3.8); ->          glTranslatef(0.0, 0.0, -1.0);
459
460          glTranslatef(0.0, 0.0, 0.9);
461          glRotatef(guy.partsRot[4][0], 1, 0, 0);
462          glRotatef(guy.partsRot[4][1], 0, 1, 0);
463          glRotatef(guy.partsRot[4][2], 0, 0, 1);
464          glTranslatef(0.0, 0.0, -0.9);
465      guy.pointer->getObjectsByName()["RightPalm"]->draw();
466      } glPopMatrix();
467      } glPopMatrix();
468      } glPopMatrix();
469
470      /*
471      glPushMatrix(); {
472      glTranslatef(0.0, 4.1, -0.7);
473      guy.pointer->getObjectsByName()["RightUpperLeg"]->draw();
474      } glPopMatrix();
475
476      glPushMatrix(); {
477      glTranslatef(0.0, 3.0, -0.7);
478      guy.pointer->getObjectsByName()["RightLowerLeg"]->draw();

```

### 9.7.5 Measuring Segment Significance: Time Taken versus Number of Changes

This Section describes the rationale of how the significance of segments was measured. Since the goal is to identify areas of high student effort, one logical approach would be to attempt to measure the time a student spends on Changes, and to identify Changes or sequences of Changes which require a significantly higher than average time to implement.

In order to use time taken as a measure of activity, it is necessary to exclude long periods of time between consecutive program compilations, since these in all likelihood indicate the student was interrupted in their work and involved in non-programming activities, and these long time periods would otherwise be such large outliers they would impact on the measure as a whole. Heuristics which address these issues were presented in Section 6.4.1.2.

Even given this correction in practice time taken is not very easy to interpret reliably, since there is no way to determine whether the student is spending the time between any two changes on programming or on some other unrelated task, such as chatting with a friend, answering a phone

call or browsing the internet. The idiosyncrasies students display in their work habits may have a considerable impact on time taken as a measurement.

Features of the problem being worked on may also contribute to time taken in ways that do not reflect the difficulty of the underlying problem. For example, segments involving the creation of data structures may require a lot of typing source code, but this code may be added without any errors. On the other hand, a different problem may involve fewer new lines being added with the student spending most of the time thinking instead of typing, making the problem more significant and interesting.

To test whether Average Time taken per Segment Change, Total Time taken by all of the Segment's Changes or the number of Changes in a Segment is the best measure of a Segment's significance, problem segments identified via human analysis (these segments are described in detail in Section 6.5) were manually categorised as being either solved with no real errors occurring during problem-solving (no error, code 1), solved with errors that were fixed efficiently (error, code 2) or solved with errors that required serious effort and involved incorrect solutions to fix (serious error, code 3). The outcome of this categorisation will be referred to as *Problem Difficulty*. Most Segments fell into the *error* or *serious error* category, probably because tasks with few errors generally took few Changes to implement.

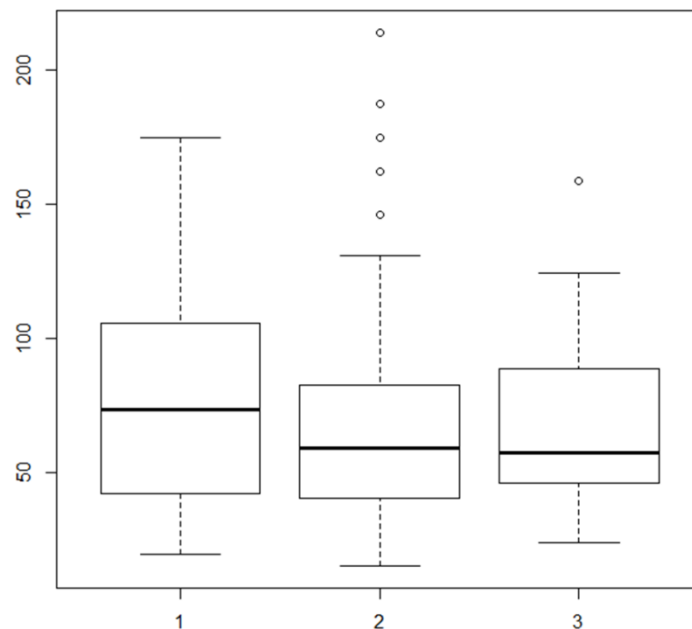
An Analysis of Variance can determine whether there is a significant correlation between Size, Average Time or Total Time and Problem Difficulty. For example, ANOVA may show that significantly more Segments with large Average Time values have a high Problem Difficulty than would be expected given a random normal distribution. This would suggest that Average Time is a good indicator of Problem Difficulty. To test which if any of the metrics is a good indicator of Problem Difficulty, ANOVA relating the metric to Problem Difficulty will be performed for each metric.

If *Size* is shown to vary more significantly according to *Problem Difficulty* than *Average Time* or *Total Time* then the hypothesis that size is the best indicator of segment significance is confirmed. *Total* is related to *size* since large segments will include more Changes and total time taken by a segment is the sum of the time taken by Changes of that segment; hence if *size* is correlated to *Problem Difficulty* then *total* is also likely to be correlated to *Problem Difficulty*. However, time only contributes positively to determining the significance of a problem if and only if *total time* shows a stronger correlation to *Problem Difficulty* than *size* does by itself.

### 9.7.5.1 Average Time

Table 74 shows descriptive statistics for Average Time by Problem Difficulty. Box plots for each of the difficulties are shown in Figure 177. The statistics show that Segments with a lower difficulty

have a higher Average Time than Segments with middle or higher difficulty, which is the opposite of the hypothesized effect. This means that if there is a significant correlation between Average Time and Problem Difficulty, it goes in the opposite direction than was expected; difficult problems have lower Average Times.



**Figure 177: Boxplot of Average Time by Problem Difficulty**

**Table 74: Descriptive Statistics for Average Time by Problem Difficulty**

var	N	mean	Sd	Median
1	40	78.22	40.98	73.67
2	86	68.61	39.55	59.28
3	54	66	28.56	57.72

The results of ANOVA of Average Time and Problem Difficulty are shown in Table 75. The p-value of 0.128 indicates that the null hypothesis is confirmed. There is no significant correlation between Segment Average Time and Problem Difficulty. This means that Average Time is not a good indicator of Problem Difficulty.

**Table 75: Average Time ~ Problem Difficulty ANOVA results**

	Df	Sum Sq	Mean Sq	F-Value	Pr (>F)

Problem Difficulty	1	3188	3188	2.343	0.128
Residuals	178	242243	1361		

### 9.7.5.2 Problem Size

Descriptive statistics for the relationship between Size and Problem Difficulty is presented in Table 76. It shows that while there is only a small difference in average Size for problems of no to moderate difficulty (15.32 to 16.72 Changes), there is a large difference to problems of significant difficulty (42.7 Changes). In fact as the boxplot shows the 75<sup>th</sup> percentile of NoError and Error is still less than the 25<sup>th</sup> percentile of SeriousError. This suggests that solving of serious problems involves many more Changes than does the solution of less serious problems.

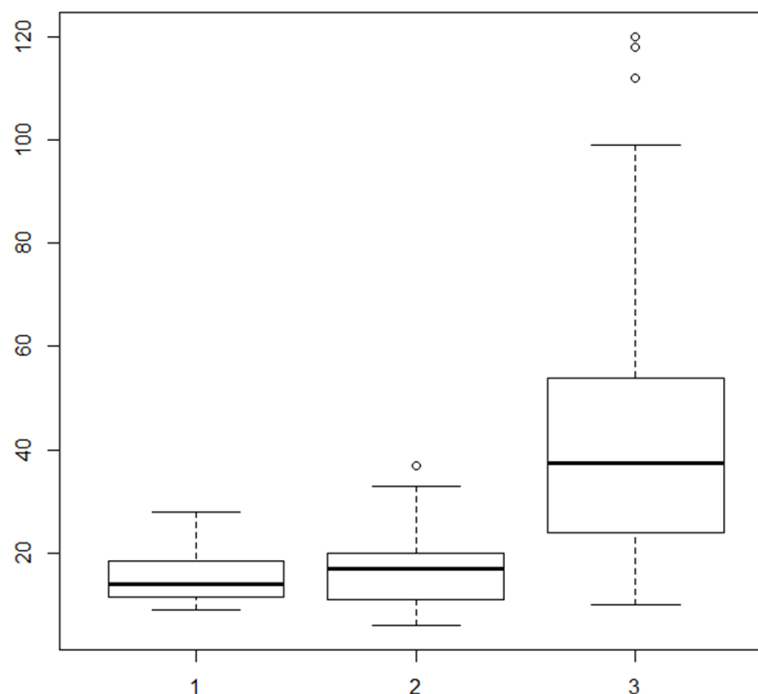


Figure 178: Boxplot for Size ~ Problem Difficulty

Table 76: Descriptive Statistics for Size ~ Problem Difficulty

var	N	Mean	Sd	Median
1	40	15.32	4.92	14
2	86	16.72	6.46	17
3	54	42.7	25.79	37.5

To confirm this intuition, ANOVA of Size according to Problem Difficulty was performed. The results are presented in Table 77. The p-value is significant at the  $p \leq 0.05$  level suggesting that Size is a good indicator of problem segment significance. The associated R-squared value is  $r^2 = 0.2982$ , indicating that Size explains a large amount of the variability of Problem Difficulty in segments.

**Table 77: ANOVA for Size ~ Problem Difficulty**

	Df	Sum Sq	Mean Sq	F-Value	Pr (>F)
Problem Difficulty	1	19735	19735	75.62	2.23E-15
Residuals	178	46456	261		

A pair-wise t-test with pooled standard deviation of Size between Problem Difficulty categories yields the results shown in Table 78.

**Table 78: Pair-wise t-test pooled standard deviation of Size ~ Problem Difficulty**

	NoError	Error
Error	0.63	-
SeriousError	3.10E-15	< 2e-16

Pairwise t-testing of the different mean Segment Size according to Problem Difficulty shows that there is no significant difference in Segment Size between NoError and Error Segments, but a very significant difference between NoError and SeriousError as well as between Error and SeriousError segments, with SeriousError segments including many more Changes as was shown in Table 78.

The outcome of ANOVA supports the hypothesis that Size is a good predictor of Problem Difficulty. This in turn validates the selection of Size as a ranking metric for Machine Generation of Segments.

### 9.7.5.3 Total Time

Size has been established as a good indicator of Problem Difficulty. However while Average Time is a poor predictor of Problem Difficulty, the relationship of Size and Time as encapsulated in Total Time may be an even better metric than either Size or Time in isolation. To test whether this is the case, an ANOVA of Total Time according to Problem Difficulty was performed.



The result of the ANOVA is presented in Table 79. The p-value is still significant. However, the F-value is half that of the ANOVA test using Size, suggesting that the Total Time metric is worse than the Size metric at predicting Problem Difficulty. The associated R-squared value  $r^2 = 0.1526$  shows that Problem Difficulty explains only half as much variance in Total Time than it does in Size.

The Total Time metric appears to be an inferior predictor of Problem Difficulty than does Size.

**Table 79: ANOVA of Total Time ~ Problem Difficulty**

	Df	Sum Sq	Mean Sq	F-Value	Pr (>F)
Problem Difficulty	1	78264522	78264522	32.06	5.91E-08
Residuals	178	4.35E+08	2441487		

#### 9.7.5.4 The best metric of Segment Significance: Problem Size

Analysis presented in this section shows that Size is a good predictor of Problem Difficulty while Average Time is not. This means that Size should be used to rank segments, and segments with larger Size can be assumed to generally involve more difficult underlying problems.

This raises the question of what variance in Average Time measures, since it does not measure Problem Difficulty directly. It seems reasonable to assume that Average Time measures the amount of time the student spends thinking about the problem during Changes. Short times may indicate a trial-and-error approach, whereas long times may indicate an approach in which the student invests significant mental effort in understanding the underlying problem. For example, debugging of event-driven code may involve the scanning of source code and would likely incur larger time spent between Changes than would a syntax error problem involving the student commenting out different lines of code. Hence, Time Spent per Change (and hence Average Time) is hypothesized to measure the problem-solving strategy underlying a Change or Segment.

It should be noted that part of the reason for why Size by itself outperformed Total Time (which can be seen as the Size multiplied by the Average Time per Change) is that the evaluation of segments into different levels of *Problem Difficulty* may have been biased by the method of analysis and the available data. While examination of Segments did also involve an examination of time-stamps, the total number of Changes that have to be examined to determine the content of the segment probably have a stronger impact on the *perceived* challenge posed by a segment than large timestamps. While the difference between Average Time per Change and Segment Size is large enough to be confident about rejecting Average Time per Change as a measure of Segment

significance, the difference between Segment Size and Total Time is probably not large enough to make a conclusive statement.

Segment Size was chosen as the primary measure of Segment significance in this research project, but further research should establish whether Total Time spent is actually an inferior metric. One approach could be to take segments of roughly equal size and compare those to see whether there is a significant difference in Total Time based on differences in Problem Difficulty.

## 9.7.6 Segment Reports / Memos

### 9.7.6.1 Assignment 1

#### 9.7.6.1.1 John

##### 9.7.6.1.1.1 Spatial

###### 9.7.6.1.1.1.1 JOHN\_A1.SP.1."Door Impl." [14]

Name: Door Impl.

Description: Spatial 2D

Occurrences: 68-75, 372-377 : total = 14 (+)

Detailed Description:

68-75, 372-377 (14 changes (+5 minutes, 7 changes), ~21 changes ++):

From 68-75, Implements door without error, one size tweak. (Basic shape and borders)

From 372-377 fixes the door drawing, which had an error in the formula for calculating the width of components.. First tries to fix by tweaking, then realises an error in the offset formula.

###### 9.7.6.1.1.1.2 JOHN\_A1.SP.2."Attempts GL transformations" [15]

Name: Attempts GL transformations

Description: Spatial 2D

Occurrences: 122-136 : total = 15 (+)

Detailed Description:

Attempts to use OpenGL transformation (rotation) but uses it away from actual drawing and w.out glBegin/glEnd so ends up rotating the entire drawing surface around an arbitrary axis.

###### 9.7.6.1.1.1.3 JOHN\_A1.SP.3."Arrow Icon" [13]

Name: Arrow Icon

Description: Spatial 2D

Occurrences: 215-227 : total = 13 (+)

Detailed Description:

Tweaks arrow size. Makes a spatial error when creating move icon but fixes in one change. Also misplaces button but then fixes in one change, Tweaks arrow over several changes 224-2276.

215-227

#### **9.7.6.1.1.1.4 JOHN\_A1.SP.4."Rotate Icon" [11]**

Name: Rotate Icon

Description: Spatial 2D

Occurrences: 284-294 : total = 11 (+)

Detailed Description:

Rotate Icon (10 changes guaranteed, +) 14 changes, 46 minutes) (~61? changes, EXCLUDE)

Done from file, can't be sure how long it took.

#### **9.7.6.1.1.1.5 JOHN\_A1.SP.5."Resize Icon" [11]**

Name: Resize Icon

Description: Spatial 2D

Occurrences: 295-305 : total = 11 (+)

Detailed Description:

295-305 Resize Icon (11 changes, 24 minutes) (~32 changes, +++)

Done from file, can't be sure how long it took.

#### **9.7.6.1.1.1.6 JOHN\_A1.SP.6."Flower Pot" [14]**

Name: Flower Pot

Description: Spatial 2D

Occurrences: 456-464, 470-474 : total = 14 (+)

Detailed Description:

Flower pot not drawn because of operator precedence (forgets that using bit shift operations to replace multiplication and division have low precedence), tries different solutions involving spatial programming before correcting the problem. Makes the same operator precedence error again when placing colour buttons

#### **9.7.6.1.1.1.7 JOHN\_A1.SP.7."Rotation around Parent" [56 ]**

Name: Rotation around Parent

Description: Spatial 2D / Math

Occurrences: 657-664, 667-687, 691-710, 714-720 : total = 56 (+++++)

Detailed Description:

Working on rotation around parent. Again gets calculating centre wrong by using  $\text{upper.x} - \text{lower.x}/2$ , and uses bad formula for calculating angle which does not allow for 360 degree rotation.

Tries casting to int and using debug messages to print out variable values. Fixes centre calculation (but uses parent centre instead of child centre), but is moving centre instead of lower-left-corner, so moving too far (hence moving the child away from the parent when rotated).

After fixing that problem, runs into problems because the int position of objects loses decimal point values, leading objects to get closer to parent as they're rotated (very common problem amongst students). Eventually solves the problem by storing parent-child distances instead of using changing child centres (fixed 684)

686-714 Tries to find a formula for rotating smoothly around 360 degrees, again gets distance formula wrong ( $(lx - sx/2)$ , instead of  $(lx - (lx - sx)/2)$ ). Finds a good formula, but still rotates the child's lower-left corner around the parent rather than its centre, so the rotation isn't entirely correct and remains that way.

Note that students aren't really drawing out points of rotation etc much... that would help... a more constrained environment with such functions, or teaching them how to do it.

#### **9.7.6.1.1.2 Event-Driven**

##### **9.7.6.1.1.2.1 JOHN\_A1.ED.1."Screen-Window convo" [15]**

Name: Screen-Window convo

Description: Screen-Window

Occurrences: 229-243 : total = 15 (+)

Detailed Description:

2. 229-243, (15 changes, +):

Implementing add-object mode, forgets y-height again

##### **9.7.6.1.1.2.2 JOHN\_A1.ED.2."Deleting Children" [11]**

Name: Deleting Children

Description: Event-Driven Program Flow

Occurrences: 568-578 : total = 11 (+)

Detailed Description:

Deleting children, ED problem w. iterator

#### **9.7.6.1.1.2.3 JOHN\_A1.ED.3."GLUT Menu" [23]**

Name: GLUT Menu

Description: Event-Driven Program Flow

Occurrences: 752, 754-775 : total = 23 (++)

Detailed Description:

GLUT menu adding, again struggles with the fact that a mouse click causes two mouse events (A mouseup and a mousedown) and separating menu events from button events and retrieving menu events even when the menu is not visible

#### **9.7.6.1.1.3 General Programming**

##### **9.7.6.1.1.3.1 JOHN\_A1.GP.1."Virtual function and Constructor" [11]**

Name: Virtual function and Constructor

Description: OO Programming / Virtual Keyword

Occurrences: 39-49 : total = 11 (+)

Detailed Description:

Implementing FlowerPot, forgetting to use virtual keyword and correct constructor.

##### **9.7.6.1.1.3.2 JOHN\_A1.GP.2."Data type for RGB" [10]**

Name: Data type for RGB

Description: C++ Data Type Syntax

Occurrences: 81-90 : total = 10 (+)

Detailed Description:

The student requires several changes to determine the correct data type for storing RGB data.

##### **9.7.6.1.1.3.3 JOHN\_A1.GP.3."Delete Mode Memory Problems" [20]**

Name: Delete Mode Memory Problems

Description: Memory Management

Occurrences: 580-599 : total = 20 (++)

Detailed Description:

Adding delete mode, pointer/memory problems lead to crashes. "Fixed" at 646 by not freeing memory.

### **9.7.6.1.2 Christopher**

#### **9.7.6.1.2.1 Spatial**

##### **9.7.6.1.2.1.1 CHRISTOPHER\_A1.SP.1."Circle Icon" [16]**

Name: Circle Icon

Description: Spatial 2D

Occurrences: 135, 141, 146-153, 156, 158-159, 165-167 : total = 16 (+)

Detailed Description:

141, 146-160, 165-167 (-135, +154, +155, +157, +160)

2. 135, 141, 146-153, 156, 158-159, 165-167 (17, +); Fixes circle icon and positions

##### **9.7.6.1.2.1.2 CHRISTOPHER\_A1.SP.2."Hit Code" [17]**

Name: Hit Code

Description: Spatial 2D

Occurrences: 259-275 : total = 17 (+)

Detailed Description:

4. 259-275 (16, +): Implementing hit functionality of drawing surface to check whether something is being drawn outside of surface. Makes simple error in formula and requires 16 changes to fix it. (Very long for such a simple error)

(Wrong: `_p.getY()<lower.getY()&&_p.getY()>upper.getY()`)

##### **9.7.6.1.2.1.3 CHRISTOPHER\_A1.SP.3."Door Icon" [20]**

Name: Door Icon

Description: Spatial 2D

Occurrences: 299-318 : total = 20 (++)

Detailed Description:

5. 299-318 (20, ++): Working on door icon. Requires 14 major modifications to make the relatively simple 2D fix.

Mod: IIIII IIIII IIII (14)

Tweak: (0)

##### **9.7.6.1.2.1.4 CHRISTOPHER\_A1.SP.4."Clipping" [10]**

Name: Clipping

Description: Spatial 2D

Occurrences: 759-760, 763-770 : total = 10 (+)

Detailed Description:

9. 759-760, 763-770 (11, +): Clipping. Tries to implement four rectangles to cover four sides of drawing area to do 'clipping'. Makes two errors, requires 9 changes to implement relatively simple rectangles.

Changes: IIIII IIII

Tweaks:

Errors: II

#### **9.7.6.1.2.1.5 CHRISTOPHER\_A1.SP.5."Move Icon" [13]**

Name: Move Icon

Description: Spatial 2D

Occurrences: 740-749, 754-756 : total = 13 (+)

Detailed Description:

8. 740-749, 754-756 (11, +): Implements move icon.

Creates triangle/arrowhead from 740-741, then copies it for all four dimensions from 741-748, making one error in placement at 746.

Works on the lines connecting the arrowheads from 754-756. Solved without real problems.

#### **9.7.6.1.2.2 Event-Driven**

##### **9.7.6.1.2.2.1 CHRISTOPHER\_A1.ED.1."Screen-Window" [9]**

Name: Screen-Window

Description: Screen-Window

Occurrences: 47-55 : total = 9 ()

Detailed Description:

1. 47-55 (10, +) Forgets to convert from screen to window coords, attempts to fix hit code and debug before fixing

##### **9.7.6.1.2.2.2 CHRISTOPHER\_A1.ED.2."Hit Code" [18]**

Name: Hit Code

Description: Hit Code Error

Occurrences: 170-187 : total = 18 (+)

Detailed Description:

2. 170-187 (18, +) Error in hit code condition prevents buttons from working

#### **9.7.6.1.2.2.3 CHRISTOPHER\_A1.ED.3."Hit Code 2" [21]**

Name: Hit Code 2

Description: Hit Code Error

Occurrences: 257-277 : total = 21 (++)

Detailed Description:

3. 257-277 (21, ++) Again, error in hit code (this time when trying to restrict drawing to draw area). Unfortunately, includes another error (using hit instead of !hit to determine whether to draw) which makes the problem slightly more complicated.

#### **9.7.6.1.2.2.4 CHRISTOPHER\_A1.ED.4."Hit Code 3" [11]**

Name: Hit Code 3

Description: Hit Code Error

Occurrences: 422-432 : total = 11 (+)

Detailed Description:

4. 422-432 (11, +) Hit code error at delete causes objects hits to not register.

#### **9.7.6.1.2.2.5 CHRISTOPHER\_A1.ED.5."ED program flow" [11]**

Name: ED program flow

Description: Event-Driven Program-Flow

Occurrences: 611, 617-618, 622-623, 643-648 : total = 11 (+)

Detailed Description:

5. 611, 617-618, 622-623, 643-648 (11, +)

611-648 (38, +++) Retrieving object to move it, then trying to retrieve it again using its position after the move with the old coordinates retrieves a null object, which is then accessed and crashes the application. ED program flow error.

#### **9.7.6.1.2.2.6 CHRISTOPHER\_A1.ED.6."ED Program flow 2" [12]**

Name: ED Program flow 2

Description: Event-Driven Program-Flow

Occurrences: 860-871 : total = 12 (+)

Detailed Description:



6. 860-871 (12, +) Using wrong variable not storing values, ED program flow error.

### **9.7.6.1.2.3 General Programming**

#### **9.7.6.1.2.3.1 CHRISTOPHER\_A1.GP.1."this' keyword" [10]**

Name: 'this' keyword

Description: C++ syntax

Occurrences: 225, 240-245, 251-252, 255 : total = 10 (+)

Detailed Description:

2. 225, 240-245, 251-252, 255 (10, +) problems with this keyword

#### **9.7.6.1.2.3.2 CHRISTOPHER\_A1.GP.2." C++ this. syntax / static functions" [12]**

Name: C++ this. syntax / static functions

Description: C++ syntax

Occurrences: 240-245, 251-256 : total = 12 (+)

Detailed Description:

5. ((240-245, 251-256), (11 changes +)) C++ this. syntax / static functions

#### **9.7.6.1.2.3.3 CHRISTOPHER\_A1.GP.3."C++ String" [22]**

Name: C++ String

Description: C++ syntax

Occurrences: 556-577 : total = 22 (++)

Detailed Description:

8. 556-577 (22, ++) (C++ Strings) String representation of point, struggling w. C++ string syntax + concatenating

#### **9.7.6.1.2.3.4 CHRISTOPHER\_A1.GP.4."Macro" [47]**

Name: Macro

Description: C++ syntax

Occurrences: 360-394, 415-416, 475-484 : total = 47 (++++)

Detailed Description:

6. 360-394, 415-416, 475-484 (47 changes, +++) (Macros) Attempting to use macros for modes, syntax is incorrect (forgets semicolon) resulting in errors when the macro is used.

#### **9.7.6.1.2.3.5 CHRISTOPHER\_A1.GP.5."Virtual Keyword" [31]**

Name: Virtual Keyword

Description: C++ Object-Oriented Programming

Occurrences: 880-893, 896-897, 901-915 : total = 31 (+++)

Detailed Description:

9. 880-893, 896-897, 901-915 (31, +++) (Virtual Keyword) Struggles w. implementing select method for DiagramObject because forgets virtual keyword

#### **9.7.6.1.2.3.6 CHRISTOPHER\_A1.GP.6."General Semantics" [10]**

Name: General Semantics

Description: Computer Science Oversight

Occurrences: 1014-1023 : total = 10 (+)

Detailed Description:

11. 1014-1023 (10, +) (General Semantics) Wall delete implementation, forgets to modify from copied delete code (door) to wall.

#### **9.7.6.1.2.3.7 CHRISTOPHER\_A1.GP.7."Confusing function bracket error" [25]**

Name: Confusing function bracket error

Description: C++ Misleading Error

Occurrences: 1060, 1063-1067, 1069-1070, 1073-1089 : total = 25 (++)

Detailed Description:

12. 1060, 1063-1067, 1069-1070, 1073-1089 (26 changes, ++) (CONFUSING) Forgetting the function brackets at a function causes strange "insufficient contextual information to determine type" error (NEW ERROR TYPE: CONFUSING C++) 1107-1021 (15, +) (Virtual Keyword) Tries making move and select virtual from DiagramObject, gives up on select. 12. 1060, 1063-1067, 1069-1070, 1073-1089 (26 changes, ++) (CONFUSING) Forgetting the function brackets at a function causes strange "insufficient contextual information to determine type" error (NEW ERROR TYPE: CONFUSING C++)

#### **9.7.6.1.2.3.8 CHRISTOPHER\_A1.GP.8."Virtual keyword 2" [15]**

Name: Virtual keyword 2

Description: C++ OO syntax

Occurrences: 1107-1121 : total = 15 (+)

Detailed Description:

1107-1021 (15, +) (Virtual Keyword) Tries making move and select virtual from DiagramObject, gives up on select.

### **9.7.6.1.3 Michael**

#### **9.7.6.1.3.1 Spatial**

##### **9.7.6.1.3.1.1 MICHAEL\_A1.SP.1."Grid Drawing" [36]**

Name: Grid Drawing

Description: Spatial

Occurrences: 133-150, 161, 164-167, 169, 172-175, 219-223, 225-227 : total = 36 (+++)

Detailed Description:

From 133-150, Trying to draw a straight line for grid, gets it right at 150. Remarkably many changes for such a simple task. From 161-175 Starts on drawing the full grid. Draws it w.out loops!

From 464-468, replaces the entire thing with loops (in 3 changes!)

8 major

20 Tweaks

##### **9.7.6.1.3.1.2 MICHAEL\_A1.SP.2."Button Position/Size" [10]**

Name: Button Position/Size

Description: Spatial

Occurrences: 252, 256-264 : total = 10 (+)

Detailed Description:

Modifying button position/size and icon position/size again (more tweaking?), as well as placement of furniture object.

Major III

Tweak IIIII II

##### **9.7.6.1.3.1.3 MICHAEL\_A1.SP.3."Button Spatial Coloring"**

Name: Button Spatial Coloring

##### **9.7.6.1.3.1.3.1 MICHAEL\_A1.SP.3.1."Button Positioning" [21]**

Name: Button Positioning

Description: Spatial

Occurrences: 389-390, 393, 396-413 : total = 21 (++)

Detailed Description:

a) 389-390, 393, 396-413 (22, ++)

(Note, the above occurrence list is incomplete) Adding new buttons. A LOT of tweaking into position.

Tweak IIIII IIIII IIIII IIII

Major II

#### **9.7.6.1.3.1.3.2 MICHAEL\_A1.SP.3.2."Button Spatial Coloring" [26]**

Name: Button Spatial Coloring

Description: Spatial/Math

Occurrences: 414-424, 429, 431-432, 434-437, 440-447 : total = 26 (++)

Detailed Description:

b) 414-424, 429, 431-432, 434-437, 440-447 (28, ++)

Working on code that colors buttons according to position. Requires several major changes and tweaks to properly set the condition.

An additional problem here may be the fact that the result of the spatial condition is only indirectly visible through button color, coordinates are not displayed (this may be a problem for many conditional statements involving coordinates).

Several times changes from lesser than to greater than.

Lesser->Greater than II

Tweak IIIII IIIII IIIII I

Major IIIII IIIII

#### **9.7.6.1.3.1.4 MICHAEL\_A1.SP.4."Line buttons" [18]**

Name: Line buttons

Description: Spatial

Occurrences: 660-661, 664-671, 679, 682-688 : total = 18 (+)

Detailed Description:

Creating/Adding/Positioning line buttons.

Some errors, but mostly tweaking things into position.

(669 is first button more or less complete)

Tweak IIIII IIIII

Major IIIII I

#### **9.7.6.1.3.1.5 MICHAEL\_A1.SP.5."Positioning button text" [26]**

Name: Positioning button text

Description: Spatial

Occurrences: 965, 969-972, 977, 980-983, 990-993, 997, 999-1000, 1007-1013, 1015-1016 : total = 26 (++)

Detailed Description:

Positioning text for buttons, a lot of tweaking.

#### **9.7.6.1.3.1.6 MICHAEL\_A1.SP.6."Clipping" [56]**

Name: Clipping

Description: Spatial

Occurrences: 1293, 1296-1306, 1308-1310, 1326-1328, 1330-1346, 1348, 1351-1352, 1362-1372, 1374-1375, 1384-1388 : total = 56 (+++++)

Detailed Description:

Working on clipping w. scissor test and coordinate test for moving objects. For a long time, forgot to take into account width and height of objects, resulting in LL coordinate test working, but not UR. Adds width and height 1387. A \*lot\* of tweaking. Again, a case where the result is not immediately visible, which may cause extended tweaking (was hard to debug).

Tweak II II II II II II II II II II II II II II II

Major II II II II

#### **9.7.6.1.3.1.7 MICHAEL\_A1.SP.7."Positioning status text" [23]**

Name: Positioning status text

Description: Spatial

Occurrences: 1048-1049, 1053-1057, 1061-1062, 1064, 1071-1072, 1075, 1077, 1244-1250, 1322-1323 : total = 23 (++)

Detailed Description:

positioning text for status box (from 1048). Few errors and \*a lot\* of tweaking into place.

#### **9.7.6.1.3.1.8 MICHAEL\_A1.SP.8."Menu" [18]**

Name: Menu

Description: Spatial

Occurrences: 1426-1427, 1430-1431, 1433, 1436-1441, 1444-1445, 1447-1448, 1450, 1452, 1454 : total = 18 (+)

Detailed Description:

Attempt at setting up menu (unsuccessful, but not due to coordinates).

Tweak IIIII IIIII IIII

Major IIII

#### **9.7.6.1.3.1.9 MICHAEL\_A1.SP.9."Circle Draw" [26]**

Name: Circle Draw

Description: Math

Occurrences: 1527, 1529, 1531, 1533-1536, 1538-1550, 1552-1557 : total = 26 (++)

Detailed Description:

Student attempts to implement flower pot object (circle) but uses wrong formula, then attempts to fix by tweaking values (which doesn't work because formula is wrong.) Gets onto the error at 1539.

Final fix 1553.

Wrong formula is responsible for most problems, but also some tweaking.

Tweak IIIII IIIII

Major IIIII IIIII IIIII III

#### **9.7.6.1.3.1.10 MICHAEL\_A1.SP.10."Circle resize" [18]**

Name: Circle resize

Description: Math

Occurrences: 1647, 1650-1652, 1657-1670 : total = 18 (+)

Detailed Description:

Working on circle resize, finding a(math formula) to properly calculate distance between button up and button down (Works at 1952). Doesn't convert screen-window.

Then tries to make it so circle cannot be resized past gui border (works at 1670).

Quite a bit of tweaking.

Tweak IIIII II

Major IIIII IIIII

#### **9.7.6.1.3.1.11 MICHAEL\_A1.SP.11."Circle boundary check" [18]**

Name: Circle boundary check

Description: Spatial

Occurrences: 1653-1670 : total = 18 (+)

Detailed Description:

Implementing "boundary check" to ensure circle can't be drawn over gui.

#### **9.7.6.1.3.1.12 MICHAEL\_A1.SP.12."Wall Icon" [21]**

Name: Wall Icon

Description: Spatial

Occurrences: 1756-1757, 1763-1768, 1773-1775, 1778, 1782-1790 : total = 21 (++)

Detailed Description:

Wall icon render and use in button. Mostly tweaking, few mistakes.

Tweak IIIII IIIII IIIII II

Major III

#### **9.7.6.1.3.1.13 MICHAEL\_A1.SP.13."Select Icon" [11]**

Name: Select Icon

Description: Spatial

Occurrences: 2060-2065, 2067-2071 : total = 11 (+)

Detailed Description:

Drawing and placing select icon.

Tweak IIIII IIIII II

Major

#### **9.7.6.1.3.1.14 MICHAEL\_A1.SP.14."Move Icon" [21]**

Name: Move Icon

Description: Spatial

Occurrences: 2077-2097 : total = 21 (++)

Detailed Description:

Move Icon

Tweak IIIII IIIII IIIII

Major IIIII I

#### **9.7.6.1.3.1.15 MICHAEL\_A1.SP.15."Delete Icon" [18]**

Name: Delete Icon

Description: Spatial

Occurrences: 2101-2103, 2105-2119 : total = 18 (+)

Detailed Description:

Delete icon and placement.

Tweak IIIII IIIII IIIII III

Major III

#### **9.7.6.1.3.1.16 MICHAEL\_A1.SP.16."Onto Icon" [14]**

Name: Onto Icon

Description: Spatial

Occurrences: 2122-2125, 2127, 2129-2130, 2132, 2134-2139 : total = 14 (+)

Detailed Description:

Onto Icon and placement.

Tweak IIIII IIIII I

Major III

#### **9.7.6.1.3.1.17 MICHAEL\_A1.SP.17."Rotatelcon" [11]**

Name: Rotatelcon

Description: Spatial

Occurrences: 2148, 2151-2152, 2154-2161 : total = 11 (+)

Detailed Description:

Working on Rotatelcon

#### **9.7.6.1.3.1.18 MICHAEL\_A1.SP.18."Resize Icon" [11]**

Name: Resize Icon

Description: Spatial

Occurrences: 2169-2170, 2172, 2174-2177, 2180-2183 : total = 11 (+)

Detailed Description:

Resize Icon

Tweak II

Major IIIII III

### **9.7.6.1.3.2 Event-Driven**

#### **9.7.6.1.3.2.1 MICHAEL\_A1.ED.1."Button highlight" [45]**

Name: Button highlight



Description: Event-Driven Program Flow

Occurrences: 639-643, 647-648, 716-735, 737-749, 757-761 : total = 45 (++++)

Detailed Description:

11. (Button highlight) (639-643,647-648, 716-735, 737-749, 757-761)(42, +++)

639-643,647-648 (7), 716-735, 737-749, 757-761 (35) (42, +++)

So it *\*does\** seem that what he wants is for it to light up ONLY while it's being pressed (rather than staying lit till a different button is pressed).

This is complicated by the fact that the reset function which is used to init button colors in display func also resets the color of the button back when it's set inside the mouse handler (so it's an ED program flow issue).

#### **9.7.6.1.3.2.2 MICHAEL\_A1.ED.2."Select ll > ur" [30]**

Name: Select ll > ur

Description: LL < UR

Occurrences: 1133, 1136-1153, 1155, 1158, 1161-1169 : total = 30 (+++)

Detailed Description:

Select doesn't work because when objects are added the ll may be larger than upper-right, which makes hit function not work.

#### **9.7.6.1.3.2.3 MICHAEL\_A1.ED.3."GLUT Menu" [28]**

Name: GLUT Menu

Description: Event-Driven Program Flow

Occurrences: 1392, 1394, 1398-1402, 1407, 1409-1410, 1412-1414, 1449, 1451, 1453, 1459-1460, 1464, 1473-1474, 1485-1486, 1490-1491, 1499-1501 : total = 28 (++)

Detailed Description:

Trying to use GLUT menu as dropdown menu. Gets it into more or less working order by end (though this isn't really how menu should be used).

Main problem is correctly attaching and detaching the menu to ensure it pops up only if the menu button is pressed, and then disappears again. It seems difficult for the student to understand the sequence of function calls that occur during event handling. (1391-1484)

#### **9.7.6.1.3.2.4 MICHAEL\_A1.ED.4."Resizing circle" [10]**

Name: Resizing circle

Description: Event-Driven Program-Flow

Occurrences: 1707-1716 : total = 10 (+)

Detailed Description:

When resizing, introduces getk and geth variables at 1702 but in resize uses x1f, x1y instead. Fixes to using getk and geth at 1716.

#### **9.7.6.1.3.2.5 MICHAEL\_A1.ED.5."???" [18]**

Name: ???

Description: NONE

Occurrences: 1845-1847, 1851-1852, 1858-1860, 1863-1867, 1869-1873 : total = 18 (+)

Detailed Description:

Copies code from furniture to door, no real problems.

### **9.7.6.1.3.3 Pipeline**

#### **9.7.6.1.3.3.1 MICHAEL\_A1.PI.1."Init render" [19]**

Name: Init render

Description: Draw Not Store

Occurrences: 92-110 : total = 19 (+)

Detailed Description:

2. (Init render) (92-110) (19, ++)

92-97, 99-100, 106, 108, 110

Attempts to use a function that draws a simple shape, but puts it into init instead of display, so it's not drawn.

At 106, instead of drawing shape inits button (which is rendered in display).

At 110 finally puts it in the right place.

#### **9.7.6.1.3.3.2 MICHAEL\_A1.PI.2."Not flushing" [10]**

Name: Not flushing

Description: No Flush

Occurrences: 228-234, 241-243 : total = 10 (+)

Detailed Description:

6. (Not flushing) (228-234, 241-243) (10, +)

228, 230-231, 233-234, 241, 249

To have diagram object drawn in front of rather than behind grid, moves the statements in front of grid BUT moves after glFlush call, leading to it not being displayed properly.

Initially everything is drawn fine, but after the first redisplay is done the things drawn after the flush tend to not get drawn anymore.

This takes a LONG time to be fixed at 648-650.

#### **9.7.6.1.3.3.3 MICHAEL\_A1.PI.3."Color button code" [10]**

Name: Color button code

Description: Experimenting glutPostRedisplay

Occurrences: 468-477 : total = 10 (+)

Detailed Description:

8. 468-477 (No Action/Error, only problem, 10, +)

Tries adding and removing glutPostRedisplay and render calls from where buttons are drawn, possibly to debug the spatial conditional code for coloring buttons. Whatever the exact purpose, the changes are useless as the initial state is the correct one.

#### **9.7.6.1.3.3.4 MICHAEL\_A1.PI.4."Highlight not stored" [24]**

Name: Highlight not stored

Description: Draw Not Store

Occurrences: 600-623 : total = 24 (++)

Detailed Description:

10. (Highlight not stored) (600-623) (24, ++)

605-609

Attempts to highlight buttons by having a function which draws the button in a different color, calls this function in the mouse handler, this again fails because the following display call will clear the screen.

Fixes at 623 after implementing a change color method and using it in the mouse handler, rather than trying to draw the object straight out. This type of error (trying to draw at a certain point instead of changing state) seems very common amongst students.

#### **9.7.6.1.3.3.5 MICHAEL\_A1.PI.5."Not storing objects" [45]**

Name: Not storing objects

Description: Draw Not Store

Occurrences: 797-841 : total = 45 (++++)

Detailed Description:

12. (Not storing objects ) (797-841) (48, +++)

777-788, 790-792, 794-798, 797-798, 810-822, 827-838, 840-841(48)

When trying to implement add functionality, just creates and renders but does not store diagramObject in mouse handler. This means it gets cleared in display. Tries to address this by adding \*yet another\* glutPostRedisplay call immediately after the render.

Improves this in 807 when actually stores a new diagramObject and renders it in display, but this always renders furnitures[0] even before button to init it is pressed.

This will lead to the display method to fail at that point, leaving most of the screen undrawn, until the object is inited. This confuses the student, so it's removed again at 810. The C++ tendency to without crash proceed from such events is pretty nasty (esp. in OpenGL context).

Fixes this by initing to default value in main() func ,but then doesn't add the new furniture's render back into display so it doesn't display (instead rendering in mouse handler where it gets cleared).

Tries to debug this by drawing various different shapes in the mouse handler and adding yet another display() call, and changing from display to glutPostRedisplay for that new call. Then uses debug cout statement (818) to see whether the condition is reached at all in mouse handler, and from 820-825 debugs ED to see whether that's at fault.

From 826-831 again tries drawing shape in mouse handler, changing its color and adding redisplay calls (the last two being frequent signs of confusion). 832, tries to move the code from the mouse down to the mouse up handler (again pointless), before going back to trying to put in more redisplay calls 834-835.

In 837-838 FINALLY puts both initialisation of DiagramObject and its render in \*display\* function back in, which causes the object to be rendered.

In 840, moves the render into the condition that is only run once (the one that contains the reset), which is wrong since it won't be drawn anymore after the first display, but moves it back in 841.

This stretch displays a definite confusion with how the pipeline works and how draw actions are sequenced, and also showcases the most common behaviors when such a problem is encountered: useless color changes and addition of redisplay or flush calls.

#### **9.7.6.1.3.3.6 MICHAEL\_A1.PI.6."Select Outline" [11]**

Name: Select Outline

Description: Pipeline Error

Occurrences: 871, 873, 878-880, 890, 892-896 : total = 11 (+)

Detailed Description:

Here, student tries to implement select outline for objects being selected. The ED part of the code (figuring out which object is selected) actually works, but due to a pipeline error (select outline being overdrawn) this is not visible to student, and student debugs ED aspect and spatial aspect (drawing of outline) (misidentifying problem).

Uses debug statement, then creates variable to store whether something was selected.

Fixes in 896, working in 897.

#### **9.7.6.1.3.3.7 MICHAEL\_A1.PI.7."Scissor test" [19]**

Name: Scissor test

Description: Pipeline Error

Occurrences: 1294-1295, 1312-1313, 1315-1320, 1353, 1355-1356, 1360-1361, 1376, 1379-1381 : total = 19 (+)

Detailed Description:

15. (Scissor test) (1294-1295, 1312-1313, 1315-1320, 1353, 1355-1356, 1360-1361, 1376, 1379-1381 (21 changes, ++)

Trying to get scissor test to work. Never does manage to get the enabling/disabling into the proper order to protect the GUI. Also involves a lot of spatial tweaking of scissor coordinates (but this is coded Sp and included in problem Sp.17)

#### **9.7.6.1.3.3.8 MICHAEL\_A1.PI.8."Wall draw"**

Name: Wall draw

##### **9.7.6.1.3.3.8.1 MICHAEL\_A1.PI.8.1."Wall Drawn Not Stored" [65]**

Name: Wall Drawn Not Stored

Description: Draw Not Store

Occurrences: 1881, 1883-1892, 1894-1897, 1899-1908, 1910-1916, 1918-1927, 1932, 1934, 1937-1946, 1948-1952, 1954, 1956-1960 : total = 65 (++++ (6))

Detailed Description:

a) (1881, 1883-1892, 1894-1897, 1899-1908, 1910-1916, 1918-1927, 1932, 1934, 1937-1946, 1948-1952, 1954, 1956-1960) (Draw not store)(65 changes, ++++(6))

1881) Starts creating the drawWalls function

1883) Has the drawWalls function draw a single point

1884) Modifies the position of that point

1885) Makes the point be at x,y, changes point's color to red

1886) Changes the point's size  
1887) Removes the drawWalls function from the display handler (because it requires the x,y)

Wall won't render, because the mouse handler has a glutPostRedisplay at the end (and the wall render is drawing only inside function, typical non-storing error).

1888) Adds a hanging glBegin call to display function  
1889) Adds the glEnd to the hanging glBegin to reset function (called when any button is pressed to reset button colors)  
1890) Adds the drawWalls call to display to try it out, but adds it to beginning so gui fill will be drawn over it.  
1891) Tries to fix the unrecognised pipeline problem by changing color before function call (though the function call itself already changes the color)  
1892) Tries manually drawing a point w.out the function in display, but at the same spot, so same pipeline problem.  
1893) (fixes above using fixed value, since display doesn't have access to x,y)  
1894) Moves the glColor call AFTER the point is drawn (which is even wronger, but won't make any difference as the call is superfluous in the first place)  
1895) Moves the glColor and point size calls outside the glBegin/End (good) but stays AFTER the point is actually drawn (useless) and the point is still overdrawn and doesn't show up  
1896) Tries to fix by changing point's position.  
1897) Moves the glColor call in front of the point drawing.  
This would be correct, but the glColor call is superfluous in the first place.  
1898) Tries changing GL\_POINTS to GL\_POINT, which is a valid macro but it is incorrect to use it with glBegin  
1899) Tries changing point position again, also moves glPointSize in front of point drawing (good) but is still getting overdraw.  
1900) Removes the hanging glEnd in the reset function (not that it will make any difference)  
1901) Removes the hanging glBegin in the display handler  
1902) Removes the drawWalls call from the mouse handler  
1903) Adds the hanging glBegin call back into the display handler, has the display handler call drawWalls if we're in the wall drawing mode  
(so still going with naive, non-store approach)  
1904) Adds the hanging glEnd back into the reset handler.  
1905) Now puts the point drawing into the correct position so at least that gets drawn.  
1906) Moves the point drawing even lower, after the button renders. Not that this makes any difference...

1907) Moves it into the loop that draws outlines for selected objects (for some reason), which means it only gets drawn (many times, but always at the same spot) when there are selected objects.

1908) Removes the dangling glBegin call from the display handler.

1909) Adds a space at 441

1910) Uses a fixed value in the drawWalls function (not that that makes any difference, since where it's being called in the mouse handler it still gets cleared via postRedisplay)

1911) Adds the hanging glBegin to the hit conditional for the wall button (so when wall button is pressed, add hanging glBegin).

The student is obviously not thinking at all about how he could have multiple walls.

1912) Adds some glVertex calls to the hanging glBegin. It does draw, but only as long as the button is pressed (because once it stops being pressed, the next display call from mouse handler will wipe it out)

It is the hanging glEnd in the reset function that makes it work...

1913) Now that he adds a glEnd call inside the button conditional, it only draws for a very brief moment.

1914) Changing the coordinatse won't help

1915) Changing the coordinatse won't help

1916) Moves the line drawing from inside the conditional to where the point is being drawn.

1917) Corrects syntax error (x and y not available).

Now the lines are drawn correctly

1918) Surrounds the line drawing with a conditional so it's only drawn when in wall drawing mode.

1919) Removes the glEnd from the line drawing, leaving it dangling.

Still draws (due to glEnd in reset)

1920) Removes the glEnd from reset, still gets drawn (must be glEnd somewhere?)

1921) Removes the glEnd from the point drawing, so now all the vertices in the line drawing are drawn as points

1922) Removes the glBegin for the line loop drawing

1923) Removes the point drawing code, so now the vertex calls for the line drawing don't have any begin (and don't get drawn anymore)

1924) Adds two glBegin statements, first a GL\_POINTS statement and then a GL\_LINE\_LOOP statement without closing the point statement, leading to the vertices being drawn as points

1925) Removes the (currently useless) GL\_LINE\_LOOP call

1926) Moves point drawing (hanging / glEnd-less) code to wall button hit test in processMouse function (so it won't be drawn, glutPostRedisplay) and removes the hanging glVertex calls left over from the line drawing.

Here, he's trying out the pipeline

1927) Removes the only glVertex call in the hanging glBegin(GL\_POINTS) statement

1928) Fixes syntax, when the button is pressed (and the hanging call is executed) points are drawn at the four corners of the drawing surface, this must be because the call overrides the call that draws the border around the drawing area.

1929) Removes everything but a hanging glVertex call from the drawWall function (which is called in processMouse when the mode is wall drawing mode)

The point is not drawn.

1930) Adds a variable to store the state of the mouse button click (up or down)

1932) Removes the conditional for wall drawing from processMouse (including drawWalls call)

1933) Inits the variable that stores state of mouse button click.

1934) Adds a conditional to display function that is triggered when in wall mode.

It's got a hanging glBegin(GL\_POINTS) as well as a call to the drawWalls function (which now only contains a single glVertex call)

Works, but point size is 1 (very small)

1936) Changes point size to make it visible.

1937) Now the point is displayed wherever he clicks (probably the idea was that the points would stay so a wall could be formed, but of course not... since it's cleared)

1938) Removes mouse check (which was wrong btw, = instead of ==), probably trying to make dots stay

1939) Adds back

1940) Removes again, probably seeing whether there's a difference

1941) Adds drawing area check

1942) Removes the dangling glEnd (which isn;t doing anything anyways)

1943) Adds it back

1944) Changes point drawing to line drawing, draws from origin to mouse pointer (because some other glBegin/End statement down the line must be drawing at origin)

1945) Adding the glEnd w.out conditional makes the line draw have only one vertex, so nothing is drawn.

1946) This is just weird experimentation

1948) Adds the conditional back in front of glEnd, making it draw a line (for all the wrong reasons) again.



The student here is not at all understanding the pipeline, this is an excellent problem to analyse in detail.

1949) Removes the glEnd again.

1950) Removes the drawWalls call that was drawing the point, so now nothing is drawn.

1951) Adds drawWalls call back to mouse handler (into conditional which checks for wall drawing mode)

1952) Removes the call as well as the drawWalls function itself.

Also removes mouseX and mouseY vars

Basically starting from scratch

1954) Re-adds the drawWalls function, so far all it does is change color.

1956) Makes drawWalls function change line width

1957) Adds glBegin/glEnd (w/out vertices) to drawWalls function

1958) Adds vertices for the glBegin/End

1959) Adds call to the function (which does draw the triangle, but also doesn't change line width back)

1960) Changes line width back

1961) Here the real new approach (w. storage) starts.

Of course, with just one array will only be able to draw one wall.

1963) Inits var

1965) Initialises all corners to 0 at beginning of program

1966) Changes wall color from red to black

1967) Changes line width of wall

1968) Changes line width of wall

1969) Changes line width of wall

1970) Adds a (rather unnecessary) flush

1971) Makes it so that inside mouse function, a click with wall mode will add the x and y coordinates to the array.

However, these points aren't rendered yet.

1972) Removes the isDrawing call, which wasn't being deactivated and was causing problems.

1973) Instead of rendering the fixed points, renders the points from the array. Wall adding actually \*kinda\* works now.

1974) Changes wall color.

1975) Changes point size at wall loop, but not drawing points.

1976) Draws the corner points.

1977) Changes point size

1978) Changes point size

1979) Makes 2D array to allow creation of more rooms.

1980) Makes it so that if not in wall mode, will switch to next room, but this will switch too often (any time not in wall mode)

1981) Changes room corner setting to use 2D array

1982) Changes init to use 2D array

1983) Changes room render to use 2D array

1984) Changes room render to use 2D array

1985) Makes it less or equal, but wrong syntax (=<)

1986) Makes it not render just cnRow (row currently at) but all rows (all walls)

1987) Fixes syntax

1988) Fixes syntax.

However, ED problems.

Not resetting the rows or cols for array for walls.

1990) Correctly moves the glEnd clause when drawing corner points to match up with the glBegin clause.

1991) Tries removing the rendering of walls to see if that will fix crash.

1992) Adds it back in (unchanged)

1993) Fixes array access inside draw function. Now draws first wall correctly, but incorrect managing of row/col indices means that multiple rooms can't be added.

1994) BAD

1995) Fixes a pipeline bug that was causing the line width to be reset inside loop (so wall was thickness 1)

1996) Gets that it's something to do with not setting one of the vars correctly, tries always incrementing cnRow even when in wall mode (wrong)

1997) Wrong

1999) Realises that he needs to also reset cnCounter.

Working room add (multi-rooms also).

#### **9.7.6.1.3.3.8.2 MICHAEL\_A1.PI.8.2."Wall Drawing ED" [13]**

Name: Wall Drawing ED

Description: Event-Driven Program-Flow

Occurrences: 1979-1980, 1985-1986, 1988, 1990-1993, 1995-1999 : total = 14 (+)

Detailed Description:

b) (1979-1980, 1985-1986, 1988, 1990-1993, 1995-1999) (Event-driven) (13 changes, +)

#### **9.7.6.1.3.4 General Programming**

##### **9.7.6.1.3.4.1 MICHAEL\_A1.GP.1."= /==" [14]**

Name: = /==

Description: General Syntax (= / ==)

Occurrences: 521-534 : total = 14 (+)

Detailed Description:

2. (= /==) (521-534) (14, +)

: Uses = instead of ==

#### **9.7.6.1.3.4.2 MICHAEL\_A1.GP.2."Inefficient render" [24]**

Name: Inefficient render

Description: Inefficient Computer Science

Occurrences: 1614-1633, 1833-1836 : total = 24 (++)

Detailed Description:

5. (Inefficient render) (1614-1633, 1833-1836) (24, ++)

Pots are rendered, but renders 1000 each redraw, so very slow (noticeable time spent in draw). This problem is never fixed effectively. It is interesting, because it is another example of an obvious inefficiency in Computer Science algorithms.

#### **9.7.6.1.3.4.3 MICHAEL\_A1.GP.3."Bitwise operator" [12]**

Name: Bitwise operator

Description: General / C++ Syntax

Occurrences: 1641-1652 : total = 12 (+)

Detailed Description:

6. (Bitwise operator) (1641-1652) (12, +)

Trying to calc distance, but uses ^ operator, which is a bitwise operator in C++. Fixed at 1652

#### **9.7.6.1.3.4.4 MICHAEL\_A1.GP.4."2D array syntax" [11]**

Name: 2D array syntax

Description: General / Array syntax

Occurrences: 1978-1988 : total = 11 (+)

Detailed Description:

7. (array syntax) (1978-1988) (11, +)

Struggles w. 2D array syntax 354-380

### **9.7.6.1.3.5 OpenGL**

#### **9.7.6.1.3.5.1 MICHAEL\_A1.GL.1."Drawing lines" [14]**

Name: Drawing lines

Description: GL Syntax

Occurrences: 116-129 : total = 14 (+)

Detailed Description:

116, 118-120, 124, 127, 129

Student is trying to draw lines, but uses Point constructor instead of glVertex call so nothing is drawn. Tries to change line color (again, modifying non-problem in an attempt to fix non-detected problem in pipeline).

Also incorrectly uses GL\_LINE instead of GL\_LINES (which has no meaning in glBegin(...)) so nothing is drawn after vertex is used 124. Spatial tweaking is attempted before the problem is finally solved in 129.

#### **9.7.6.1.3.5.2 MICHAEL\_A1.GL.2."Color exp" [14]**

Name: Color exp

Description: Color

Occurrences: 179-180, 183-186, 197-204 : total = 14 (+)

Detailed Description:

Color experimenting for Gui fill and clear color

#### **9.7.6.1.3.5.3 MICHAEL\_A1.GL.3."Color tweak" [11]**

Name: Color tweak

Description: Color

Occurrences: 277-287 : total = 11 (+)

Detailed Description:

277, 285-287 (Move?)

Tweaks color for circle draw because circle is not drawing. The circle draw actually has two problems (formula is correct). Uses glVertex3f (277) to draw in x-z dimension instead of x-y (so won't appear on 2D projection) and doesn't include glFlush() (which would make it not redraw). So the color change is again an attempted fix for a different problem where no visual cues are present to identify the nature of the problem. (Show drawn pixel count or something?)

#### **9.7.6.1.3.5.4 MICHAEL\_A1.GL.4."Button color tweak" [11]**

Name: Button color tweak

Description: Color

Occurrences: 295-302, 332-334 : total = 11 (+)

Detailed Description:

Tweaks button color

### **9.7.6.1.4 Ida**

#### **9.7.6.1.4.1 Spatial**

##### **9.7.6.1.4.1.1 IDA\_A1.SP.1."Move Icon" [21]**

Name: Move Icon

Description: Spatial

Occurrences: 395-415 : total = 21 (++)

Detailed Description:

395-415 (20 Changes, ++) Move Icon

##### **9.7.6.1.4.1.2 IDA\_A1.SP.2."Rotate Icon" [41]**

Name: Rotate Icon

Description: Spatial

Occurrences: 418-458 : total = 41 (++++)

Detailed Description:

418-458 (25 Changes, ++) Rotate Icon (Mostly radian/degree related)

##### **9.7.6.1.4.1.3 IDA\_A1.SP.3."Object rotation" [23]**

Name: Object rotation

Description: Spatial

Occurrences: 765-787 : total = 23 (++)

Detailed Description:

5. (Object rotation) (765-787) (20 changes, +)

(20 Changes, 15 Relevant +), Object rotation: Implementation of object rotation using GL transforms. Initially the student has problems using matrix stacks to correctly limit the changes, but figures this out quickly and does not make this mistake again in A1 or A3. Student then uses the transformation and render at a point in the code where it will be overdrawn by the display function, and tries to fix

this problem by changing the rotation formula before discovering the true source of the problem. Also initially uses degrees instead of radians again. Rotation then works, but rotates around the origin. The student correctly uses translate calls to reposition the pivot, but makes several errors while doing so, indicating problems composing a series of transformations.

#### **9.7.6.1.4.2 General Programming**

##### **9.7.6.1.4.2.1 IDA\_A1.GP.1."Variable naming" [21]**

Name: Variable naming

Description: Semantic Oversight

Occurrences: 95-115 : total = 21 (++)

Detailed Description:

1. (Variable naming) (95-115) (21 changes, ++)

Uses incorrect variable, bad variable naming (\_r instead of r)

##### **9.7.6.1.4.2.2 IDA\_A1.GP.2."Switch statement" [19]**

Name: Switch statement

Description: Switch Statement / C++ syntax

Occurrences: 168-178, 183-184, 187-190, 193-194 : total = 19 (+)

Detailed Description:

2. (Switch statement) (171-192) (168-178, 183-184, 187-190, 193-194) (21 changes, ++)

Trying to implement ED code for actions but uses switch statement incorrectly.

##### **9.7.6.1.4.2.3 IDA\_A1.GP.3."Colour buttons" [18]**

Name: Colour buttons

Description:

Occurrences: 475-483, 485, 487-494 : total = 18 (+)

Detailed Description:

4. (Colour buttons) (475-483, 485, 487-494) (total: 18, +)

The student forgets to set a variable from input parameters in the constructor, resulting in the provided colour value not setting the icon's colour.

##### **9.7.6.1.4.2.4 IDA\_A1.GP.4."Static functions" [21]**

Name: Static functions

Description: C++ OO (static func, overriding)

Occurrences: 200-220 : total = 21 (++)

Detailed Description:

3. (Static functions) (200-220) (21 changes, ++)

While trying to implement button hit method, Problem using static functions, function overriding, method calling

#### **9.7.6.1.4.2.5 IDA\_A1.GP.5."Hit code and methods" [17]**

Name: Hit code and methods

Description: C++ OO

Occurrences: 505-512, 515, 517-524 : total = 17 (+)

Detailed Description:

5. (Hit code and methods)(505-512, 515, 517-524) (total: 17, +)

The student makes some general syntax errors and also fails to properly call an overridden function. Some students like this one initially struggled with OO concepts.

#### **9.7.6.1.4.2.6 IDA\_A1.GP.6."Loop error" [20]**

Name: Loop error

Description: General Computer Science Algo

Occurrences: 716-735 : total = 20 (++)

Detailed Description:

6. (Loop error) (716-735) (20 changes, ++)

In hit function for button, returns -1 inside loop instead of outside.

#### **9.7.6.1.4.2.7 IDA\_A1.GP.7."Child object rotate"[56]**

Name: Child object rotate

Description: Rounding Error

Occurrences: 1140-1152, 1168-1210 : total = 56 (+++++)

Detailed Description:

7. (Child object rotate) (1140-1152, 1168-1210) (54 Changes, +++++)

In trying to implement rotation of child objects around their parents, the student chooses to transform coordinates directly instead of using GL transforms. The student struggles with correctly placing the 'pivot'. Again, the problems the student faces seem to suggest that mentally compositing more complex spatial tasks is difficult, especially when mathematical functions are involved. A

rounding error is responsible for child objects shrinking over time; other students face similar rounding errors and this point and workarounds should be explained to students.

#### **9.7.6.1.4.3 Event-Driven**

##### **9.7.6.1.4.3.1 IDA\_A1.ED.1."Hit Code " [11]**

Name: Hit Code

Description: Screen-Window

Occurrences: 154-160, 162, 164, 166-167 : total = 11 (+)

Detailed Description:

1. (Hit Code ) (154-160, 162, 164, 166, 167) (total = 11, +)

The student forgets to convert from screen to window coordinates and tries to adjust various unrelated parts of the event-driven code before figuring out the problem.

##### **9.7.6.1.4.3.2 IDA\_A1.ED.2."Dynamic Object Storing" [19]**

Name: Dynamic Object Storing

Description: Draw Not Store

Occurrences: 226-244 : total = 19 (+)

Detailed Description:

Dynamic Object Storing (Drawing in Handler?) (226-244, 20 ++)

##### **9.7.6.1.4.3.3 IDA\_A1.ED.3."Logic Ops" [37]**

Name: Logic Ops

Description: Logic Ops / Pipeline Order

Occurrences: 304-306, 309, 315, 317-319, 322-327, 334, 339-342, 346-359, 361-364 : total = 37 (+++)

Detailed Description:

1. (Logic Ops) (304-306, 309, 315, 317-319, 322-327, 334, 339-342, 346-359, 361-363, 364) (total = 37, +++)

(Logic Ops) Logical Ops Rubber Shapes: Probably the most interesting ED segment in the assignment; the student struggles with logical operations, but misinterprets this as an ED error. The student frequently breaks ED code in an attempt to fix the problem. In the ED code, the student overwrites variables before they are accessed, loses track of what variable refers to what (306) in the ED code and resets variables in the wrong place. This highlights that complex ED code can stress a student's ability to maintain understanding of program flow.



#### **9.7.6.1.4.3.4 IDA\_A1.ED.4."Flower pot move" [15]**

Name: Flower pot move

Description: Event-Driven Program Flow

Occurrences: 992, 994-995, 998, 1002-1004, 1006, 1008, 1015, 1017-1020, 1022 : total = 15 (+)

Detailed Description:

4. (Flower pot move) (992, 994-995, 998, 1002-1004, 1006, 1008, 1015, 1017-1020, 1022) (total = 15, +), The student forgets to adjust the centre variable for flower pots, affecting parts of the code like drawing connectors that rely on the flower pot's centre.

#### **9.7.6.1.4.3.5 IDA\_A1.ED.5."Selection" [22]**

Name: Selection

Description: Event-Driven Program Flow

Occurrences: 590-592, 594-600, 602, 604-614 : total = 22 (++)

Detailed Description:

3. (Selection) (590-592, 594-600, 602, 604-613, 614) (total = 22, ++)

The student again shows a breakdown in understanding the program flow regarding what state the 'action' variable is in at a given time in the code, leading to deadlocks in which no objects can be selected and the incorrect setting of the action to none.

#### **9.7.6.1.4.4 OpenGL**

##### **9.7.6.1.4.4.1 IDA\_A1.GL.1." Line stipple" [31]**

Name: Line stipple

Description: Line Stipple

Occurrences: 1079-1109 : total = 31 (+++)

Detailed Description:

2. 1079-1126 (1079-1109) (48 changes, Line stipple +++) Student requires many changes getting a proper line stipple to work.

#### **9.7.6.1.5 Thomas**

##### **9.7.6.1.5.1 Spatial**

##### **9.7.6.1.5.1.1 THOMAS\_A1.SP.1."Button Spatial Modulus" [20]**

Name: Button Spatial Modulus

Description: Math

Occurrences: 137-149, 151, 153-156, 158-159 : total = 20 (++)

Detailed Description:

1. 137-149, 151, 153-156, 158-159 (20, ++)

Trying to implement buttons being positioned depending on a single variable (using modulus), so combination of general spatial and maths.

Gets it into place bit by bit.

Major: IIIII IIIII II (12)

Tweak: IIII (4)

#### **9.7.6.1.5.1.2 THOMAS\_A1.SP.2."Door Drawing" [10]**

Name: Door Drawing

Description: Spatial

Occurrences: 610-614, 622-623, 627-628 : total = 9 ()

Detailed Description:

5. 610-614, 622-623, 627-628 (10, +)

Door drawing. Draws a door made of a triangle and two rectangles for sides.

Fairly straight-forward, though accidentally switches height and width (fixes 622).

Major: IIIII III

Tweak:

#### **9.7.6.1.5.1.3 THOMAS\_A1.SP.3."Wall Move/Resize" [60]**

Name: Wall Move/Resize

Description: Spatial

Occurrences: 663-681, 716-724, 737-768 : total = 60 (+++++ (6))

Detailed Description:

663-681, 716-724, 737-768 (57, +++) Spatial

#### **9.7.6.1.5.1.4 THOMAS\_A1.SP.4."Rotation"**

Name: Rotation

##### **9.7.6.1.5.1.4.1 THOMAS\_A1.SP.4.1."Object rotate maths" [19]**

Name: Object rotate maths

Description: Math

Occurrences: 795, 801, 813-815, 817-827, 832-834 : total = 19 (+)

Detailed Description:

813-835 (23, ++) Initial spatial / maths

a. Spatial (795, 801, 813-815, 817-827, 832-834) (19 changes, +)

#### **9.7.6.1.5.1.4.2 THOMAS\_A1.SP.4.2."Object rotate angle Math Formula" [10]**

Name: Object rotate angle Math Formula

Description: Math

Occurrences: 838, 846, 848-849, 854-856, 870-872 : total = 10 (+)

Detailed Description:

835-872 (39, +++) Problem w. math formula for determining angle

b. Math/Formula (838, 846, 848-849, 854-856, 870-872) (10 changes, +)

#### **9.7.6.1.5.1.5 THOMAS\_A1.SP.5."Drawing Arrow" [20]**

Name: Drawing Arrow

Description: Spatial

Occurrences: 1025-1029, 1031-1045 : total = 20 (++)

Detailed Description:

12. 1025-1029, 1031-1045 (Drawing arrow) (20 changes, ++)

Working on an arrow image that can be faced into any compass direction (1025-1054) and using it for the move icon as well as the rotate icon

#### **9.7.6.1.5.1.6 THOMAS\_A1.SP.6."Move" [13]**

Name: Move

Description: Spatial

Occurrences: 1060-1072 : total = 13 (+)

Detailed Description:

14. Move (1060-1072, 13 +)

#### **9.7.6.1.5.1.7 THOMAS\_A1.SP.7."Color button spatial" [12]**

Name: Color button spatial

Description: Spatial

Occurrences: 1137, 1139, 1141, 1143-1146, 1152, 1154, 1156, 1160-1161 : total = 12 (+)

Detailed Description:

17. 1137, 1139, 1141, 1143-1146, 1152, 1154, 1156, 1160-1161 (11, Sp+)

Works on drawing color buttons.

Some errors placing button, and some tweaking on size and placement. Again, fairly simple but does take several changes.

Major: IIIII (5)

Tweak: IIIII I (6)

#### **9.7.6.1.5.1.8 THOMAS\_A1.SP.8."Wall Rotate" [13]**

Name: Wall Rotate

Description: Spatial

Occurrences: 961-963, 1192-1201 : total = 13 (+)

Detailed Description:

22. (961-963, 1192-1201) (13 changes, +) (Wall Rotate)

Attempts to implement wall rotate again, gives up

#### **9.7.6.1.5.2 General Programming**

##### **9.7.6.1.5.2.1 THOMAS\_A1.GP.1."= instead of ==" [10]**

Name: = instead of ==

Description: General Syntax (= / ==)

Occurrences: 85-94 : total = 10 (+)

Detailed Description:

3. 85-94 (Uses = instead of ==) (10, +)

In ED code, uses = instead of == in conditional, leading to the conditional statement not working properly.

##### **9.7.6.1.5.2.2 THOMAS\_A1.GP.2."Child-Parenting Algorithm" [114]**

Name: Child-Parenting Algorithm

Description: General Computer Science

Occurrences: 166, 170, 172-175, 177-181, 184-187, 189-191, 193-200, 202-203, 205-209, 211, 216-219, 221-227, 229-232, 234-235, 238-239, 241-245, 247, 249, 252-260, 263, 267, 269-270, 275-276, 278-284, 286-296, 298, 301, 303-307, 311-316, 318-325 : total = 114 (+++++ (11))

Detailed Description:

4. 166-327 (161, +++++(16))

166, 170, 172-175, 177-181, 184-187, 189-191, 193-200, 202-203, 205-209, 211, 216-219, 221-227, 229-232, 234-235, 238-239, 241-245, 247, 249, 252-260, 263, 267, 269-270, 275-276, 278-284, 286-296, 298, 301, 303-307, 311-316, 318-324, 325

This is programming supporting the adding of child objects to parent objects.

The student struggles because instead of passing and storing children by reference, he passes and stores them by value. This means that the child object stored in the parent is a copy, and since no copy constructor is provided it is a copy with empty variables.

Even the object stored in the vector is not the same as the object pushed back into the vector, as the vector too must create a copy of the object, which is why an approach attempting to use integer ids instead of passing the object also fails (id is set on the object variable, which does not affect the vector object).

The student spends the whole time debugging this problem, using cout statements to print object ids and vector data (size etc) as well as putting cout statements to verify that certain parts of the code have indeed run.

The problem is finally solved starting at 323 when the student uses object pointers instead, but the student spent a significant part of the assignment on this non-relevant problem. In this instance the complexity of C++ object passing was to blame.

#### **9.7.6.1.5.2.3 THOMAS\_A1.GP.3."= instead of == (2)" [14]**

Name: = instead of == (2)

Description: General Syntax (= / ==)

Occurrences: 521-534 : total = 14 (+)

Detailed Description:

1. 521-534 (14, +)

Uses = instead of ==, tries to adjust ED code to fix this.

#### **9.7.6.1.5.2.4 THOMAS\_A1.GP.4."Pointer syntax" [20]**

Name: Pointer syntax

Description: C++ syntax (pointer)

Occurrences: 520-524, 529-543 : total = 20 (++)

Detailed Description:

7. 520-524, 529-543 (10, +)

Has children store pointer to their parents.

Struggles w. pointer syntax.

#### **9.7.6.1.5.2.5 THOMAS\_A1.GP.5."Circular Parent-Child" [54]**

Name: Circular Parent-Child

Description: General Computer Science

Occurrences: 411, 422, 424-429, 431, 441-464, 528, 547-552, 555-560, 562-563, 565, 568, 573-576 :  
total = 54 (+++++)

Detailed Description:

8. (411, 422, 424-429, 431, 441-464 | 528, 547-552, 555-560, 562-563, 565, 568, 573-576)  
(33+21=54 changes, +++++)

From 411-464, first comes across the problem that circular parent-child relationships cause infinite recursion with visit functions. Initially tries some different algorithm approaches to detect cycles which don't work. Finally ends up using flags on individual diagram objects to ensure that recursive functions (such as draw) only visit objects once (then toggle flag) [where is this?].

From 528-576, while working on parent-child connectors, struggles with preventing circular parent-child relationships which cause recursive algorithms to recurse infinitely.

First stops objects being 're-parented' once they were parented (528), then tries to use a function to detect whether a candidate child is already a parent's child (555-568) but doesn't succeed (GP problems) so eventually settles on simply not allowing children to be parented to one parent.

#### **9.7.6.1.5.2.6 THOMAS\_A1.GP.6."Loss of Precision" [31]**

Name: Loss of Precision

Description: General Syntax

Occurrences: 877-878, 882-886, 888-894, 897-900, 902, 905-908, 911-916, 918-919 : total = 31 (+++)

Detailed Description:

873-916 (44, +++) Loss of precision

c. Loss of Precision (877-878, 882-886, 888-894, 897, 897-900, 902, 905-908, 911-916, 918-919) (31 changes,+++)

#### **9.7.6.1.5.2.7 THOMAS\_A1.GP.7."Unintentional Rounding" [43]**

Name: Unintentional Rounding

Description: General Syntax

Occurrences: 682-715, 728-736 : total = 43 (++++)

Detailed Description:

682-715, 728-736 (43, +++) Unintentional Rounding

#### **9.7.6.1.5.3 THOMAS\_A1.ED."Event-Driven"**

##### **9.7.6.1.5.3.1 THOMAS\_A1.ED.1."Not Updating Variable" [20]**

Name: Not Updating Variable

Description: Event-Driven Program-Flow

Occurrences: 917-936 : total = 20 (++)

Detailed Description:

917-936 (20, ++) Not updating variable (Program flow) (Is this duplicate?)

### **9.7.6.2 Assignment 3**

#### **9.7.6.2.1 Christopher**

##### **9.7.6.2.1.1 Spatial**

###### **9.7.6.2.1.1.1 CHRISTOPHER\_A3.SP.1."AvatarAssembly"**

Name: AvatarAssembly

###### **9.7.6.2.1.1.1.1 CHRISTOPHER\_A3.SP.1.1."Initial Avatar Assembly" [36]**

Name: Initial Avatar Assembly

Description: Spatial

Occurrences: 212-228, 258-260, 262-265, 267, 272, 274, 701-707, 709-710 : total = 36 (+++)

Detailed Description:

From 212-228, initial avatar assembly. The avatar is assembled incorrectly in the naïve manner. From 258-274, again attempting to assemble avatar using naive method, also runs into bug regarding algorithm for accessing different limbs (which makes it difficult to test whether avatar assembly is working). From 701-710, Tests assembly by manipulating limb rotation values and fixes assembly (still wrong overall).Major 32Tweak 4

###### **9.7.6.2.1.1.2 CHRISTOPHER\_A3.SP.2."Animation"**

Name: Animation

#### **9.7.6.2.1.1.2.1 CHRISTOPHER\_A3.SP.2.1."Pickup Anim" [24]**

Name: Pickup Anim

Description: Spatial

Occurrences: 946, 949-950, 953, 955, 957-962, 964, 966-968, 971, 974, 1191-1197 : total = 24 (++)

Detailed Description:

Working on pickup animation. Major: 946 953 955 957-962 964 966-968 971 974 1191-1192, 1194-1197  
Tweak: 949-950, 1193 ---Axis Changes: 5Bodypart changes: I (1)Direction Changes: 10Translate commenting: III (3)

#### **9.7.6.2.1.1.3 CHRISTOPHER\_A3.SP.3."OtherSpatial"**

Name: OtherSpatial

##### **9.7.6.2.1.1.3.1 CHRISTOPHER\_A3.SP.3.1."Early Experiment" [14]**

Name: Early Experiment

Description: Spatial

Occurrences: 138-139, 146-147, 149, 155-157, 165-166, 169-172 : total = 14 (+)

Detailed Description:

Early experimentation with transformations.

##### **9.7.6.2.1.1.3.2 CHRISTOPHER\_A3.SP.3.2."Room Creation" [29]**

Name: Room Creation

Description: Spatial

Occurrences: 408-412, 414, 419, 422-426, 429, 435-440, 442-444, 447-449, 457-460 : total = 29 (++)

Detailed Description:

Major: 17Tweak: 9From (408-412, 414), initial room creation using quads. Will go back and forth several times between quad and glutSolidCube approach. Indicates problem w. understanding difference between object interiors and exteriors, normal etc.From (419, 422-426, 429) attempt to create room using glutSolidCube, then attempts to use quads from (435-440), going back to attempting to use a cube from (442-444, 447-449) and finally tweaks the cube position from (457-460).

##### **9.7.6.2.1.1.3.3 CHRISTOPHER\_A3.SP.3.3."Walk In ViewDir" [25]**

Name: Walk In ViewDir

Description: Math

Occurrences: 623-634, 641-649, 1072-1074, 1077 : total = 25 (++)



Detailed Description:

Implementing walking in view direction. Broken assembly become apparent once more.

#### **9.7.6.2.1.1.3.4 CHRISTOPHER\_A3.SP.3.4."Buttons Pos" [14]**

Name: Buttons Pos

Description: Spatial

Occurrences: 845-852, 857-858, 860, 865-866, 872 : total = 14 (+)

Detailed Description:

Positioning buttons, minor errors.Major III (3)Minor IIIII IIII (9)

#### **9.7.6.2.1.1.3.5 CHRISTOPHER\_A3.SP.3.5."Pickup / Drop Object" [47]**

Name: Pickup / Drop Object

Description: Math

Occurrences: 999, 1010-1011, 1015-1017, 1020-1022, 1024, 1029-1031, 1036-1040, 1042, 1048-1050, 1053-1062, 1079-1080, 1082-1085, 1105-1106, 1120-1126 : total = 47 (++++)

Detailed Description:

From 999-1031, first tries to determine whether an object is close enough for pickup at (999), then uses cout statements to print out the avatar's coordinates at (1010-1011) to debug the formula.Modifying distance formula to attempt to determine distance to pickup object (wrong, using separate formulae for x and z), using debug statements to print out vals (look at role of debug statements!)From 1036-1062, attempting to drop object in front of man. At 1054 fixes from using man rotation (which is always 0) to torso rotation. From 1105-1126, trying to add scale to pickup/drop, fails. This is interesting, as this problem has to do with the fact that the assembly is incorrect (scale before? Rotate/move)(999-1031 Object close enough for pickup? 1036-1062 Drop in front of guy)Major: 37Minor: 7

#### **9.7.6.2.1.2 Animation**

##### **9.7.6.2.1.2.1 CHRISTOPHER\_A3.ANIMATION.1."Animation Algo" [18]**

Name: Animation Algo

Description: Anim

Occurrences: 464-465, 470, 495-504, 506-507, 511-513 : total = 18 (+)

Detailed Description:

464-465 First attempt at animation, using a loop to increment value (but no display call, so doesn't work) Creates first simple "anim" using glutTimerFunc (several changes after spent getting it to work, at 495, then adds required display call at 498), of the hand just flying forward along one axis.

### **9.7.6.2.1.3 General Programming**

#### **9.7.6.2.1.3.1 CHRISTOPHER\_A3.GP.1."Setting NULL / This keyword" [13]**

Name: Setting NULL / This keyword

Description: C++ Syntax

Occurrences: 21-33 : total = 13 (+)

Detailed Description:

Trying to set to NULL, struggling with NULL and this

#### **9.7.6.2.1.3.2 CHRISTOPHER\_A3.GP.2."SceneTree traversal" [121]**

Name: SceneTree traversal

Description: General Computer Science

Occurrences: 107-112, 115-120, 122-127, 130-136, 139-147, 149-152, 154-157, 161-177, 179-185, 189-192, 197-211, 295-305, 307-308, 310-318, 321-325, 561-569 : total = 121 (+++++ (12))

Detailed Description:

Implementing/Debugging getNodeByName method that is supposed to return named SceneTreeNode (recursive search algo) has bugs. From 295-325, working on limb retrieval function again, breaks it worse initially. Still not quite working at 326. Algorithm still not working properly at the end.

#### **9.7.6.2.1.3.3 CHRISTOPHER\_A3.GP.3."glutTimerFunc function pointer" [25]**

Name: glutTimerFunc function pointer

Description: C++ Syntax

Occurrences: 469, 471-481, 483-495 : total = 25 (++)

Detailed Description:

implementing anim function and using timer. Problem is proper syntax for using function pointer with glutTimerFunc. Function pointers (23 changes, ++)

#### **9.7.6.2.1.3.4 CHRISTOPHER\_A3.GP.4."C++ Pointer/Direct syntax" [12]**

Name: C++ Pointer/Direct syntax

Description: C++ Syntax

Occurrences: 884-895 : total = 12 (+)

Detailed Description:

Trying to retrieve class member's data, problem w. C++ pointer / direct member syntax.

#### **9.7.6.2.1.4 View**

##### **9.7.6.2.1.4.1 CHRISTOPHER\_A3.VIEW.1."FP Camera" [47]**

Name: FP Camera

Description: View

Occurrences: 608-615, 653-654, 712, 731-747, 975-993 : total = 47 (++++)

Detailed Description:

Implementing first-person view, minimap

Initially, tries to look at avatar position from "at" position, which is fixed (probably mixing up the eye/lookat). Fixes this and moves to an approach trying to look "ahead of" avatar (`gluLookAt(fromX,fromY,fromZ, fromX+2,fromY+2,fromZ+2, 0,1,0)`) which shows a misunderstanding of "ahead of". Also, the student's incorrect compositing of a scale transform when assembling the avatar means that the avatar's position as stored in the variables doesn't correspond to his coordinates on the screen.

From 731-747, FP View, adjusts view to take scale of avatar into consideration, adjusts view to provide proper FP. Semi-working (w.out head rotation) at 740. Tries to tweak the eye/lookAt to provide a proper "look-ahead", but of course that won't work.

From 975-993, working on incorporating spherical co-ords to look into the avatar direction.

##### **9.7.6.2.1.4.2 CHRISTOPHER\_A3.VIEW.2."TP Camera" [15]**

Name: TP Camera

Description: View

Occurrences: 616, 653, 655, 688-691, 713, 987-993 : total = 15 (+)

Detailed Description:

Works on topdown camera

##### **9.7.6.2.1.4.3 CHRISTOPHER\_A3.VIEW.3."View Spherical Coordinates" [14]**

Name: View Spherical Coordinates

Description: Math

Occurrences: 975, 977-978, 980, 982, 985-993 : total = 14 (+)

Detailed Description:

Figures out to use circle formula for view. Has to tweak math formula a little. Adjusts FP view to look into direction avatar is turned towards and also applies this to TP view.

### **9.7.6.2.2 John**

#### **9.7.6.2.2.1 General Programming**

##### **9.7.6.2.2.1.1 JOHN\_A3.GP.1."Static function pointer" [13]**

Name: Static function pointer

Description: C++ Syntax

Occurrences: 175-187 : total = 13 (+)

Detailed Description:

Struggles w. passing function pointer to glutIdleFunc, trying to pass class method (doesn't work)

##### **9.7.6.2.2.1.2 JOHN\_A3.GP.2."WaveFrontImporter" [12]**

Name: WaveFrontImporter

Description: WaveFrontImporter

Occurrences: 374-385 : total = 12 (+)

Detailed Description:

Experimenting w. using WaveFrontImporter

##### **9.7.6.2.2.1.3 JOHN\_A3.GP.3."Object-Oriented" [23]**

Name: Object-Oriented

Description: C++ OO

Occurrences: 568-569, 571-574, 576-578, 582-585, 587-588, 590, 593-599 : total = 23 (++)

Detailed Description:

Trying to implement child class, takes several changes to sort out problems of visibility and general syntax. (OO problems)

#### **9.7.6.2.2.2 View**

##### **9.7.6.2.2.2.1 JOHN\_A3.VIEW.1."Minimap" [37]**

Name: Minimap

Description: View

Occurrences: 52-54, 61-66, 68-79, 247-250, 257-258, 410-418, 450 : total = 37 (+++)

Detailed Description:

2. (Minimap) (52-54, 61-66, 68-79, 247-250, 257-258, 410-418, 450) (37 changes, +++)

From 52-54 doesn't load identity matrix before using new projection (52-54),

From, 61-79 working on minimap.

61 Uses wrong projection (ortho2D), 62 skewed projection and lookAt that won't look at anything, fixes lookat in 65 and fixes projection in 66, tweaks the lookat over several changes (including breaking lookAt several times), forgets to reset modelview matrix when drawing minimap background, resulting in it not being drawn properly and not being visible (69-76), and then also forgets to reset the projection matrix (76-79)

A lot of time also tweaking the view's eye position.

From 247-250, changes minimap projection from perspective to ortho, uses incorrect coordinates that produce a non-functional projection before correctly using the glOrtho call.

At 258, 450, tries to make minimap follow avatar, but uses incorrect gluLookAt call that looks from avatar to origin. Fixed at 450.

#### **9.7.6.2.2.2 JOHN\_A3.VIEW.2."FP view" [21]**

Name: FP view

Description: View

Occurrences: 210-213, 215-226, 681, 688-689, 700-701 : total = 21 (++)

Detailed Description:

From 197, 210-228, working on top-down and first-person view. Implements perspective view with angle that is too large (180 degrees) at 197, then spends 210-215 trying to fix it by adjusting lookat before fixing projection. Messes with angle quite a bit. Tweaks values from 215-226, including using 'wrong' values for the perspective angle and the perspective ratio. Sets up 3rd person view in 227 (works immediately) and tweaks in 228. From 413-418, tries to set up FP and 3rd Person views so that the camera gluLookAt follows the avatar's orientation, makes mistake with orienting the view correctly, then tweaks before fixing in 418. From 554-556, 614-616, tries to implement rotation around the avatar in 3rd person view (using glRotate), but centre of rotation is centre of the room instead of avatar. Fixes in 614-616 by instead of using the glguy's rotation for the view using a separate global rotation instead, which allows rotation of the view and rotation of the guy (though rotation of the guy will not now rotate the view, could have incremented both view and guy rotation when rotating guy) From 699-701, for some reason switches the sign on rotation applied to the view

for FP and TD views, which results in incorrect rotation and incorrect orientation of the avatar respectively, then changes back.

#### **9.7.6.2.2.3 JOHN\_A3.VIEW.3."TP View" [15]**

Name: TP View

Description: View

Occurrences: 227-228, 448-449, 555-557, 613-617, 691-692, 699 : total = 15 (+)

Detailed Description:

From 197, 210-228, working on top-down and first-person view. Implements perspective view with angle that is too large (180 degrees) at 197, then spends 210-215 trying to fix it by adjusting lookat before fixing projection. Messes with angle quite a bit. Tweaks values from 215-226, including using 'wrong' values for the perspective angle and the perspective ratio. Sets up 3rd person view in 227 (works immediately) and tweaks in 228. From 413-418, tries to set up FP and 3rd Person views so that the camera gluLookAt follows the avatar's orientation, makes mistake with orienting the view correctly, then tweaks before fixing in 418. From 554-556, 614-616, tries to implement rotation around the avatar in 3rd person view (using glRotate), but centre of rotation is centre of the room instead of avatar. Fixes in 614-616 by instead of using the glguy's rotation for the view using a separate global rotation instead, which allows rotation of the view and rotation of the guy (though rotation of the guy will not now rotate the view, could have incremented both view and guy rotation when rotating guy). From 699-701, for some reason switches the sign on rotation applied to the view for FP and TD views, which results in incorrect rotation and incorrect orientation of the avatar respectively, then changes back.

#### **9.7.6.2.2.3 Spatial**

##### **9.7.6.2.2.3.1 JOHN\_A3.SP.1."Scale/Transforms" [12]**

Name: Scale/Transforms

Description: Spatial

Occurrences: 362-373 : total = 12 (+)

Detailed Description:

(Mod: 364-365, 366-367, 369, 371-373)(Twe: 363, 368) Introduces a scaling factor for objects, but only scales objects in main scene, not in minimap. Then spends time tweaking scale factors to fix problem before finally fixing it in 368.

#### **9.7.6.2.2.3.2 JOHN\_A3.SP.2."Pickup Teapot anim " [14]**

Name: Pickup Teapot anim

Description: Spatial

Occurrences: 396-407, 439, 611 : total = 14 (+)

Detailed Description:

Adds code to animation modifying teapot position. Spends changes tweaking rate of teapot ascent to match avatar animation, Tweaks teapot y-translation (423-424), 11. (Swim anim) (451-467, 471, 474-549, 560-570) (104 changes, +++)From 451-517, 528-549, working on Swim animation. From 560-570 working on swim anim again (legs)Also fixes avatar assembly at 468-470.

#### **9.7.6.2.2.3.3 JOHN\_A3.SP.3."Furniture positioning, tweaking" [12]**

Name: Furniture positioning, tweaking

Description: Spatial

Occurrences: 651-661, 667 : total = 12 (+)

Detailed Description:

Moves around furniture. Several simple mistakes w. axes..

#### **9.7.6.2.2.3.4 JOHN\_A3.SP.4."Assembly"**

Name: Assembly

##### **9.7.6.2.2.3.4.1 JOHN\_A3.SP.4.1."Initial naive assembly" [26]**

Name: Initial naive assembly

Description: Spatial

Occurrences: 7-32 : total = 26 (++)

Detailed Description:

Assembles with one error (24), mostly tweaking. But the assembly is incorrect (naive), which will become apparent during animations.

##### **9.7.6.2.2.3.4.2 JOHN\_A3.SP.4.2."Final assembly" [33]**

Name: Final assembly

Description: Spatial

Occurrences: 119-123, 126, 130-151, 468-470, 472-473 : total = 33 (+++)

Detailed Description:

Fixes assembly for all body parts (any errors pop up later?) This assembly is not as straight-forward as primitive assembly. Most of the assembly is MajorSpatial, with only few periods of tweaking.(Mod: 130-137, 139-143, 149-151)(Twe: 138, 144, 146-148)

#### **9.7.6.2.2.3.5 JOHN\_A3.SP.5."Animation"**

Name: Animation

##### **9.7.6.2.2.3.5.1 JOHN\_A3.SP.5.1."Walk anim" [19]**

Name: Walk anim

Description: Spatial

Occurrences: 320-330, 332, 440, 442-447 : total = 19 (+)

Detailed Description:

From 442-447, Modifies walk animation, all changes w.out tweaks (7 direction mods, 2 axis mods)

##### **9.7.6.2.2.3.5.2 JOHN\_A3.SP.5.2."Pickup Anim" [99]**

Name: Pickup Anim

Description: Spatial

Occurrences: 158-163, 202-209, 259-272, 274-283, 285-318, 336-351, 389-395, 421-424 : total = 99  
(+++++ (9))

Detailed Description:

56 (Mod: 259-264, 271-282, 284-288, 290-299, 301-306, 308-309, 312, 317-318, 321-324, 326, 336-350)10 (Twe: 265, 267-270, 289, 327-330, 332, 351)Starts work on first animation, 157-161. Initially gets direction of transforms wrong in 157, then fixes and tweaks from 158-161.From 202-209 Working on pickup anim, makes one error in the direction of movement of the right lower arm (203) which is fixed in 205, the rest of changes are spent tweaking.Starts almost from scratch w. pickup anim, removing all but the first 'keyframes' (removes 4). Milestone at 271, 279, 323 (looking pretty good) DEBUG LINES from 313-316Will do some more work on pickup later (??-??)Long periods of modification with some short episodes of tweaking between. When breaking down into the different kind of changes, there are 31 axis changes (changing a transform from one axis to another, like rotating around y instead of x), 20 direction changes (rotating around y clockwise instead of counterclockwise) and 3 limb changes (changing an arm transform to a leg transform).The very high number of axis changes suggests the student has a very poor grasp of the orientation of transformation axes in space in this complex transformation.From 389-395 modifying and adding to the animation. At one change, forgets glutPostRedisplay call, then adds next change. No tweaks.



#### **9.7.6.2.3.5.3 JOHN\_A3.SP.5.3."Swim Anim" [120]**

Name: Swim Anim

Description: Spatial

Occurrences: 451-570 : total = 120 (++++ (12))

Detailed Description:

451-570

From 451-517, 528-549, working on Swim animation. From 560-570 working on swim anim again (legs)

Also fixes avatar assembly at 468-470.

(Mod: 451-456, 458-465, (Figures out assembly wrong), 472-481, 486, 488-493, 495-497, 501-502, 506-507, 518-523, 544-548, 560-563, 569-570)

(Twe: 466-467, 471, 483-484, 487, 498-500, 503-505, 508-517, 524, 528-534, 537-543, 549, 564-568)

20 Direction Changes, 17 Axis Changes, 10 Limb Change

#### **9.7.6.2.2.4 Animation**

##### **9.7.6.2.2.4.1 JOHN\_A3.ANIMATION.1."Animation Algorithm" [19]**

Name: Animation Algorithm

Description: Anim

Occurrences: 157, 164, 166, 168, 175, 187-199, 201 : total = 19 (+)

Detailed Description:

Initially using simple loop for animation (FAIL)Then tries to use busy loops, experimenting with various numbers of iterations (FAIL)From 186-196, In 189, tries to use glut time and sleep w. different values(FAIL)Also tries to use superfluous glutSwapBuffers()/glutPostRedisplay() calls to fixFixes in 197 by using glutIdleFunc, binding when anim starts and unbinding when ends.

#### **9.7.6.2.3 Michael**

##### **9.7.6.2.3.1 Spatial**

###### **9.7.6.2.3.1.1 MICHAEL\_A3.SP.1."Initial Assembly" [47]**

Name: Initial Assembly

Description:

Occurrences: 143-189 : total = 47 (++++)

Detailed Description:

Initial (naive) assembly.

Major: IIIII IIIII II (12)

Tweak: IIIII IIIII IIIII IIIII IIIII IIIII (30)

#### **9.7.6.2.3.1.2 MICHAEL\_A3.SP.2."Furniture Move" [12]**

Name: Furniture Move

Description: Spatial

Occurrences: 285-290, 292, 298, 304, 306-307, 315 : total = 12 (+)

Detailed Description:

4. (Furniture move) (285-290, 292, 298, 304, 306-307, 315) (12 changes, +)

From 281-317, programmatically adds and moves around furniture. Most changes here are actually dealing with the WaveFrontImporter, but there are several positioning changes (which are however spread fairly thin over the longish stretch).

Major: 7

Tweak: 5

#### **9.7.6.2.3.1.3 MICHAEL\_A3.SP.3."Partially Proper Assembly" [71]**

Name: Partially Proper Assembly

Description:

Occurrences: 318-338, 385-389, 403-405, 407-410, 416-417, 434, 436-439, 441, 446-447, 449-451, 465-475, 482, 485-487, 523-532 : total = 71 (+++++ (7))

Detailed Description:

From 318-338, works on avatar assembly again. Some parts (such as palm) are now semi-functional, though the point of rotation is still wrong.

The larger problem is still that he's not compositing all transformations together, so the palm transform is independent of all other transforms.

Involves both tweaking into position and several major changes.

Major: IIIII IIIII (10)

Tweak: IIIII IIIII (10)

From 385-389, tweaks left palm rotation point (no real improvement) and applies left palm transform structure to right palm.

Major: IIII (4)

Tweak: (0)

Works on avatar assembly again. Not succesful.

First, tries to implement semi-functional approach for foot. (403-412)

From 428-434 implements movement of the avatar by adding the avatar's x and z coordinates to each limbs' translate function instead of compositing transforms (terrible approach)

During 436-487 To rotate entire body, instead of compositing transforms, adds the same `glRotate` transforms to EACH BODY PART. This will of course not work correctly with any of the transforms for the individual body parts, so puts these calls into if/else clauses, meaning EITHER the entire body can be rotated OR individual body parts can be rotated. This approach is a road to nowhere (fast).

Since this mostly involves using spatial coordinates he's worked out previously, doesn't make many coordinate-based errors but it's still spatially wrong (wrong approach).

This is really where it's left at in terms of assembly.

Major: IIIII IIIII IIIII IIIII IIII (24)

Tweak: I (1)

From 523-532 tries to address assembly again by adding body part transforms (rotations) to the if clause that includes the whole-body transform.

The student actually works out how to structure the transforms to get the head rotating around the correct point (where neck meets torso in original construction) with this code:

```
glRotatef(bodyx, 1, 0, 0);
glRotatef(bodyy, 0, 1, 0);
glRotatef(bodyz, 0, 0, 1);
glTranslatef(0+movex,5.15,0+movez);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
gObjects.at(objectAt)->getObjectsByName()["Head"]->draw();
```

but still doesn't composite transforms so head will not be rotated along with torso (same for other limbs), only along with body transform (which is global since it's copied everywhere).

This is the last change to the construction of avatar.

Major: IIIII IIIII (10)

### 9.7.6.2.3.2 View

#### 9.7.6.2.3.2.1 MICHAEL\_A3.VIEW.1."View" [35]

Name: View

Description:

Occurrences: 113, 115, 117-118, 121-126, 191-194, 204-205, 231-243, 414-418, 425 : total = 35 (+++)

Detailed Description:

Working on moving view with avatar, changes eye position but not lookAt (113,115), then experiments with fixed values for eye position from 121-126 (this indicates the need for better mechanisms for tweaking and/or display of variables such as eye coords, since the keys did not provide that info and were thus apparently too confusing to use).

From 191-194 again experiments w. fixed values for eye coord of view.

From 204-205 experiments by moving eye using only one coordinate (x) instead of all three.

From 231-243 trying to create a view using head position (so probably a first-person view), experiments w. tweaking eye position.

From 235-242 uses head coord for eye position only, then tries to use it for lookAt as well, trying to "look ahead" by adding to the head x and z, but this of course means it's only possible to look into one direction.

Major: IIII I

Tweak: IIII

From 414-418, 425 Implements the view's lookAt component being controlled through key-set variables. This isn't really a good first-person or third-person view (as it doesn't follow person's facing) but it's as close as the student gets.

Major: III

Tweak:

### **9.7.6.2.3.3 General Programming**

#### **9.7.6.2.3.3.1 MICHAEL\_A3.GP.1."Inefficient Limb Selection" [23]**

Name: Inefficient Limb Selection

Description:

Occurrences: 340-342, 344-348, 352, 355-356, 361-362, 365-374 : total = 23 (++)

Detailed Description:

340-342, 344-348, 352, 355-356, 361-362, 365-374, 380, 381-382, 384, 391-395, 399-403, 406 (39 ++, no real problems just a LOT of implementing, this one will get recognised as a problem. Could code as 'inefficiency/primitive approach')

Implements controls for switching between limbs for rotation.

Approach is very primitive (boolean value for EVERY limb, when limb is switched sets last to FALSE and current to TRUE)

Implementing limb selection; instead of using a single variable and assigning each limb a number, uses different boolean for each limb... bad approach and requires a lot of debugging. (Kind of duplicated from the ED, probably should move the ED here)

#### **9.7.6.2.3.3.2 MICHAEL\_A3.GP.2."Inefficient Limb Rotation" [28]**

Name: Inefficient Limb Rotation

Description:

Occurrences: 448, 452, 454-455, 457, 460, 463, 466, 471, 500-505, 507-511, 515-522 : total = 28 (++)

Detailed Description:

3. 448, 452, 454-455, 457, 460, 463, 466, 471, 500-505, 507-511, 515-522 (28, ++)

ED code to support the rotation of limbs and body, and the separate rotation of either a single limb and the whole body. No problems but INEFFICIENT.

#### **9.7.6.2.3.4 Animation**

##### **9.7.6.2.3.4.1 MICHAEL\_A3.ANIMATION.1."Anim Algo" [60]**

Name: Anim Algo

Description:

Occurrences: 539-540, 542-584, 587-601 : total = 60 (+++++ (6))

Detailed Description:

1. 539-540, 542-567, 568-577, 578-584, 587-601 (50, +++++)

The student attempts to implement animations, but uses a very poor approach that ends up not working. Uses a frame-based approach, but by using a timer and glutPostRedisplay the last frame will probably be overwritten by the next (most frames will be skipped), and since there are only 4 (400ms for entire anim) all of them will probably be skipped, nothing animated.

This is in addition to an initial error where the animation is always stopped after the first animation step.

In the end, never resolves the problem of properly timing the anim and actually rendering the anim steps, so the "animation" is only active while a button is pressed and runs at a very high speed.

#### **9.7.6.2.3.5 Lighting**

##### **9.7.6.2.3.5.1 MICHAEL\_A3.LIGHTING.1."Lighting Attenuation" [60]**

Name: Lighting Attenuation

Description:

Occurrences: 8-19, 21, 23-25, 50-53, 55-69, 79-81, 83-99, 109-111 : total = 58 (+++++)

Detailed Description:

1. 8-14, 15-19, 21, 23-25, 50-53, 55-61, 61-69, 79-81, 83-99, 109-111 (60,+++++)

From 8-25, Starts experimenting w. lighting. Too dark due to very large attenuation. Tweaks position to fix it, but uses very large values for light position which makes the scene very dark and the position of the light almost impossible to determine (8-14).

From 50-111, again modifying light position, again several times using values that are far too large.

Attenuation remains unchanged. Also changes color of light.

#### **9.7.6.2.3.5.2 MICHAEL\_A3.LIGHTING.2."Final Lighting" [19]**

Name: Final Lighting

Description:

Occurrences: 210-228 : total = 19 (+)

Detailed Description:

2. 210-229 (20, ++)

Finally addresses attenuation (210-220) and creates a fairly decent lighting effect.

### **9.7.6.2.4 Thomas**

#### **9.7.6.2.4.1 Spatial**

##### **9.7.6.2.4.1.1 THOMAS\_A3.SP.1."Initial Anim Experiment" [10]**

Name: Initial Anim Experiment

Description: Spatial

Occurrences: 297-299, 302-306, 309-310 : total = 10 (+)

Detailed Description:

At first, works on a walk animation (297-309) but realises that the construction is wrong (naive) so starts working on construction again.

##### **9.7.6.2.4.1.2 THOMAS\_A3.SP.2."Moving Avatar"**

Name: Moving Avatar

##### **9.7.6.2.4.1.2.1 THOMAS\_A3.SP.2.1."Angle Conversion" [12]**

Name: Angle Conversion

Description: Math

Occurrences: 457-458, 460-461, 472-478, 481 : total = 12 (+)

Detailed Description:

Math (angle conversion) 457-458, 460-461, 472-478, 481 (12, Math+)

#### **9.7.6.2.4.1.2.2 THOMAS\_A3.SP.2.2."Avatar Movement" [36]**

Name: Avatar Movement

Description: Math

Occurrences: 415, 417, 423, 427-428, 430-431, 436, 444-445, 447-450, 452-456, 459, 461-464, 466-468, 470-474, 476-478, 480 : total = 36 (+++)

Detailed Description:

Summary:

423-425 – Trying to change avatar's x,y position via naïve rotate

426-436 - Trying to move avatar forward by using animation's timer as glTranslate param

437-481 - Implements walking w. polar coordinates (works great)

Working on moving avatar while animating the walking anim 415-436), then works on moving avatar in view direction.

The initial approach is wrong, because it doesn't keep track of avatar location, instead merely translating it along a fixed axis. This together with a rotation can make the avatar walk forward in a given direction, but doesn't allow for other movement.

After 444, keeps track of the avatar position (correct approach) and tries to move avatar into view direction (good).

He uses the correct formula (adding cos/sin of direction to X/Y coord) but initially forgets to convert from degree to radian for C++ math functions., an error that he addresses at 456, but there are two more errors.

But then when adding the transformations 444-447 to move the body to the correct position, gets the order wrong (rotate-translate instead of translate-rotate)

After printing out angle values, apparently thinks that negative angles are to blame so introduces code to convert negative angles to positive ones (458). But since this isn't the fix, problem remains.

Then, changes the correct angle-conversion code to incorrect code at 461 (angle = angle \* -1 if neg, shows basic poor knowledge of angles) which means that even after he fixes the incorrect order of transformations at 466 the result is still wrong.

He returns the transformations to their correct order again at 474, then finally fixes the code for converting negative to positive angles at 481 from using `*-1` to adding 360 degrees, after which movement in view directions works.

Tweak: IIII (4)

Sign change: IIII II (7)

Pipe change: IIII I (6)

Name: Room Construction

Occurrences: 645-648, 650-651, 653-662, 664-674 : total = 27 (++)

Working on room walls. Starts using a good approach (using the exterior of scaled cubes, instead of the interior) immediately.

Major: IIIII IIIII IIII (14)

Dime change: IIIII III (8)

Limb change:

Pipe change:

Name: Carrying Lamp

Occurrences: 732-733, 735-740, 742-760, 766-770, 802, 808, 817-818 : total = 36 (+++)

Works on carrying and moving the lamp with the pickup anim.



From 732-740 works on attaching the lamp to the avatar's palm transform to carry it (good).  
 Then, from 742-761 works on dropping the lamp in front of the avatar, at first by using constant integers (which will not drop it "in front" of the avatar) then by using the circle formula using the avatar's direction until it works very well.  
 Finally gets there largely through experimentation of switching signs, trying to multiply values, then finally adding a fixed value to the circle (cos/sin) values.  $\text{lampX} = \text{currentPosX} - 4 + \cos(\text{direction} * 0.0174532925) * 2$ ;  $\text{lampY} = \text{currentPosY} - 2 + \sin(\text{direction} * 0.0174532925) * 2$ ;  
 Finally, tries to implement a check to pick up the lamp only when it's in front of avatar (using atan2) and close enough (using distance formula). Fascinatingly, gets the distance formula wrong twice before giving up. The incorrect implementations are:  
 $(\text{lampX} * \text{lampX} - \text{currentPosX} * \text{currentPosX}) + (\text{lampY} * \text{lampY} - \text{currentPosY} * \text{currentPosY})$   
 and  
 $\text{sqrt}((\text{lampX} * \text{lampX} - \text{currentPosX} * \text{currentPosX}) + (\text{lampY} * \text{lampY} - \text{currentPosY} * \text{currentPosY}))$   
 Major: IIIII IIIII IIIII IIIII IIIII (25)  
 Tweak: IIIII II (7)

#### 9.7.6.2.4.1.5 THOMAS\_A3.SP.5."Assembly"

Name: Assembly

##### 9.7.6.2.4.1.5.1 THOMAS\_A3.SP.5.1."Naive Assembly" [39]

Name: Naive Assembly

Description: Spatial

Occurrences: 8-11, 13-17, 20-27, 29-46, 51-53, 122 : total = 39 (+++)

Detailed Description:

Initial (naive) assembly. Constructs from 8-51, then tries rotating arm/leg 34, 38, 52-53 (which rotates around wrong pivot, of course). (Does he get it yet?)

Puts in rotations for limbs from 122-135, doesn't realize/fix naïve construction yet.

Major: IIIII IIIII IIIII IIIII IIIII IIIII I (31)

Tweak: IIIII (5)

Dime change: IIIII IIIII IIIII (15)

Sign change: IIIII II (7)

Limb change:

#### **9.7.6.2.4.1.5.2 THOMAS\_A3.SP.5.2."Proper Assembly" [55]**

Name: Proper Assembly

Description: Spatial

Occurrences: 311-332, 334-336, 338-367 : total = 55 (+++++)

Detailed Description:

Working on construction. At 332, manages to get rotation of leg correct (basic approach found).

During, frequently applies (and removes)(fixed) rotations to see whether he's getting it right.

Has a pretty solid construction at 365, and fixes up the walking anim at 366 and it's looking good.

Major: IIIII IIIII IIIII IIIII IIIII IIIII IIIII III

Tweak: IIIII IIIII

Dime change: IIIII IIIII IIIII IIIII I

Sign change: IIIII II

Limb change:

Pipe change: IIIII III

#### **9.7.6.2.4.1.6 THOMAS\_A3.SP.6. "Animations"**

##### **9.7.6.2.4.1.6.1 THOMAS\_A3.SP.6.1."Walk Animation" [36]**

Name: Walk Animation

Description: Spatial

Occurrences: 366-370, 375-377, 380-392, 400-411, 502-504 : total = 36 (+++)

Detailed Description:

At first, works on a walk animation (297-309) but realises that the construction is wrong (naive) so starts working on construction again.

From 502-504 tweaking walk anim again (tweaking lower arm pos).

Working on arm movement for the walking animation.

Ends up with a plain but functional walking anim.

Major: IIIII IIIII IIIII IIIII I (21)

Tweak: IIIII I (9)

Dime change: IIIII IIIII IIIII I (16)

Sign change: IIIII (5)

Limb change:

Pipe change:

##### **9.7.6.2.4.1.6.2 THOMAS\_A3.SP.6.2."Pickup Animation" [21]**

Name: Pickup Animation

Description: Spatial

Occurrences: 678-686, 688-690, 693-701 : total = 21 (++)

Detailed Description:

Works on the second animations.

Major: IIIII IIIII IIIII (15)

Tweak: IIIII III (8)

Dime change: IIIII (5)

Sign change: II (2)

Limb change:

Pipe change:

#### **9.7.6.2.4.1.6.3 THOMAS\_A3.SP.6.3."Waving animation" [20]**

Name: Waving animation

Description: Spatial

Occurrences: 702-721 : total = 20 (++)

Detailed Description:

Works on the second and third animations.

Major: IIIII IIIII IIIII I (16)

Tweak: III (3)

#### **9.7.6.2.4.2 Animation**

##### **9.7.6.2.4.2.1 THOMAS\_A3.ANIMATION.1."Animation Algorithm" [46]**

Name: Animation Algorithm

Description: Animation

Occurrences: 149-152, 166-171, 173, 175, 187, 197-200, 203, 207, 232-240, 243-244, 246, 250-251, 253-254, 257, 290, 292-295, 297-301 : total = 46 (++++)

Detailed Description:

Initially adds one animation function for every limb, later (183) rationalises that to use only one animation function that takes a param that identifies the limb.

Initially (166-170) tries a very naïve approach in which the 'animation' simply consists of a static `glRotate` call (with no variables), which does nothing but rotate the avatar initially.

The student then (171-180) implements the animation algorithm used throughout the assignment. It uses an incrementing variable that stores the current frame. This variable is incremented in a function that is run each time the display function is called.

The student then implements (200-232) a data structure (he calls "frame") for storing the rotation of each limb.

Each part of the animation has one 'frame' for each body part, with three values that specify the x,y,z rotation for the bodypart. The student then spends from 233-254 experimenting with the animation algorithm. It doesn't work correctly, as every call to the anim function increments the animation variable, meaning that a single 'frame' will increment the variable once for each limb. He figures this out and moves the increment to a separate function, and creates another function to reset the animation variable (253).

At this point the animation algorithm works, but it doesn't have a timing component, so it only executes when some event (such as a mouse or keyboard event) triggers a call to the display function.

From 290-301 works on creating some way of timing the animation. At 300-301 introduces a busy loop into the display function to "time"/slow-down the animation. This is a poor approach, but it works well enough that the student never chooses to go to the (correct) timer function approach.

Overall, the student uses a unique animation algorithm. When analysing his development of animations, it seems this approach may be superior (he develops them rapidly) quite possibly because the use of 'frames' allows the student to 'see' the position of each limb in the frame, instead of having to keep the details of the animation in his head.

#### **9.7.6.2.4.3 View**

##### **9.7.6.2.4.3.1 THOMAS\_A3.VIEW.1."Third-Person Camera" [21]**

Name: Third-Person Camera

Description: View

Occurrences: 498-499, 509-522, 524, 527-528, 531-532 : total = 21 (++)

Detailed Description:

Starts with a 3rd-person view and very quickly uses circle formula to orient it, but at first keeps the lookAt at the origin (509-513). From 516-522 moves to the correct approach of setting the lookAt to be the avatar position and using the circle formula to position the eye up and behind the avatar, but requires many changes as he at first gets sin and cos mixed up and tries changing the sign of the direction variable instead (wrong), before correctly switching sin and cos at 520.

From 523, works on 'orbital' view which is supposed to rotate around the avatar and allow for zooming in and out.

The zoomable view is implemented by 527 with no real errors. The student then implements the 'orbit' part from 527-532 by introducing a new CAMDIR variable to be added to the direction in the lookAt:

```
gluLookAt(currentPosX-  
cos(camDIR*0.0174532925+direction*0.0174532925)*at*camIN,at*camIN,currentPosY-  
sin(camDIR*0.0174532925+direction*0.0174532925)*at*camIN , currentPosX,4,currentPosY, 0,1,0);
```

#### **9.7.6.2.4.3.2 THOMAS\_A3.VIEW.2."Ortho/Top-down Camera" [20]**

Name: Ortho/Top-down Camera

Description: View

Occurrences: 542-561 : total = 20 (++)

Detailed Description:

From 542 works on the minimap. Initially uses gluOrtho from 542-546 but incorrectly uses gluLookAt with the projection so nothing is visible. Switches to gluOrtho2D (done at 551) which produces a 2D view of the avatar from the side (instead of the top), then tweaks the dimensions of the projection and goes back to using glOrtho (554).

The student then tries tweaking the projection by changing the z-axis size (possibly to try to make it look top-down) which doesn't do anything because the scene is being looked on by the x-y plane. The student then rotates the scene (556) after which a skewed top-down 2D view of the scene is visible. The student then makes the projection's x-y-z size equal again and tweaks it, producing a functional top-down view (but it is never implemented as a mini-map).

#### **9.7.6.2.4.4 General Programming**

##### **9.7.6.2.4.4.1 THOMAS\_A3.GP.1."Importer Problems" [39]**

Name: Importer Problems

Description: WaveFrontImporter

Occurrences: 592-630 : total = 39 (+++)

Detailed Description:

works on importing the lamp, struggles w. syntax of my importer

##### **9.7.6.2.4.4.2 THOMAS\_A3.GP.2."Bracketing" [8]**

Name: Bracketing

Description: General Syntax (Bracketing)

Occurrences: 792-794, 796-799, 801 : total = 8 ( )

Detailed Description:

4. 792-794, 796-799, 801 (10, Bracketing+)

Whilst trying to implement the lamp-distance formula, has some problems with bracketing.

#### **9.7.6.2.4.4.3 THOMAS\_A3.GP.3."Accidental Octal" [14]**

Name: Accidental Octal

Description: General Syntax (AccidentalOctal)

Occurrences: 366-379 : total = 14 (+)

Detailed Description:

5. 366-379 (14, AccidentalOctal+)

Realises that by using 000 or 00 notation, is using non-decimal number systems.

#### **9.7.6.2.4.5 Event-Driven**

##### **9.7.6.2.4.5.1 THOMAS\_A3.ED.1."Forgets hit code impl." [10]**

Name: Forgets hit code impl.

Description: Oversight

Occurrences: 259-268 : total = 10 (+)

Detailed Description:

Implementing the first button, forgets to implement hit code (so function always returns false)

##### **9.7.6.2.4.5.2 THOMAS\_A3.ED.2."Forgets screen-window conversion" [21]**

Name: Forgets screen-window conversion

Description: Screen-Window

Occurrences: 268-288 : total = 21 (++)

Detailed Description:

Implementing the first button, fairly straightforward but forgets to convert screen->window. Tries to fix by modifying ED code and hit code before finally converting at 287.

#### **9.7.6.2.4.6 Lighting**

#### **9.7.6.2.4.6.1 THOMAS\_A3.LIGHTING.1."Simple Lighting" [13]**

Name: Simple Lighting

Description: Lighting

Occurrences: 563-571, 578-581 : total = 13 (+)

Detailed Description:

Adds a light, at first uses only specular then uses only diffuse and ambient light color.

The student is happy with the simple lighting he achieves.

#### **9.7.6.2.4.7 GL**

##### **9.7.6.2.4.7.1 THOMAS\_A3.GL.1."Avatar movement keys" [10]**

Name: Avatar movement keys

Description: GL Syntax

Occurrences: 83-86, 88-93 : total = 10 (+)

Detailed Description:

Tries to implement cam movement via arrow keys, but uses keyboard handler instead of special handler, and forgets glutPostRedisplay.

Finally moves the code to the processSpecialKeys function in 92.

#### **9.7.6.2.5 Ida**

##### **9.7.6.2.5.1 Spatial**

###### **9.7.6.2.5.1.1 IDA\_A3.SP.1."Viewport" [17]**

Name: Viewport

Description: Spatial

Occurrences: 422-426, 428-439 : total = 17 (+)

Detailed Description:

5. (Viewport) (422-426, 428-439) (17 total changes, +) The student adjusts the size of the viewport, and as a result has to modify the viewport border line. The viewport is adjusted from 422-426, while the adjustment of the line takes from 427-439. The initial viewport adjustment is almost entirely tweaking, whereas the first part of the border adjustment includes real spatial errors (428-433). The remainder of changes from 434-439 are tweaks. This segment requires many changes for a fairly simple problem; it consists mostly of tweaking to achieve pixel-perfection.

###### **9.7.6.2.5.1.2 IDA\_A3.SP.2."GUI Background" [16]**

Name: GUI Background

Description: Spatial

Occurrences: 310-325 : total = 16 (+)

Detailed Description:

4. (The GUI Background) (310-325) (15 total changes, +) The GUI background is drawn from 310-325. Considering the simplicity of the task (2D rectangles framing the view window) it is remarkable how long it takes the student to complete. The reason becomes clearer when analysing the graph. Only changes 310-313 are modification changes, the rest are tweak changes to precisely position the rectangles. Pixel-perfect positioning is apparently not easy in this case, though the task is trivial. (Note: this is a case where simple grids/visual aids may be especially helpful).

#### **9.7.6.2.5.1.3 IDA\_A3.SP.3."Assembly"**

Name: Assembly

##### **9.7.6.2.5.1.3.1 IDA\_A3.SP.3.1."Naive Assembly" [24]**

Name: Naive Assembly

Description: Spatial

Occurrences: 47-69, 78 : total = 24 (++)

Detailed Description:

1. Assembly (78 for assembly, 11 for data structures) This task involves building the GLGuy by assembling his limbs, then using the GLGuy to produce three animations.

Data structures for assembly are put in place from 15-46, and the first attempt at assembly occurs from 47-78. However, while the initial assembly looks correct, it breaks when limbs are rotated as transformations do not properly position the rotate point at the joint (a mistake almost all students make).

This becomes apparent from 79-83, when the student implements a simple animation using the `glutIdleFunc`. As the arms are lifted, they rotate about their own centres instead of the joints.

The student fixes most of these assembly problems (84-131) and implements keyboard actions to rotate limbs to test the assembly.

Some limbs are still incorrectly assembled.

Until change 102, the student incorrectly assembles the `glGuy`, while from 102-131 the student applies the successful

assembly method learned from 84-102 to the rest of the assembly.

- Naïve assembly (47-78, 32 changes +++)



#### **9.7.6.2.5.1.3.2 IDA\_A3.SP.3.2."Proper assembly" [28]**

Name: Proper assembly

Description: Spatial

Occurrences: 84-91, 95-103, 121-131 : total = 28 (++)

Detailed Description:

- Proper assembly (84-131) (84-103, 121-131), 32 changes +++

#### **9.7.6.2.5.1.4 IDA\_A3.SP.4. "Animation"**

##### **9.7.6.2.5.1.4.1 IDA\_A3.SP.4.1."The Walk Animation" [66]**

Name: The Walk Animation

Description: Spatial

Occurrences: 144-150, 153-157, 160-183, 185-203, 205-215 : total = 66 (+++++ (6))

Detailed Description:

2. The Walk Animation (68 total changes) The student creates the first working animation (the walk animation) from 132-216, fixing an assembly error in the overall positioning (which the student would not have noticed when just moving limbs) of GLGuy in 143. Analysing the error/problem graph for spatial actions for this interval shows periods of problem (incorrect animation) intermingled with periods without problem, which represent tweaking of values. This suggests that the effort in producing the animation is both due to the student's difficulty correctly visualising the spatial transformation direction and due to the student's difficulty approximating the magnitude of spatial transformations. Mathematical operations such as 186 seem to present special difficulty to visualisation, usually producing extremely unintended results initially. The pattern of modification-tweak-modification (MTM) is interesting, and seems present in most spatial programming (is it in other programming?). Most changes during the modification phase are themselves errors; this indicates the modification phase changes are not trivial and present a significant challenge to the student. If this were not the case, the student could move straight from problem to solution without attempting so many incorrect solution steps.

Walking animation (144-150, 153-157, 160-215) (68 changes, +++)

##### **9.7.6.2.5.1.4.2 IDA\_A3.SP.4.2."The Pickup/Drop Animation"**

Name: The Pickup/Drop Animation

#### **9.7.6.2.5.1.4.2.1 IDA\_A3.SP.4.2.1."Pickup Animation" [79]**

Name: Pickup Animation

Description: Spatial

Occurrences: 271-297, 326-362, 364-378 : total = 79 (++++ (7))

Detailed Description:

Pickup Animation (271-297, 326-362, 364-378) 79 changes, ++++

#### **9.7.6.2.5.1.4.2.2 IDA\_A3.SP.4.2.2."Teapot Animation" [14]**

Name: Teapot Animation

Description: Spatial

Occurrences: 551-564 : total = 14 (+)

Detailed Description:

Teapot Animation (551-564) 13 changes +

#### **9.7.6.2.5.1.4.3 IDA\_A3.SP.4.3."Star Jump" [30]**

Name: Star Jump

Description: Spatial

Occurrences: 415-416, 441-442, 455-471, 473-478, 509-511 : total = 30 (+++)

Detailed Description:

6. (The Star Jump) (415-416, 441-442, 455-471, 473-478, 509-511) (30 total changes, +++)

Initial work including keyboard binding for the animation key are carried out from 414-417. The bulk of the work for this animation is carried out from 441-478. From 441-454 the student works on the animation mechanism for the animation (coded animation), with spatial work on the animation occurring from 455-478 (visible on the graph). In coding this animation, the student does not go through MTM cycles; the entire stretch involves modification, with only a single tweaking change. Keyboard events to trigger the star jump are modified from 450-454.

Tweaks the anim again from 509-511.

### **9.7.6.2.5.2 Lighting**

#### **9.7.6.2.5.2.1 IDA\_A3.LIGHTING.1."Lighting" [26]**

Name: Lighting

Description: Lighting

Occurrences: 219-221, 226, 228-237, 239-240, 243-246, 248-250, 252, 259, 270 : total = 26 (++)

Detailed Description:

219-221, 226, 228-237, 239-240, 243-246, 248-250, 252, 259, 270

#### **9.7.6.2.5.2.2 IDA\_A3.LIGHTING.2."Lighting 2" [21]**

Name: Lighting 2

Description: Lighting

Occurrences: 569-572, 646-649, 660-661, 666, 671-680 : total = 21 (++)

Detailed Description:

569-572, 646-649, 660-661, 666, 671-680

#### **9.7.6.2.5.3 General Programming**

##### **9.7.6.2.5.3.1 IDA\_A3.GP.1."Passing Light W. Array" [11]**

Name: Passing Light W. Array

Description: C++ Syntax

Occurrences: 251-259, 262-263 : total = 11 (+)

Detailed Description:

3. Passing light position to GL function through array 253-265 (251-259, 262-263) (11 Changes, +) Creating an array of the correct format to pass to the glLightfv function.

##### **9.7.6.2.5.3.2 IDA\_A3.GP.2."WaveFront importer" [21]**

Name: WaveFront importer

Description: WaveFrontImporter

Occurrences: 485, 488-501, 503-508 : total = 21 (++)

Detailed Description:

4. Using Importer for models 485-508 (24 Changes, ++) Figures out code to allow importing of models. Initially, unsure of how to import more models, resulting in models not being displayed.

#### **9.7.6.2.5.4 View**

##### **9.7.6.2.5.4.1 IDA\_A3.VIEW.1."View" [17]**

Name: View

Description: View

Occurrences: 390-400, 605-610 : total = 17 (+)

Detailed Description:

1. (Implementing Views) (390-400, 605-610) (17 Changes, +) Implements the views. Immediately uses sin/cos (spherical coords) approach for first-person view, and correct coords for third-person view. Gets the sign on sin/cos wrong way around, but most of the changes are tweaks, positioning the view into place. Implements view correctly despite the rotation not being applied correctly yet to avatar (resulting in view not looking from correct place).

#### **9.7.6.2.5.5 Animation**

##### **9.7.6.2.5.5.1 IDA\_A3.ANIMATION.1."Animation Algo" [15]**

Name: Animation Algo

Description: AnimAlgo

Occurrences: 79-83, 132-141 : total = 15 (+)

Detailed Description:

1. (Animation Algorithm) (79-83, 132-141) (12 changes, +)

The initial attempt at creating an animation happens at 79; use of the idle func rather than timers is not optimal but functional. However, when the animation runs it becomes clear the arms (and all other limbs) are not rotating correctly.

A working animation algorithm is implemented very quickly. Even the first change is largely correct, using a function to be used with glutIdleFunc which will unbind itself when it reaches the terminating condition. At 83 a semi-working animation is implemented but this shows the student the incorrect avatar assembly.

When work on the anim is continued at 132, small program flow errors are quickly fixed.

#### **9.7.6.2.5.6 Pipeline**

##### **9.7.6.2.5.6.1 IDA\_A3.PI.1."Star Jump Anim" [12]**

Name: Star Jump Anim

Description: Pipeline Oversight

Occurrences: 443-454 : total = 12 (+)

Detailed Description:

1. (443-454)(10 changes, +) Forgets to use glutIdleFunc(animate) for the star jump animation, leading to nothing happening.

## 9.7.7 Analysis of Segment Contents

### 9.7.7.1 In-depth Analysis of Segments

#### 9.7.7.1.1 Qualitative Analysis of Segments implementing Key Tasks

##### 9.7.7.1.1.1 Rotation Task

###### 9.7.7.1.1.1.1 Ida

The student produces an initial approach to child-parent rotation at (1141) at shown in Figure 179. The student attempts to calculate the distance between the parent and the child's lower and upper points using  $a^2 = \sqrt{b^2 + c^2}$ . However, the formula used is incorrect. The student attempts to calculate the b and c values by deducting the child's lower-left corner x and y values from the parent's centre values and then dividing the result by two. The division by two is incorrect and leads to the radius used to rotate the object shrinking at each rotate action.

```
57 void DiagramObject::rotate(Point p){
58
59     if(hasParent()){
60         Point c = epicenter;
61         double DL = pow(pow((epicenter.x-lower.x)/2, 2)
62             +pow((epicenter.y-lower.y)/2, 2), 0.5);
63         double DU = pow(pow((epicenter.x-upper.x)/2, 2)
64             +pow((epicenter.y-upper.y)/2, 2), 0.5);
65         double angle = atan((p.y-epicenter.y)/(p.x-epicenter.x))
66             - atan((pivot.y-epicenter.y)/(pivot.x-epicenter.x));
67         int t = int((angle+0.785398163)*2/3.141592654);
68         lower.x = c.x + DL*int(cos(t*2/3.141592654));
69         lower.y = c.y + DL*int(sin(t*2/3.141592654));
70         upper.x = c.x + DU*int(cos(t*2/3.141592654));
71         upper.y = c.y + DU*int(sin(t*2/3.141592654));
72     }
73     else{
74         double midy = (upper.y + lower.y)/2;
75         double midx = (upper.x + lower.x)/2;
76         theta = (atan((p.y-midy)/(p.x-midx))
77             - atan((pivot.y-midy)/(pivot.x-midx)))*180/6.28;
78     }
79 }
```

```

void DiagramObject::render()
{
    ...
    int temp = 0;
    if(lower.x > upper.x){
        temp = upper.x;
        upper.x = lower.x;
        lower.x = temp;
    }
    if(lower.y > upper.y){
        temp = upper.y;
        upper.y = lower.y;
        lower.y = temp;
    }
    ...
}

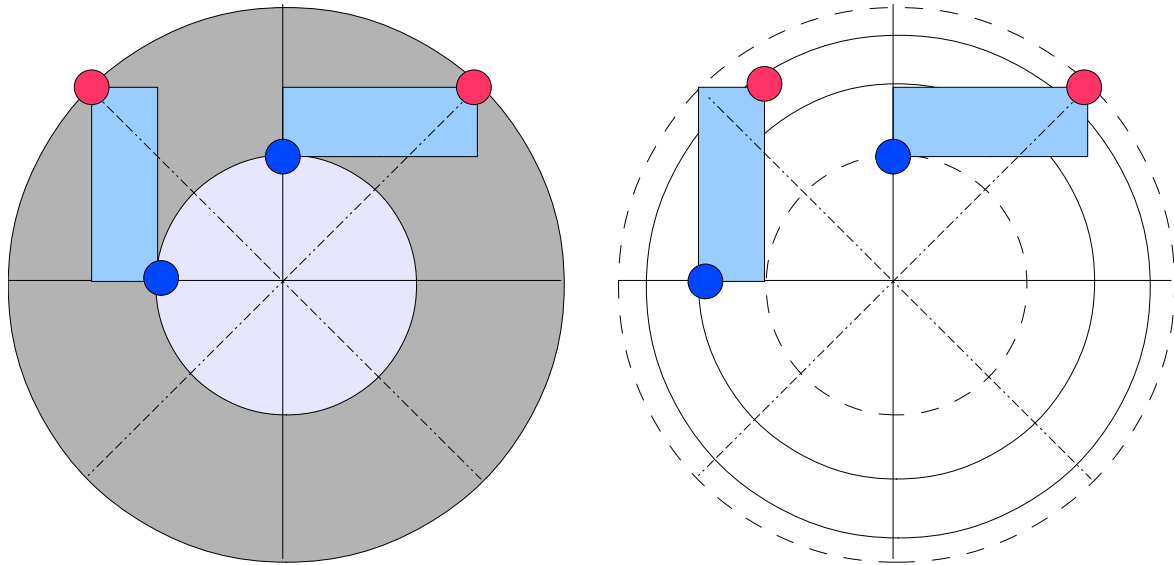
```

**Figure 179: Initial approach to rotating objects around their parent at (1141)**

The student then calculates the angle between the child's position and the parent position and then subtracts the angle between the mouse-down and the parent position. She uses the atan function, which will result in a lack of range as it can only recognise angles between 0-180 degrees. The student also incorrectly 'converts' the angle produced by the atan calculation to degrees when using it with the C++ trigonometry functions. This is incorrect because C++ functions use radian units, and hence the conversion leads to incorrect angle values being used.

In addition, the student uses integer variables to store object position, which will lead to a loss of precision as objects are rotated.

However, the most significant problem is with the rotation method itself. It rotates the lower and upper points of a rectangle separately. This leads to the object changing shape (as shown in Figure 180, left), sometimes appearing to disappear altogether. Furthermore, the student's implementation includes a check inside the object's display function (see , bottom) which switches an object's lower and upper coordinates if lower > upper. Figure 180 (right) shows the effect of this switch, with the dotted lines representing the old rotation circle, and the solid lines representing the post-swap rotation circles. As the diagram makes clear this can lead to the two points converging, with the object ultimately disappearing.



**Figure 180: Incorrect rotation of an object based on rotating its lower and upper points separately**

At this stage the object disappears rapidly when rotated, largely due to the error in the distance formula dividing the distance from parent by 2.

To debug the problem, the student outputs the angle to standard output and casts the result of lower/upper calculations to INT to no effect.

The student moves to a new solution approach at (1146). Instead of explicitly calculating the distance between lower/upper points to the parent, as Figure 181 shows the student instead translates the child object to the origin by subtracting the parent object's position. At this stage, the child object's distance from origin will be the child's distance to the parent. The student then performs the rotation. The student finally translates the object back into position by adding the previously subtracted parent position. The effect of this approach is essentially the same as the previous approach functionally, but it does not include the error in the distance calculation and hence works better. However, the student still rotates the lower and upper points separately, leading to the errors discussed previously. The object does 'rotate' several times while changing shape compared to the previous implementation which made it disappear straight away.

```

Point c = epicenter;
double angle = atan((p.y-epicenter.y)/(p.x-epicenter.x))
               - atan((pivot.y-epicenter.y)/(pivot.x-epicenter.x));
int t = int((angle+0.785398163)*2/3.141592654);
lower.x -= c.x;
lower.y -= c.y;
upper.x -= c.x;
upper.y -= c.y;
lower.x = int(lower.x*cos(t*2/3.141592654)
               - lower.y*sin(t*2/3.141592654));
lower.y = int(lower.y*cos(t*2/3.141592654)
               + lower.x*sin(t*2/3.141592654));
upper.x = int(upper.x*cos(t*2/3.141592654)
               - upper.y*sin(t*2/3.141592654));
upper.y = int(upper.y*cos(t*2/3.141592654)
               + upper.x*sin(t*2/3.141592654));
lower.x += c.x;
lower.y += c.y;
upper.x += c.x;
upper.y += c.y;

```

Figure 181: New approach to rotating objects at (1146)

The student experiments with incorrect modifications to the calculation (such as first translating the object away from the origin instead of towards it) until (1150). After working on unrelated issues between (1151-1168) the student fixes an error introduced into the calculation in debugging earlier and replaces the rotation value based on mouse coordinates with a fixed value of  $\pi/2$  from (1169-1172). While the fixed rotation angle does not resolve the problem, it does mask it, as the rotation by the fixed value both slows down the rate at which an object is rotated (lessening the effect of loss of precision problems) as well as essentially forcing the object into a 'square' shape, after which the problems with the rotation method no longer make it disappear altogether, as shown in Figure 182, though the object still changes shape.

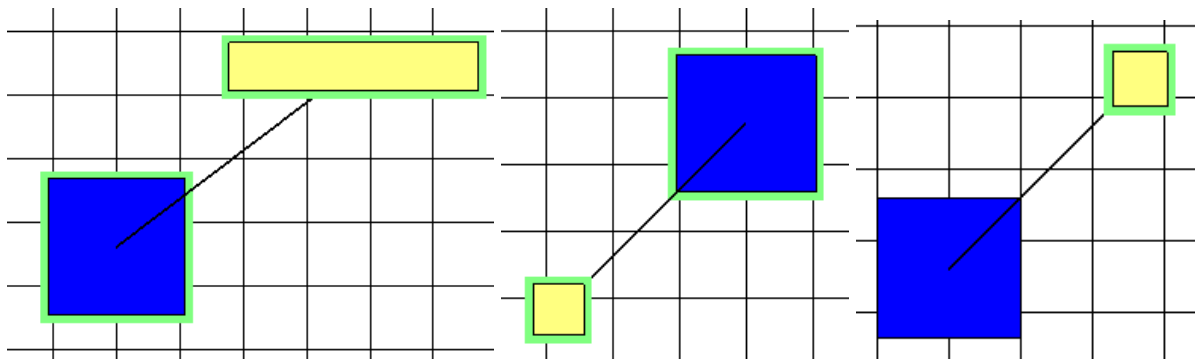


Figure 182: Rotation of a child object at (1172)



After analysing the angle and lower/upper point values using standard output statements, the student adds a new variable temp to the DiagramObject. The student appears to have reasoned that she can stop the 'degrading' of the object's lower and upper points by using a value that is not changed in the calculation to calculate the points, much like John calculated a distance value on the first rotation, to be used in all subsequent rotations to prevent loss-of-precision errors. The code for the implementation of the temp variable is shown in Figure 183. A call to setTemp, made before rotation occurs, sets the temp1 /temp2 variables equal to the lower and upper points respectively. However, since this occurs before every rotation action, this means the temp values are equal to the lower/upper values anyways, meaning the mechanism has no effect. The original problem of the incorrect rotation persists, with the student working on experimenting with the temp mechanism, with fixed versus dynamic (calculated on mouse values) rotation values and removing and re-adding different parts of code between (1173-1210) while printing out coordinate and angle values to standard output. The student frequently breaks rotation altogether by not updating the temp variables correctly, but gets no closer to a working rotation solution. At (1210) the student gives up, using a fixed value which masks the problems with the rotation algorithm but only allows rotation by 45-degree increments. Also, due to a combination of the unresolved problems (loss of precision and incorrect angle calculation) the distance between child and parent objects still decreases as the child object is rotated.

```
void DiagramObject::setTemp() {  
    temp1 = lower;  
    temp2 = upper;  
}
```

```

else if(action == "rotate"){
    draw = "rotate";
    for(int i=0; i<tableCount; i++){
        ...
        if(newPieceOfFurniture[i]->isSelected()){
            newPieceOfFurniture[i]->setPivot(clickDown);
            newPieceOfFurniture[i]->setTemp();
        }
    }
}

if(draw == "rotate")
{
    for(int i=0; i<tableCount; i++)
    {
        if(newPieceOfFurniture[i]->isSelected())
        {
            newPieceOfFurniture[i]->rotate(clickUp);
            newPieceOfFurniture[i]->setTemp();
            newPieceOfFurniture[i]->render();
        }
    }

    temp1.x -= c.x;
    temp1.y -= c.y;
    temp2.x -= c.x;
    temp2.y -= c.y;

    lower.x = int(temp1.x*cos(angle) - temp1.y*sin(angle));
    lower.y = int(temp1.y*cos(angle) + temp1.x*sin(angle));
    upper.x = int(temp2.x*cos(angle) - temp2.y*sin(angle));
    upper.y = int(temp2.y*cos(angle) + temp2.x*sin(angle));

    temp1.x += c.x;
    temp1.y += c.y;
    temp2.x += c.x;
    temp2.y += c.y;
}

```

**Figure 183: State of the source code implementing the 'temp' variable at (1174)**

Despite working on the problem for a considerable time, the student never recognised the fundamental error in the rotation approach used. Rotation of the object based on a single point (mid, lowerleft or upperright) would have resolved the major problem in her implementation and been conceptually simpler, but she did not develop this solution. This student's work shows the problems when multiple problems intermingle; this is especially important in Computer Graphics programming, where so many different aspects of programming (in this case ED, mathematical, GP) intermix and interact; in this case, the GP problem produces spatial errors which lead to a fatal GP problem (crash); the problem is made worse by poor ED code producing difficult-to-control rotations that makes debugging hard, and is apparently but not really addressed by the ED solution of going

from dynamic to incremental (45-degree) rotation . Such a mix of problems and the complexity of the code associated with different aspects of the problem overwhelm the student's ability to pinpoint individual errors, meaning that the student will be unable to isolate and resolve the associated problems. This is an issue that should be addressed in assignment design.

#### 9.7.7.1.1.1.2 John

John begins implementation of the rotate functionality at (658) (see Figure 184). The initial approach contains two problems. First, he incorrectly calculates the object centres with the formula  $centre = upper - lower$  instead of the correct  $centre = lower + (upper - lower)/2$  as shown in Figure 185. The student corrects this error at (662) after some experimentation and outputting coordinates to standard output (to verify the values) at (661).

```

60 void DiagramObject::rotate(int newx, int newy) {
61     int px = lower.x + ((upper.x-lower.x)>>1); /* parent's center's X */
62     int py = lower.y + ((upper.y-lower.y)>>1); /* parent's center's Y */
63     /* get angle between mouse cursor and parent's center on the X axis*/
64     float angle = atan(float(newy-py)/float(newx-px));
65
66     vector<pair<DiagramObject*,int> >::iterator it=children.begin();
67     for(; it!=children.end(); it++) {
68         int distance;
69         if (it->second)
70             distance = it->second;
71         else {
72             int tmpx = it->first->lower.x;
73             int tmpy = it->first->lower.y;
74             distance = int(sqrt( (tmpx-px)*(tmpx-px)+(tmpy-py)*(tmpy-py) ));
75         }
76         /* move the child to the new location */
77         it->first->move(int(px + distance*cos(angle)),
78                     int(py + distance*sin(angle) ));
79     }
80 }

```

Figure 184: Initial child-parent rotate implementation at (658)

```

658, ADDED(O):    int childCenterx = ((*it)->upper.x - (*it)->lower.x)>>1;
662, MUTATED(O): int childCenterx = lower.x + (((*it)->upper.x - (*it)->lower.x)>>1);
669, MUTATED(O): int childCenterx = (*it)->lower.x + (((*it)->upper.x - (*it)->lower.x)>>1);
670, MUTATED(O): int childCenterx = (*it)->lower.x + ( (*it)->upper.x - ((*it)->lower.x)>>1) );
671, MUTATED(O): int childCenterx = (*it)->lower.x + ( (*it)->upper.x - (((*it)->lower.x)>>1) );
672, MUTATED(O): int childCenterx = (*it)->lower.x + ( ((*it)->upper.x-(*it)->lower.x)>>1);
673, MUTATED(O): int cx = (*it)->lower.x; /* child lower.x */

```

Figure 185: Line History for the calculation of the object centre; at (673) the student changes to utilising the lower-left corner instead

The approach still contains an error though. The student utilises the parent's and child's centre coordinates for rotation. However, the object move function sets an object's lower-left position to the input variable rather than the centre, meaning that when an object is rotated, its lower-left point is moved to its centre point, leading to objects moving by (width/2, height/2) each time they are rotated, until they disappear from the screen.

The student experiments with modifying the mathematical formula used to calculate the angle between parent and child objects as well as modifying the formula used to calculate the mid-point (breaking it in the process, as is shown in Figure 185, Changes 669-672) before discovering the source of the error. The student switches to utilising the lower left corner in the calculation at (673), resolving the problem.

While the mathematics of the approach is now correct, this reveals a loss-of-precision error underlying the student's approach. Since an object's coordinates are stored using Integer values, each recalculation of its new rotated position using the circle equation leads to the floating-point part of the new position being lost, which over time moves the child object toward or away from the parent object (the code containing this fault is shown in Figure 186). The student realises the problem very quickly. In (675) he attempts to modify the rotation and move algorithms to store the distance from the parent to the child and then using this fixed distance in the calculation in line 85 as shown in Figure 187. Unfortunately, the student does not store the distance value after it is calculated, meaning it is recalculated during every rotation, resulting in the same loss of precision as before.

```

60 void DiagramObject::rotate(int newx, int newy) {
61     int px = lower.x + ((upper.x-lower.x)>>1); /* parent's center's X */
62     int py = lower.y + ((upper.y-lower.y)>>1); /* parent's center's Y */
63     /* get angle between mouse cursor and parent's center on the X axis*/
64     float angle = atan(float(newy-py)/float(newx-px));
65
66     vector<DiagramObject*>::iterator it=children.begin();
67     for(; it!=children.end(); it++)
68     {
69         int cx = (*it)->lower.x; /* child lower.x */
70         int cy = (*it)->lower.y; /* child lower.y */
71         float radius = sqrt((cx-px)*(cx-px) + (cy-py)*(cy-py)); /* distance from
72         /* move the child to the new location */
73         (*it)->move(int(px + radius*cos(angle)), int(py + radius*sin(angle) ));
74     }
75 }

```

Figure 186: Storage of the new position causes loss of precision in (673)

```

60 void DiagramObject::rotate(int newx, int newy) {
61     int px = lower.x + ((upper.x-lower.x)>>1); /* parent's center's X */
62     int py = lower.y + ((upper.y-lower.y)>>1); /* parent's center's Y */
63     /* get angle between mouse cursor and parent's center on the X axis*/
64     float angle = atan(float(newy-py)/float(newx-px));
65
66     vector<pair<DiagramObject*,int> >::iterator it=children.begin();
67     for(; it!=children.end(); it++) {
68         int distance;
69         if (it->second)
70             distance = it->second;
71         else {
72             int tmpx = it->first->lower.x;
73             int tmpy = it->first->lower.y;
74             distance = int(sqrt( (tmpx-px)*(tmpx-px)+(tmpy-py)*(tmpy-py) ));
75         }
76         /* move the child to the new location */
77         it->first->move(int(px + distance*cos(angle)),
78             int(py + distance*sin(angle) ));
79     }
80 }

```

**Figure 187: The calculated distance is not stored, meaning it is recalculated every time at (675)**

The student spends the Changes between (675-685) debugging this problem, modifying different mathematical formulas involved in the calculation of angles and the calculation of the child object's new position as well as event-driven code without success. At (686) he identifies the actual underlying error and implements a function (shown in Figure 188) to store the distance value when the object is moved (or the first time the object is rotated), and then reuses this stored distance value when calculating the child object's new position in line 85, fixing the loss-of-precision problem (In fact precision is lost, but only in the original calculation and not in subsequent rotations).

```

70 #include <iostream> /* debug */
71 void DiagramObject::rotate(int newx, int newy) {
72     int px = lower.x + ((upper.x-lower.x)>>1); /* parent's center's X */
73     int py = lower.y + ((upper.y-lower.y)>>1); /* parent's center's Y */
74     /* get angle between mouse cursor and parent's center on the X axis */
75     float angle = atan(float(newy-py)/float(newx-px));
76
77     vector<pair<DiagramObject*,int> >::iterator it=children.begin();
78     for(; it!=children.end(); it++) {
79         int distance;
80         if (it->second)
81             distance = it->second;
82         else
83             updateDistance();
84         /* move the child to the new location */
85         it->first->move(int(px + distance*cos(angle)),
86                     int(py + distance*sin(angle) ));
87     }
88 }

59 void DiagramObject::updateDistance() {
60     int px = lower.x + ((upper.x-lower.x)>>1); /* parent's center's X */
61     int py = lower.y + ((upper.y-lower.y)>>1); /* parent's center's Y */
62     vector<pair<DiagramObject*,int> >::iterator it=children.begin();
63     for(; it!=children.end(); it++) {
64         int tmpx = it->first->lower.x;
65         int tmpy = it->first->lower.y;
66         it->second = int(ceil(sqrt( (tmpx-px)*(tmpx-px)+(tmpy-py)*(tmpy-py) )) );
67     }
68 }

```

**Figure 188: The correction at (686) only calculates the distance when the object is moved in relation to the parent.**

The student's event-driven method of determining the amount (in radians) to rotate by is developed as is shown in the Line History in Figure 189. The student initially attempts to use the `asin` and then the `atan` function to produce an angle value from two points, but then switches to the `atan2` function which allows full 360 degree rotation rather than the 180 degree rotation afforded by the `atan` function. The student does not spend a large amount of time on implementing this effective way of calculating rotation angles. However, as can be seen in the Project History, the student spends a considerable number of Changes after having correctly implemented the calculation formula at (665) on modifying the formula. Most of those Changes are actually attempts to debug the previously discussed problems by modifying (and thereby breaking) his correct angle calculation formula as he attempts to identify the source of the problems.

(833):	(658, 664-667, 673, 693, 699, 710-714, 777 : total = 14)
658, ADDED(O):	float angle = asin(float(newy-centery)/float(newx-centerx));
664, MUTATED(O):	float angle = atan(float(newy-centery)/float(newx-centerx));
665, MUTATED(X):	float angle = atan2(float(newy-centery)/float(newx-centerx));
666, MUTATED(X):	float angle = atan2(double(newy-centery)/double(newx-centerx));
667, MUTATED(O):	float angle = atan(float(newy-centery)/float(newx-centerx));
673, MUTATED(O):	float angle = atan(float(newy-py)/float(newx-px));
693, MUTATED(X):	float newAngle = atan(float(newy-py)/float(newx-px));
699, MUTATED(O):	float offsetAngle = atan(float(newy-py)/float(newx-px));
710, MUTATED(X):	float offsetAngle = atan2(float(newy-py)/float(newx-px));
711, MUTATED(X):	float offsetAngle = atan2(double(newy-py)/double(newx-px));
712, MUTATED(X):	float offsetAngle = std::atan2(double(newy-py)/double(newx-px));
713, MUTATED(X):	float offsetAngle = std::atan2(double(newy-py), double(newx-px));
714, MUTATED(O):	float offsetAngle = atan2(double(newy-py), double(newx-px));
777, MUTATED(X):	float offsetAngle = std::atan2(double(newy-py), double(newx-px));

Figure 189: Line History showing the development of the angle calculation

#### 9.7.7.1.1.1.3 Thomas

The student's initial implementation of child-parent rotation at (834) (see Figure 190) calculates the angle by which to rotate by calculating the distance between the mouse-down point and the current mouse location while the mouse is being dragged as shown in Figure 190, top.

```

381 void execRotateFurniture(int _x, int _y){
382     if(selected!=-1){
383         int dist = sqrt((_x-mouse_x)*(_x-mouse_x)+(_y-mouse_y)*(_y-mouse_y));
384         furn.at(selected).angle=dist;
385     }
386 }

168 radius = sqrt((x-centrex)*(x-centrex)+(y-centrey)*(y-centrey));
169 glBegin(GL_POLYGON);
170 glVertex2i(centrex+radius*cos(angle), centrey+radius*sin(angle));
171 glVertex2i(centrex+radius*cos(angle), centrey+radius*sin(angle)+height);
172 glVertex2i(centrex+radius*cos(angle)+width, centrey+radius*sin(angle)+height);
173 glVertex2i(centrex+radius*cos(angle)+width, centrey+radius*sin(angle));
174 glEnd();

```

Figure 190: Initial child-parent rotation implementation at 834

Inside the object's draw function (shown in Figure 190, bottom), the distance of a child object to its parent (in the case of parent objects, the distance is 0) is calculated. The child object is drawn using the circle formula, with the centre of the circle at the parent's position.

This approach successfully rotates the child object around its parent, but effectively ignores the child's actual position. The child's x,y position is used only to calculate the radius (distance from parent) meaning that attempting to move the child will move it only along the vector along the child object's current angle to the parent, which is calculated independently of the child's position (it is based only on the mouse position).

The student attempts to address this problem by storing the angle between the parent and the child at the time it is added to the parent inside the append function which creates parent-child links as shown in Figure 191. However, the student uses the incorrect asin function (which will not calculate the angle, he should be using atan2). Furthermore, calculating the angle only once when the child is added will not fix the problem, as the angle is still being modified independently of the child's position when the child is rotated. Between (834-869) the student tries several other things to enable independent child movement, including adding debug output to standard out to observe the numeric position of objects, converting (incorrectly) from radian units to angle units for C++ trigonometric functions (e.g. 848, 854), casting variables in the angle calculation to the type double and changing the variables storing the centre point (centrex, centrey) to type double.

```
246 void append(furniture &f) {
247     children.push_back(&f);
248     f.centrex=x;
249     f.centrey=y;
250     f.radius
251         = sqrt((f.x-f.centrex)*(f.x-f.centrex)
252             +(f.y-f.centrey)*(f.y-f.centrey));
253     if(f.radius!=0)
254         f.angle=asin((f.y-f.centrey)/f.radius);
255     cout<<f.angle;
256 }
```

**Figure 191: At (845) The student attempts to store the angle between parent and child when the child is first added to the parent**

The student attempts to calculate the actual angle using the child's position and the parent's position inside the display function as shown in Figure 192, but as the x,y coordinates are not updated when the child is rotated this means that the angle calculated does not change during rotation meaning that the object can no longer rotate. He does switch to atan for calculating the angle (better) and then switches to atan2 at (872), meaning that at least the angle calculation is correct.



```

166 radius = sqrt((x-centrex)*(x-centrex)+(y-centrey)*(y-centrey));
167 if(x-centrex!=0)
168     angle=atan((x-centrex)/(y-centrey));
169 glBegin(GL_POLYGON);
170 glVertex2i(int(centrex+radius*cos(angle)), int(centrey+radius*sin(
171 glVertex2i(int(centrex+radius*cos(angle)), int(centrey+radius*sin(
172 glVertex2i(int(centrex+radius*cos(angle)+width), int(centrey+rad:
173 glVertex2i(int(centrex+radius*cos(angle)+width), int(centrey+rad:
174 glEnd();

```

**Figure 192: Student attempts to calculate the parent-child angle inside the display function at (870)**

The student solves the issue at (876) by changing the child's x/y coordinates as it is rotated (see Figure 193). This means that the child's x,y coordinates as modified by the object move function again have the desired effect of moving the child to a new position, and rotation via the rotation function still works as the angle is once more calculated using mouse coordinates and is hence malleable.

```

381 void execRotateFurniture(int _x, int _y){
382     if(selected!=-1){
383         double angle=atan2((_y-mouse_y), (_x-mouse_x));
384         furn.at(selected).x
385             = furn.at(selected).centrex+furn.at(selected).radius*cos(angle);
386         furn.at(selected).y
387             = furn.at(selected).centrey+furn.at(selected).radius*sin(angle);
388     }
389 }

166 radius = sqrt((x-centrex)*(x-centrex)+(y-centrey)*(y-centrey));
167 angle=atan2((y-centrey), (x-centrex));
168 cout<<angle*(180/3.1415)<<endl;
169 glBegin(GL_POLYGON);
170 glVertex2i(int(centrex+radius*cos(angle)), int(centrey+radius*sin(angle)));
171 glVertex2i(int(centrex+radius*cos(angle)), int(centrey+radius*sin(angle)+height));
172 glVertex2i(int(centrex+radius*cos(angle)+width), int(centrey+radius*sin(angle)+height));
173 glVertex2i(int(centrex+radius*cos(angle)+width), int(centrey+radius*sin(angle)));
174 glEnd();

```

**Figure 193: At (876) the child's coordinates are changed as it is rotated, meaning the angle is correctly calculated**

However, as the student uses integer variables to store the x,y coordinates, this introduces a loss-of-precision error, causing the child object's distance to the parent to change during rotation.

The student utilises many incorrect solution approaches in an attempt to fix this problem. Between 877-915, he converts radian units to degrees and back again (878, 882, 884, 897), casts components of calculations to INT (879), increases the decimal accuracy of PI (888-890), calculates the object's

x/y position on a combination of the current and previous angles rather than just the current angle (891), adds debug cout statements, uses modulus statements to ensure the angle is in the 0-360 range (893) and slows down the rotation by dividing the angle used by a constant (897-899) as well as adding a conditional which prevents rotation for small angles (900-902, 905-911). None of these solution attempts address the root of the problem, but they show the student's uncertainty about the root cause.

The student addresses the loss of precision problem by removing the casts to INT from calculations and changing the type of the variable storing an object's x/y coordinates to double at (915-916), fixing the problem with very minor modifications to the code text.

The student's implementation now correctly rotates a child around its parent, and allows the child to be moved independently and then rotated again. However, the implementation is still buggy. The parent's centre is not updated when the parent is moved, resulting in the child object moving around the parent's previous position.

The student attempts to address this problem with several different approaches to updating the parent's centre, but the implementation is made more difficult by the fact that the implementation must ensure not to modify the centres of child objects since otherwise they will rotate around their own centres (and hence not at all). Solutions such as shown in Figure 194 are conceptually wrong; in this case, the parent's centre is set to the child's position, which makes no sense.

```

150 void move(int toX, int toY){
151     hasMoved=true;
152     for(unsigned int i=0;i<children.size();i+=1){
153         double xoffset = x-children.at(i)->x;
154         double yoffset = y-children.at(i)->y;
155         bool flagx=false;
156         bool flagy=false;
157         if(children.at(i)->x==centrex)
158             flagx=true;
159         if(children.at(i)->y==centrey)
160             flagy=true;
161         if(!(children.at(i)->hasMoved))//break clause to av
162             children.at(i)->move(toX-xoffset,toY-yoffset);
163         if(flagx)
164             centrex=children.at(i)->x;
165         if(flagy)
166             centrey=children.at(i)->y;
167     }
168     x=toX;
169     y=toY;
170 }

```

Figure 194: At (931) the parent's centre is incorrectly set to equal the child's position

The student's final solution is shown in Figure 195. The student updates an object's rotation centre to be x/y position whenever it is moved, and updates the centre of all of its children to be its position. This means that a child object's rotation centre becomes its own centre when it is moved, making rotation about its parent impossible. Moving the parent will change the child's centre to be the parent's location again, making rotation about the parent possible once more. Therefore, under the final buggy solution, a child move must be followed by the parent being moved in order to re-enable rotation around the parent.

```

150 void move(int toX, int toY){
151     hasMoved=true;
152     for(unsigned int i=0;i<children.size();i+=1){
153         double xoffset = x-children.at(i)->x;
154         double yoffset = y-children.at(i)->y;
155         if(!(children.at(i)->hasMoved))//break clause to avoid infinit
156             children.at(i)->move(int(toX-xoffset),int(toY-yoffset));
157         children.at(i)->centrex=toX;
158         children.at(i)->centrey=toY;
159     }
160     x=toX;
161     y=toY;
162     centrex=x;
163     centrey=y;
164 }

```

Figure 195: The student's final solution at (943)

The student is apparently too confounded with the program flow involved in propagating centres to produce a fix for the problem. The complexity of the student's implementation of centres has led to this difficulty, which could have been addressed by simply utilising the parent's x/y coordinates instead of having a separate centrex/centrey coordinate (the student's implementation is shown in Figure 196). Apparently the student was unable to make the conceptual leap to the much simpler conceptual model of using a parent's position, instead of having to maintain a parent's position in a separate variable, which was what led to the program flow issues which prevented a working solution due to the difficulty of correctly updating these variables.

```

258 void append(furniture* f){
259     children.push_back(f);
260     f->centrex=x;
261     f->centrey=y;
262     f->parent = this;
263 }

171 if(parent != NULL)
172 {
173     cout << "Setting centre from parent " << endl;
174     centrex = parent->x;
175     centrey = parent->y;
176 }
177 cout << x << ", " << y << " : " << centrex << ", " << centrey << endl;
178 glColor3f(red, green, blue);
179 radius = sqrt((x-centrex)*(x-centrex)+(y-centrey)*(y-centrey));
180 angle=atan2((y-centrey),(x-centrex));
181 glBegin(GL_POLYGON);
182 glVertex2i(int(centrex+radius*cos(angle)), int(centrey+radius*sin(angle)));
183 glVertex2i(int(centrex+radius*cos(angle)), int(centrey+radius*sin(angle)+height));
184 glVertex2i(int(centrex+radius*cos(angle)+width), int(centrey+radius*sin(angle)+height));
185 glVertex2i(int(centrex+radius*cos(angle)+width), int(centrey+radius*sin(angle)));
186 glEnd();

```

Figure 196: The student stores parent and child centres separately

#### 9.7.7.1.1.1.4 Christopher

Christopher did not implement functionality to rotate a child object around the parent object.

#### 9.7.7.1.1.1.5 Michael

Michael did not implement functionality enabling objects to be parented to other objects and hence did not implement functionality to rotate a child object around the parent object.

### 9.7.7.1.1.2 Assembly Task

#### 9.7.7.1.1.2.1 Christopher

##### Naïve Assembly

Christopher's implementation of a naïve assembly is contained in segment [CHRISTOPHER\\_A3.SP.1.1."Initial Avatar Assembly" \[36\]](#). In his assignment, the student implemented a Scene Tree algorithm which he could have used to develop the hierarchical model (see Figure 197) but the student adds all limbs to a single root node at (78), producing a flat graph and a non-hierarchically assembled model.

```

void draw() {
    glPushMatrix();
    glTranslatef(translate.x,translate.y,translate.z);
    glRotatef(rotation.x,1,0,0);
    glRotatef(rotation.y,0,1,0);
    glRotatef(rotation.z,0,0,1);
    glScalef(scale.x,scale.y,scale.z);
    cout << "Drawing object" << endl;
    if (obj!=NULL) {
        obj->draw();
    } else {
        cout << "No object for node " << name << endl;
    }
    cout << "Drawing child nodes " << endl;
    for (int i =0; i<childNodes.size();i++) {
        childNodes[i]->draw();
    }
    //end for
    glPopMatrix();
}

```

**Figure 197: The Scene Tree implementation, based on lecture notes, supports hierarchical transformations**

As the student commences work on the assembly, he makes several axis errors in the initial assembly at (213-215), misplacing the torso and arms. The assembly is shown in Figure 199. It is a simple non-hierarchical assembly, with each limb being placed directly from the origin into position via a single translate statement. The student makes additional axis errors in positioning the upper leg (220) and the lower leg (223-226) (see Line History in Figure 200). At this point in the assembly, the student has produced an axis error every time a limb was newly added (excluding copied limbs or limbs continued from parent limbs). As the Line History shown in Figure 200 demonstrates, some limbs are placed with multiple axis and sign errors.



Figure 198: Change 223, Incorrect reversal of y-translate sign (y -> -y).

```

225 scene->getNodeByName("Head")->setTranslate(0,3,0);
226 scene->getNodeByName("Torso")->setTranslate(0,1,0);
227 scene->getNodeByName("RightUpperArm")->setTranslate(0,2,1);
228 scene->getNodeByName("RightLowerArm")->setTranslate(0,2,2);
229 scene->getNodeByName("LeftUpperArm")->setTranslate(0,2,-1);
230 scene->getNodeByName("LeftLowerArm")->setTranslate(0,2,-2);

```

Figure 199: Excerpt of the Initial naive assembly at (225)

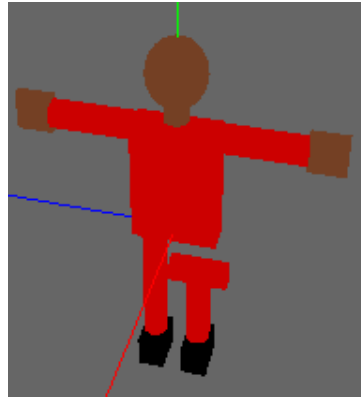
```

>220, ADDED(O): scene->getNodeByName("RightUpperLeg")->setTranslate(0,2,1);
>221, MUTATED(O): scene->getNodeByName("RightUpperLeg")->setTranslate(0,-2,1);
>222, MUTATED(O): scene->getNodeByName("RightUpperLeg")->setTranslate(0,-1,1);
>223, MUTATED(O): scene->getNodeByName("RightUpperLeg")->setTranslate(0,1,1);
>224, MUTATED(O): scene->getNodeByName("RightUpperLeg")->setTranslate(0,-1,1);
>225, MUTATED(O): scene->getNodeByName("RightUpperLeg")->setTranslate(0,-1,0.5);
<257, DELETED(O): DELETED (scene->getNodeByName("RightUpperLeg")->setTranslate(0,-1,0.5);)

```

Figure 200: Line History for modifications to the right upper leg's translate during naive assembly

From (229-232) the student experiments with different rotation values for the avatar's upper leg. This exposes the non-hierarchical nature of the assembly method, as the leg rotates about its own centre rather than its joint as it is rotated as shown in Figure 201.



**Figure 201: Naive assembly becomes apparent when the upper leg rotation leaves the lower leg unaffected at (229-232)**

After a ten-minute gap pondering the problem at 233 the student realises the problem with his assembly method and moves to a hierarchical assembly method from (233-255), adding child limbs to parent limbs using his scene graph (shown in Figure 202), leading to the hierarchical assembly shown in Figure 202. However, while the assembly is hierarchical it is joint-naïve, since it does not translate the limb onto the parent-child joint before orienting it. This means that the limb will rotate about its own centre rather than the parent-child joint. Comparison of limb movement with the provided executable would have made the student aware of the incorrect construction.

```

211 SceneTreeNode* manNode;
212     manNode=new SceneTreeNode("man");
213     scene->childNodes.push_back(manNode);
214 SceneTreeNode* TorsoNode;
215     TorsoNode=new SceneTreeNode("Torso");
216     TorsoNode->obj=glObjects[0]->getObjectsByName()["Torso"];
217     manNode->childNodes.push_back(TorsoNode);
218     SceneTreeNode* HeadNode;
219     HeadNode=new SceneTreeNode("Head");
220     HeadNode->obj=glObjects[0]->getObjectsByName()["Head"];
221     HeadNode->setTranslate(0,2,0);
222     TorsoNode->childNodes.push_back(HeadNode);
223     SceneTreeNode* RightUpperArmNode;
224     RightUpperArmNode = new SceneTreeNode("RightUpperArm");
225     RightUpperArmNode->obj=glObjects[0]->getObjectsByName()["RightUpperArm"];
226     RightUpperArmNode->setTranslate(0,0,1);
227     TorsoNode->childNodes.push_back(RightUpperArmNode);
228     SceneTreeNode* RightLowerArmNode;
229     RightLowerArmNode = new SceneTreeNode("RightLowerArm");
230     RightLowerArmNode->obj=glObjects[0]->getObjectsByName()["RightLowerArm"];
231     RightLowerArmNode->setTranslate(0,0,1);
232     RightUpperArmNode->childNodes.push_back(RightLowerArmNode);
233     SceneTreeNode* RightPalmNode;

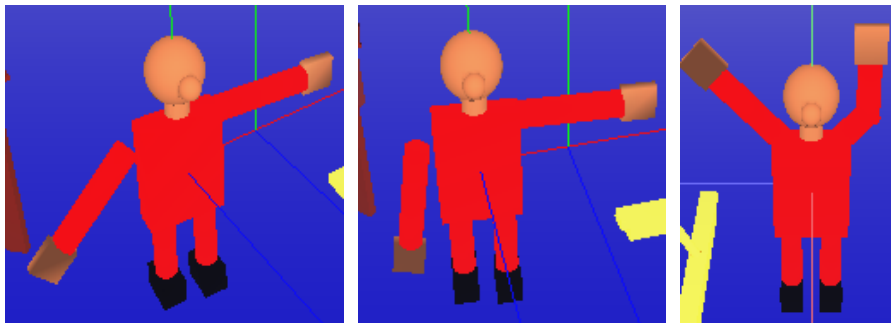
```

**Figure 202: Excerpt from Christopher's hierarchical assembly at (255)**

After correcting the assembly, the student corrects the position of limbs from (256-268) since the original implementation positioned limbs in respect to the origin while the new implementation

positions them in respect to one another. The student makes no errors in correcting the limb's positions.

From (701-710) while working on animations, the student again experiments with limb orientation. The naïve construction is clearly apparent at 704 (see Figure 203) which means at this stage the student must be aware of the incorrect avatar assembly. The student is apparently unable to conceptualise the correct approach. He switches to using a different arm orientation to hide the broken assembly as shown in Figure 203 (right panel). The student does no further work on the avatar assembly. His final submission contains a hierarchical but joint-naïve avatar assembly.



**Figure 203: The incorrect assembly is apparent at (704), the re-orientation of limbs to hide the broken assembly at (705)**

#### **9.7.7.1.1.2.2 John**

##### **Naïve assembly**

The student's first non-hierarchical assembly is contained in segment [JOHN\\_A3.SP.4.1."Initial naive assembly"](#) [26]. The student adds separate push/pop statements around each limb, meaning that each limb is transformed from the origin to its position. The student only makes a single axis error while implementing the non-hierarchical assembly, otherwise adding all other limbs correctly. Most of the changes are spent tweaking limbs into place.

From (102-118), the student implements keys to allow user interaction with the application, including implementation of keys to orient limbs. This allows the student to discover that the avatar assembly is incorrect. As Figure 204 (left panel) shows, because of the non-hierarchical assembly limbs rotate around their own centres. Furthermore, if a limb was moved, its attached limbs would not move with it.



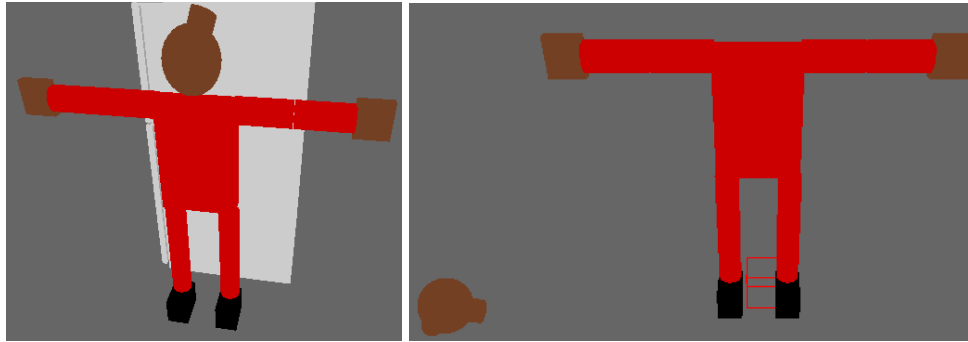


Figure 204: Rotation around own centre (118), around origin at (119)

### Proper Assembly

The student implements a proper avatar construction in segment [JOHN\\_A3.SP.4.2."Final assembly" \[33\]](#). At (119) the student experiments with moving the head's translate call, moving it before the head's rotation block which incorrectly causes the limb to rotate around the origin (as shown in Figure 204, right).

The student first works on discovering a method to properly place the 'joint'. In (120) the student introduces a second push/pop including another translate statement to the head's assembly block. The student then experiments with placement of the translate statement from (120-123) (source code shown in Figure 205) resulting in the output shown in Figure 206. After incorrectly placing the translate call in (120-122), the student correctly uses a 'towards-joint' translate before the rotation block and an 'away-from-joint' translate after the rotation block in (123), producing correct rotation around the head-chest joint.

120:

```
glPushMatrix(); {
    glTranslatef(0.0, 8.0, 0.0);
    glPushMatrix(); {
        glTranslatef(0.0, 1.0, 0.0);
        glRotatef(guy.partsRot[0][0], 1,0,0);
        glRotatef(guy.partsRot[0][1], 0,1,0);
        glRotatef(guy.partsRot[0][2], 0,0,1);
        guy.pointer->getObjectsByName()["Head"]->draw();
    } glPopMatrix();
} glPopMatrix();
```

121

```

glPushMatrix(); {
    glTranslatef(0.0, 8.0, 0.0);
    glPushMatrix(); {
        glRotatef(guy.partsRot[0][0], 1,0,0);
        glRotatef(guy.partsRot[0][1], 0,1,0);
        glRotatef(guy.partsRot[0][2], 0,0,1);
        glTranslatef(0.0, -1, 0.0);
        guy.pointer->getObjectsByName() ["Head"]->draw();
    } glPopMatrix();
} glPopMatrix();

```

122:

```

glPushMatrix(); {
    glRotatef(guy.partsRot[0][0], 1,0,0);
    glRotatef(guy.partsRot[0][1], 0,1,0);
    glRotatef(guy.partsRot[0][2], 0,0,1);
    glPushMatrix(); {
        glTranslatef(0.0, 8.0, 0.0);
        guy.pointer->getObjectsByName() ["Head"]->draw();
    } glPopMatrix();
} glPopMatrix();

```

123:

```

glPushMatrix(); {
    glTranslatef(0.0, 8.0, 0.0);
    glPushMatrix(); {
        glTranslatef(0.0, -1.0, 0);
        glRotatef(guy.partsRot[0][0], 1,0,0);
        glRotatef(guy.partsRot[0][1], 0,1,0);
        glRotatef(guy.partsRot[0][2], 0,0,1);
        glTranslatef(0.0, 1.0, 0);
        guy.pointer->getObjectsByName() ["Head"]->draw();
    } glPopMatrix();
} glPopMatrix();

```

Figure 205: Experimentation with translate statements for positioning the head joint from (120-123)

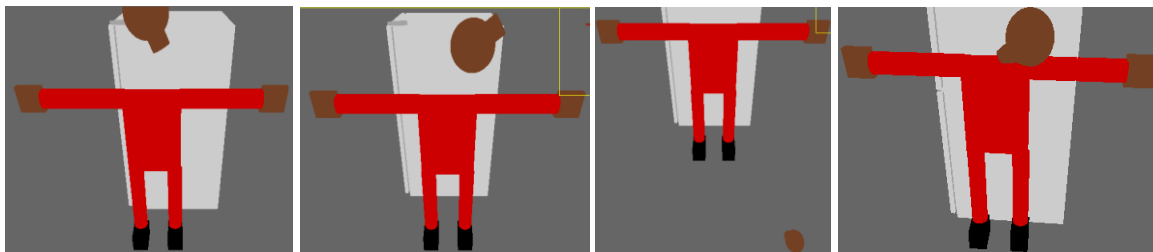
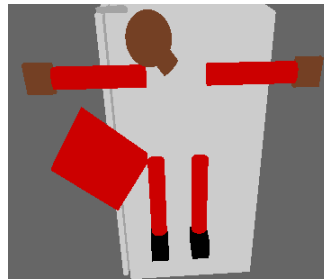


Figure 206: Effect of different attempts at positioning the head joint at (120-123)

However, the student still has not implemented a hierarchical assembly, meaning limbs move independently of each other, but the student cannot observe this behaviour because other limbs do not have orientation blocks which permit their rotation yet.

The student transfers the same approach used to assemble the head to the chest at (126). Since both limbs are isolated in their own push/pop blocks (non-hierarchical), rotation of the chest leads to the avatar breaking apart as shown in Figure 207.



**Figure 207: Incorrect assembly at (126)**

To address the problem of the avatar ‘breaking apart’ when the chest is rotated, the student experiments with the head’s transform block from 119-122 as shown in Figure 208. The student attempts to modify the transformations in such a way that the head is rotated about the same point as the chest, so that both can remain connected. However the problem does not lie with the head’s assembly but rather with the fact that the assembly is non-hierarchical, so modifications involving the moving of translate statements only introduces additional errors.

*Change 118, Initial State (from naïve construction)*

```
glPushMatrix(); {
    glTranslatef(0.0, 8.0, 0.0);
    glRotatef(guy.partsRot[0][0], 1,0,0);
    glRotatef(guy.partsRot[0][1], 0,1,0);
    glRotatef(guy.partsRot[0][2], 0,0,1);
    guy.pointer->getObjectsByName()["Head"]->draw();
} glPopMatrix();
```

*Change 119, translate moved after rotation stack*

```
411      glPushMatrix(); {
413          glRotatef(guy.partsRot[0][0], 1,0,0);
414          glRotatef(guy.partsRot[0][1], 0,1,0);
415          glRotatef(guy.partsRot[0][2], 0,0,1);
412      glTranslatef(0.0, 8.0, 0.0);
416          guy.pointer->getObjectsByName()["Head"]->draw();
417      } glPopMatrix();
```

*Change 120, translate moved back before stack, additional translate added before stack together with a new set of push/pop statements for the new translate statement and the rotation stack*

```

411      glPushMatrix(); {
415      └─ glTranslatef(0.0, 8.0, 0.0);
-1 : └─ glPushMatrix(); {
-1 : └─ glTranslatef(0.0, 1.0, 0.0);
412      │ glRotatef(guy.partsRot[0][0], 1,0,0);
413      │ glRotatef(guy.partsRot[0][1], 0,1,0);
414      │ glRotatef(guy.partsRot[0][2], 0,0,1);
416      │ guy.pointer->getObjectsByName("Head")->draw();
417      │ } glPopMatrix();
-1 : └─ } glPopMatrix();
418
419      glPushMatrix(); {

```

*Change 121, second translate has sign reversed, is moved after stack*

```

413      glPushMatrix(); {
415      └─ glRotatef(guy.partsRot[0][0], 1,0,0);
416      │ glRotatef(guy.partsRot[0][1], 0,1,0);
417      │ glRotatef(guy.partsRot[0][2], 0,0,1);
414      │ glTranslatef(0.0, 1.0, 0.0); -> glTranslatef(0.0, -1, 0.0);
418      │ guy.pointer->getObjectsByName("Head")->draw();
419      │ } glPopMatrix();

```

*Change 122, stack and second translate moved before first translate*

```

411      glPushMatrix(); {
414      └─ glRotatef(guy.partsRot[0][0], 1,0,0);
415      │ glRotatef(guy.partsRot[0][1], 0,1,0);
416      │ glRotatef(guy.partsRot[0][2], 0,0,1);
417      │ glTranslatef(0.0, -1, 0.0);
413      │ glPushMatrix(); {
412      │ │ glTranslatef(0.0, 8.0, 0.0);
418      │ │ guy.pointer->getObjectsByName("Head")->draw();
419      │ │ } glPopMatrix();
420      │ } glPopMatrix();

```

**Figure 208: Experimentation with the head's transform from 119-122**

At change 130 the student discovers the correct assembly method, moving the torso's transformations into the head's outer push/pop block (see Figure 209), thereby creating a hierarchical transformation. In the same change, the student also correctly reduces the head's y-translate value, taking into account the chest's y-translate to position the head in relation to the chest (as opposed to in relation to the global coordinate system as was the case before).

```

411: 411      glPushMatrix(); {
423: 412          glTranslatef(0.0, 6.0, 0.0);
424: 413
425: 414          glTranslatef(0.0, -3.0, 0);
426: 415          glRotatef(guy.partsRot[1][0], 1,0,0);
427: 416          glRotatef(guy.partsRot[1][1], 0,1,0);
428: 417          glRotatef(guy.partsRot[1][2], 0,0,1);
429: 418          glTranslatef(0.0, 3.0, 0);
430: 419          guy.pointer->getObjectsByName()["Torso"]->draw();
-1: 420
422: 421      glPushMatrix(); {
412: 422          glTranslatef(0.0, 8.0, 0.0); ->          glTranslatef(0.0, 2.0, 0.0);
413: 423
414: 424          glTranslatef(0.0, -1.0, 0);
415: 425          glRotatef(guy.partsRot[0][0], 1,0,0);
420: 430      } glPopMatrix();
421: 431
431: 432      } glPopMatrix();

```

**Figure 209: The student discovers the correct assembly method**

From change (130-151), the student attempts to apply the hierarchical construction to other limbs. During this segment, the student makes three axis errors, initially getting most of the reassembly translates incorrect, which is somewhat surprising considering that the student had previously constructed the avatar. The student incorrectly translates the arm along the x instead of the z-axis at (132), incorrectly translates the leg along the z instead of the y-axis at (137), and incorrectly translates the leg along the x-axis at (139).

At one point the student also shows he has not completely internalised the hierarchical assembly method. At (134) he makes an error when attaching the lower to the upper arm. Instead of translating the lower arm half-way onto the joint, orienting, and then moving it the second half of the way, the pre-orientation translate calls cancel each other out (see Figure 210). As a result the arm is oriented without being placed at the joint and then moved half-way, placing the 'joint' at the upper arm's centre as shown in Figure 211. He then tweaks the translate values (starting with a tweak of 0.1) at (135, 147-149, 151) slowly moving the joint away from the upper arm. However, considering the student knows the dimensions of both arm segments from the initial construction, it seems that this tweak approach may indicate that he still has not fully developed a conceptual understanding of the effect of the hierarchical transforms, instead relying on a trial-and-error approach based on the successful past implementation of the head-torso assembly.

```

glPushMatrix(); {
    glTranslatef(0.0, 1.0, -1.5);

    glTranslatef(0.0, 0.0, 1.4);
    glRotatef(guy.partsRot[2][0], 1,0,0);
    glRotatef(guy.partsRot[2][1], 0,1,0);
    glRotatef(guy.partsRot[2][2], 0,0,1);
    glTranslatef(0.0, 0.0, -1.4);
    guy.pointer->getObjectsByName() ["RightUpperArm"]->draw();

    glPushMatrix(); {
        glTranslatef(0.0, 0.0, -1.3);

        glTranslatef(0.0, 0.0, 1.3);
        glRotatef(guy.partsRot[3][0], 1,0,0);
        glRotatef(guy.partsRot[3][1], 0,1,0);
        glRotatef(guy.partsRot[3][2], 0,0,1);
        glTranslatef(0.0, 0.0, -1.3);
        guy.pointer->getObjectsByName() ["RightLowerArm"]->draw();
    } glPopMatrix();
} glPopMatrix();

```

Figure 210: Assembly implementation at (134)



Figure 211: Error in (134) caused by pre and post-translates cancelling each other out

At (151) the student completes the proper assembly of all limbs, creating a fully functional avatar. The student moved from a naïve non-hierarchical assembly approach to a non-hierarchical assembly approach allowing the rotation of a limb around a joint at (123) to finally discovering a hierarchical assembly approach with correct joint-rotation at (130). As the timeline shown in Figure 212 shows, time between Changes for this segment tended to be long. All but three Changes exceeded the median time per Change for this student (68 seconds per Change), suggesting the student was engaging in spatial reasoning between versions.

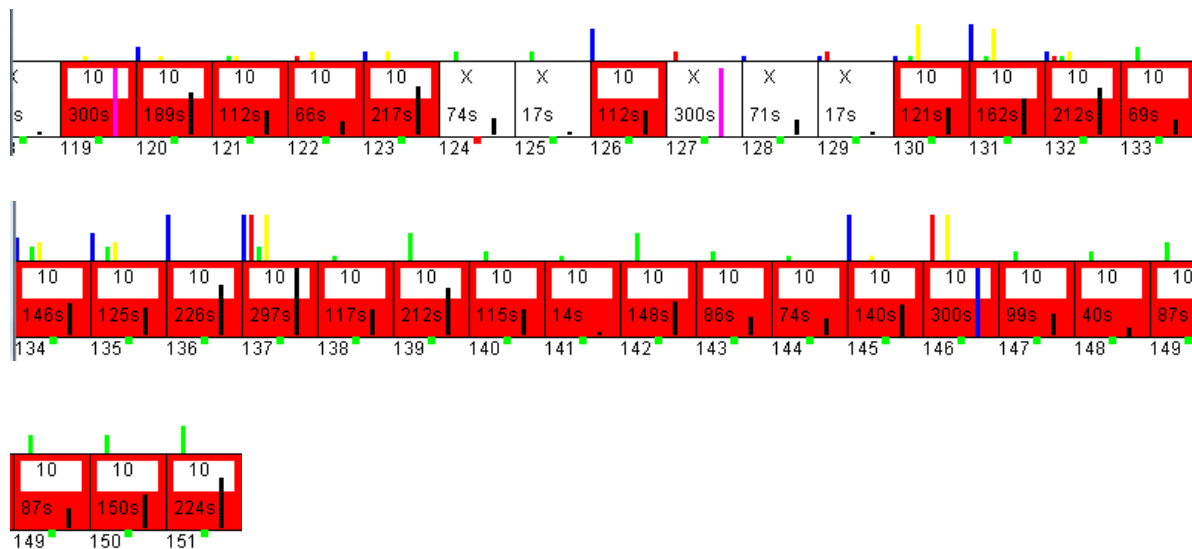


Figure 212: Timeline for Changes implementing the proper assembly method

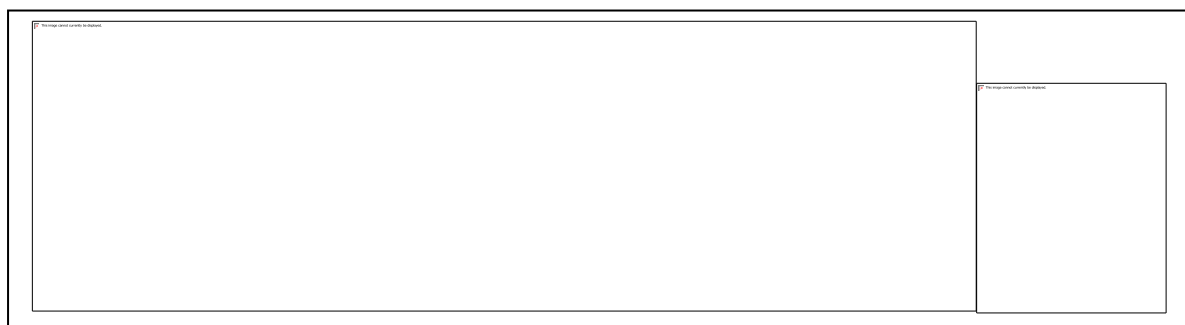
### 9.7.7.1.1.2.3 Thomas

#### Naïve Assembly

Thomas's initial naïve avatar assembly implementation is contained in segment THOMAS\_A3.SP.5.1."Naive Assembly" [39]. The avatar's assembly is the first task the student works on after commencing the development of his assignment project.

The student first positions the chest, head and upper arms from (7-10) with one axis error.

At (11) the student assembles the lower arm, using a hierarchical assembly method as shown in Figure 213, correctly placing the lower arm's translate statement inside the upper arm's push/pop transformation block. This contrasts with John, Christopher and Michael who used non-hierarchical approaches in their initial assembly methods. However, the student apparently does not understand the way in which the upper and lower arm's translation statements will be composited as he translates the lower arm by (0,1,-1.5), meaning that it is moved towards the chest rather than away from it (because of the negative z-value) and moved above the upper arm (because of the y-value).



**Figure 213: At Change 10, the student creates a hierarchical assembly but incorrectly translates the arm back to the origin ( $z = -1.5$ ) and up ( $y = 1.5$ )**

The student then experiments with the lower arm's translate call from (12-18, see Figure 214), moving the lower arm back to the origin (13), then incorrectly translating it first along the y-axis (14), then the x-axis (15-16), then again the y-axis at (17) before removing the lower arm altogether at (18) (see Figure 215).

(1296): (11, 13-18 : total = 7)
11, ADDED(O): glTranslatef(0.0, 1.0, -1.5);
13, MUTATED(O): glTranslatef(0.0, 0.0, 0.0);
14, MUTATED(O): glTranslatef(0.0, 1.0, 0.0);
15, MUTATED(O): glTranslatef(1.0, 0.0, 0.0);
16, MUTATED(O): glTranslatef(2.0, 0.0, 0.0);
17, MUTATED(O): glTranslatef(0.0, 1.0, 0.0);
18, DELETED(O): DELETED (glTranslatef(0.0, 1.0, 0.0);)

Figure 214: Line History showing the modification to the lower arm's translate statement from 13-18

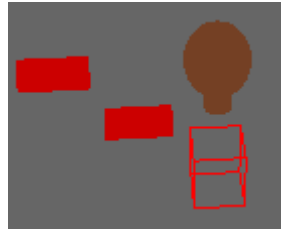


Figure 215: Effect of experimentation with the lower-arm translate statement

This indicates the student is struggling with understanding the composition of hierarchical transformations. Despite his initial approach at (10) being essentially correct, he has not understood how to translate the lower arm into the correct position relative to the upper arm based on the upper arm's local coordinate system.

The student first stops all but the head, upper and lower arm limbs from being rendered, presumably to be able to better observe the effect of transformations on these limbs. The student then begins to work on the lower arm again from (20), where he copies the upper arm's translate in front of the lower arm, which correctly translates it away from the upper arm on the z-axis but incorrectly translates it above the upper arm on the y-axis, producing the assembly shown in Figure 216. The student corrects the translate call from (21-22), removing the y-dimension and leaving the arm correctly constructed.





**Figure 216: At (20), the student incorrectly copies the upper arm's (0, 1.0, 1.5) translate in front of the lower arm.**

When attempting to copy the correctly constructed left arm to create the right arm at (24) (Figure 217 shows the effect of incorrect transform calls, Figure 218 shows the text of the incorrect calls), the student reverses the arm translate call's y dimension instead of the z-dimension. He corrects the upper but not the lower arm's translate call in (25). In an attempt to correct the error the student incorrectly inverts the upper arm's y-translate (26), before correcting both the upper arm's y-translate and the lower arm's z-translate at (27), at which point both the left and right lower and upper arms are correctly assembled.



**Figure 217: Effect of incorrect translations at (24,25,26) and correction at (27)**

(1307): (23-27, 29-30, 66, 241, 359-360 : total = 11)	
23, ADDED(O):	glTranslatef(0.0, -1.0, 1.5);
24, MUTATED(O):	glTranslatef(0.0, 1.0, -1.5);
25, MUTATED(O):	glTranslatef(0.0, -1.0, -1.5);
26, MUTATED(O):	glTranslatef(0.0, 1.0, -1.5);
27, MUTATED(O):	glTranslatef(0.0, 0.0, 1.5);
29, MUTATED(O):	glTranslatef(0.0, 0.0, 0.5);
30, MUTATED(O):	glTranslatef(0.0, 0.0, 1.0);

**Figure 218: Line History containing the right arm's translate statement**

The student adds the left palm and tweaks it into position from (27-30). The student also removes the right arm for reasons that are not apparent.

The student begins adding the right arm back by copying the left arm's transformations at (31) then attempts to move it to the right at (32) but reverses the y instead of the z-value, again making an axis error with a fairly simple spatial problem. He corrects this error at (33) by reversing the z instead of the y-value for the upper arm.

At (34) rather than reversing the lower arm's and the palm's z-value, he instead applies a 180-degree x-rotation to the arm as shown in Figure 219, thereby placing it into the correct position. However as Figure 220 shows, this unintentionally rotates the local y-coordinate axis downward and also twists the x-axis around itself, meaning an x-rotate applied to the right arm will have the opposite effect to an x-rotate about the left arm (the same goes for a y-translate). This will have unintended consequence when the student later returns to implement a new proper assembly for limbs. With the arms appearing to be correctly positioned, the student moves on to leg assembly.

```
glPushMatrix();
glTranslatef(0.0, 1.0, -1.5);
glRotatef(180,1.0,0.0,0.0);
glObjects.at(objectAt)->getObjectsByName()["RightUpperArm"]->draw();
glPushMatrix();
glTranslatef(0.0, 0.0, 1.5);
glObjects.at(objectAt)->getObjectsByName()["RightLowerArm"]->draw();
glPushMatrix();
glTranslatef(0.0, 0.0, 1.0);
glObjects.at(objectAt)->getObjectsByName()["RightPalm"]->draw();
```

Figure 219: A rotation used to transform the left arm's assembly transformations to produce the right arm

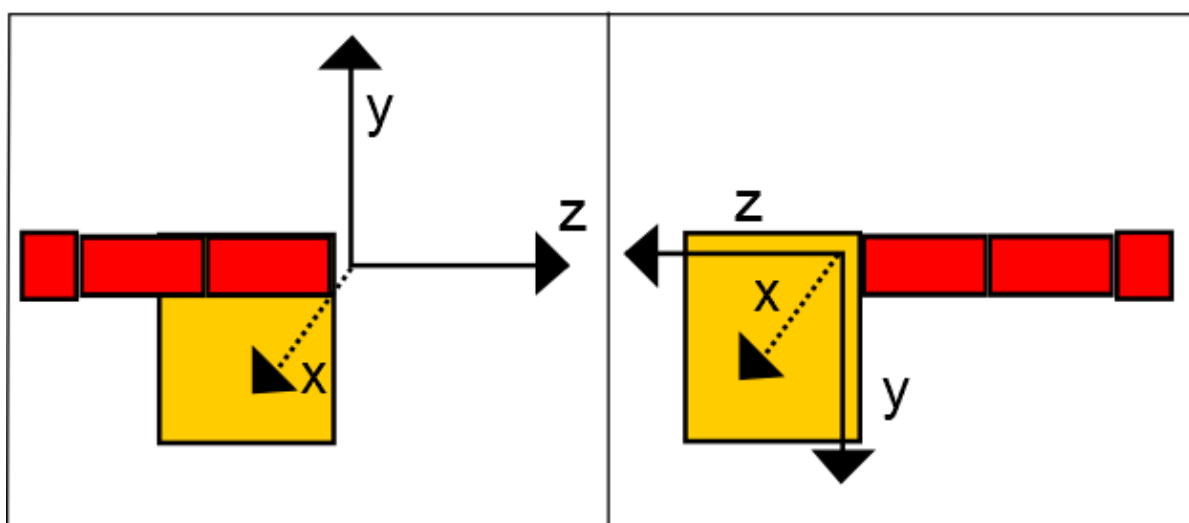
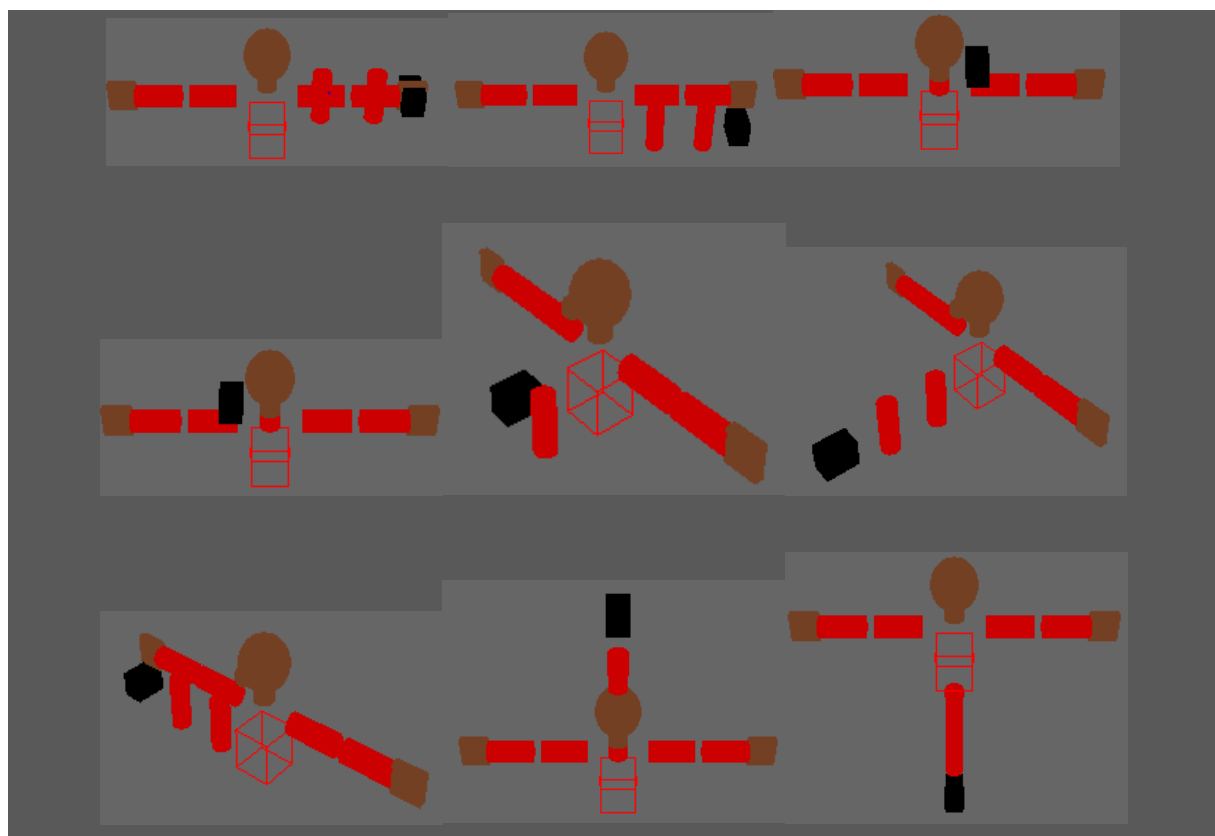


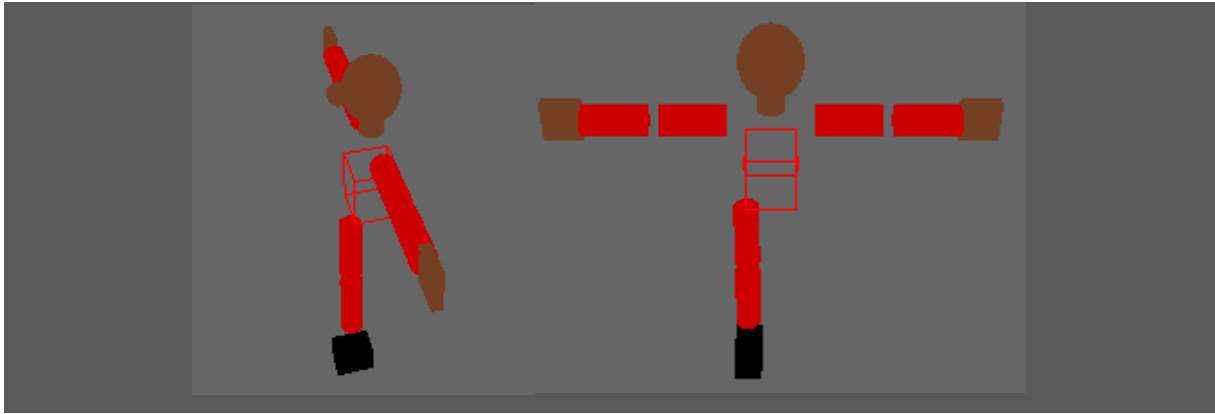
Figure 220: Creating the arm by rotating it rotates the arm's local axes which will cause transformations based on global axes to not have the desired effect

When beginning the leg assembly, the student again demonstrates a poor understanding of space. While building the leg (see Figure 222 for screen captures), he initially copies the right arm's translate calls, including the 180-degree x-rotation, and does not correct the z or y translates, leading to the legs being positioned on top of the arms. He then removes the upper leg's y-translate (keeping the z-translate) at (36), before changing the upper leg's y-translate to a y-translate, and making the lower leg's translate (0,0,0), meaning it will have the same position as the upper leg.

(1330): (35-37, 39, 41-45,: total = 9)	
35, ADDED(O):	glTranslatef(0.0, 1.0, -1.5);
36, MUTATED(O):	glTranslatef(0.0, 0.0, -1.5);
37, MUTATED(O):	glTranslatef(0.0, 1.5, 0.0);
39, MUTATED(O):	glTranslatef(1.5, 0.0, 0.0);
41, MUTATED(O):	glTranslatef(0.0, 0.0, 1.5);
42, MUTATED(O):	glTranslatef(0.0, 1.5, 0.0);
43, MUTATED(O):	glTranslatef(0.0, -1.5, 0.0);
44, MUTATED(O):	glTranslatef(0.5, -1.5, 0.0);
45, MUTATED(O):	glTranslatef(0.0, -1.5, 0.5);

Figure 221: Line History containing the leg translate statement





**Figure 222: Incorrect translations during assembly from (35-44), with the final correct translation at (45)**

The student removes the rotate call at (38), before incorrectly changing the upper and lower leg's and foot's y-transform to an x-transform from (39-40). He then again changes the transform to a z-transform (thereby having incorrectly tried both the x and the z-axis, as well as the y-axis in the wrong direction) at (41), before changing it to a positive y-transform at (42). At (43) the student changes the positive to a negative y-transform for all leg limbs, properly positioning them along the y-axis.

However, when attempting to move the leg left into position at (44), the student translates along the (wrong) x-axis, translating it forward instead, before correcting and completing the assembly at (45).

At (46) he copies the leg assembly from the left to the right leg, correctly changing the z-value for the right upper leg's translate call.

Finally, from (52-54) the student adds a rotate call in front of the leg to produce 'orientation'/rotation of the leg limb (see Figure 223). The student tries rotating it about the x, y and z axis in turn as shown in Figure 224, presumably in order to test leg rotation since he has not implemented any keyboard keys to rotate limbs. The incorrect assembly is apparent as the leg rotates about the origin rather than the torso-leg joint. However, the student either does not understand this or chooses to ignore the problem for now, only returning to the incorrect assembly after attempting to implement the first animation at (297-310).

```

/**/
glPushMatrix();
glRotatef(90,0.0,1.0,0.0);
/**/
glPushMatrix();
glTranslatef(0.0, -1.5, -0.5);
glObjects.at(objectAt)
->getObjectsByName() ["RightUpperLeg"]->draw();
glPushMatrix();
glTranslatef(0.0, -1.5, 0.0);
glObjects.at(objectAt)
->getObjectsByName() ["RightLowerLeg"]->draw();
glPushMatrix();
glTranslatef(0.0, -1.5, 0.0);
glObjects.at(objectAt)
->getObjectsByName() ["RightFoot"]->draw();
glPopMatrix();
glPopMatrix();
glPopMatrix();
/**/
glPopMatrix();
/**/

```

Figure 223: A rotate command added in front of the leg's assembly block to enable leg rotation

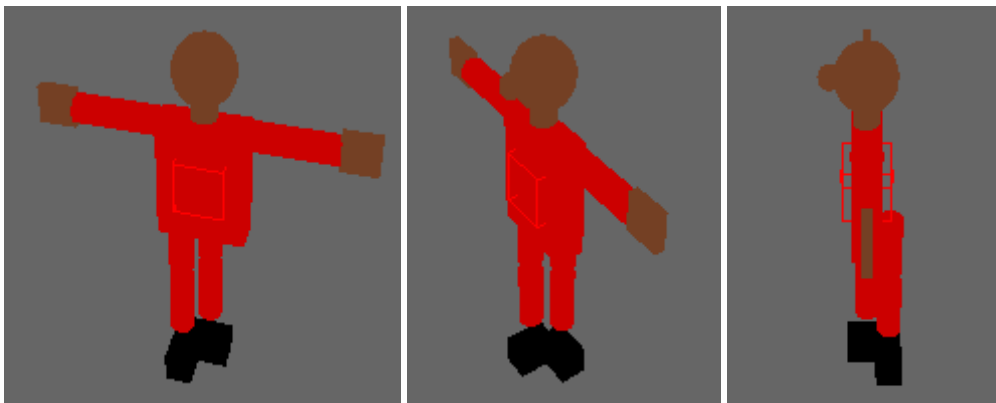


Figure 224: A view of the incorrect rotation of the leg about the origin at (53)

### Proper Assembly

It is during the implementation of the first animation that the student realises the assembly of the avatar is incorrect, rotating limbs about their parent's centres instead of the proper child-parent joint. Implementation of the proper assembly is contained in segment [THOMAS\\_A3.SP.5.2."Proper Assembly" \[55\]](#).

From (310-319) the student experiments with moving the leg translate call from after the rotate block to before the rotate block and then back again several times (see Figure 225). The effect of these modifications to the leg's transformations is shown in Figure 226.

```

glPushMatrix();
glTranslatef(0.0, -1.5, 0.5);
glRotatef(limbRot[LEFTUPPERLEG][0],1.0,0.0,0.0);
glRotatef(limbRot[LEFTUPPERLEG][1],0.0,1.0,0.0);
glRotatef(limbRot[LEFTUPPERLEG][2],0.0,0.0,1.0);
animateBody(LEFTUPPERLEG);

gObjects.at(objectAt)->getObjectsByName()["LeftUpperLeg"]->draw();

```

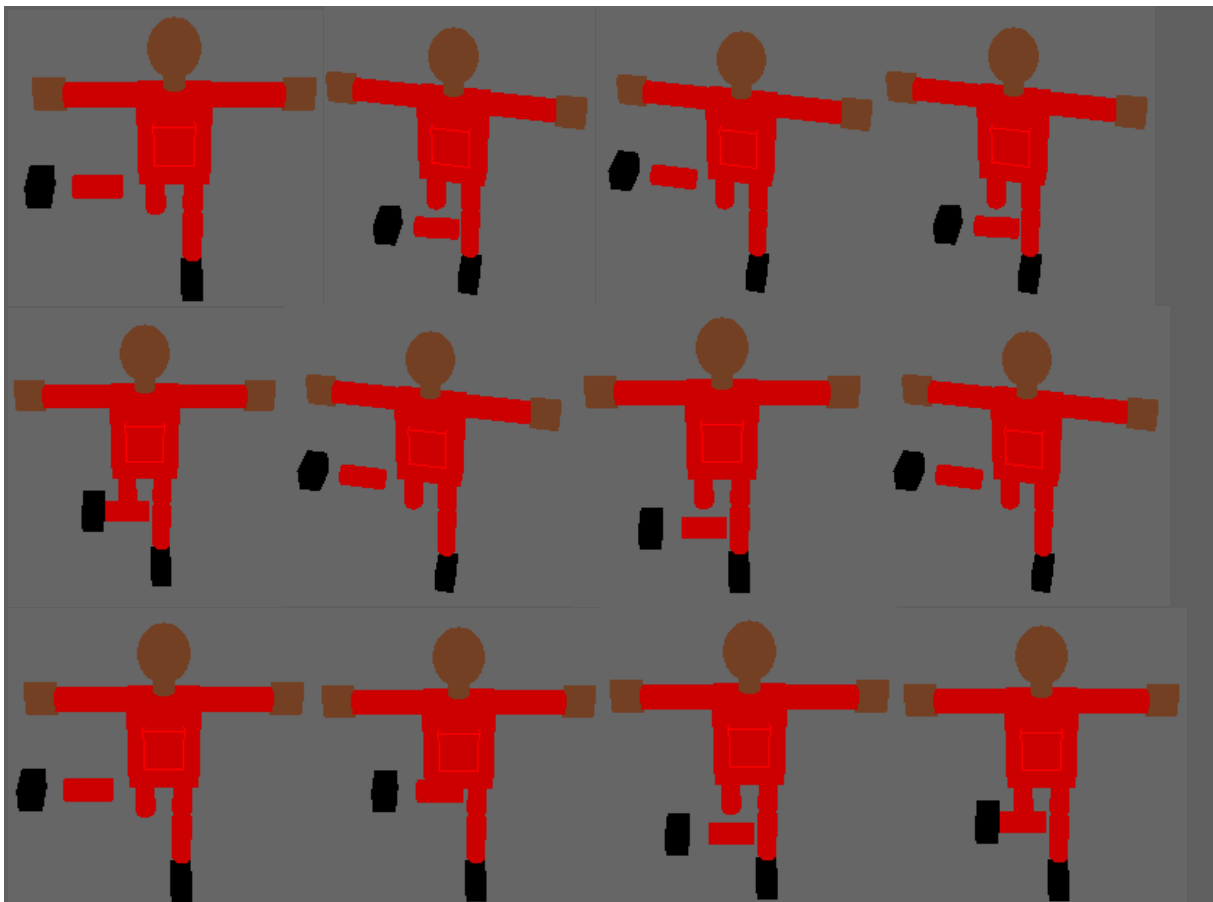
```

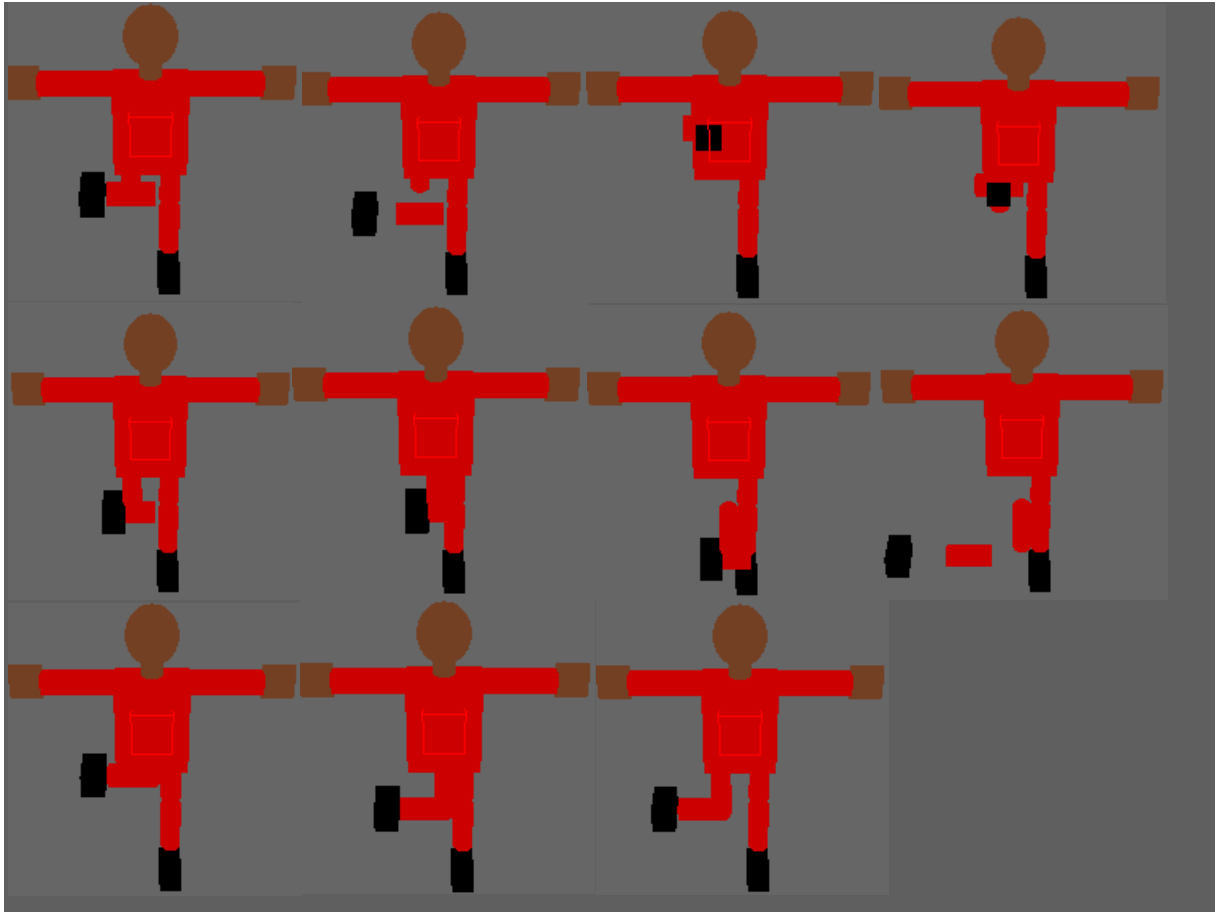
glPushMatrix();
glRotatef(limbRot[LEFTUPPERLEG][0],1.0,0.0,0.0);
glRotatef(limbRot[LEFTUPPERLEG][1],0.0,1.0,0.0);
glRotatef(limbRot[LEFTUPPERLEG][2],0.0,0.0,1.0);
animateBody(LEFTUPPERLEG);

glTranslatef(0.0, -1.5, 0.5);
gObjects.at(objectAt)->getObjectsByName()["LeftUpperLeg"]->draw();

```

Figure 225: The two assemblies (first introduced at (310) and (311)) the student switches between from (310-319)





**Figure 226: The steps of the student's construction of the left leg from (309-331)**

The student also experiments with tweaking the y-position of limbs (313, 320-322, 324, 326), moving the limb's centres closer to their limb-parent joint, but without adding a second translation to translate the limb away from the joint after orientation this simply places the child limb onto the parent limb.

At (327-328) the student adds a second transform to the leg limbs, mirroring the first translate call (see Figure 227). While this call is wrong and merely rotates the leg about the same incorrect point as earlier before translating it away even farther from that point, the addition of a second translate call puts the student on the path to the correct solution, which he achieves after tweaking the values of the translate calls from (329-330), achieving a completely correct leg assembly (Figure 228) after re-adding the z-component to the upper leg's initial translate (thereby placing it on the left-hand side). The construction method is copied correctly to the right leg at (332).

```

glPushMatrix();
glTranslatef(0.0, -1.5, 0.0);
glRotatef(limbRot[LEFTUPPERLEG][0],1.0,0.0,0.0);
glRotatef(limbRot[LEFTUPPERLEG][1],0.0,1.0,0.0);
glRotatef(limbRot[LEFTUPPERLEG][2],0.0,0.0,1.0);
animateBody(LEFTUPPERLEG);
glTranslatef(0.0, -1.5, 0.0);
glObjects.at(objectAt)->getObjectsByName()["LeftUpperLeg"]->draw();

```

Figure 227: At (327) the student adds a second translate call, mirroring the first

```

366 void drawLeftLeg() {
367     glPushMatrix();
368     glTranslatef(0.0, -1.5, 0.0);
369     glRotatef(limbRot[LEFTUPPERLEG][0],1.0,0.0,0.0);
370     glRotatef(limbRot[LEFTUPPERLEG][1],0.0,1.0,0.0);
371     glRotatef(limbRot[LEFTUPPERLEG][2],0.0,0.0,1.0);
372     animateBody(LEFTUPPERLEG);
373     glTranslatef(0.0, -0.5, 0.0);
374     glObjects.at(objectAt)->getObjectsByName()["LeftUpperLeg"]->draw();

```

Figure 228: Proper assembly of the upper leg at (330)

From 334-367, the student works on applying the construction method to the other limbs. When adding translate calls to the right arm (334), the student wrongly translates limbs by the y (down) instead of the z-axis (right) as he did for the leg.

At (335-336) the student reveals his incomplete conceptual understanding of the correct assembly method discovered during the leg's assembly. He changes the upper arm's translate to (0,0,0) and then (0,0,-0.5), and adds the comment "limb is at centred origin, need to move it so that the edge is at origin". The student understands the need to move the limb in order to center it on the 'joint', but is aiming to first rotate it at the origin before moving it into position, going back to the first incorrect conception / solution approach of the composition of transformations at (310). This indicates the understanding of transformation composition is not yet internalised despite the discovery of the correct method. The incorrect assembly leads to the limb being positioned wrongly as shown in Figure 229.



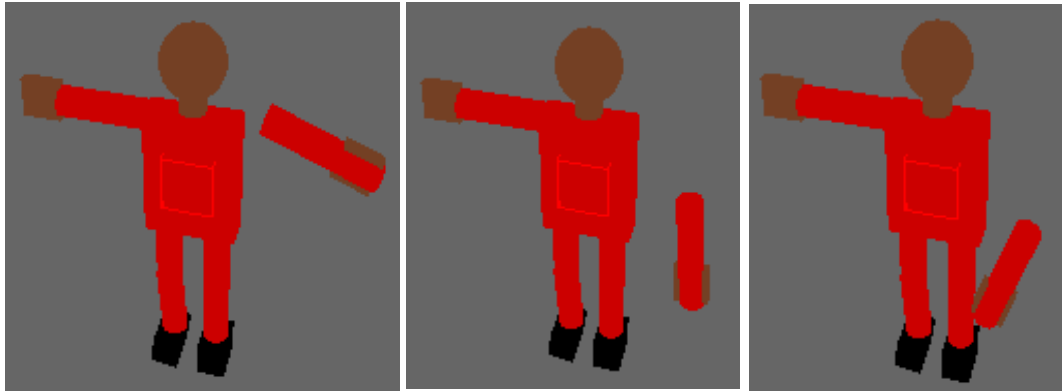


Figure 229: Screen captures of the resulting incorrect limb rotation at (336)

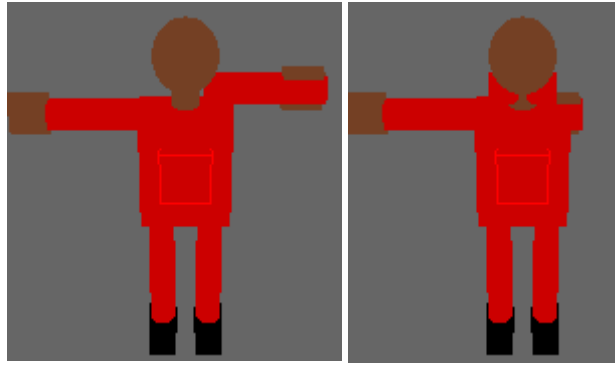
The student corrects the assembly method to position the upper arm prior to rotating it from (337-340). However, the student has been using positive z-translates to position the right arm's limbs from (335) onward. This would be correct, except that during the initial assembly the student used a `glRotate` call (line 346 in Figure 230) to mirror the left arm to create the right arm, which means the right arm's z-axis is reversed compared to the global coordinate system as is shown in Figure 220. This leads to the post-rotate lower arm and palm being translated onto their parents, leading to the incorrect construction shown in (Figure 231).

```

340:340      glTranslatef(0.0, 1.0, -1.5);
341:341      glRotatef(limbRot[RIGHTUPPERARM][0],1.0,0.0,0.0);
342:342      glRotatef(limbRot[RIGHTUPPERARM][1],0.0,1.0,0.0);
343:343      glRotatef(limbRot[RIGHTUPPERARM][2],0.0,0.0,1.0);
344:344      animateBody(RIGHTUPPERARM);
345:345      glTranslatef(0.0, 0.5, 0.0); -> glTranslatef(0.0, 0.0, 0.0);
346:346      glRotatef(180,1.0,0.0,0.0);
347:347      glObjects.at(objectAt)->getObjectsByName()["RightUpperArm"]->draw();
348:348      glPushMatrix();
349:349      glTranslatef(0.0, 0.5, 0.0); -> glTranslatef(0.0, 0.0, 0.5);
350:350      glRotatef(limbRot[RIGHTLOWERARM][0],1.0,0.0,0.0);
351:351      glRotatef(limbRot[RIGHTLOWERARM][1],0.0,1.0,0.0);
352:352      glRotatef(limbRot[RIGHTLOWERARM][2],0.0,0.0,1.0);
353:353      animateBody(RIGHTLOWERARM);
354:354      glTranslatef(0.0, 0.0, 0.5);
355:355      glObjects.at(objectAt)->getObjectsByName()["RightLowerArm"]->draw();

```

Figure 230: The almost-correct arm construction at (335), foiled by the upper arm's additional (180, 1,0,0) rotate call



**Figure 231: The use of +z instead of -z values leads to the affected right arm's limbs (lower arm and palm) being translated onto their parent. Screen captures of the construction at (340, 342).**

The student does not understand the state/orientation of the upper arm's (pre-rotate) local coordinate system in relation to the lower arm's (post-rotate) coordinate system, as he struggles with the direction of the z-translation from (335-355). Instead of reversing the z-coordinate of the translate calls after the `glTranslate(180,1,0,0)` call, the student reverses the z-coordinate of the call preceeding the rotation, thereby merely translating the upper arm further into the wrong direction at (341). At (342) the student modifies the upper arm's pre-rotation translation from (0, 0, 0.5) to (0, 0.5, 0), introducing a further error by removing the upper arm's correct z-rotation.

The student removes and then adds back the `glRotate(180,1,0,0)` call at (343-344), indicating he is aware of the source of the problem. However, despite understanding the source of the problem the student does not have an understanding of the composition of transformations that would allow him to apply the correct solution (reversing the sign of the z-translates after the rotate call).

Instead, the student reverses the z-translate before the rotate call at (345) from -z to z before changing it back from z to -z at [346] (see Figure 232 for the effect of this and following Changes) . He removes part of the first translate at (347), then includes an x-translate in the first translate at (348). At (350) the student returns to the erroneous reversal of the first translate's z-value first attempted at (350). While he correctly undoes this change at (351), he simultaneously incorrectly reverses the y-transform. In (352), he correctly undoes the y-change, but again incorrectly reverses the z-axis, leading back to the erroneous state of (350). From (353-355) he finally removes the `glRotatef(180,1,0,0)` call, thereby avoiding having to understand the effect of that call on the lower arm and palm's local coordinate system.

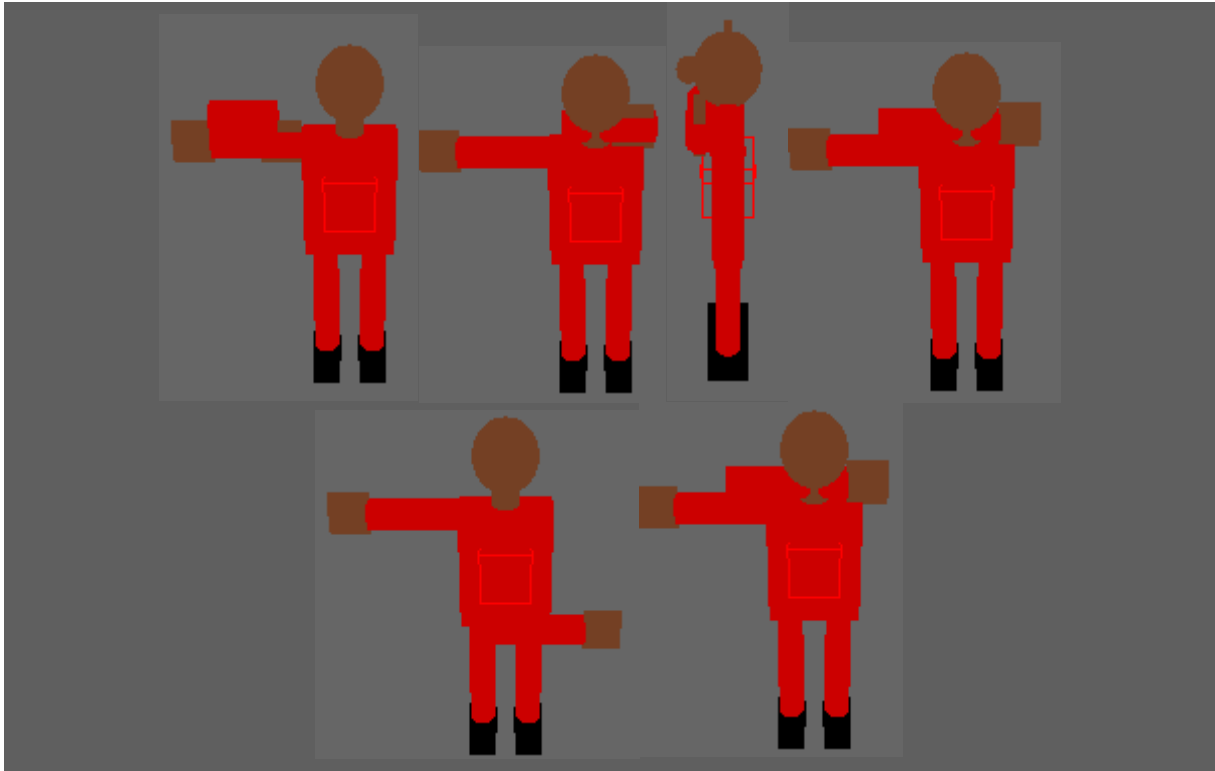


Figure 232: Changes (345), (347), (348), (350), and (351) (which is identical to 348)

After removing the rotate call, the student very quickly implements the correct arm assembly, initially correctly reversing the z-coordinate for the arm's limbs but simultaneously incorrectly inverting the y-coordinate (356) before correcting that error in (357) and tweaking the y-value for the upper arm in (358). The correct assembly is copied to the left arm from (359-360). Finally the student assembles the head without error from (361-362). This completes the student's work on avatar assembly.

### Summary

Unlike other students, the student's first implementation was hierarchical. However, it was joint-naïve and the student required a long period of experimentation and thought to implement a proper assembly implementation. Even after having discovered the correct method of assembly by assembling the leg, the student struggled when applying that method to the arm. The student also struggled to understand the effect of a `glRotate` call on the right arm, ultimately removing the call altogether after a long period of unsuccessfully trying to compensate for the change to the local coordinate system caused by the call.

#### 9.7.7.1.1.2.4 Ida

##### Naïve assembly

The student's initial avatar assembly is captured in segment [IDA\\_A3.SP.3.1."Naive Assembly" \[24\]](#). The student begins the avatar construction with an assembly method that already correctly includes

hierarchical push/pop statements (see Figure 233) but does not properly position limbs at the parent-child joint for rotation, leading limbs to rotate about their own centres.

```
glPushMatrix();{
    glRotatef(rotation[lu_arm][xc],1,0,0);
    glRotatef(rotation[lu_arm][yc],0,1,0);
    glRotatef(rotation[lu_arm][zc],0,0,1);
    glObjects.at(objectAt)->getObjectsByName()["LeftUpperArm"]->draw();

glPushMatrix();{
    glRotatef(rotation[ll_arm][xc],1,0,0);
    glRotatef(rotation[ll_arm][yc],0,1,0);
    glRotatef(rotation[ll_arm][zc],0,0,1);
    glObjects.at(objectAt)->getObjectsByName()["LeftLowerArm"]->draw();
```

**Figure 233: The student's initial assembly is already properly hierarchical**



**Figure 234: The student incorrectly adds an x-translate to the assembly**

The student makes two axis errors early on (48-50) while placing the upper arm. Aside from a further minor error at (63) when the student reverses both the x and z axes instead of just the z-axis when copying the left leg's assembly to the right leg, the rest of the assembly of arms, legs and head proceed without error. The remainder of changes is taken up with minor tweaks, revealing the need to show students how to better measure on-screen distances. All in all, the student assembled the avatar very quickly and efficiently when compared to the other students.

### **Proper Assembly**

*Arm Construction* The problem with the naïve assembly becomes apparent as the student begins to implement animations, with the upper arms rotating about their own centres rather than the torso-arm joint (see Figure 235) so the student returns to working on the avatar assembly in segment [IDA\\_A3.SP.3.2."Proper assembly" \[28\]](#).

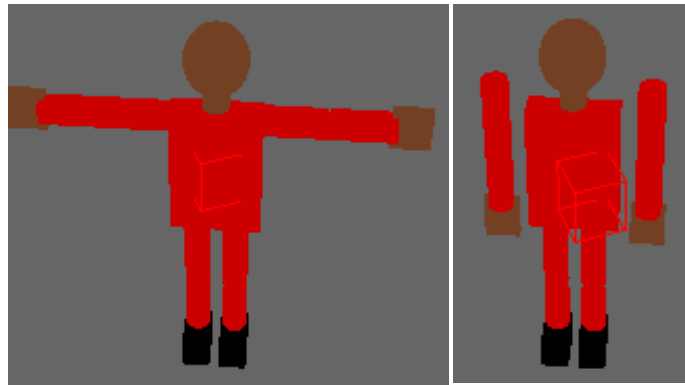


Figure 235: At (83), the incorrect centres of rotation (incorrect joints) become apparent

Screen captures for selected Changes to accompany the following description are shown in Figure 236.

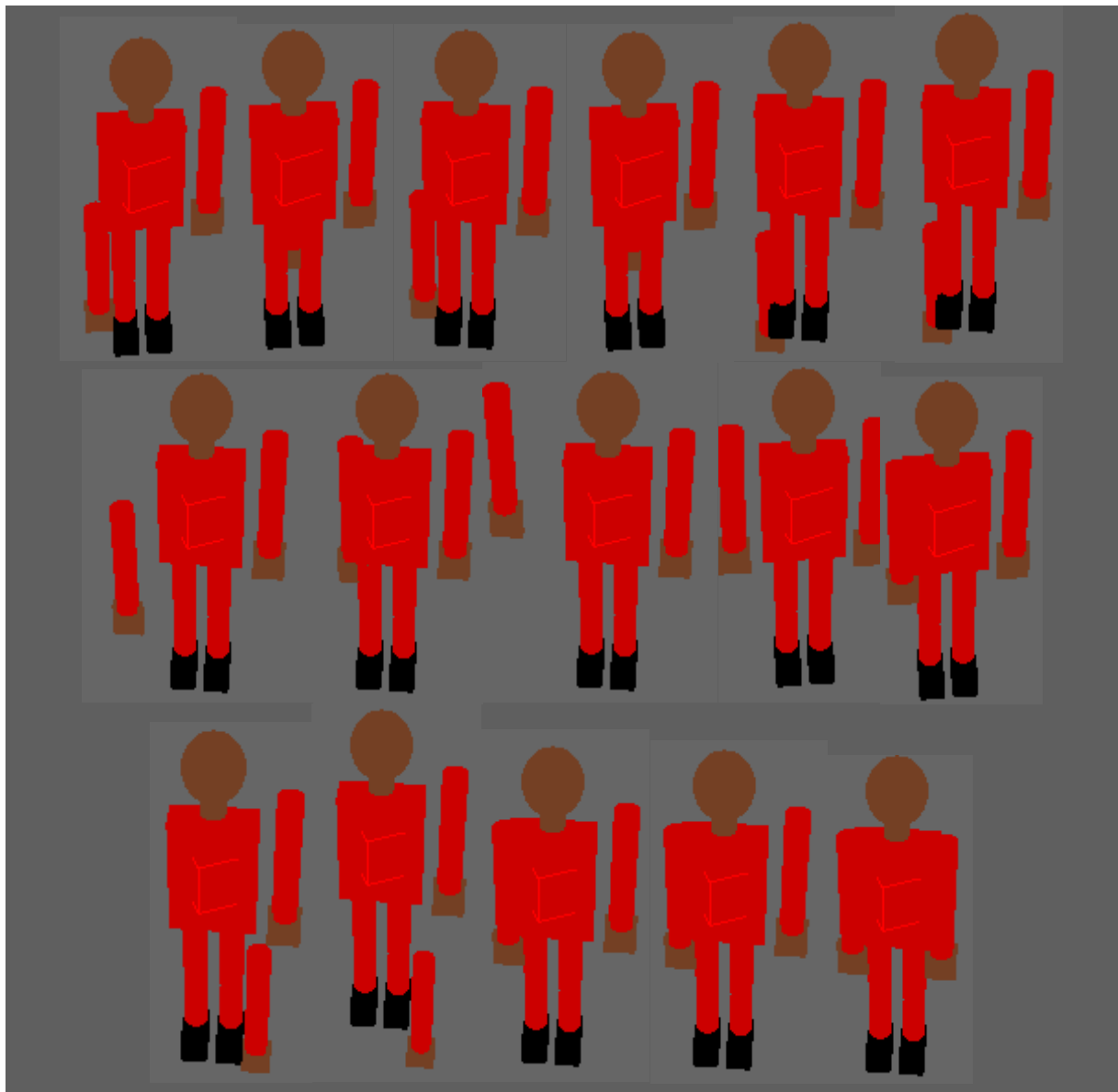


Figure 236: Screen capture of work on arm rotation at (84-88, 90-91, 95-103)

Initially, the student experiments with moving the translate call from before to after the orientation block in (84).

After removing the translate call altogether (85), probably to observe the effect of the translate call on the limb's rotation, the student adds pre-post translates to the limb (see Figure 237) using the body's position. The student is attempting to translate the limb towards the origin using the body's position, rotate it, and then translate it away from the origin (back to the original point) using the body's position before finally applying the limb's own translate call. The underlying idea of moving the local coordinate system via a pre-post translate to rotate it and move it back into position is correct, but the attempt to use the origin as the point of rotation is not, as the limb will be rotated about the origin instead of the proper joint. Furthermore, attempting to translate the limb by the body's position would rotate the limb not about the body's local coordinate system, but about the global coordinate system's origin.

```
glTranslatef(-position[0],-position[1],-position[2]);  
glRotatef(rotation[ru_arm][xc],1,0,0);  
glRotatef(rotation[ru_arm][yc],0,1,0);  
glRotatef(rotation[ru_arm][zc],0,0,1);  
glTranslatef(position[0],position[1],position[2]);  
glTranslatef(0,1.0,1.5);  
glObjects.at(objectAt)->getObjectsByName()["RightUpperArm"]->draw();
```

**Figure 237: The upper arm's pre/post translates added to the upper arm rotation assembly (86)**

The student removes the initial (limb-placing) translate call at (87), which will rotate and place the limb at the origin without moving it at all.

At (88-89) the student correctly replaces the body's translate value in the pre/post translate calls with the limb's translate value divided by 2 (thereby placing the limb at the 'joint') but incorrectly uses both the limb's y and z values in the translation, instead of just the z-value, thereby misplacing the joint. The student correctly removes the y-translate at (91) (see Figure 238) but also incorrectly reverses the pre/post translate's z-translations (from -z/+z to +z/-z), thereby translating the limb away from the joint for the rotate, then back towards the joint after the rotate, which in fact puts the actual joint farther away from the correct joint. The initial limb translate is also still wrongly placed after the rotate block instead of before it.

```

380 glPushMatrix();
381     glTranslatef(0,0,0.75);
382     glRotatef(rotation[ru_arm][xc],1,0,0);
383     glRotatef(rotation[ru_arm][yc],0,1,0);
384     glRotatef(rotation[ru_arm][zc],0,0,1);
385     glTranslatef(0,0,-0.75);
386     glTranslatef(0,1.0,1.5);
387     glObjects.at(objectAt)
388         ->getObjectsByName() ["RightUpperArm"]->draw();

```

**Figure 238: Assembly at (91) after removal of the y-translate**

After implementing keyboard keys to manually rotate limbs (92-93) which allows the student to better observe the effect of rotation on limbs, the student first removes (incorrect) and then moves the initial translate to before the orientation block (correct) at (95-96). The student then removes the pre/post translates at (97) before adding them back at (98) and simultaneously correctly reversing the pre/post z-coordinates, creating a working arm construction including proper rotation about the parent-child joint as shown in Figure 239.

```

385 glPushMatrix();{
386     glTranslatef(0,1.0,1.5);
387     glTranslatef(0,0,-0.5);
388     glRotatef(rotation[ru_arm][xc],1,0,0);
389     glRotatef(rotation[ru_arm][yc],0,1,0);
390     glRotatef(rotation[ru_arm][zc],0,0,1);
391     glTranslatef(0,0,0.5);
392     glObjects.at(objectAt)
393         ->getObjectsByName() ["RightUpperArm"]->draw();

```

**Figure 239: Proper arm construction at (98)**

The student then breaks the arm assembly at (99-100) by modifying the pre/post translate calls; the reason for this is unclear (since the assembly was working) but indicates a misunderstanding of the spatial composition used to achieve the correct assembly. The student returns to the correct assembly at (101). The left arm is correctly assembled by copying the right arm's construction at (102-103).

*Leg Construction (121-131)* The student then applies the upper arm construction method to the legs and the head from (121-131), making some errors as shown in Figure 240. Most changes are spent making correct additions of translate calls or tweaks to existing calls, with two sign errors occurring when adding pre/post translate calls at (122) and (128). The assembly is also applied to the head at 126-127 as shown in Figure 241.

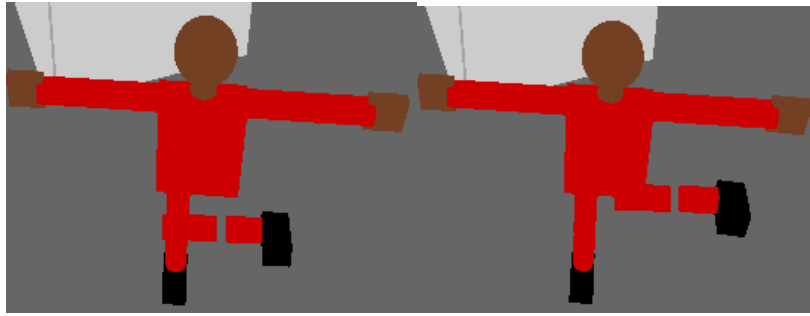


Figure 240: Leg assembly at (121) and (122)



Figure 241: Head assembly at (126) and (127)

## Summary

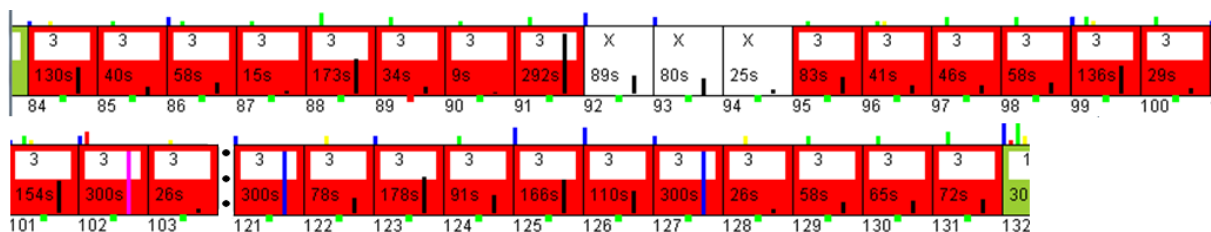


Figure 242: Timeline for Ida's proper assembly of avatar

Unlike John, Michael and Christopher, Ida's initial avatar assembly is hierarchical. However, as is the case with all students it is joint-naïve, rotating limbs about their own centres rather than about the parent-child joint.

The student experiments with moving translation calls before discovering the correct solution involving a pre and post-translate onto the joint. While the student discovers the proper approach more quickly than other students, she nevertheless experiments with multiple incorrect solutions before discovering the correct solution.

### 9.7.7.1.1.2.5 Michael

The student creates a naïve avatar construction in segment [MICHAEL\\_A3.SP.1."Initial Assembly"](#) [47] from (143-189) (see Figure 243), making only a single error early on at (145). Other than that, all



limbs are correctly assembled, with many tweaking changes occurring in the segment (see Figure 244, 159-182).

```

301 glPushMatrix();
302 glTranslatef(0.0, 4.9, -1.5);
303 glObjects.at(objectAt)
304     ->getObjectsByName() ["LeftUpperArm"]->draw();
305 glPopMatrix();

```

Figure 243: The student's initial naïve construction at (182)

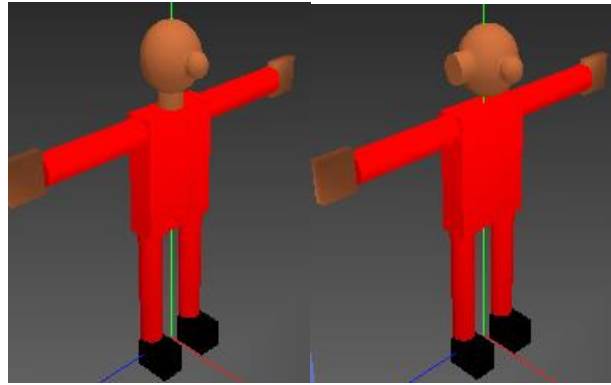
(2009): (28, 44-45, 156, 158-163, 182, 430, 434 : total = 13)	
28, ADDED(O):	glTranslatef(0.0, 1.0, 2.0);
44, MUTATED(O):	glTranslatef(5.0, 1.0, 2.0);
45, MUTATED(O):	glTranslatef(0.0, 1.0, 2.0);
156, MUTATED(O):	glTranslatef(0.0, 0, 0);
158, MUTATED(O):	glTranslatef(0.0, 0, -2.0);
159, MUTATED(O):	glTranslatef(0.0, 5.0, -2.0);
160, MUTATED(O):	glTranslatef(0.0, 7.0, -1.5);
161, MUTATED(O):	glTranslatef(0.0, 6.0, -1.5);
162, MUTATED(O):	glTranslatef(0.0, 5.7, -1.5);
163, MUTATED(O):	glTranslatef(0.0, 5.9, -1.5);
182, MUTATED(O):	glTranslatef(0.0, 4.9, -1.5);

Figure 244: The upper arm's assembly modifications from (28-182)

The student also places limbs one dimension at a time (see Figure 244, changes 156-159) which may have to do with the student having difficulty thinking about more than one dimension at a time.

### Semi-Naïve Assembly

The student becomes aware that the assembly is incorrect after implementing keys for rotating the head, since the head rotates about its own centre as shown in Figure 245.

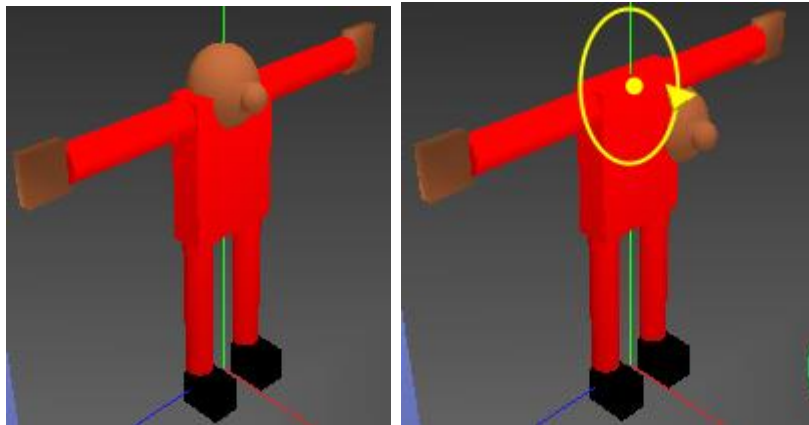


**Figure 245: The incorrect assembly becomes apparent at (310-317) after the student implements keys to rotate the head**

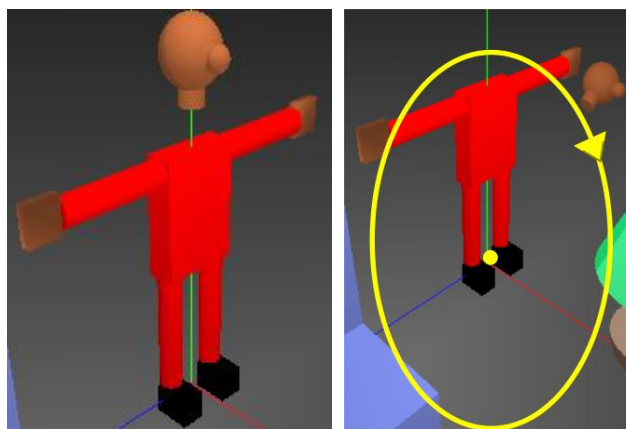
The student begins by adding a second transform to the head after the orientation block, and then tweaking the first translate until the two translates add up to roughly the same value as the original head translate from (319-321) (see Figure 246, Figure 247), but the first translate does not correctly position the head on the head-torso joint, which is obvious since the pre and post-translate do not add up to the value the student had used earlier to position the head. This shows the student does not yet understand how to properly position the head onto the joint for rotation, since the student knows the dimensions of the joints from the initial naïve assembly and could work out the necessary translate statements if he understood the method. The student then modifies the head assembly to rotate the head about the origin as shown in Figure 248.

```
glPushMatrix();
//glTranslatef(0,5.9,0);
glTranslatef(0.0, 4.0, 0.0);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,1,0);
glObjects.at(objectAt)->getObjectsByName()["Head"]
->draw();
glPopMatrix();
```

**Figure 246: First attempt at rotation about a joint at (321)**



**Figure 247: Rotation of the head about the misplaced joint at (321)**



**Figure 248: Rotation about the origin at (334)**

The student experiments with removing the second translate as well as with tweaking the y-value of both the first and the second value, but always uses incorrect translate statements (322-333). At (334-335) the student tries adding back the original translate in front of the orientation block where it is added to the existing pre-translate, translating the head even higher than in the initial naïve construction.

The student tweaks this translation from (336-338), moving the head to the correct torso-head joint and allowing for proper rotation (see Figure 249). However, while the student has in fact created a correct joint rotation, the student has not implemented a hierarchical transformation for the avatar, meaning that limbs will not rotate together (the head is rotating about the global point at which it meets the torso in the initial assembly, not about the local point at which it meets the torso).

```

glPushMatrix();
//glTranslatef(0,5.9,0);
//glTranslatef(0.0, 1.0, 0.0);
glTranslatef(0.0, 5.15, 0.0);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
//glTranslatef(0,5.9,0);
glObjects.at(objectAt)
    ->getObjectsByName() ["Head"]->draw();
glPopMatrix();

```

**Figure 249: The corrected assembly, with the pre and post-translate adding up to the head's original translate (first commented-out translate)**

From change (339-374) the student implements manual rotation of all limbs via key strokes. This will allow the student to observe that the avatar's construction is incorrect and non-hierarchical, since the rotation of an upper limb will not rotate the associated lower (child) limbs.

**380:**

```

glPushMatrix();
glTranslatef(0.0, 5.15, 0.0); -> glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.75,0);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName() ["LeftPalm"]->draw();
glPopMatrix();

```

**385:**

```

glPushMatrix();
glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.75,0); -> //glTranslatef(0,0.75,0);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName() ["LeftPalm"]->draw();
glPopMatrix();

```

**386:**

```

glPushMatrix();
glTranslatef(0.0, 4.9, -3.7); -> //glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
//glTranslatef(0,0.75,0); -> glTranslatef(0,0.75,0);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();

```

**387:**

```

glPushMatrix();
//glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.75,0); -> glTranslatef(0,0,0);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();

```

**389:**

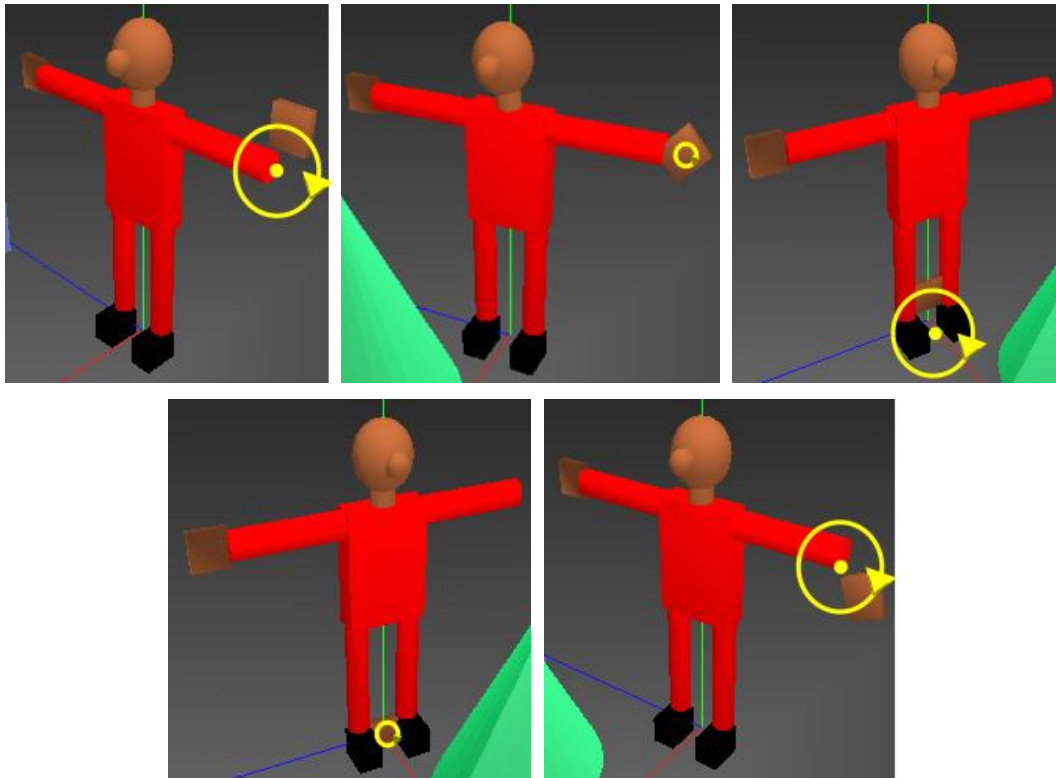
```

glPushMatrix();
//glTranslatef(0.0, 4.9, -3.7); -> glTranslatef(0.0, 4.5, -3.3);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.4,-0.4);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
glPopMatrix();

```

**Figure 250: Experimentation with addition and removal of pre and post-translates in the placing of the left palm at (380 & 384, 385, 386, 387, 389)**

The student works on the proper joint orientation of the palm from (375-390) (see Figure 251 for screen captures for some of these changes), still without implementing a hierarchical model. However, since the student is working on extremities this problem does not become apparent, as no child limbs will be put out of place by their rotation.



**Figure 251: Effect of the moving of translation statements at (380 & 384, 385, 386, 387, 389)**

From (375-380) the student copies the head's orientation rotate block and the pre/post translate calls across to the palm, but after realising that this places the palm at the head, the student replaces the first translate call with the palm's original translate call (without taking into account the need to translate the palm onto the joint) and keeps the second translate call intact, showing a poor spatial understanding of the method the student had developed to position and orient the head. The resulting incorrect rotation is shown in the first panel of Figure 251.

After implementing keys for orienting the palm from (381-384) the student removes the palm's second translate call (that is still unchanged from when it was copied from the head translate call, and hence wrong), thereby having the palm rotate about its own centre again rather than about the correct joint. The student removes the palm's pre-translate and adds back its post-translate in (386), then removes the post-translate in (387), all of which lead to incorrect rotations as shown in Figure 251.

In (388) the student adds back the post-translate, correctly including a z-dimension coordinate in the correct direction, but incorrectly adding a y-coordinate, which shows the student is not properly visualising the positioning of the limb (see Figure 252). This post-translate will position the palm at the lower end of the arm, as becomes apparent after the student adds back the pre-translate

(correctly deducting the amount added at the post-translate) at (389). The student never corrects this error, instead copying the assembly to the right palm at (390).

```
glPushMatrix();
//glTranslatef(0.0, 4.9, -3.7);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0,0); -> glTranslatef(0,0.4,-0.4);
//glTranslatef(0.0, 4.9, -3.7);
glObjects.at(objectAt)->getObjectsByName()["LeftPalm"]->draw();
```

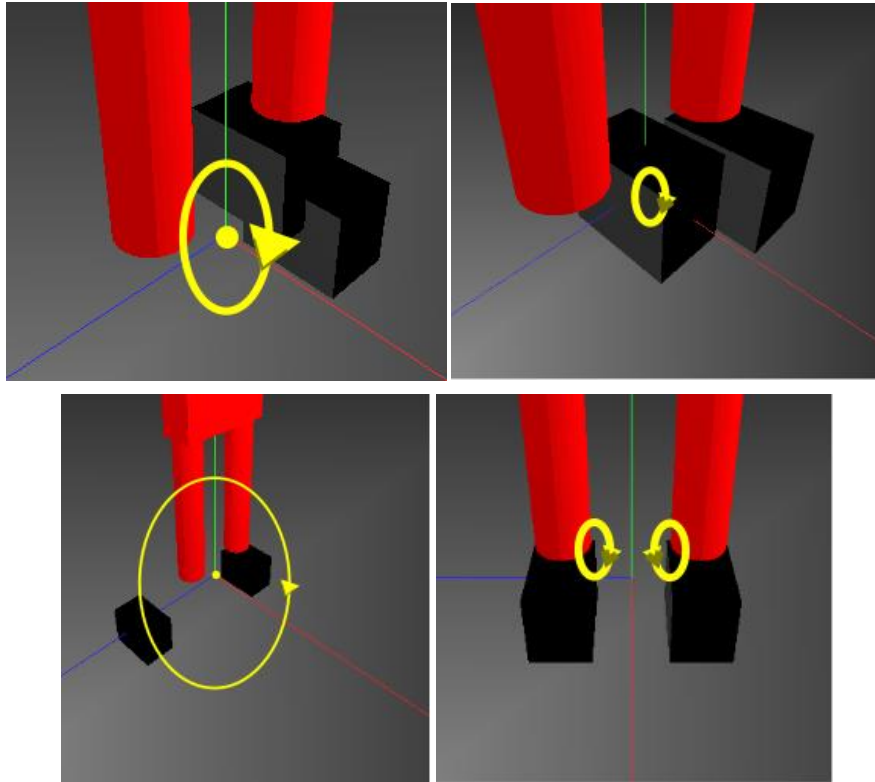
**Figure 252: Adding a y and z dimension to the post-translate at (388)**

(Feet, 404-411) The student works on applying the palm construction technique to the feet as shown in Figure 253. Figure 254 shows the effect of experiments with the foot assembly. Initially at (404), the student removes the pre-translate and adds a post-translate; however, the translate incorrectly translates along the y-axis. The student removes the post-translate at (405), leaving the foot to rotate about the origin, before adding it back at (407) with an incorrect x-translate and a correct z-translate. After tweaking these values from (408-410) the student adds a pre-translate to place the feet at the correct position in relation to the legs.

```
glPushMatrix();
glTranslatef(0.3, 0.0, 0.65); -> //glTranslatef(0.3, 0.0, 0.65);
//glTranslatef(0.0, 4.5, -3.3);
glRotatef(lpx, 1, 0, 0);
glRotatef(lpy, 0, 1, 0);
glRotatef(lpz, 0, 0, 1);
glTranslatef(0,0.4,-0.4);

glObjects.at(objectAt)->getObjectsByName()["RightFoot"]->draw();
glPopMatrix();
```

**Figure 253: Foot assembly at (404)**



**Figure 254: Rotation of the foot caused by different assemblies at (404, 405, 407 and 411)**

The student does not attempt to enable rotation for the non-extremity limbs (the upper/lower arms and legs). It is likely that he realises that the rotation of limbs with child limbs would fail to rotate the child limbs properly, but he does not attempt to find a correct (hierarchical) solution to the problem, and the proper construction of other limbs is never attempted.

#### **Avatar Movement (Part of Semi-Naïve assembly)**

(Avatar movement, 428-435) The student spends (412-427) working on Viewing before returning to avatar assembly, captured in segment [MICHAEL\\_A3.SP.3."Partially Proper Assembly" \[71\]](#). The student attempts to implement avatar movement by summing the x/z coordinates of limb's translate statements with move variables as shown in Figure 255. This moves each limb into place individually before it is then individually rotated/oriented. This approach shows the student has still not developed an understanding of how to utilise glPush/glPop statements to build hierarchical models.



```

glPushMatrix();
//glTranslatef(0,5.9,0);
//glTranslatef(0.0, 1.0, 0.0);
glTranslatef(0.0+movex, 5.15, 0.0+movez);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
//glTranslatef(0,5.9,0);
glObjects.at(objectAt)->getObjectsByName()["Head"]->draw();
glPopMatrix();

```

**Figure 255: Enabling avatar movement at (434); move variables highlighted in yellow**

(*whole-body orientation/rotation 436-532*) The student next attempts to add a whole-body orientation/rotation block to the avatar. Using hierarchical transformations, this would be simple to achieve as shown in Figure 256 by adding a whole-body translate and rotate call.

<p>PUSH BODY STACK          TRANSLATE TO BODY_POSITION          ROTATE BODY</p> <p>(For each limb, do: )          PUSH LIMB STACK          TRANSLATE TO LIMB JOINT          ROTATE LIMB JOINT          TRANSLATE TO LIMB POSITION          POP LIMB STACK</p> <p>POP BODY STACK</p>
---

**Figure 256: An approach to proper whole-body positioning and orientation**

But since the student does not have a conceptual understanding of how to composite transformations to create hierarchical transformations he does not develop this straight-forward solution.

At (440-441) the student begins by surrounding the head's transformation calls with an if/else conditional block (see Figure 257), which uses the head's standard translation into place (developed earlier to orient the head about the head-chest join) for any orientation mode except for body orientation mode, and uses a simple post-translate when in body-orientation mode. This leads to the head being oriented about the origin when in body-orientation mode, with the rest of the body staying in place, as shown in the left panel in Figure 258. The student removes the if/else clause again at (445), removing all the changes made from (440-445).

```

if (sltbody)
{
    glRotatef(headx, 1, 0, 0);
    glRotatef(heady, 0, 1, 0);
    glRotatef(headz, 0, 0, 1);
    glTranslatef(0+movex,5.9,0+movez);
}

else {
glTranslatef(0.0+movex, 5.15, 0.0+movez);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
}

```

Figure 257: The if block enables body-orientation, the else block enables orientation of the head only at (444)

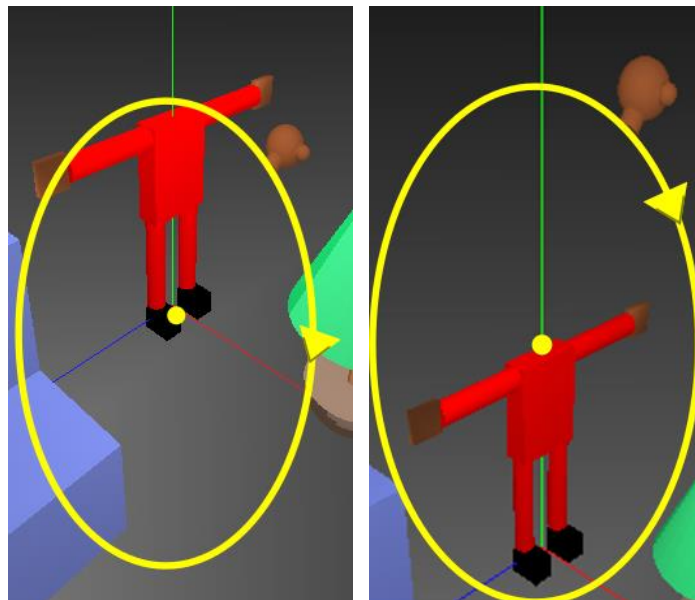


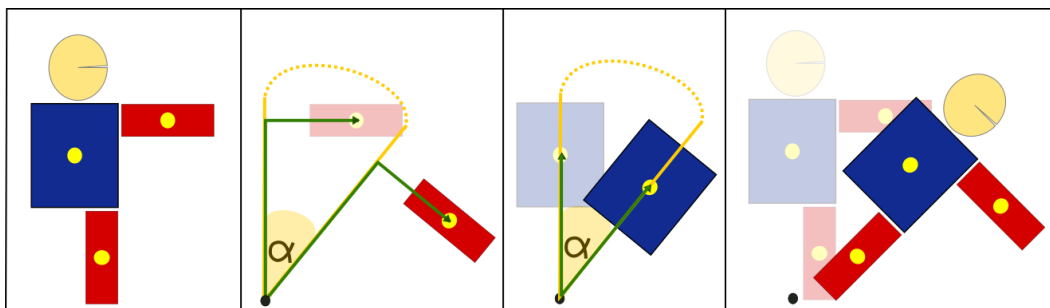
Figure 258: Effect of the body orientation on the head's rotation at (444) and (446)

At (446) the student changes the post-translate from the correct value of (0,0.75,0) to (0,5.9,0), which is a summation of the pre and post translate calls, which results in the head being placed at twice the correct height, but still orienting about the correct point. This error is a repeat of the same experiment in (437).

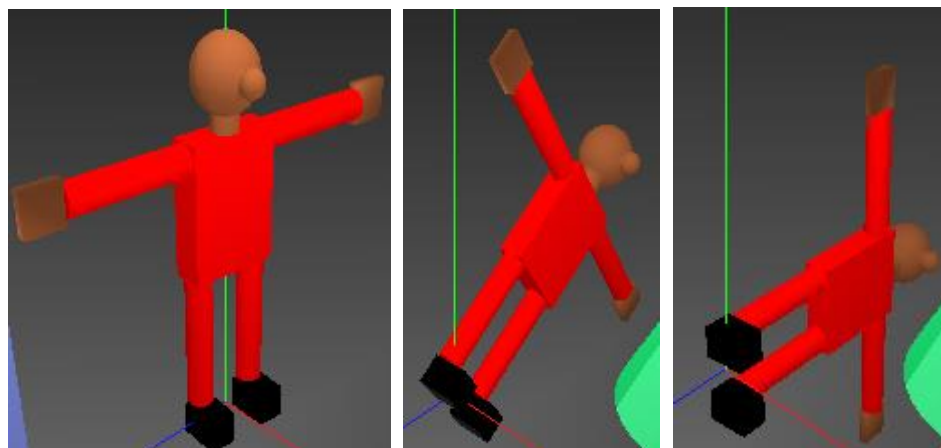
From (447-450) the student again adds an if/else if block, copying the head's correct transform block (pre/post translates and orientation block) into both the if and the elseif conditional clauses, which means that the head is positioned and oriented as usual if in head or body mode, but is left at the origin and not oriented when in any other limb mode.

At (451-452) the student again combines the pre/post translates for the body's conditional clause as he did in change (444), leading to the same incorrect result. (The student switches from using an elseif block to using an else block in [452]).

The student addresses the problem of the head rotating by itself from (453-457) by adding an orientation block using the head's rotate variables. The same block is then added to all other limbs from (458-487). The resulting assembly will apply the same pre-rotation to all limbs before the limb's individual translation call when body orientation is selected as shown in Figure 259. This has the same effect as a single rotation call with the same value preceding all statements would have had, rotating the whole avatar about the origin as shown in Figure 260. The student does not understand how compositing of transformations works, so he does not apply this simpler and better solution.



**Figure 259: Individual rotation and translation of limbs to achieve rotation of the whole avatar**



**Figure 260: Rotation of the body by individual rotation and translation of limbs at (487)**

This approach allows for both the rotation of the whole avatar OR for the rotation of individual extremities but not both at the same time. In what is likely an attempt to fix this shortcoming from (489-506) the student makes the keys that modify the head rotation variable simultaneously modify the body rotation variable by the same amount as shown in Figure 261. The student also switches

from using the head rotate variables to using body rotate variables in the body orientation as shown in Figure 262.

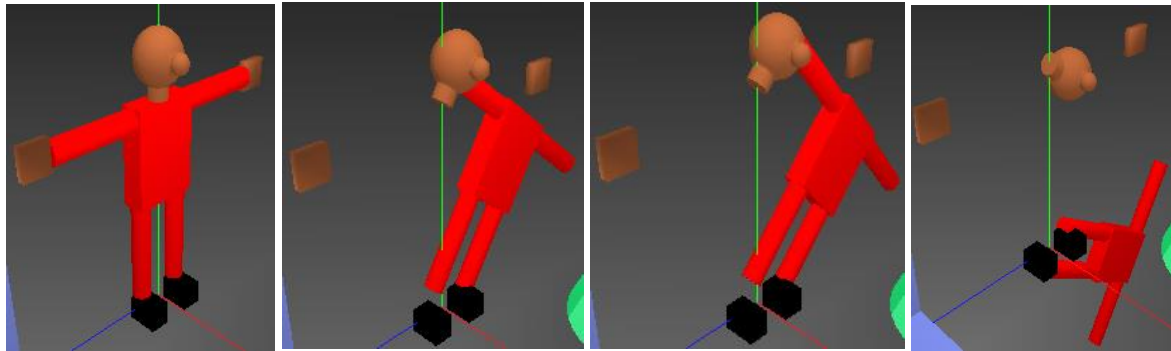
```
glPushMatrix();
if (sltbody){
glRotatef(headx, 1, 0, 0); -> glRotatef(bodyx, 1, 0, 0);
glRotatef(heady, 0, 1, 0); -> glRotatef(bodyy, 0, 1, 0);
glRotatef(headz, 0, 0, 1); -> glRotatef(bodyz, 0, 0, 1);
glTranslatef(0+movex,4.9,3.7+movez);
}
else {
glTranslatef(0.0+movex, 4.5, 3.3+movez);
glRotatef(rpx, 1, 0, 0);
glRotatef(rpy, 0, 1, 0);
glRotatef(rpz, 0, 0, 1);
glTranslatef(0,0.4,0.4);
}
glObjects.at(objectAt)->getObjectsByName()["RightPalm"]->draw();
glPopMatrix();
```

Figure 261: Construction method at (506) using a universal 'body' rotate value for whole-avatar rotation

```
else if(key == '4')
{
    if (slthead || sltbody)
    {
        headx += 5;
        bodyx += 5;
    }
    else if (sltlpalm) lpx += 5;
    else if (sltrpalm) rpx += 5;
    else if (sltlfoot) lfx += 5;
    else if (sltrfoot) rfx += 5;
}
```

Figure 262: Rotation of the head or body increments both the head's and body's rotate value at (506)

This means that in body orient mode, the extremities are rotated along with the body. However, when another limb is selected the non-extremities maintain their body orientation, while the extremities use their own (head, palm, foot) orientations, leading to the extremities remaining at the origin while the non-extremities are oriented about the origin as shown in Figure 263.



**Figure 263: As the head rotates about its global 'joint' with the body, the body rotates away from the head at (505)**

By modifying the extremity and body variables by the same amount, the student is trying to achieve body rotation for both types of limbs, while simultaneously maintaining the separate rotation of extremities. The correct and simple solution to this problem would be to utilise a hierarchical assembly method, but the student does not develop an understanding of this concept. His alternative approach of manually applying the same transformations to individual limbs cannot be successful; an understanding of the effect of transformations on local coordinate systems would have informed the student that this approach is a dead end.

From (507-511) the reverts to the previous solution approach from (487), using the head rotate variable for all limbs in body orient mode, which allows either extremities or the whole body to be rotated.

```

glPushMatrix();
if (sltbody){
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0+movex,5.9,0+movez);
}
else{
glTranslatef(0.0+movex, 5.15, 0.0+movez);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
}
gObjects.at(objectAt)->getObjectsByName() ["Head"]->draw();
glPopMatrix();

glPushMatrix();
if (sltbody){
glRotatef(bodyx, 1, 0, 0);
glRotatef(bodyy, 0, 1, 0);
glRotatef(bodyz, 0, 0, 1);
}
glTranslatef(0.0+movex, 4.0, 0.0+movez);
gObjects.at(objectAt)->getObjectsByName() ["Torso"]->draw();
glPopMatrix();

```

Figure 264: Assembly at (522)

From (513-522) the student goes back to using the body variable for all non-extremities, and also modifies the keyboard keys to modify the body variable when in body mode, and the head/palm/foot variable when in head/palm/foot mode.

```

glRotatef(bodyx, 1, 0, 0);
glRotatef(bodyy, 0, 1, 0);
glRotatef(bodyz, 0, 0, 1);
glTranslatef(0.0+movex, 5.15, 0.0+movez);
glRotatef(headx, 1, 0, 0);
glRotatef(heady, 0, 1, 0);
glRotatef(headz, 0, 0, 1);
glTranslatef(0,0.75,0);
gObjects.at(objectAt)->getObjectsByName()["Head"]->draw();
glPopMatrix();

glPushMatrix();
glRotatef(bodyx, 1, 0, 0);
glRotatef(bodyy, 0, 1, 0);
glRotatef(bodyz, 0, 0, 1);
glTranslatef(0.0+movex, 4.0, 0.0+movez);
gObjects.at(objectAt)->getObjectsByName()["Torso"]->draw();
glPopMatrix();

```

Figure 265: Final construction at (532)

From (523-532) the student implements the final construction approach by removing the if/else conditionals and applying the body orientation block to all limbs (extremities and non-extremities), while applying the individual orientation blocks to extremities. This allows the extremity orientation to be applied at the same time as the body orientation as shown in Figure 266. However, the student cannot apply this approach to non-extremities, since it is not a hierarchical construction. This is masked by the fact that extremities have no children and hence their rotation will not break the avatar.

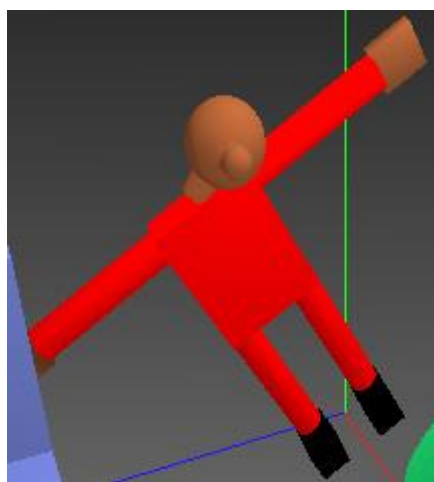


Figure 266: Rotation can be applied separately to the extremities and the body at (532)

The student shows his continued lack of development of spatial concepts from (533-601) while attempting to develop an animation. Given the construction approach used, a proper animation is impossible to achieve, so the student settles on applying a simple rotation to the entire avatar. However rather than using the existing assembly, he implements an entirely new assembly with a fixed rotate value which is called when the animation button is pressed (see Figure 267).

```
glPushMatrix();  
glRotatef(50, 0, 1, 0);  
glTranslatef(0,5.9,0);  
glObjects.at(objectAt)->getObjectsByName()["Head"]->draw();  
glPopMatrix();  
  
glPushMatrix();  
glRotatef(50, 0, 1, 0);  
glTranslatef(0.0, 4.0, 0.0);  
glObjects.at(objectAt)->getObjectsByName()["Torso"]->draw();  
glPopMatrix();
```

**Figure 267: Animation attempt involving the implementation of an entirely new avatar assembly at (589)**

### **Discussion**

While the student discovers the correct method for rotating limbs around their parent-child joints, the student never develops a hierarchical assembly preventing him from applying the approach to non-extremity limbs. His assembly method precludes any rotation of non-extremity limbs, as such rotation would cause the limb to rotate by itself away from the avatar and hence visibly break the avatar assembly.

In addition, the avatar's rotation occurs before its translation into place, leading to rotations applied to the avatar incorrectly rotating it about the origin from its global position.

The student spent considerable time in attempting to properly assemble the avatar, including lengthy experimenting with the positioning of transformation calls, but due to his failure to develop an understanding of the underlying concepts of local and global coordinate systems and the effect of the compositing of transformations he ended up utilising an incorrect dead-end assembly approach, applying individual rotations to each limb in order to orient the avatar.

#### **9.7.7.1.1.3 Animation Algorithm**

##### **9.7.7.1.1.3.1 Christopher**



The first implementation is added at 464-465 (see Figure 268). It is executed via a keyboard press. It lacks a timer and a call to `glutPostRedisplay()`. The lack of timer means that the entire loop will be executed at once resulting in an 'animation' too fast to see, whereas the lack of the `glutPostRedisplay()` call means that only the final 'frame' of the animation is displayed at  $i=60$  when the animation ends. This results in activation of the 'animation' simply rotating the head to an  $i-30=60-30=30$  degree angle. The translation of the head to the origin is also pointless and probably grounded in a student misperception about the compositing of transformations; it is removed in 468.

```

case 'r':
    scene->getNodeByName("Head")->setTranslate(0,0,0);
    scene->getNodeByName("Head")->setRotation(0,0,0);
    for (int i=0;i<60;i++) {
        scene->getNodeByName("Head")->setRotation(i-30,0,0);
    }

```

**Figure 268: Initial Attempt at Animation Algorithm at Change 465**

The student creates a separate head rotate function from 469-470 and attempts to use the `glutTimerFunc` timer function (the actual call is `glutTimerFunc(1/10,animateHeadShake*(i+1),0)`) to implement proper time-based behaviour for the animation, executing it at a controlled pace. While this approach is correct, the student still does not include a `glutPostRedisplay` call and also makes several errors regarding the passing of the function pointer to the `glutPostRedisplay` function. For the pointer to be successfully passed, the function should be static which it is not. The student also tries to pass parameters to the function pointer, which produces a further syntax error.

The student experiments with the timer function's syntax unsuccessfully from (471-476) without getting the code to compile. He then tries to remove the integer parameter  $i$  from the function and from the function pointer being passed to `glutTimerFunc` from 476-482. Removing the parameter from the function pointer is correct and addresses the related syntax error; however, the `glutTimerFunc` expects the function pointer to include an INT parameter, so its removal from the function itself causes a new syntax error. The student continues to experiment with different syntax from 482-489, 491-492 without getting the code to compile. At 490 the student correctly makes the function being passed static, but it is still missing the required INT parameter.

The student re-adds the required parameter at 493-494, which allows the program to compile. At Change (495) the algorithm is almost correct. However, the student is still missing the

glutPostRedisplay statement, causing only the final 'frame' of the animation to be shown. He experiments with removing the animation reset statement (setting the rotation amount to 0 at the beginning of the animation) but this does not address the problem.

The student correctly adds the glutPostRedisplay call at 498, and after some syntax correction produces the code shown in Figure 269. This code correctly produces a simple head-spinning animation and marks the point at which the student has mastered the required concepts relating to time-driven animation behaviour. From 500-513 the student does further work on the animation algorithm, adding a termination condition to terminate the animation after a certain amount of rotation has been applied.

```
500 static void animateHeadShake (int i) {  
501     scene->getNodeByName ("Head")->rotation.y+=1;  
502     glutPostRedisplay();  
503     glutTimerFunc(10, animateHeadShake, 0);  
504 }
```

Figure 269: The first working Animation Algorithm

#### 9.7.7.1.1.3.2 John

John implements an animation algorithm in segment [JOHN\\_A3.ANIMATION.1."Animation Algorithm"](#) [19].

The student first tries to implement an animation algorithm at (164-165) (see Figure 270). This initial attempt involves a single function which is to execute the entire animation (all of its 'frames'). The student adds two blocks of transforms and adds glutPostRedisplay() calls between them. However, since glutPostRedisplay() calls do immediately repaint the screen, it is unlikely that anything but the last set of transforms will be shown on screen; even if this were not the case, without a timer the different steps would execute so quickly as to not be visible to the user.

```

591 void Person::pick() {
592     reset();
593
594     /* initial position (relaxed...) */
595     guy.partsRot[PERSON_LUA][X] = -35;
596     guy.partsRot[PERSON_LLA][X] = -100;
597     guy.partsRot[PERSON_RUA][X] = 35;
598     guy.partsRot[PERSON_RLA][X] = 100;
599     guy.partsRot[PERSON_LP][Y] = -45;
600     guy.partsRot[PERSON_RP][Y] = 45;
601     glutPostRedisplay();
602
603     guy.partsRot[PERSON_RUA][Z] = 10;
604     guy.partsRot[PERSON_RP][Y] = 15;
605     glutPostRedisplay();
606 }

```

Figure 270: Initial animation approach at (165)

The student replaces the second block of transformations with a loop at (166) (see Figure 271) and includes a `glutPostRedisplay` call in the loop to show each ‘frame’ of the animation. However, the algorithm still suffers from the same problem; the `glutPostRedisplay` call only executes in breaks during execution, so it will not execute until after the loop is complete and will show only the final frame. A direct display call would still not show the proper animation, since without a timer it would execute too fast to be visible. The student decreases the increment by which limbs are rotated at 169 assuming that the rotation may be occurring too fast to be visible, but this does not resolve the real underlying problem.

```

591 void Person::pick() {
592     reset();
593
594     /* initial position (relaxed...) */
595     guy.partsRot[PERSON_LUA][X] = -35;
596     guy.partsRot[PERSON_LLA][X] = -100;
597     guy.partsRot[PERSON_RUA][X] = 35;
598     guy.partsRot[PERSON_RLA][X] = 100;
599     guy.partsRot[PERSON_LP][Y] = -45;
600     guy.partsRot[PERSON_RP][Y] = 45;
601     glutPostRedisplay();
602
603     unsigned int i;
604     for(i=0; i<15; i++ {})
605         guy.partsRot[PERSON_RUA][Z]++;
606         guy.partsRot[PERSON_RP][Y]++;
607         glutPostRedisplay();
608     }
609 }

```

**Figure 271: Addition of iterating loop in animation function**

The student adds a busy loop to the animation algorithm at 172-174 hoping to slow down the animation and make the individual frame visible (see Figure 272). However, he adds it at the wrong position in the program flow, between the initial transforms and the loop implementing the transformation, so even without the problems discussed earlier this approach would not work.

The student moves to using `glutIdleFunc` at (175), struggling with function pointer syntax from (176-186). However, the student does not use the `glutIdleFunc` properly, instead simply passing the broken animation function shown in Figure 272, which shows a misunderstanding of time-driven behaviour since the idle function should be used to produce individual frames, whereas the function called by the student is attempting to draw all frames belonging to the animation in one go.

```

600     guy.partsRot[PERSON_LP][Y] = -45;
601     guy.partsRot[PERSON_RP][Y] = 45;
602     glutPostRedisplay();
603
604     float i;
605     /* pause in this position for a while */
606     for(i=0; i<100; i+=INC) {}
607
608
609     for(i=0; i<15; i+=INC) {
610         guy.partsRot[PERSON_RUA][Z] += INC;
611         guy.partsRot[PERSON_RP][Y] += INC;
612         glutPostRedisplay();
613     }
614 }

```

Figure 272: Addition of a busy loop to 'time' the animation at (172)

The student implements a simple timer in the animation function at (188). However, as shown in Figure 273 the timer is again incorrectly placed between the block of initial transformations and the loop implementing the animation.

```

604     guy.partsRot[PERSON_RP][Y] = 45;
605     glutPostRedisplay();
606
607     while(glutGet(GLUT_ELAPSED_TIME) < timer+1000)
608         Sleep(1);
609     float i;
610     for(i=0; i<15; i+=INC) {
611         guy.partsRot[PERSON_RUA][Z] += INC;
612         guy.partsRot[PERSON_RP][Y] += INC;
613         glutPostRedisplay();
614     }

```

Figure 273: Implementation of a 'timer' using the C++ Sleep function at (188)

The student addresses this error at (190) (see Figure 274) by placing the Sleep call into the animation loop. This approach is almost correct. However, the glutPostRedisplay call will not execute a call to the display function while the loop is running, as it only calls display when processor time is available. Hence display will be called only after the loop terminates, again only showing the final frame of the animation. This problem could be addressed by a direct call to display, but the student does not discover this fix. Instead, between (191-195) the student adds glutSwapBuffer calls, modifies the Sleep time, the increment at which limbs are rotated as well as the number of iterations of the busy loop preceding the animation loop, all to no effect.

```

610     for(i=0; i<15; i+=INC) {
611         guy.partsRot[PERSON_RUA][Z] += INC;
612         guy.partsRot[PERSON_RP][Y] += INC;
613         glutPostRedisplay();
614         Sleep(1);
615     }

```

Figure 274: The Sleep function is placed into the animation loop at (190)

The student discovers a correct animation approach at (196-197). The animation function, which is passed as a static function pointer to `glutIdleFunc`, is modified to draw a single frame of the animation each time it is called instead of attempting to draw all of the animation's frames in one go. Hence each time OpenGL has 'idle' time, the animation function will be called to produce a single frame. The interval of time at which `glutIdleFunc` calls the animation function is large enough for the animation to appear smooth (though the student should not have been relying on this for the implementation). A conditional inside the function unbinds it from `glutIdleFunc` once the animation has completed execution. This approach is used as the core animation algorithm for the rest of the assignment and is shown in Figure 275.

```

345         case 'p': //guy.pick(); break;
346         glutIdleFunc(animate); break;

645 void animate(void){
646     //guy.reset();
647     static const float INC = 1;
648
649     if (guy.partsRot[PERSON_LUA][X] <= -100 ||
650         glutIdleFunc(NULL);
651     else {
652         guy.partsRot[PERSON_LUA][X] -= INC;
653         guy.partsRot[PERSON_RUA][X] += INC;
654         glutPostRedisplay();
655     }
656 }

```

Figure 275: Correct animation algorithm implemented at (197), with the function drawing a single frame each time it is called by `glutIdleFunc`

### 9.7.7.1.1.3.3 Thomas

Thomas implements a more complex animation algorithm than the other students. The implementation is captured by segment [THOMAS\\_A3.ANIMATION.1."Animation Algorithm"](#) [46].

The student's initial animation algorithm function is launched inside the function drawing the avatar (see Figure 276) and not via a keyboard key, meaning the 'animation' is run whenever the avatar is redrawn, which occurs when the screen is redrawn (via the display() function).

```

381 void drawBody() {
382     glPushMatrix();
383     //rotate the body by the given amount
384     glRotatef(limbRot[BODY][0],1.0,0.0,0.0);
385     glRotatef(limbRot[BODY][1],0.0,1.0,0.0);
386     glRotatef(limbRot[BODY][2],0.0,0.0,1.0);
387     animateBody();
388     glPushMatrix();
389     glRotatef(limbRot[TORSO][0],1.0,0.0,0.0);
390     glRotatef(limbRot[TORSO][1],0.0,1.0,0.0);

```

**Figure 276: Initially (153) the animation is triggered inside the function drawing the body**

At (166-170) (see Figure 277) the student implements the animation functions' bodies. There is one separate animation function for every limb. The first 'animation' function simply rotates the avatar's body by a fixed amount when the animation is active (whenever the animation's counter value is not -1). This results in the avatar's body always being rotated by the fixed value since the animation functions are called inside the function drawing the avatar's body.

```

63 void animateBody(){
64     if(animation0!=-1){
65         glRotatef(91,1,0,0);
66         //do transforms
67     }else if(animation1!=-1){
68         //do transforms
69     }else if(animation2!=-1){
70         //do transforms
71     }
72 };

```

**Figure 277: The first animation algorithm produces a single static rotation at (166)**

The student then begins to implement an actual animation algorithm from (171-180). Each limb's animation function draws a single frame, then increments the animation variable for that animation to move to the next frame, and once the counter exceeds the number of frames in the animation the animation terminates by setting the animation variable to -1 (see Figure 278). However, the implementation will cause the animation's variable to be incremented at each limb's animation function instead of once for the entire frame (involving each limb's animation function). The student does not realise this as he does not yet implement any transformations.

```

67         if(animation0!=-1){
68             //play the animation frame by frame.
69             //reset when done
70             if (animation0<MAXANIMATION0){
71                 animation0++;
72                 //do transforms
73             }else{animation0=-1;}

```

**Figure 278: The first proper attempt at implementing an animation at (178)**

From (200-232) the student implements a ‘frame’ data structure using an array, with each row in the array holding one sub-array for every avatar limb. These sub-arrays contain three values specifying the rotation value to apply to the limb along the x, y and z axis. Each row in the array is one ‘frame’ of the animation.

The student experiments with the algorithm from (233-254) by specifying an animation in the data structure, but it is not executed properly due to each limb’s animation function incrementing the animation variable. The student realises this error in (251) and separates out the animation variable incrementation to a separate function (see Figure 279). After this the animation executes properly, but only as long as redraw events are triggered, since the animation algorithm is not yet time-based and relies on the avatar being redrawn (the animation function is triggered inside the avatar’s draw function).

```

128 void updateAnimationFrames(){
129     if (animation0!=-1){
130         animation0++;
131         if !(animation0<MAXANIMATION0){
132             animation0=-1;
133         }
134     }
135 }

```

**Figure 279: The function which increments the animation variable**

The student then works on implementing a button to execute the animation from (255-290).

After successfully implementing functionality which allows the animation to be triggered via pressing of a user interface button, the student experiments with an animation from (295-299), but the animation executes too rapidly to observe since there is no timing mechanism. The student then introduces a simple busy loop at (300-301) which slows down the animation sufficiently. This produces a working animation algorithm.



Thomas's animation algorithm is unique in that it specifies every limb's x,y,z rotation values at every frame, and each row of the animation storage array represents exactly one frame as shown in Figure 280. This approach is very verbose and requires a great deal of modification for every implemented transformation as the limb's transform needs to be changed in every row (frame). However, it also allows for a clearer overview of an animation through examination of the array in which the values are hard-coded.

```
frame keyFrame0[MAXANIMATION0][MAXSELECT+1] = {
//body      head      rua      rla      rp      rul      rll      rf
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,025},{000,000,-10},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,025},{000,000,-10},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,010},{000,000,-15},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,010},{000,000,-15},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,005},{000,000,-05},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,005},{000,000,-05},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,-20},{000,000,000},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,-20},{000,000,000},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,-25},{000,000,000},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,-25},{000,000,000},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,-30},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,-30},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,005},{000,000,-15},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,005},{000,000,-15},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,010},{000,000,-15},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,010},{000,000,-15},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,020},{000,000,-10},{000,000,000},
{{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,000},{000,000,020},{000,000,-10},{000,000,000},
},
};
```

Figure 280: An animation stored using a three-dimensional array

#### 9.7.7.1.1.3.4 Ida

Ida implements the animation algorithm in segment `IDA_A3.ANIMATION.1."Animation Algo"` [15]. Her initial implementation in (81) is correct and functional (see Figure 281); when executed, it applies a rotation to the upper arms and then calls `glutPostRedisplay` to redraw the avatar. The animation is executed by being made the `glut` idle function. Pressing the animation key binds the animation function to the `glut` idle function (see Figure 282); once the conditional inside the animation indicates the animation is complete, it unbinds itself from `OpenGL`, terminating the animation. From (132-141) the student implements a method to terminate an animation prematurely.

```

458 void animate(void) {
459     if(rotation[lu_arm][xc]<=-50||rotation[ru_arm][xc]>=50)
460         glutIdleFunc(NULL);
461     else{
462         rotation[lu_arm][xc]--;
463         rotation[ru_arm][xc]++;
464         glutPostRedisplay();
465     }
466 }

```

Figure 281: Animation algorithm in (81)

```

268         else if(key == 'q')
269             glutIdleFunc(animate);

```

Figure 282: At (81), the 'q' key is set to bind the animation function to the glutIdleFunc

Ida's implementation of the animation is extremely efficient; she correctly implements an animation algorithm on the first attempt, suggesting that she has a good grasp on the OpenGL pipeline and an understanding of the event-driven programming model.

#### 9.7.7.1.1.3.5 Michael

Michael's attempt to implement an animation algorithm is contained in segment [MICHAEL\\_A3.ANIMATION.1."Anim Algo"](#) [60]. The student's first attempt is implemented in Changes 534-539. The student creates four separate functions, each containing a single 'frame' of the animation which is simply a rectangle translated to different positions on the screen (see Figure 283). The 'frames' are drawn in the display function as shown in Figure 284, with the current frame being drawn using a switch statement. However, the algorithm is incorrect since it always disables the animation after the first frame is drawn by setting anim to FALSE. Finally, the 'timer' function (see Figure 285) moves the animation from one frame to the next by incrementing the current\_frame variable and then calling glutPostRedisplay after registering itself with the OpenGL timer to be called again in 100ms to move to the next frame. This algorithm is a good start, but unfortunately glutPostRedisplay does not immediately call the display function; combined with the short frame\_delay of 100ms, it is likely that most of the frames will not be drawn since the current\_frame variable will have been incremented again already by the time display is finally called. This problem, combined with the problem with the animation being deactivated after the first frame is drawn means that the animation will not be seen on screen.

```

95 void drawFrame0 ()
96 {
97     glRecti(10, 100, 100, 180);
98 }
99 void drawFrame1 ()
100 {
101     glPushMatrix();
102     glTranslatef(150, 0, 0);
103     glRecti(10, 100, 100, 180);
104     glPopMatrix();
105 }

```

Figure 283: Two of the animation's 'frames' as implemented in (538), consisting of a moving rectangle

```

if (anim){
    switch(current_frame)
    {
        case 0: drawFrame0(); break;
        case 1: drawFrame1(); break;
        case 2: drawFrame2(); break;
        case 3: drawFrame3(); break;
    }
    anim = FALSE;
}

```

Figure 284: The frame-drawing mechanism, implemented inside the display function at (534)

```

121 void timer(int value)
122 {
123     //advance one frame
124     current_frame = (current_frame + 1) % num_frames;
125     //reset the timer
126     glutTimerFunc(frame_delay, timer, 0);
127     //display the new frame
128     glutPostRedisplay();
129 }

```

Figure 285: The 'timer' function moves the animation to the next frame after 'frame\_delay' milliseconds

Since the animation algorithm does not produce any visible output, the student attempts to debug it. He tries moving the frame-drawing code inside the display function in front of the `gluLookAt` at (548), exchanges the code drawing rectangles for code drawing the avatar's limbs inside the `drawFrame` functions (see Figure 286) at (549) before reverting to rectangles at (550). The student changes the rectangle colour and clear colour at (552-554) and moves the animation block inside the display function from (555-556). The student tries tweaking the timer delay from (558-564) and from (574-577) and again replaces the rectangle-drawing code inside the frames with code drawing the avatar at (568-572). The student removes the frame functions and moves the frame's display code

directly into the display function at (579) and then moves the (incorrect) `anim = FALSE` command to different positions from (580-583). None of these approaches address the underlying issues. The first problem involves the `anim=FALSE` call deactivating the animation after the first 'frame'. The second problem involves the interaction of the `glutPostRedisplay` call and the `glut timer` function, with the `glutPostRedisplay` call being unlikely to redraw the screen frequently enough to display frames as the timer function switches between them. As a result, none of the solution attempts listed above cause the animation to display anything on screen.

```
94 void drawFrame0 ()
95 {
96     glPushMatrix();
97
98     glRotatef(bodyx, 1, 0, 0);
99     glRotatef(bodyy, 0, 1, 0);
100    glRotatef(bodyz, 0, 0, 1);
101    glTranslatef(0.0+movex, 5.15, 0.0+movez);
102    glRotatef(headx, 1, 0, 0);
103    glRotatef(heady, 0, 1, 0);
104    glRotatef(headz, 0, 0, 1);
105    glTranslatef(0,0.75,0);
106    glObjects.at(objectAt)->getObjectsByName()["Head"]->draw();
107    glPopMatrix();
108 }
```

Figure 286: The student replaces rectangles with avatar assembly inside the frame functions at (549)

```

363         if (anim){
364             switch(current_frame)
365             {
366                 case 0:
367                 {
368                     ... Non-modified avatar assembly ...
369                 }
370                 break;
371                 ...
372                 case 2:
373                 {
374                     int spin =100;
375                     glPushMatrix();
376                     glRotatef(spin, 0, 1, 0);
377                     glTranslatef(0,5.9,0);
378                     glObjects.at(objectAt)->getObjectsByName()["Head"]->draw();
379                     glPopMatrix();
380                     ... More avatar assembly with spin ...
381                 }
382                 break;
383                 ...
384                 case 6:
385                 {
386                     int spin =300;
387                     glPushMatrix();
388                     glRotatef(spin, 0, 1, 0);
389                     ... Tweaked Avatar construction ...
390                 }
391                 break;
392             }
393         }

```

**Figure 287: The final 'animation' algorithm consists of a single transformation applied to the avatar, active while a key is pressed**

From (587-601) the student copies the entire avatar's construction into the frame rendering mechanism in the display function six times, each preceded by a rotation with a spin value which increases from frame to frame. The animation is designed to spin the avatar about its y-axis, the only 'animation' implemented by the student.

Since the student never addresses the problem relating to the glutPostRedisplay function, the animation mechanism does not function properly and the animation is only displayed while the animation button is depressed.

Despite spending more Changes on implementation of an animation algorithm than other students and producing an initial approach that included use of the glut timer function as well as a call to redisplay the scene after each frame, the student's final implementation is non-functional. The two

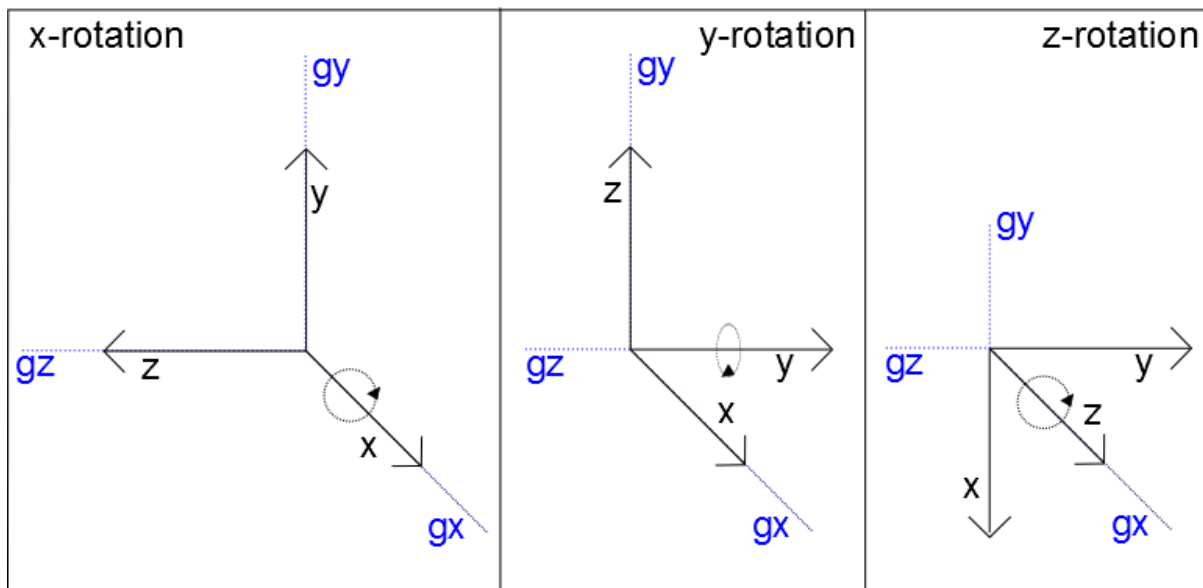
chief problems that prevented the student from implementing the animation algorithm correctly were the incorrect deactivation of the animation due to a program flow error (the `anim=FALSE` statement being executed after the first frame is drawn) and the use of `glutPostRedisplay` which redraws the screen after an indefinite time instead of immediately as would have been required by the algorithm. The `glutPostRedisplay` issue shows how opaque details of the workings of OpenGL functions can hinder student learning; while the student attempts many different solution approaches, none correctly identify the underlying problem. Learning material detailing the precise function of different OpenGL commands on the OpenGL pipeline and the role of those commands in the OpenGL framework would support students in resolving such errors. The student's issues are most likely compounded by the fact that the animation algorithm does not produce any output. Since the student does not write to standard output or use any similar debugging techniques, he remains in the dark as to the nature of the problem. The student would most likely have benefitted from being instructed in methods of debugging OpenGL program flow and pipeline issues.

#### **9.7.7.1.1.4 Animation Implementation**

##### **9.7.7.1.1.4.1 A short explanation of Gimbal Lock**

Gimbal lock is a problem which occurs when compositing three-dimensional rotations. The sequence of spatial actions that leads to gimbal lock is not in itself complex; a sequence of three rotations (x-axis, y-axis, z-axis) can produce gimbal lock.

Figure 288 shows how gimbal lock occurs. The global axes are shown in blue ( $g_x, g_y, g_z$ ), the local axes in black ( $x, y, z$ ). The approach used orients an object by a sequence of rotations around the x, y and z axes. Rotation about the x-axis in the first panel produces the local coordinate system shown in the second panel. A 90-degree rotation about the y-axis in the second panel produces the local coordinate system in the third panel. As can be seen when examining the first and third panels, the rotation about the local x and z axes both produce a rotation about the global x-axis, and none of the rotation steps can produce a rotation about the global y-axis, meaning the series of transformations has lost a rotation axis. The initial rotation about the x-axis is irrelevant; if there is a 90-degree rotation about the y-axis, the local x-axis from the first step will always overlay the local z-axis in the third step (though not necessarily on the global x-axis) and one axis of rotation and hence one degree of freedom will be lost.



**Figure 288: Gimbal Lock, produced by a 90-degree rotation about the local y-axis in the second step, leads to the local x-axis and z-axis rotating about the same global axis**

Anecdotally, students find it very difficult to understand gimbal lock and why it occurs, even though it only requires understanding the composition of three rotations. Evidence for student difficulties, if they can be found, in understanding gimbal lock are indicative of cognitive limitations of the human spatial system in dealing with and compositing three-dimensional transformations. In fact Section 9.7.7.1.1.4.6 deal with a student problem segment in which the student encounters gimbal lock and is unable to resolve the problem despite a good deal of trial-and-error manipulation

To avoid gimbal lock, students could use different approaches such as stacking transformations rather than using a fixed set of three transformations, to add a fourth rotation to the sequence of rotations or to avoid rotation by 90 or 270 degrees for the y-rotation transformation.

#### **9.7.7.1.1.4.2 Ida**

Ida implements three assignments, the walk ([IDA\\_A3.SP.4.1."The Walk Animation" \[66\]](#)) and pickup ([IDA\\_A3.SP.4.2.1."Pickup Animation" \[79\]](#)) animations as well as a jumping animation called Star Jump ([IDA\\_A3.SP.4.3."Star Jump" \[30\]](#)). She keeps the transformations simple (most of the applied transformations are not composite transformations) but spends considerable time on tweaking animation values and on using sine and cosine functions to produce smooth animations.

#### **Walking Animation**

The student implements the Walking animation in segment [IDA\\_A3.SP.4.1."The Walk Animation" \[66\]](#). The core animation consists of three different transforms.

The leg transform is first added to the upper leg at (144) but rotates about the wrong axis before being corrected at (145). The same transform is then copied to the lower leg at (147). The Arm transform is added to the upper arm at (160) but rotates about the wrong axis before being corrected at (161). It is copied to the lower arm at (165). The body transform is added correctly at (185). Two of the three core transforms are added incorrectly initially.

The student adds a z-transform to the lower and upper arm at (210). The transforms themselves are correct, but the rotation of the upper arm causes the lower arm's y-rotation to become incorrect (swinging the arms along the avatar's side / x-axis rather than the front / z-axis, see Figure 289). The student attempts to address this problem using two incorrect solutions, first modifying the upper arm's rotation to along the x-axis at (211), then by modifying the construction of the upper arm at (215) by moving the y-transformation after the z-transformation for that limb, incorrectly assuming that reversing the order of these transformations would address the problem with the lower limb's orientation. The student discovers the correct solution at (216) by adding an x-transform to the lower arm to correctly orient it after the re-orientation of the upper arm.



**Figure 289: Implementation of the Walking animation's arm movement at (210), (211) (two images) and (215)**

**Table 80: List of Rotations implemented as part of work on the Walking**

Limb	Transform Modifications	Comment	# (1-3)
Upper Leg I (L1)	+xc (144), +zc (145)		2 Axis
Lower Leg I (L2)	Continued from Upper Leg (+zc 147)	Continued	-
Lower Arm II (L1)	Copied from first experiment (+xc (157))	Copied	-
Upper Arm I	-zc (160), -yc (161), -xc	Composite	4 Axis



(L2)	(211), -yc (213)		
Lower Arm I (L1)	Continued from UpperArm I (-yc 165)	Continued	-
Body I (L1)	+xc (185)		1 Axis
Lower-Upper Arm I (L2)	+zc (210), LowerArm->UpperArm (216)	Initially breaks Lower Arm I, fixed by moving transform from lower to upper arm. Also attempts to fix by modifying construction (not included).	1 Axis, 1 Limb,

Of four unique transformations making up the final animation, three are initially added incorrectly. Implementation of the four transformations requires a total of 8 axis modifications to transformation calls, resulting in a ratio of  $8/4 = 2$  transformation attempts per correct transformations.

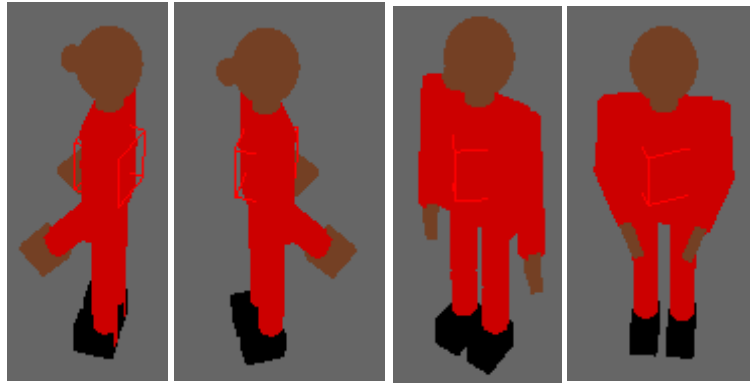
### **Pick-up/Drop Animation**

The student implements the Pick-up/Drop animation in segment [IDA\\_A3.SP.4.2.1."Pickup Animation" \[79\]](#), working on the pick-up part of the animation from (271-308, 326-328, 331-362) and the drop part from (368-378). The animation involves the upper and lower arm as well as the torso, with the leg's transforms being copied across from the walking animation and tweaked.

For the pickup animation, the student implements two torso translations at (298) and (338) and one torso rotation at (271), all of which do not require correction.

The student adds an x-rotation to the upper arm at (326) but changes it to a y-rotation at (328). The student adds an x-rotation to the lower arm at (331) which does not require correction, and then copies that transformation to the upper arm at (332), creating a working composite animation for the upper arm.

The student begins implementation of the drop animation at (368) by adding a y-rotate to the upper arm, before modifying the rotation to affect the lower arm at (370) and then reversing the rotation's direction at (371) and (372) (see Figure 290). The student changes the axis of rotation to z at (373) and then to x at (374), having tried all axes. The student copies the upper arm's x and y rotations from the pickup phase of the animation to the drop phase at (377).



**Figure 290: Development of the pickup animation's arm animation, bringing together the arms to pick up an object at with incorrect attempts at (370, 371 and 373) and the correct solution at (374)**

**Table 81: List of Rotations implemented as part of work on the Pickup animation**

Limb	Transform Modifications	Comment	# (1-3)
Torso I	+z (271)		1 Axis
UpperArm I	+x (326), +y (328)		2 Axis
LowerArm I	+x (331)	Composite	1 Axis
UpperArm II	Continued from LowerArm (+x [331])	Continued	-
UpperArm/LowerArm I	-y (368), UpperArm->LowerArm (370), +y (371), -y (372), -z (373), -x (374)	Composite	3 Axis 1 Limb 2 Direction
UpperArm III	Copied from Pickup (+y (375))	Copied	-
UpperArm IV	Copied from Pickup (+y (377))	Copied	-

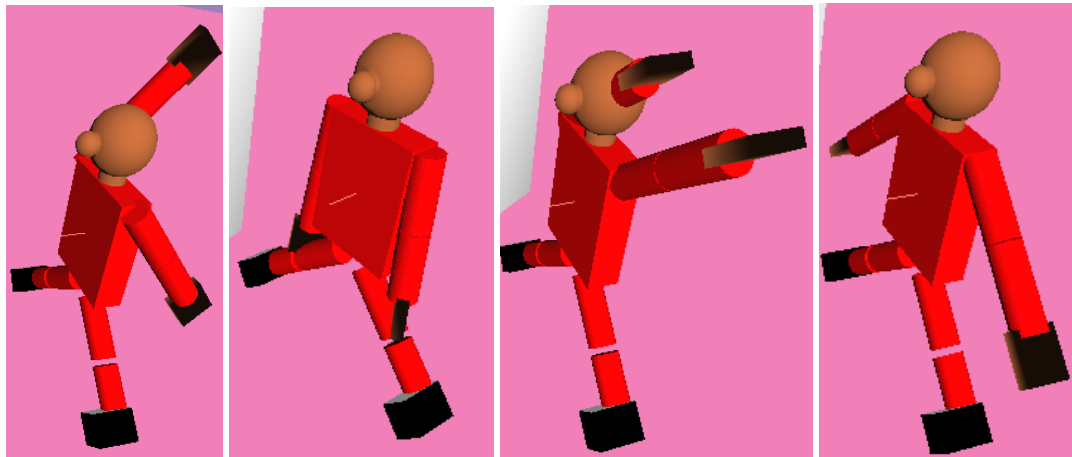
The four unique implemented transformations required a total of 7 axis modifications, leading to a ratio of  $7/4 = 1.75$  transformation attempts per successfully implemented transformation. Of the four added transformations, two are added correctly, while two are added with errors.

### **Star Jump Animation**

The student's final animation is the Star Jump animation implemented in segment [IDA\\_A3.SP.4.3."Star Jump" \[30\]](#) which causes the avatar to jump into the air and spread its arms and legs. It is implemented in Changes (415-417, 441-444, 455-478, 509-511).

The final animation consists of five unique transformations. The Upper Leg transformation is incorrectly added as a z-rotation at (415) before being modified to an x-rotation at (441). A rotation around the z-axis is correctly added to the lower leg at (470).

A y-rotation is added to the upper arm at (456) before being corrected to a z-rotation at (457), then to an x-rotation at (458) with the direction of the rotation being reversed at (459). The lower arm's rotation is added as an x-rotation at (471), changed to a y-modification at (473), then to a z-modification at (474) before being changed back to an x-rotation at (475) and finally having the direction of the x-rotation reversed at (476). The effect of some of these experiments is shown in Figure 291.



**Figure 291: Implementation of avatar arm movement for the Star Jump animation at (456-459)**

The student tweaks the amount of limb rotations at Changes (442, 455, 460, 461, 462, 469, 477, 509-511).

While implementing the Star Jump animation, the student only added two of the five transformations correctly to start with. The student did not add more than a single transformation to any single limb, but did add rotations in different directions to both the leg and the arm, but as a result made many axis errors while adding those transforms, especially while adding the arm transforms for which the student tried all three axes for both the upper and the lower arm, indicating a poor spatial understanding supplemented by visual experimentation with different rotation axes.

As shown in Table 82, of the four new rotation transformations added during implementation, three were added with errors. A total of 10 axis modifications were required to implement the four rotations, meaning 2.5 axis modifications were made for every successfully implemented rotation call.

**Table 82: List of Rotations implemented as part of work on the Start Jump animation**

Limb	Transform Modifications	Comment	# (1-3)
UpperLeg	+z (415), +x (441)		2 axis
LowerLeg	+z (470)	Composite	1 axis
UpperLeg	Continued from UpperLeg (+x 442)	Continued	-
UpperArm	+y (456), +z (457), +x (458), -x (459)		3 axis 1 direction
LowerArm	+x (471), +y (473), +z (474), -x (475), +x (475)	Composite	4 axis 1 direction

### **Conclusion**

The student generally implements relatively simple transformations, but makes errors frequently while implementing these simple transformations. The student's implementation of compound transformations in the pick-up / drop animation and implements rotations around different axes for upper and lower limbs in the star jump animation require many corrective axis modifications, indicating that the student has difficulty visualising the compounding of transformations.

#### **9.7.7.1.1.4.3 Thomas**

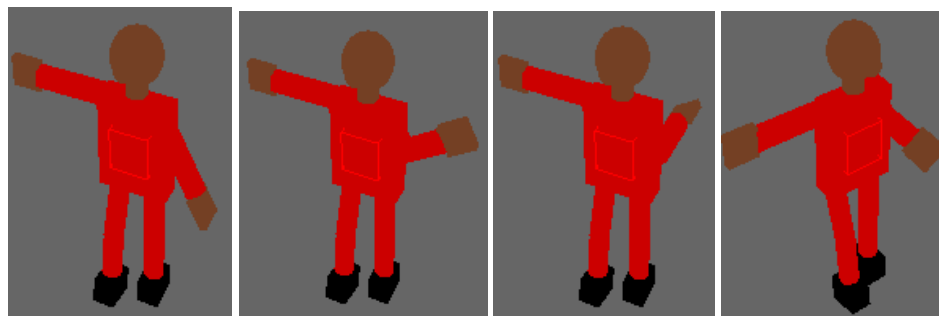
Thomas utilises an animation algorithm and data structure that stores all transformations as part of a single two-dimensional array, a method different to that of other students. As a result, implementation of animations is somewhat more natural as each 'key frame' can be implemented as one row in the array. The student implements a walking animation ([THOMAS\\_A3.SP.6.1."Walk Animation" \[36\]](#)), a pickup animation ([THOMAS\\_A3.SP.6.2."Pickup Animation" \[21\]](#)) and a wave animation ([THOMAS\\_A3.SP.6.3."Waving animation" \[20\]](#)).

### **Walking Animation**

The student implements the Walking animation in segment [THOMAS\\_A3.SP.6.1."Walk Animation" \[36\]](#) from (302-309) and then from (366-411), with the break due to the student's fixing the initial naïve assembly as he realises limbs do not properly rotate when implementing the animation.

The student begins implementing the first transforms of the animation at (306), correctly adding an x-transform to both the upper and lower leg to produce a working 'walking' leg.

After an interval fixing the avatar's assembly, the student returns to implementing the walking animation by working on the upper and lower arms. The student adds a z-transform to the upper arm at (373) which is incorrect and only makes the arm turn about its own joint axis. It is eventually corrected to a y-transform at (381). However, before that correction the student also adds an x-transform to the upper arm at (376), creating a compound transformation. The student reverses the direction of the rotation at (377). A z-transformation is added to the lower arm at (391) (see Figure 292), but is modified to an x-rotation at (404) before being changed to a y-rotation at (405) and then having its direction changed to produce a negative y-rotation at (406). The student also adds a z-rotation to the palms at (407).



**Figure 292: Modification of the lower arm's rotation from z (391) to x (404) to y (405) with a direction reversal at (406)**

In the walking animation the student adds a compound transformation (see Table 83 for a summary of all transformations during the implementation of the animation) for the upper arm but hence requires an axis correction and a direction correction to implement it. The student also struggles with producing the correct transformation for the lower arm, trying all axes before discovering the correct y-rotation. This may be due to the difficulty of visualising the interaction of the lower arm's transformation with the compound transformation applied to the upper arm. The rotation applied to the lower and upper leg (a z-rotation) does not require correction.

**Table 83: List of Rotations implemented as part of work on the Walking animation**

Limb	Transform Modifications	Comment	# (1-3)
UpperLeg	+x (306)		1 axis
LowerLeg	Copied from UpperLeg at 306 (+x)		-
UpperArm 1	+z (373), +y (381)		2 axes
UpperArm 2	+x (376), -x (377)		1 axis 1 direction
LowerArm ( <b>L2</b> )	+z (391), +x (404), +y (405), -y (406)	Composite	3 axis 1 direction

Palm	+z (407)		1 axis
------	----------	--	--------

Of five unique rotations added during the animation's implementation, two were added with initial axis errors. The student required a total of 8 axis modifications for implementation of 5 transformations, a ratio of 1.6 axis modifications per successfully implemented rotation.

### **Pickup/Drop Animation**

The pickup animation is implemented in segment [THOMAS\\_A3.SP.6.2."Pickup Animation" \[21\]](#) and consists only of transformations applied to the arms (see Table 84), with the transformation for the lower arms being copied from the upper arm transform at (683) and (685).

**Table 84: List of Rotations implemented as part of work on the Pickup animation**

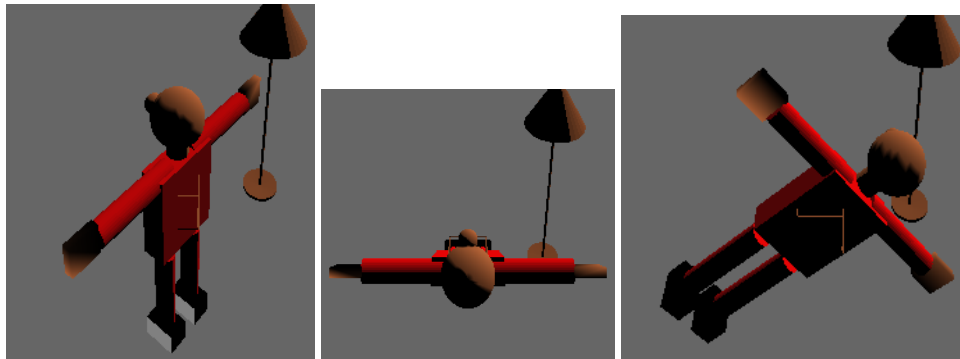
Limb	Transform Modifications	Comment	# (1-3)
UpperArm 1	+y (678), -y (679)		1 axis 1 direction
UpperArm 2 (recheck)	-x (679)		1 axis
LowerArm 1	Copied from UpperArm 1 at 383 (+x)		-
LowerArm 2	Copied from UpperArm 2 at 385 (+y)		-

A y-transformation is added to the upper arm at (678) and its direction is reversed at (679). The student also adds an additional x-transformation to the upper arm at (679), producing a compound transformation. The remainder of the Changes are spent tweaking these animations and copying the upper arm's transformations to the lower arm. Both unique transformations are added without need for axis correction, leading to an axis-modification to rotation ratio of 1.

### **Wave Animation**

The student produces a Waving animation as his third animation, implemented in segment [THOMAS\\_A3.SP.6.3."Waving animation" \[20\]](#). The final animation involves transformation of upper and lower arms, the upper leg and the body; however, all transforms occur along the same axis (the x-axis), meaning the final animation is very simple.

As shown in Figure 293 the student begins by adding a y-transform to the body at (702), then modifies that translation to a z-translation at (703) before changing it to an x-translation at (704) having tried all axes at that point.



**Figure 293: Implementation of the full-body rotation of the avatar for the Waving animation; the initial attempt is a y-rotation at (702) modified to a z-rotation at (703) and finally to an x-rotation at (704)**

This z-transformation is copied to the upper leg at (707), the upper arm at (718) and the lower arm at (721). The only really original transformation added during this animation is the original body x-translation, and the student has to try all axes to discover the correct axis for the transformation. Most of the other Changes in the animation are Tweaks to the transformations. The transformations are summarised in Table 85.

**Table 85: List of Rotations implemented as part of work on the Wave animation**

Limb	Transform Modifications	Comment	# (1-3)
Body	+y (702), +z (703), +x (704), -x (705)		3 axis 1 direction
LowerLeg	+x (707)		1 axis
UpperArm	+x (710), DEL (711), +x (718), -x (720)	The development occurs on left and right arms separately, rationalised to a single (right) arm here for simplicity  Composite	4 axis
LowerArm	Copied from UpperArm at 715 (-x)		-

Of three unique rotation transformations added, two are added with initial axis errors. The modification-to-transformation ratio is 2.3.

### **Conclusion**

The student's implementation of the walking animation is quite complex, including a compound transformation of the arms. The other two animations involve few animation steps and are relatively simple, especially the third animation which in essence involves only a single transformation (all limbs are transformed along the same axis).

The student makes several axis errors and direction errors while implementing the animations, including some like the lower arm transformation in the Walking animation and the body transformation in the Wave animation in which he tries all axes before discovering the correct axis. However, compared to other students he makes relatively fewer errors in the second and third animations, though this may also have to do with the simplicity of those animations. The low complexity of the second and third animation may also indicate a discomfort with spatial reasoning.

#### **9.7.7.1.1.4.4 Christopher**

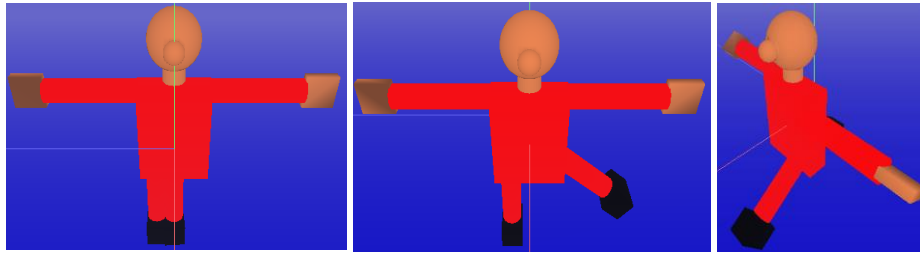
Christopher implements only the walking and pick-up animations ([CHRISTOPHER\\_A3.SP.2.1."Pickup Anim" \[24\]](#)), failing to implement the third animation. While implementing both animations, the student frequently encounters General Programming problems which hinder his progress; the most significant of which is his struggle with bugs in the scene graph (SG) algorithm implementation (discussed further in Section 9.7.7.1.2.4) which causes limbs to not be affected by transformations. When this occurs, the student often requires several Changes to identify the root cause as the SG implementation rather than with the transformations themselves; the student's difficulty with understanding the compositing of transformations as observed during his avatar assembly as discussed in Section 9.7.7.1.1.2.1 likely increases his uncertainty in such situations and leads to uncertainty about the problem source.

### **Walking Animation**

The student implements a simple Walking animation in Changes (541, 550, 553, 556-557, 559, 571-576, 584-588), with the work on the actual animation often interrupted by work on General Programming.

The student starts implementing the animation by working on the upper leg at 541 (see Figure 294), incorrectly rotating it about the x-axis. The student then incorrectly rotates the upper leg about the y-axis at 571 before rotating it about the correct z-axis after having tried all alternatives at 576.





**Figure 294: Implementation of the leg's movement for the walking animation, using an x-rotation at (541), modifying that to a y-rotation at (571) and finally correcting it to a z-rotation at (576)**

This is the sole transformation the student creates for the whole animation (see Table 86), simply transferring that z-transformation to other limbs including arms where it produces an incorrect and unnatural animation.

**Table 86: List of Rotations implemented as part of work on the Walking animation**

Limb	Transform Modifications	Comment	# (1-3)
Body	+x (585)		1 axis
UpperLeg	+x (541), +y (571), +z (752)		3 axis
LowerLeg	Copied from UpperLeg at 757 (+z), -z (756)		-

In total, the animation consists of one transformation with incorrect axis and direction, one incorrect axis modification, one correct axis modification and one correct direction modification (both correcting errors for the first and only real transformation, essentially by trial and error). The student copies that transformation to the arm (for which it does not work) and to the lower leg.

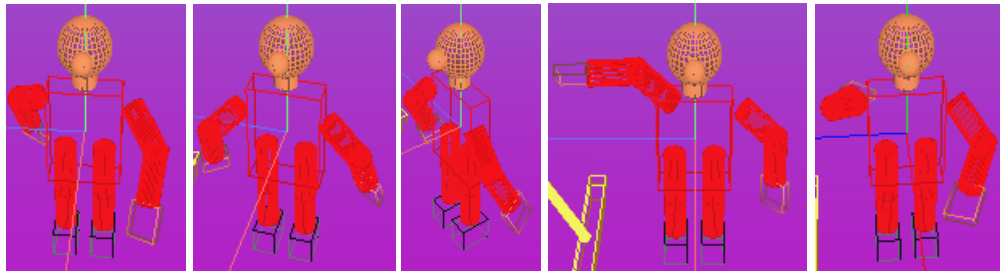
The student does not utilise any composited transformations (such as transforming a limb about two axes, or transforming a connected child limb about an axis different to its parent). The student is avoiding any transformations that would require mental compositing of transformations.

The axis modification to final rotation ratio is  $4/2 = 2.0$ .

### **Pickup Animation**

The pick-up animation is implemented in segment [CHRISTOPHER\\_A3.SP.2.1."Pickup Anim" \[24\]](#). The student does not encounter problems with his scene graph algorithm, but spends several Changes dealing with miscellaneous General Programming problems.

The pick-up animation consists of a transformation of the torso, a transformation of the arms and a transformation of the legs copied from the walking animation. The arm transformation is added at (938) (see Figure 295). It is added as a y-transform, but the student changes it to a z-transform at (971) before changing it to an x-transform at (1195) and then back to a y-transform at (1197) (see the Line History shown in Figure 296).



**Figure 295 : Implementation of the rotation of the upper arm at (938, 971, 1194, 1195, and 1197)**

(1935): (934, 938, 945, 971, 1191-1195, 1197 : total = 10)
938, MUTATED(X): scene->getNodeByName("RightUpper")->rotation.y=60*sin(i*PI/180);
945, MUTATED(O): scene->getNodeByName("RightUpperArm")->rotation.y=60*sin(i*PI/180);
971, MUTATED(O): scene->getNodeByName("RightUpperArm")->rotation.z=60*sin(i*PI/180);
1191, MUTATED(O): scene->getNodeByName("RightUpperArm")->rotation.z=-60*sin(i*PI/180);
1192, MUTATED(O): scene->getNodeByName("RightUpperArm")->rotation.z=60*sin(i*PI/180);
1193, MUTATED(O): scene->getNodeByName("RightUpperArm")->rotation.z=2*60*sin(i*PI/180);
1194, MUTATED(O): scene->getNodeByName("RightUpperArm")->rotation.z=60*sin(i*PI/180);
1195, MUTATED(O): scene->getNodeByName("RightUpperArm")->rotation.x=60*sin(i*PI/180);
1197, MUTATED(O): scene->getNodeByName("RightUpperArm")->rotation.y=60*sin(i*PI/180);

**Figure 296: Line History for the development of the upper arm transform**

The transformation for the upper body is added at 962 but incorrectly rotates about the y-axis; the transformation is corrected to a z-rotation at 957.

The third and final transformation used is the correct transformation of the leg added at 962, but it is copied from and identical to the leg transformation used in the walking animation.

**As shown in**

Table 87 the animation consists of one copied transform, one transform that undergoes three axis modifications and one transform that has one axis modification; none of the 'new' transforms are added correctly, despite being non-composited transforms.

**Table 87: List of Rotations implemented as part of work on the Pickup animation**

Limb	Transform Modifications	Comment	# (1-3)
UpperArm	+y (945), +z (971), -z (1191), +z (1192), +x (1195), +y (1197)		4 axis 2 direction
Man	+y (953), +z (957), -z (958), +z (960), -z (968)		2 axis 3 direction
UpperLeg	Copied from man at 962 (+z), Del (966), +z (968)	Copied	-
LowerArm	Copied from UpperArm at (1194)		-

Both of the unique rotations used in the animation initially rotate about the incorrect axis. The two rotations require a total of 6 axis modifications, leading to a rotation to axis modification ratio of 3.0

The failure of the student to produce a more complex second animation with any composite transforms, and the fact that none of the transforms are added without error (in the case of the arm transform undergoing extensive experimentation) indicate the student is still very uncomfortable with spatial programming in the final stage of his assignment programming.

### **Conclusion**

Since the student's assignment consists of more Changes than most other students' assignment implementations, it can be assumed that the student invested significant effort in the assignment's development. However, the student's work on animations was not very ambitious, despite the student's general hard work and the fact that animations made up a large part of the total marks for the assignment. This suggests that the student felt too uncertain and non-confident about his spatial understanding. It may also have to do with his poor avatar assembly, which he had to hide while implementing animations.

#### **9.7.7.1.1.4.5 Michael**

Michael did not manage to correctly implement an animation mechanism or to properly assemble the avatar, and hence did not work on a real animation.

#### **9.7.7.1.1.4.6 John**

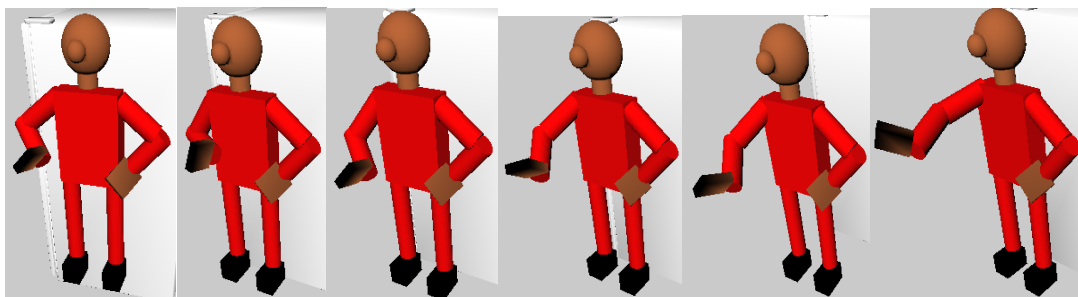
John implements a pick-up ([JOHN\\_A3.SP.5.2."Pickup Anim" \[99\]](#)) and walking animation ([JOHN\\_A3.SP.5.1."Walk anim" \[19\]](#)), as well as a swimming animation ([JOHN\\_A3.SP.5.3."Swim Anim" \[120\]](#)). John's pick-up and swimming animations are some of the most complex attempted by any student, and involve several composite transforms.

During the implementation of the pick-up animation the student encounters gimbal lock (see 9.7.7.1.1.4.1 A short explanation of Gimbal Lock) which requires a significant amount of work to resolve.

### **PICKUP ANIMATION**

The student's work on the pickup animation is captured in segment [JOHN\\_A3.SP.5.2."Pickup Anim" \[99\]](#). When first starting on the pick-up animation (158-163), the student identifies a bug in his animation approach and spends (164-201) implementing a working animation algorithm.

From (202-209) the student adds two more animation frames to the animation and adds rotations for the lower arm, upper arm and palm (see Figure 297). It seems the student is unable to properly visualize the transformations required to achieve the desired animation since the animation achieved makes the arm move in front of the body rather than away from it and the added rotations are all deleted when the student continues work on the animation from (259) (essentially from scratch). He interrupts his efforts in creating the animation to deal with a different problem, working on different views from (210-258).



**Figure 297: First attempt at arm movement for pickup animation at (202, 204, 205, 206, 207, and 209)**

The student returns to working on the pickup animation from (259-318). From (259-275), the student removes all but the initial frames and begins programming the animation from scratch. In this segment, the student correctly adds or modifies to be correct three rotation transformations, while incorrectly adding or modifying 6 rotation transformations (one wrong direction, five wrong axis rotations) as shown in Figure 298. This shows the student is struggling with understanding the compositing of transformations, even though the student adds the transformations one by one (presumably to reduce the amount of spatial visualization required), and therefore only has to

visualise the one new rotation impacting on the state of the previous rotations. The student also carries out 6 tweak actions.

266

```
672      guy.partsRot[PERSON_LUA][X]-=INC;
-1 : 1    guy.partsRot[PERSON_LLA][Z]+=INC;
673      glutPostRedisplay();
674      if(guy.partsRot[PERSON_LUA][X]<=-85)
```

268

```
672      guy.partsRot[PERSON_LUA][X]-=INC;
673      guy.partsRot[PERSON_LLA][Y]+=10*INC; -> guy.partsRot[PERSON_LLA][Y]-=INC;
674      glutPostRedisplay();
675      if(guy.partsRot[PERSON_LUA][X]<=-85)
```

269

```
672      guy.partsRot[PERSON_LUA][X]-=INC;
673      guy.partsRot[PERSON_LLA][Z]+=10*INC; -> guy.partsRot[PERSON_LLA][Y]+=10*INC;
674      glutPostRedisplay();
675      if(guy.partsRot[PERSON_LUA][X]<=-85)
```

**Figure 298: Changes 266, 268 and 269 illustrate the student's difficulty in correctly understanding the composition of rotations.**

From (276-318) the student encounters gimbal lock for the lower arm transform. For more details on Gimbal Lock, see Section 9.7.7.1.1.4.1. Gimbal Lock is a problem encountered when compositing transformations, and an understanding of why and how Gimbal Lock occurs requires an understanding of the composition of the involved transformations. In this case, the transformations in questions are the three transformations used to position limbs (see Figure 299), where gimbal lock occurs when the y-rotation is 90 or 270 degrees, in which case the x-axis from the first transformation is rotated onto the z-axis of the third transformation and one degree of freedom is lost.

```
514 glPushMatrix(); {
515 glTranslatef(0.0, 0.0, -1.3);
516
517 glTranslatef(0.0, 0.0, 0.5);
518 glRotatef(guy.partsRot[PERSON_LLA][X], 1,0,0);
519 glRotatef(guy.partsRot[PERSON_LLA][Y], 0,1,0);
520 glRotatef(guy.partsRot[PERSON_LLA][Z], 0,0,1);
521 glTranslatef(0.0, 0.0, -0.5);
522 guy.pointer->getObjectsByName()["LeftLowerArm"]->draw();
```

**Figure 299: The sequence of rotations used to orient limbs, which causes gimbal lock when the Y-rotation value is 90 or 270 (or -90/-270) degrees.**

In attempting to resolve the problem caused by gimbal lock from (276-310), the student repeatedly changes lower arm's x to a z transform and back again to observe the effects of the gimbal lock (as

can be seen in the Line Histories shown in Figure 300 and Figure 301) as well as experimenting with the modification of the orientation of other limbs (which has no effect on the gimbal-locked limb). The student also modifies the first frame's y-rotation or adds another y-rotation to the second frame (at 280, for example) which resolves the gimbal lock, but the student apparently is seeking to pinpoint and understand the problem and thus removes these approaches again.

(1059): (275-280, 282-285, 287-290, 292, 294-296, 298-300 : total = 21)
275, ADDED(O): guy.partsRot[PERSON_LLA][X]+=INC;
276, MUTATED(O): guy.partsRot[PERSON_LLA][X]-=INC;
277, MUTATED(O): guy.partsRot[PERSON_LLA][Z]-=INC;
278, MUTATED(O): guy.partsRot[PERSON_LLA][Y]-=INC;
279, MUTATED(O): guy.partsRot[PERSON_LLA][X]-=INC;
280, MUTATED(O): guy.partsRot[PERSON_LLA][X]+=INC;
282, MUTATED(O): //guy.partsRot[PERSON_LLA][X]+=INC;
283, MUTATED(X): /uy.partsRot[PERSON_LLA][X]-=INC;
284, MUTATED(O): guy.partsRot[PERSON_LLA][X]-=INC;
285, MUTATED(O): // guy.partsRot[PERSON_LLA][X]-=INC;
287, MUTATED(O): guy.partsRot[PERSON_LLA][X]-=INC;
288, MUTATED(O): guy.partsRot[PERSON_LLA][Z]-=INC;
289, MUTATED(O): guy.partsRot[PERSON_LLA][Z]-=INC2;
290, MUTATED(O): guy.partsRot[PERSON_LLA][X]-=INC2;
292, MUTATED(O): guy.partsRot[PERSON_LLA][Y]-=INC2;
294, MUTATED(O): guy.partsRot[PERSON_LLA][X]-=INC2;
295, MUTATED(O): guy.partsRot[PERSON_LLA][Z]-=INC2;
296, MUTATED(O): guy.partsRot[PERSON_LLA][Y]-=INC2;
298, MUTATED(O): //guy.partsRot[PERSON_LLA][Y]-=INC2;
299, MUTATED(O): guy.partsRot[PERSON_LLA][X]-=INC2;
300, DELETED(O): DELETED (guy.partsRot[PERSON_LLA][X]-=INC2;)

**Figure 300: First Line History showing experimentation with the lower arm rotation**

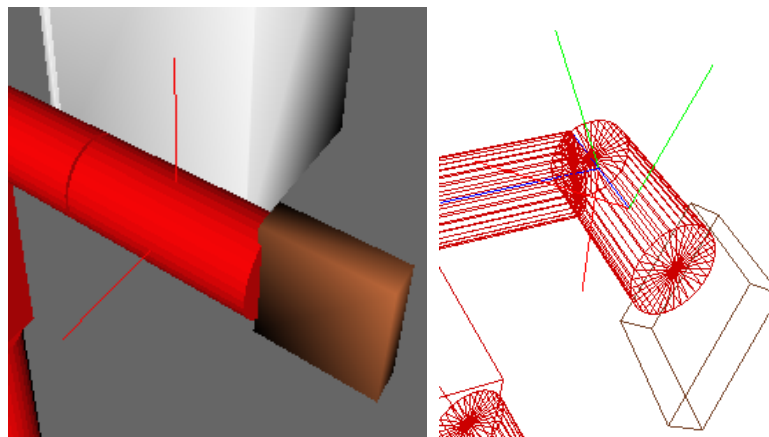
(1067): (301, 303, 306, 308-309, 312, 392 : total = 7)
301, ADDED(O): guy.partsRot[PERSON_LLA][X]+=INC;
303, MUTATED(O): guy.partsRot[PERSON_LLA][Z]+=INC;
306, MUTATED(O): guy.partsRot[PERSON_LLA][X]+=INC;
308, MUTATED(O): guy.partsRot[PERSON_LLA][Z]+=INC;
309, MUTATED(O): guy.partsRot[PERSON_LLA][Y]+=INC;

```
312, MUTATED(O): guy.partsRot[PERSON_LLA][X]+=INC;
```

**Figure 301: Second Line History showing experimentation with the lower arm rotation**

The student appears to make little progress, removing the entire gimbal-locked frame at 300 and then adding a new (also gimbal locked, since the lock occurs in the preceding frame) frame at 301 and continues trying to change the lower arm transform axis. He also removes a static declaration specifying the amount of rotation at 307, possibly thinking it is somehow influencing the transformations.

The student then seeks assistance from an instructor (the researcher) during a practical class. The instructor suggests he look up the concept of gimbal lock in his lecture notes and also instructs him to add coordinate axis lines to the avatar's limb rotations. The student adds coordinate axis lines to the program after the lower arm's transformations at (310-311) (the coordinate lines can be seen in Figure 302), and adds additional coordinate axis lines to before the lower arm's transformations at (313). The student then uses these coordinate axis lines together with the limb rotation keys and standard output statements which print the current value of the lower arm's transform (added at 317-318) to debug the gimbal lock problem from (310-318).



**Figure 302: Separate grid lines show the local coordinate system for the lower and upper arms**

Before resolving the problem, he then works on creating a room from (319-335). It is interesting to observe how the student's switching to work on a different, unrelated task seems to divide the different sub-segments of this problem, each of which comprises a different solution approach. While it is impossible to verify the student's mental state without the student's input, it may be conjectured that the student's working on a different problem was an attempt to step away from the problem and prepare himself for a different approach by clearing his mind of the details of his current approach.

At (336) the student apparently grasps the root cause of the gimbal lock. The student modifies the first frame of the animation (see Figure 303), making transition to the second frame no longer conditional on the lower arm's y-rotation being 90 degrees, instead (after several changes) maintaining a y-rotation but moving to the second frame after a 30-degree rotation.

Change 336:

```

713     switch(counter) {
714         case 0: {
715             /* initial position (relaxed...) */
716             guy.partsRot[PERSON_LUA][X] = -35; -> guy.partsRot[PERSON_LUA][X] = -90;
717             guy.partsRot[PERSON_LLA][X] = -100;
718             guy.partsRot[PERSON_RUA][X] = 35;
719             guy.partsRot[PERSON_RLA][X] = 100; -> guy.partsRot[PERSON_RUA][X] = 90;
720             guy.partsRot[PERSON_LPI][Y] = -45;
721             guy.partsRot[PERSON_RPI][Y] = 45;
722             glutPostRedisplay();
723             counter++;
724             break;
725         }
726         case 1: {
727             guy.partsRot[PERSON_LUA][X]=INC; -> guy.partsRot[PERSON_LUA][X]=INC;
728             guy.partsRot[PERSON_LLA][Y]=INC2; -> guy.partsRot[PERSON_LLA][X]=INC;
729             guy.partsRot[PERSON_LPI][Y]=INC;
730             glutPostRedisplay();
731             if(guy.partsRot[PERSON_LLA][Y]<=90) -> if(guy.partsRot[PERSON_LUA][X]>30)
732                 counter++;
733             break;
734         }/*

```

Change 337:

```

720     }
721     case 1: {
722         guy.partsRot[PERSON_LUA][X]=INC; -> guy.partsRot[PERSON_LUA][Y]=INC;
723         guy.partsRot[PERSON_LLA][X]=INC;
724         glutPostRedisplay();
725         if(guy.partsRot[PERSON_LUA][X]>30)

```

Change 338:

```

720     }
721     case 1: {
722         guy.partsRot[PERSON_LUA][Y]=INC; -> guy.partsRot[PERSON_LUA][Y]=INC;
723         guy.partsRot[PERSON_LLA][X]=INC;
724         glutPostRedisplay();
725         if(guy.partsRot[PERSON_LUA][X]>30) -> if(guy.partsRot[PERSON_LUA][Y]<=30)
726             counter++;
727         break;
728     }/*

```

Figure 303: Changes addressing the gimbal lock problem by not making the first frame change the y-rotation to a value close to 90 degrees

The student then continues working on the animation from (336-351). The student again shows difficulty in correctly composing transformations, producing the final animation through a trial-and-error process involving 4 correct or corrected transformations and 8 incorrect transformations (5 incorrect axis rotations, 3 incorrect sign/direction rotations). The line history in Figure 304 shows the



trial and error approach the student uses for a single rotation in this segment in order to achieve the desired outcome.

```
(860): (164, 166, 170, 259, 261, 263-264, 336-338, 340, 342-343, 347-348 : total = 15)
336, MUTATED(O): guy.partsRot[PERSON_LUA][X]+=INC;
337, MUTATED(O): guy.partsRot[PERSON_LUA][Y]+=INC;
338, MUTATED(O): guy.partsRot[PERSON_LUA][Y]-=INC;
340, MUTATED(O): guy.partsRot[PERSON_LUA][X]-=INC;
342, MUTATED(O): guy.partsRot[PERSON_LUA][X]+=INC;
343, MUTATED(O): guy.partsRot[PERSON_LUA][Z]+=INC;
347, MUTATED(O): guy.partsRot[PERSON_LUA][Y]+=INC;
348, MUTATED(O): guy.partsRot[PERSON_LUA][Y]-=INC;
```

**Figure 304: A line history showing changes to a single transformation in the segment 336-351, involving 3 axis changes and 3 direction changes.**

After working on room creation from 352-388, the student works on the avatar's torso orientation from 389-395. Initially, the torso has no rotation applied to it at all, so it is somewhat surprising that the student makes two errors when applying the torso orientation (as shown in Figure 305, one axis error and one sign error), again demonstrating that even relatively simple spatial reasoning remains difficult. Overall, this segment contains two correct and two incorrect orientation actions.

```
(1183): (389-391 : total = 3)
389, ADDED(O): guy.partsRot[PERSON_T][X]+=INC;
390, MUTATED(O): guy.partsRot[PERSON_T][Z]+=INC;
391, MUTATED(O): guy.partsRot[PERSON_T][Z]-=INC;
```

**Figure 305: The torso rotation Line History**

The student completes work on the animation from 421-424 by modifying the y-value of the object being lifted during the transformation without any errors. A summary of transformations is shown in Table 88.

**Table 88: List of Rotations implemented as part of work on the Pickup animation**

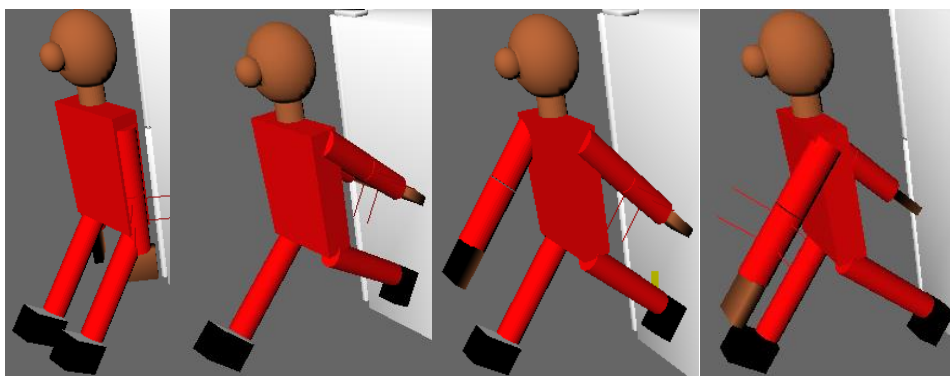
Limb	Transform Modifications	Comment	# (1-3)
UpperArm 1	+x (157), -x (159)		1 axis 1 direction
LowerArm 1	Copied from UpperArm at 159 (-x)		-
Palm 1	+y (164), DEL (271), +y (271), DEL	Composite	4 axis

	(336)		
UpperArm 2	+z (164), +x (259), -x (263), +y (337), -y (338), -x (340), +x (342), +z (343), +y (347), -y (348)	Composite	6 axis 4 direction
LowerArm 3	+x (259), DEL (264), +x (301), +z (303), +x (306), +z (308), +y (309), +x (312), DEL (314), +z (350)	Comprised of 3 separate Line Histories  Composite	8 axis 2 delete
LowerArm 4	Continued from UpperArm at 349 (-y)		-
Body 1	+x (389), +z (390), -z (391)		2 axis 1 direction
UpperLeg 1	Copied from Walk Anim at 391		-
LowerArm 5	Copied from Pickup Anim (earlier) at 395		-

### **Walk Animation (320-330, 332, 440, 442-447)**

Work on the walking animation is captured in segment [JOHN\\_A3.SP.5.1."Walk anim" \[19\]](#). The walk animation is relatively simple, especially when compared to the student's other two implemented assignments.

The student's initial walk animation implementation is added at (320). The initial implementation first puts the arms at the avatar's side via an x-rotation of the upper arms (see Figure 306), and then attempts to move the arms and legs forward using a z-rotation of the upper arms and legs. However, the student does not take into account the effect of the initial transformation on the arms, causing them to rotate incorrectly. The student fixes this in the next Change (321) by changing the arm rotate axis (z -> y) and then changes the direction of the rotation in (322-323).



**Figure 306: Development of leg movement from (320-323)**

The student adds a second case statement to the animation in (324), reversing the initial animation's transformations to create a smooth back-and-forth motion. The student then tweaks the animation speed from (325-335).

From (442-447) the student copies the upper leg's z-transform to the lower leg and also adds a rotation to the lower arms (see Figure 307). The rotation to the lower arms is initially added at (443). The direction of the rotation is reversed at (444), with the rotation axis modified from x -> y at (445). The student reverses the direction of the rotation at (446) before reversing it again at (447).

(1255): (443-447 : total = 5)

```

443, ADDED(O): guy.partsRot[PERSON_RLA][X] = -0.5*abs(guy.partsRot[PERSON_RUL][Z]);
444, MUTATED(O): guy.partsRot[PERSON_RLA][X] = 0.5*abs(guy.partsRot[PERSON_RUL][Z]);
445, MUTATED(O): guy.partsRot[PERSON_RLA][Y] = 0.5*abs(guy.partsRot[PERSON_RUL][Z]);
446, MUTATED(O): guy.partsRot[PERSON_RLA][Y] = -0.5*abs(guy.partsRot[PERSON_RUL][Z]);
447, MUTATED(O): guy.partsRot[PERSON_RLA][Y] = 0.5*abs(guy.partsRot[PERSON_RUL][Z]);

```

**Figure 307: Line History for the lower arm rotation**

The walking animation is the simplest animation implemented by the student, consisting of four unique rotations (upper leg and upper arm rotations, a lower leg rotation as well as the initial orientation of the arm). The upper arm animation is composited, since it consists of an initial orientation rotation and a further rotation to produce the animation. The transformations are summarised in Table 89.

**Table 89: List of Rotations implemented as part of work on the Walk animation**

Limb	Transform Modifications	Comment	# (1-3)
UpperArm 1	Copied from pickup at 320 (-x)		-
UpperArm 2	+z (320), -y (321), +y (322), -y (322)	Composite	2 axis 2 direction
UpperLeg	+z (320)		1 axis
Body	+z (326), +x (329)		2 axis
LowerLeg	Copied from UpperLeg at 440 (+z), -z (442)	Copied	-
LowerArm	-x (443), +x (444), +y (445), -y (446), +y (447)	Composite	2 axis 3 direction

The student initially makes an error in the implementation of two of the four rotations. The ratio of axis modifications to rotations is  $7 / 4 = 1.75$ .

## SWIMMING ANIMATION

The student's third animation, a swimming animation, is implemented in segment [JOHN\\_A3.SP.5.3."Swim Anim"](#) [120]. From (451-490) the student starts work on the swimming animation (some stages of the work are shown in Figure 308). Initially the student again makes axis and sign errors while orienting limbs (454-456) (see Figure 277) even though the composition of rotations is very simple at that point.

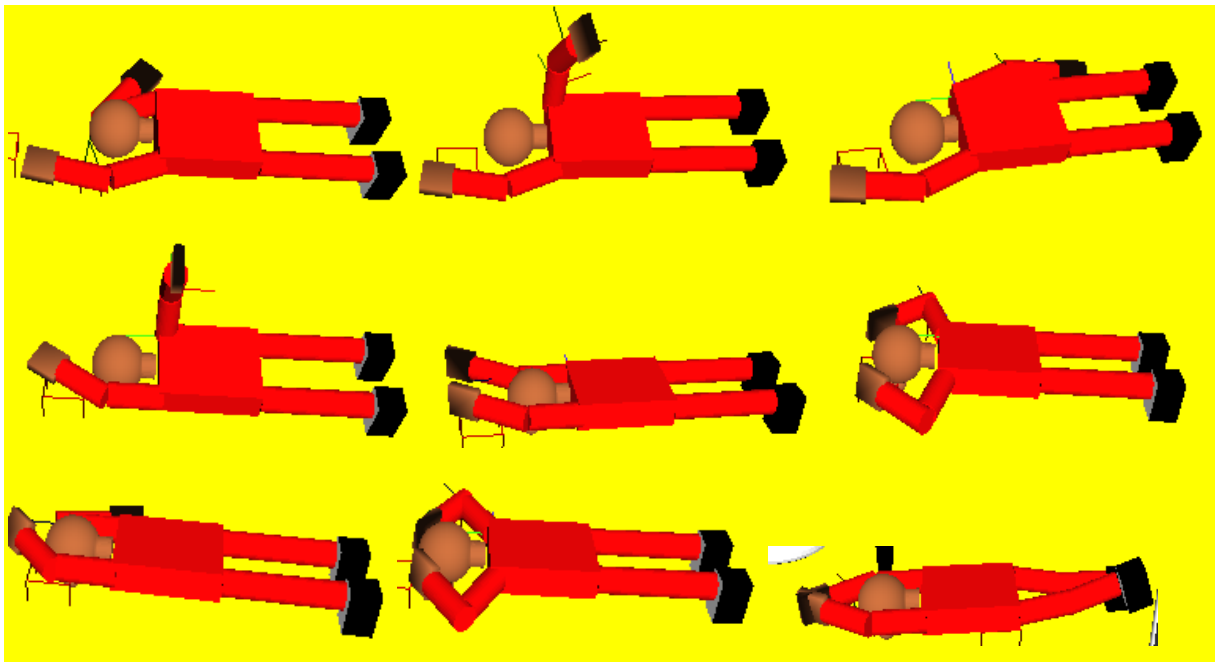


Figure 308: Some different stages of the swim animation's implementation at (463, 473, 476, 490, 491, 506, 507, 509, and 570)

454

```
854      guy.partsRot[PERSON_T][Z] -= 90;
-1 : i   guy.partsRot[PERSON_T][Y] += 20;
855      guy.partsRot[PERSON_LUA][X] += 70;
856      guy.partsRot[PERSON_RUA][X] += 70;
```

455

```
854      guy.partsRot[PERSON_T][Z] -= 90;
855      guy.partsRot[PERSON_T][Y] += 20; ->   guy.partsRot[PERSON_T][X] += 20;
856      guy.partsRot[PERSON_LUA][X] += 70;
857      guy.partsRot[PERSON_RUA][X] += 70;
```

456

```
854      guy.partsRot[PERSON_T][Z] -= 90;
855      guy.partsRot[PERSON_T][X] += 20; ->   guy.partsRot[PERSON_T][X] -= 20;
856      guy.partsRot[PERSON_LUA][X] += 70;
857      guy.partsRot[PERSON_RUA][X] += 70;
```

Figure 309: Changes (454-456) involve one axis change and one sign change for a simple initial orientation

As the student adds two frames and the animation becomes more complex, the student's understanding of the underlying rotations breaks down, leading to many axis change errors. Finally, the student adds the coordinate lines first used in the implementation of the pick-up animation to understand the orientation of limbs during the different frames of the animation from (467-470).

The student then attempts to fix the animation from (471-490) (some modifications are shown in Figure 310) but continues to struggle with the upper and lower arm's orientation and finally gives up on the approach at (490), removing the third frame and most of the second frame. In the segment from (451-490), the student performs nine axis changes and six sign changes and fails to produce a working animation.

From (491-523) the student corrects the rotations in the first frame with only one significant problem in (506-507, 509). The remainder of work on the arm animation from 524-543 involves tweaking of the magnitude of the orientation commands.

(1289): (458, 473, 476-477, 480, 482-483, 489-491, 491, 496-497 : total = 13)
458, ADDED(O): guy.partsRot[PERSON_RUA][Y] -= INC;
473, MUTATED(O): guy.partsRot[PERSON_RUA][X] -= INC;
476, MUTATED(O): guy.partsRot[PERSON_RUA][Z] -= INC;
477, MUTATED(O): guy.partsRot[PERSON_RLA][Z] -= INC;
490, MUTATED(O): guy.partsRot[PERSON_RUA][Y] -= INC;
491, MUTATED(O): guy.partsRot[PERSON_RLA][Z] = -90;
491, MOVED(O): MOVED (guy.partsRot[PERSON_RLA][Z] = -90;)
496, MUTATED(O): guy.partsRot[PERSON_RLA][Z] = 90;
497, MUTATED(O): guy.partsRot[PERSON_RLA][Z] = -90;

**Figure 310: Line History for one of the swim animation's upper arm rotations**

From (544-570) the student implements the leg component of the swim animation. The leg movement is far less complex than the arm movement, since it has only one 'frame' with one orientation command for the upper leg and one for the lower leg, which are added without modification to direction or axis. An additional arm transform is added and then removed some changes later (see Figure 311), with the remainder of changes from (547-570) being spent on tweaking the rate of change in orientation.

(1371): (544-546 : total = 3)
544, ADDED(O): guy.partsRot[PERSON_LLA][Y] = -0.5*abs(guy.partsRot[PERSON_LUL][Z]);
545, MUTATED(O): guy.partsRot[PERSON_LLA][Y] = -0.5*abs(guy.partsRot[PERSON_LUA][X]);

```
546,      DELETED(O):      DELETED      (guy.partsRot[PERSON_LLA][Y]      =      -
0.5*abs(guy.partsRot[PERSON_LUA][X]));
```

**Figure 311: The removed arm transform.**

As the summary Table 90 shows, of five unique rotations added, one was added without an initial error. The five rotations required a total of 14 axis modifications, producing a ratio of 2.8 axis modifications per implemented rotation.

**Table 90: List of Rotations implemented as part of work on the Swim animation**

Limb	Transform Modifications	Comment	# (1-3)
Body 1	Copied at 451 (+z), -z (452)	Copied	-
UpperArm 1	Copied from pickup anim at 452 (+x)	Copied	-
LowerArm 1	Continued from UpperArm at 453 (+x)	Continued	-
Body 2	+y (454), +x (455), -x (456), DEL (491)		2 direction 2 axis DELETED
LowerArm 2	-y (458), -x (473), -z (476), Limb Change to LowerArm (477), +z (480), -z (489), +z (496), -z (497)	Composite	3 axis 1 limb 4 direction
UpperArm 2	-z (474), -y (475), DEL (476), +x (481), - y (490), DEL (491)	Composite	6 axis DELETED
LowerArm 3	Copied at 485 (+z), -z (489), DEL (490)	Copied	-
LowerArm 4	Copied at 485 (-x), DEL (490)	Copied	-
UpperArm 3	Copied from same anim at 493 (-x)	Copied	-
LowerArm 5	Copied from same anim at 495 (-x)	Copied	-
LowerArm 6	Copied from same anim at 496 (+x)	Copied	-
LowerArm 7	Copied from same anim at 502 (-z)	Copied	-
LowerArm 8	Copied from same anim at 502 (-x)	Copied	-
LowerArm 9	Copied from same anim at 507 (-x)	Copied	-
LowerLeg/UpperLeg	-z (455)		1 axis
LowerLeg 1	Continued from UpperLeg at 465 (+z), -z (466)	Continued	-
Body 3	Copied at 569 (+z), +x (570)		2 axis

## Discussion

John produces the most complex animations attempted by any of the students examined. The pick-up and swim animations in particular are complex, involving arm and leg movement and several composite rotations.

As a result, the student makes many errors when implementing the animations, requiring several axis modifications for the implementation of key rotations. The student also encounters gimbal lock, with which he struggles over a period of many Changes in the implementation of the pick-up assignment.

As John's work on animations shows, even high-achieving students struggle with the compositing of transformations, requiring many Changes and making many errors.

#### 9.7.7.1.1.5 View Implementation

##### 9.7.7.1.1.5.1 John

##### First-Person View

John's implementation of the first-person View is captured in segment [JOHN\\_A3.VIEW.2."FP view"](#) [21]. John begins implementation at (210). The Line History showing development of the `gluLookAt` statement implementing the View is shown in Figure 312. The student sets the `lookAt` statement's eye value to the avatar's head position (`guy.x,guy.y+8,guy.z`) and the `lookAt` value to the avatar's head position plus an offset along the z-axis (`guy.x,guy.y+8,guy.z+10`). The intent is to produce a View that looks 'ahead of' the avatar, but it instead looks sideways due to the incorrect use of the z-axis. The student fixes this error in three Changes (211-213) by moving the offset from the z to the x-axis, causing the view to correctly 'look ahead' of the avatar.

(912): (197, 210-213, 216, 218, 220-221, 224, 226, 410, 413, 417, 619, 688 : total = 16)	
197, MUTATED(X):	<code>gluLookAt(guy.x,guy.y,guy.z , 0,0,0, 0,1,0);</code>
210, MUTATED(O):	<code>gluLookAt(guy.x,guy.y+8,guy.z, guy.x,guy.y+8,guy.z+10, 0,1,0);</code>
211, MUTATED(O):	<code>gluLookAt(guy.x,guy.y+8,guy.z, guy.x,guy.y+8,guy.z-10, 0,1,0);</code>
212, MUTATED(O):	<code>gluLookAt(guy.x,guy.y+8,guy.z, guy.x,guy.y+8,guy.z+10, 0,1,0);</code>
213, MUTATED(O):	<code>gluLookAt(guy.x,guy.y+8,guy.z, guy.x+10,guy.y+8,guy.z, 0,1,0);</code>
216, MUTATED(O):	<code>gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+10,guy.y+8,guy.z, 0,1,0);</code>
218, MUTATED(O):	<code>gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+1000,guy.y+8,guy.z, 0,1,0);</code>
220, MUTATED(O):	<code>gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+5,guy.y+8,guy.z, 0,1,0);</code>
221, MUTATED(O):	<code>gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+3,guy.y+8,guy.z, 0,1,0);</code>
224, MUTATED(O):	<code>gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+100,guy.y+8,guy.z, 0,1,0);</code>
226, MUTATED(O):	<code>gluLookAt(guy.x+1,guy.y+8,guy.z, guy.x+3,guy.y+8,guy.z, 0,1,0);</code>

```

410, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z,
    guy.x + cos(guy.xrot),guy.y+8,guy.z + sin(guy.zrot), 0,1,0);
411, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z,
    guy.x + cos(guy.yrot),guy.y+8,guy.z + sin(guy.yrot), 0,1,0);
412, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z,
    guy.x + cos(guy.yrot * ratio),guy.y+8,guy.z + sin(guy.yrot * ratio), 0,1,0);
413, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z,
    guy.x + cos(guy.yrot * ratio),guy.y+8,guy.z - sin(guy.yrot * ratio), 0,1,0);
700, MUTATED(O): gluLookAt(guy.x, guy.y+8, guy.z,
    guy.x-cos(guy.yrot * ratio), guy.y+8, guy.z-sin(guy.yrot * ratio), 0,1,0);
701, MUTATED(O): gluLookAt(guy.x, guy.y+8, guy.z,
    guy.x+cos(guy.yrot * ratio), guy.y+8, guy.z-sin(guy.yrot * ratio), 0,1,0);

```

**Figure 312: Line History showing John's implementation of the first-person View; the three phases of development are highlighted in red (fixed), blue (linear movement) and green (spherical movement)**

However, the more fundamental error is that the View is fixed and will not change facing when the avatar is rotated. The student experiments with different lookAt x-offsets and adds x and z-offsets to the eye point, incorrectly moving the camera away from the avatar, in Changes (216, 218, 220, 221, 224, 226). The student is apparently stuck with the implementation as he moves on to other tasks and only returns to working on the first-person view about two-hundred Changes later at (410).

At (410) the student discovers the correct spherical-coordinate solution. In the initial Change (410) the student makes a spatial error by incorrectly calculating the spherical coordinates using the avatar's x-axis and z-axis rotation for the calculation of the view's spherical lookAt x and z coordinates, rather than using the y-coordinate to calculate the spherical coordinate for both to produce a rotation on the x-z plane. The student also incorrectly uses the sine's and cosine's positive values. Both of these errors are corrected at (413) where the student reverses the direction of the cosine calculation for the look-at's z coordinate and uses the avatar's y-rotation to calculate both the x and z spherical coordinates.

This produces a correct first-person View, originating at the avatar's head and looking in the direction the avatar is facing.

### **Third-Person (Shoulder) View**

John implements the top-down View in segment [JOHN\\_A3.VIEW.3."TP View"](#) [15], starting at Change 227. His initial implementation at (227-228) produces a fixed top-down View looking from above and behind the avatar at (guy.x-10,guy.y+15,guy.z) to in front of the avatar at head level



(guy.x+4,guy.y+8,guy.z). Work on the lookAt call is shown in Figure 313. The student suspends work on the View until (414), which is after the successful implementation of spherical coordinates to produce a non-fixed first-person View.

**(1003): (227-228, 414-418, 448, 556, 613, 615-617, 619, 689, 699, 701 : total = 17)**

227, ADDED(O): gluLookAt(

guy.x-5,guy.y+15,guy.z,  
guy.x+1,guy.y+8,guy.z, 0,1,0);

228, MUTATED(O): gluLookAt(

guy.x-10,guy.y+15,guy.z,  
guy.x+4,guy.y+8,guy.z, 0,1,0);

414, MUTATED(O): gluLookAt(

guy.x+cos(guy.yrot\*ratio)-10,guy.y+15,guy.z-sin(guy.yrot\*ratio),  
guy.x+cos(guy.yrot\*ratio),guy.y+8,guy.z-sin(guy.yrot\*ratio), 0,1,0);

415, MUTATED(O): gluLookAt(

guy.x-cos(guy.yrot\*ratio),guy.y+15,guy.z+sin(guy.yrot\*ratio),  
guy.x+cos(guy.yrot\*ratio),guy.y+8,guy.z-sin(guy.yrot\*ratio), 0,1,0);

416, MUTATED(O): gluLookAt(

guy.x-cos(guy.yrot\*ratio)-10,guy.y+15,guy.z+sin(guy.yrot\*ratio),  
guy.x+cos(guy.yrot\*ratio),guy.y+8,guy.z-sin(guy.yrot\*ratio), 0,1,0);

417, MUTATED(O): gluLookAt(

guy.x-cos(guy.yrot\*ratio),guy.y+15,guy.z+sin(guy.yrot\*ratio),  
guy.x+cos(guy.yrot\*ratio),guy.y+8,guy.z-sin(guy.yrot\*ratio), 0,1,0);

418, MUTATED(O): gluLookAt(

guy.x-3\*cos(guy.yrot\*ratio),guy.y+15,guy.z+3\*sin(guy.yrot\*ratio),  
guy.x+cos(guy.yrot\*ratio),guy.y+8,guy.z-sin(guy.yrot\*ratio), 0,1,0);

448, MUTATED(X): gluLookAt(

guy.x-zoom\*cos(guy.yrot\*ratio),guy.y+15,guy.z+zoom\*sin(guy.yrot\*ratio),  
guy.x+cos(guy.yrot\*ratio),guy.y+8,guy.z-sin(guy.yrot\*ratio), 0,1,0);

556, MOVED(O): MOVED (gluLookAt(

guy.x-zoom\*cos(guy.yrot\*ratio),guy.y+15,guy.z+zoom\*sin(guy.yrot\*ratio),  
guy.x+cos(guy.yrot\*ratio),guy.y+8,guy.z-sin(guy.yrot\*ratio), 0,1,0);)

613, MOVED(O): MOVED (gluLookAt(

guy.x-zoom\*cos(guy.yrot\*ratio),guy.y+15,guy.z+zoom\*sin(guy.yrot\*ratio),  
guy.x+cos(guy.yrot\*ratio),guy.y+8,guy.z-sin(guy.yrot\*ratio), 0,1,0);)

```

615, MUTATED(O): gluLookAt(
    guy.x-zoom*cos(yrot*ratio),guy.y+15,guy.z+zoom*sin(yrot*ratio),
    guy.x,guy.y+8,guy.z, 0,1,0);
616, MUTATED(O): gluLookAt(
    guy.x+zoom*cos(yrot*ratio),guy.y+12,guy.z+zoom*sin(yrot*ratio),
    guy.x,guy.y+8,guy.z, 0,1,0);
617, MUTATED(O): gluLookAt(
    guy.x-zoom*cos(yrot*ratio),guy.y+12,guy.z+zoom*sin(yrot*ratio),
    guy.x,guy.y+8,guy.z, 0,1,0);
619, MUTATED(O): gluLookAt(
    guy.x-zoom*cos(yrot*ratio),guy.y+12, guy.z+zoom*sin(yrot*ratio),
    guy.x,guy.y+8,guy.z, 0,1,0);
689, MUTATED(O): gluLookAt(
    guy.x-zoom*cos(guy.yrot*ratio), guy.y+12, guy.z+zoom*sin(guy.yrot *ratio),
    guy.x,guy.y+8,guy.z, 0,1,0);
691, MUTATED(O): gluLookAt(
    guy.x-zoom*cos(yrot*ratio), guy.y+12, guy.z+zoom*sin(yrot *ratio),
    guy.x,guy.y+8,guy.z, 0,1,0);
699, MUTATED(O): gluLookAt(
    guy.x+zoom*cos(yrot*ratio), guy.y+12, guy.z-zoom*sin(yrot *ratio),
    guy.x,guy.y+8,guy.z, 0,1,0);
701, MUTATED(O): gluLookAt(
    guy.x-zoom*cos(yrot*ratio), guy.y+12, guy.z-zoom*sin(yrot *ratio),
    guy.x,guy.y+8,guy.z, 0,1,0);

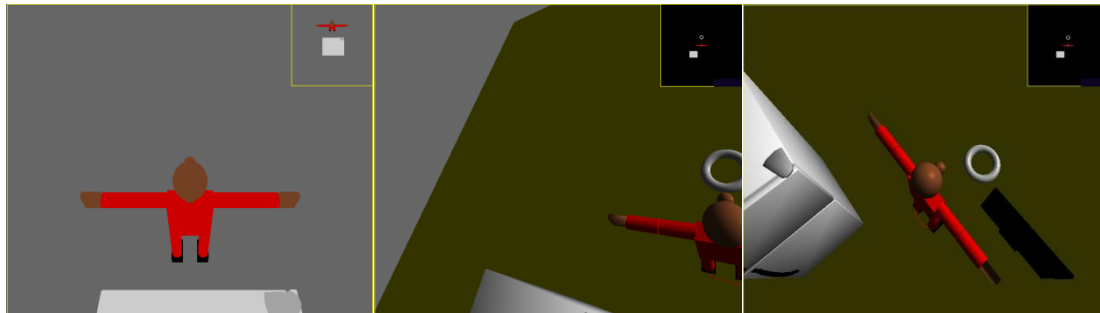
```

**Figure 313: Line History for the gluLookAt call containing John's implementation of the first-person View**

At (414) the student applies the spherical-coordinate approach successfully used in the development of the first-person view (discussed earlier) to the third-person View by adding spherical coordinates to the eye and lookAt points. This is a conceptual error as the spherical coordinates should only be applied to the eye value, not the lookAt value.

As a result as shown in the middle panel in Figure 314, both the camera's position and the point it looks at are rotated and the avatar does not remain in view as the camera is rotated. This error indicates the student incorrectly visualized the position and rotation of the camera's eye and lookAt points. The student then works on an incorrect solution during (415-418), modifying the fixed offset values of the eye and lookAt. At (418, 448) the student multiplies the View's spherical x and z

coordinates with a 'zoom' factor which allows the third-person View to be moved toward or away from the avatar. However, the View still does not properly orbit the avatar.



**Figure 314: The initial attempt at (227) using fixed points, and the incorrect lookAt using spherical coordinates at (614), the correct lookAt with spherical coordinates removed from the lookAt at (615)**

The student moves on to other tasks before returning to the implementation of the top-down View at (555). The student attempts a new solution, adding a `glRotate` call (rotating about the y-axis) in front of the View's `gluLookAt` call as shown in Figure 313, and when that doesn't work moves the rotate call after the `gluLookAt` statement at (556). This does not correct the `gluLookAt` statement's erroneous lookAt coordinate implementation. The student then surrounds the rotate and view statements with push/pop statements, but this merely eliminates the effect of the rotate and View statements altogether and also suggests errors in the student's model of transformations and their effect on the matrix stack.

(1383): (555, 557, 615, 618 : total = 4)	
555, MUTATED(O):	<code>glRotatef(yrot, 0, 1, 0);</code>
557, MOVED(O):	<code>MOVED (glRotatef(yrot, 0, 1, 0);)</code>
615, MUTATED(O):	<code>//glRotatef(yrot, 0, 1, 0);</code>
618, DELETED(O):	<code>DELETED (//glRotatef(yrot, 0, 1, 0);)</code>

**Figure 315: Line History showing a rotation applied in an attempt to 'orbit' the View**

The student corrects the `gluLookAt` statement at (615) by removing the spherical coordinates from the lookAt point, producing a working third-person view that orbits the avatar.

The student then tweaks the top-down view by reversing the direction in which the view's rotation operates and by modifying the fixed coordinates added to the eye position to position it behind the avatar at (616, 617, 689, 699, and 701).

### **Minimap View**

Implementation of the minimap 'View' is contained in segment [JOHN\\_A3.VIEW.1."Minimap"](#) [37]. The Line History for the line implementing the minimap's projection is shown in Figure 316, whereas the gluLookAt all producing the camera positioning is shown in Figure 317.

(537): (62, 65-66, 68, 71, 73-75, 258, 450 : total = 10)
62, MUTATED(O): gluLookAt(0,10,0, 0,0,0, 0,1,0);
65, MUTATED(O): gluLookAt(0,10,0, 0,0,0, 0,0,-1);
66, MUTATED(O): gluLookAt(0,20,0, 0,0,0, 0,0,-1);
68, MUTATED(O): gluLookAt(0,10,0, 0,0,0, 0,0,-1);
71, MUTATED(O): gluLookAt(0,10,0, 0,0,0, 1,0,0);
73, MUTATED(O): gluLookAt(0,40,0, 0,0,0, 1,0,0);
74, MUTATED(O): gluLookAt(0,250,0, 0,0,0, 1,0,0);
75, MUTATED(O): gluLookAt(0,25,0, 0,0,0, 1,0,0);
258, MUTATED(O): gluLookAt(guy.x,25,guy.z, 0,0,0, 1,0,0);
450, MUTATED(O): gluLookAt(guy.x,25,guy.z, guy.x,guy.y,guy.z, 1,0,0);

**Figure 316: Line History showing the gluLookAt call for John's implementation of the minimap view**

(526): (52, 56, 58, 62, 66, 247, 249-250 : total = 8)
52, MUTATED(O): gluOrtho2D(0, WIDTH, 0, HEIGHT);
56, MUTATED(O): gluOrtho2D(internalLeftX, internalRightX, internalLowerY, internalUpperY);
58, MUTATED(X): gluOrtho2D(internalLeft, internalRight, internalLower, internalUpper);
62, MUTATED(O): gluPerspective(60, WIDTH/HEIGHT, 0.1, 100.0);
66, MUTATED(O): gluPerspective(60, 1, 0.1, 100.0); // square aspect
247, MUTATED(X): glOrtho(0, 100, 0, 100, 0, 100);
249, MUTATED(O): glOrtho(-25, 25, -25, -25, 0, 50);
250, MUTATED(O): glOrtho(-25, 25, -25, 25, 0, 50);

**Figure 317: Line History containing the projection call for John's implementation of the minimap View**

The student begins setting up the minimap View at (52-53) but loads the new projection on top of the existing projection, creating a skewed view. This is fixed at (54) when the student loads the identity matrix into the projection matrix. Another error in the initial set-up is the use of gluOrtho2D (for two-dimensional orthographic projections) rather than gluOrtho (for three-dimensional projections) which leads to the scene not being drawn on the minimap at (61).

The projection is changed to a skewed but workable gluPerspective projection at (62), but the student also adds a gluLookAt statement in which the 'view vector' equals the 'up vector'; this 'breaks' the View and therefore nothing is visible despite the perspective having been fixed. The

student modifies the scene-drawing loop at (63) and removes it altogether at (64) to no effect. At (65) the student correctly modifies the `gluLookAt` statement by changing its up-vector, causing the scene to be rendered in the minimap.

After removing the skew from the projection at (66) the student attempts to have the minimap background (drawn via a `glRecti` call) use a separate `gluOrtho2D` projection at (69) as shown in Figure 318 but positions the before the modelview matrix is reset, leading to the background not being drawn in the correct place (and not being visible). The student uses multiple incorrect solution attempts. During (70-75) he disables the OpenGL depth test, modifies the up-vector of the `gluLookAt` call (since that was the approach that fixed the previous View problem he'd had earlier) and tweaks the fixed view's eye position.

```
glMatrixMode(GL_PROJECTION);
glViewport(POSX, POSY, SIZEX, SIZEY);
//Draw Background
glLoadIdentity();
gluOrtho2D(internalLeft, internalRight, internalLower, internalUpper);
glColor3ub(0,255,0); // green
glRecti(internalLeft, internalLower, internalRight, internalUpper);
//Draw Scene
glLoadIdentity();
gluPerspective(60, 1, 0.1, 100.0); // square aspect
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0,10,0, 0,0,0, 0,0,-1);

glObjects[0]->draw();
```

**Figure 318: Source code for drawing the minimap background with a separate projection at (69)**

The student fixes part of the problem at (76) by resetting the modelview matrix before drawing the minimap background. However, the student fails to reset the projection matrix after drawing the background, thereby loading the minimap projection onto the `gluOrtho2D` projection used to draw the background, leading to only the background being drawn (with the projection being skewed when the scene is drawn, leading it to not render to screen). The student changes the solid background fill to a border at (77-78), most likely to see whether the scene is being overdrawn by the background (which it is not). The student fixes the error at (79) by resetting the modelview matrix after the drawing of the background. With the minimap in working order, the student turns to working on other problems until (247).

At (247) the student replaces the perspective projection used for the minimap with an orthogonal (glOrtho) projection. However, the boundaries of the new projection are chosen from 0-100, meaning the origin (the center of the scene) is at the lower-right point of the projection and the scene is not very visible. The student corrects the projection at (249-250), producing a working orthogonal projection. The minimap's lookAt is modified at (418) to be centered on the avatar's position. Finally, from (410-418) the minimap view is made to rotate with the avatar by copying the orbiting technique (based on spherical coordinates) from the third-person view.

The final view meets all assignment requirements and functions. The major problems encountered by the student involve pipeline issues of not resetting modelview and projection matrices, which at several points leads to multiple views / projections being loaded onto the matrix stack. This produces a blank view, which causes the student to misidentify the problem as it does not provide any information on what particular part of the set-up code is responsible for the problem. The student also at one point produces a gluLookAt call with equal view and up vectors.

#### 9.7.7.1.1.5.2 Ida

##### First Person View

Ida's work on the first-person View is captured in segment [IDA\\_A3.VIEW.1."View"](#) [17] and occurs from (381-400) as shown in Figure 319. It seems the student may have experimented with and implemented the View in an external project or during the practical exercise on Viewing because an almost-correct spherical view is added at (381).

(1275): (381, 391-392, 394, 399-400 : total = 6)

```
381, MUTATED(X): gluLookAt(
    position[xc] ,position[yc]+7, position[z],
    position[xc]+cos(angle) ,position[yc]+7, position[z]+sin(angle), 0, 1, 0);
391, MUTATED(O): gluLookAt(
    position[xc], position[yc]+7, position[z],
    position[xc]+cos(angle), position[yc]+7, position[z]-sin(angle), 0, 1, 0);
392, MUTATED(O): gluLookAt(
    position[xc], position[yc], position[z],
    position[xc]+cos(angle), position[yc], position[z]-sin(angle), 0, 1, 0);
394, MUTATED(O): gluLookAt(
    position[xc], position[yc]+7, position[z],
    position[xc]+cos(angle), position[yc]+7, position[z]-sin(angle), 0, 1, 0);
399, MUTATED(O): gluLookAt(
```

```

        position[xc], position[yc]+7, position[zc],
        position[xc]+cos(angle), position[yc]+7, position[zc]-sin(angle), 0, 1, 0);
400, MUTATED(O): gluLookAt(
        position[xc], position[yc]+7, position[zc],
        position[xc]+cos(angle), position[yc]+7, position[zc]-sin(angle), 0, 1, 0);

```

**Figure 319: Line History containing Ida's gluLookAt implementation for the first-person View**

Data on time between Changes supports this hypothesis. The 32-minute gap (equivalent to around 21-32 Changes) between the Change in which the spherical look-at is implemented at (381) and the preceding Change (380) indicates that the student most likely implemented a spherical view external to the project and/or worked out the view on paper or discovered an external resource detailing the implementation of such a view.

The only real modification is the change of the sign preceding the sine function at (391). After that the student experiments with removing the offsets added to the eye's and lookAt's y-coordinate at (392) before re-adding them at (394). The remaining Changes at (399) and (400) are formatting-related.

#### **Top Down View (replacing the Minimap View)**

The student implements the top-down view (the student's version of the minimap view) in Changes (381, 395-399, 644, 649-650, 661, 666). Development of the lookAt call is shown in Figure 320; the two projections used are shown in Figure 321.

(1262): (381, 395-399, 644, 649-650, 661, 666 : total = 11)
381, MUTATED(X): gluLookAt(at,at,at,0,0,0, 0,1,0);
395, MUTATED(X): gluLookAt(position[x], 10, position[z], 0, 0, 0, 0, 1, 0);
396, MUTATED(O): gluLookAt(position[xc], 10, position[zc], 0, 0, 0, 0, 1, 0);
397, MUTATED(O): gluLookAt(0, 10, 0, 0, 0, 0, 0, 1, 0);
398, MUTATED(O): gluLookAt(position[xc], 10, position[zc], position[xc], 0, position[zc], 0, 1, 0);
399, MUTATED(O): gluLookAt(position[xc], 10, position[zc], position[xc], 0, position[zc], 1, 0, 0);
644, MUTATED(O): gluLookAt(position[xc], 9, position[zc], position[xc], 0, position[zc], 1, 0, 0);
649, MUTATED(O): gluLookAt(position[xc], radius+5, position[zc], position[xc], 0, position[zc], 1, 0, 0);
650, MUTATED(O): gluLookAt(position[xc], 9, position[zc], position[xc], 0, position[zc], 1, 0, 0);
661, MUTATED(O): gluLookAt(position[xc], 10, position[zc], position[xc], 0, position[zc], 1, 0, 0);
666, MUTATED(O): gluLookAt(position[xc], 9, position[zc], position[xc], 0, position[zc], 1, 0, 0);

**Figure 320: Line History for the lookAt call implementing Ida's top-down View**

```
(1259): (381, 395 : total = 2)
```

```
381, MUTATED(X): gluPerspective(60, 800.0/600.0, 0.1, 100.0);
```

```
395, MUTATED(X): glOrtho(-30, 30, -30, 30, 0, 60);
```

**Figure 321: Line History implementing the projection for Ida's minimap View**

The orthogonal projection is added correctly at (395) and is not modified in any way after initially being added. This indicates the student may have looked for a worked example or previous practical exercise or experimented with the projection beforehand (which is also suggested by the 9 minutes between the current and previous versions).

The student also begins working on the camera at (395). The student changes the view's eye coordinate to equal the avatar's position, but does not modify the lookAt coordinate and hence leaves the camera looking at the origin. At (397) the student removes the avatar's position from the eye position and leaves the eye positioned at the fixed position (0,10,0) looking at the origin. However, this approach does not produce a working view because the view vector (eye-lookAt) is aligned with the same axis (y-axis) as the up-vector.

The student reintroduces the avatar's position at (398), correctly utilising it for both the eye and the lookAt point x and z values, but does not change the up-vector and leaves it aligned with the view vector. This means the View still does not display anything. The student corrects this error at (399) by setting the up-axis to be the x-axis which produces a working top-down view.

The student tweaks the view slightly at (644) before adding a variable to the eye's y-value. Based on the name of the variable ('radius') the student means to enable the view to rotate about the avatar from the top, but in fact changing the y-position would merely zoom/unzoom the view. The variable is removed at (650), after which the student tweaks the view's y-position (661, 666) and completes work on the View.

Overall, the student implements the view fairly quickly. The student initially fails to set the gluLookAt call's lookAt point, encounters problems when the lookAt's view vector equals the up vector at (397) and (398) and shows a lack of understanding regarding the effect of the modification of the eye's y-coordinate at (649).

### **Third-Person View**

The student implements the third-person camera at (573) and (575) as shown in Figure 322. It appears that the student again consults external materials and/or experiments with the camera in a



different project, because the time difference to the preceding version is 29 minutes, after which the view is added almost correctly with one typing error which is corrected at (575).

```
(1269): (381, 573, 575 : total = 3)
381, MUTATED(X): gluLookAt(at,at,at,0,0,0, 0,1,0);
573, MUTATED(X): gluLookAt(position[xc]-radius*cos(yrot*3.14/180), 11,
position[yc]+radius*sin(yrot*3.14/180), position[xc], position[yc]+7, position[zc],0,1,0);
575, MUTATED(O): gluLookAt(position[xc]-radius*cos(yrot*3.14/180), 11,
position[zc]+radius*sin(yrot*3.14/180), position[xc], position[yc]+7, position[zc],0,1,0);
```

**Figure 322: Line History showing the implementation of the lookAt call producing the third-person View**

The implemented camera includes the ability to orbit the avatar and to zoom and unzoom from the avatar. The period of 29 minutes (equivalent to 19-29 Changes based on average time per Change) indicates that significant effort was expended in the implementation of the view. However, since the implementation did not occur inside the project and details are unavailable what if any problems the student faced is unknown.

### **Origin View**

The student implements a fourth View which has its lookAt fixed at the centre of the room (the associated Line History is shown in Figure 323). It is implemented at (605) by copying the third-person camera view and removing the avatar's position from it. The remainder of Changes (606-608) tweak the eye and lookAt coordinate by constant factors. No real problems occur since it is a de-facto copy of the third-person view.

```
(421): (11, 381, 573, 605-609 : total = 8)
11, MUTATED(O): gluLookAt(at,at,at , 0,0,0, 0,1,0);
381, MUTATED(X): gluLookAt(at,at,at,0,0,0, 0,1,0);
573, MUTATED(X): gluLookAt(at,at,at,0,0,0,0,1,0);
605, MUTATED(O): gluLookAt(-radius*cos(yrot*3.14/180), 11, radius*sin(yrot*3.14/180), 0, 7,
0,0,1,0);
606, MUTATED(O): gluLookAt(-radius*cos(yrot*3.14/180), 11, radius*sin(yrot*3.14/180), 0, 0, 0 ,
1 , 0);
607, MUTATED(O): gluLookAt(-radius*cos(yrot*3.14/180), 15, radius*sin(yrot*3.14/180), 0, 0, 0 ,
1 , 0);
608, MUTATED(O): gluLookAt(-2*radius*cos(yrot*3.14/180), 15, 2*radius*sin(yrot*3.14/180), 0, 0, 0
, 0 , 1 , 0);
```

```
609, MUTATED(O): gluLookAt(-radius*cos(yrot*3.14/180), 15, radius*sin(yrot*3.14/180), 0, 0, 0 , 0 ,
1 , 0);
```

**Figure 323: Line History showing the implementation of the Origin View**

### 9.7.7.1.1.5.3 Thomas

#### Third-person camera

Thomas begins working on Views by implementing a third-person View in segment [THOMAS\\_A3.VIEW.1."Third-Person Camera"](#) [21]. The Line History detailing the development of the associated lookAt statement is shown in Figure 324.

```
(1144): (7, 509-522, 524, 527-528, 531-532 : total = 20)
7, MUTATED(O): gluLookAt(at,at,at , 0,0,0, 0,1,0);

//uses avatar's current position as eye position, but looks at origin
509, MUTATED(O): gluLookAt(currentPosX,at,currentPosY , 0,0,0, 0,1,0);

//goes back to using at variable as eye position
510, MUTATED(O): gluLookAt(at,at,at , 0,0,0, 0,1,0);

//adds spherical coordinates (correctly) for eye position, but looks at origin
511, MUTATED(X): gluLookAt(
    currentPosX-cos(direction*PI/180.0)*at,at,currentPosY-sin(direction*PI/180.0)*at ,
    0,0,0, 0,1,0);

//corrects GP error
512, MUTATED(O): gluLookAt(
    currentPosX-cos(direction*PI/180.0)*at,at,currentPosY-sin(direction*PI/180.0)*at ,
    0,0,0, 0,1,0);

//removes spherical coordinates, for the first time modifies lookAt. LookAt set to avatar's position (correct),
now have an eye looking from (at,at,at) at avatar
513, MUTATED(O): gluLookAt(
    at,at,at ,
    currentPosX,0,currentPosY, 0,1,0);

//also adds the current position to the eye coordinate, making it a sum of the current position and the at
variable
514, MUTATED(O): gluLookAt(
```

```

currentPosX-at,at,currentPosY-at ,
currentPosX,0,currentPosY, 0,1,0);

//adds spherical coordinates to the eye
515, MUTATED(O): gluLookAt(
    currentPosX-sin(direction*PI/180.0)*at,at,currentPosY-cos(direction*PI/180.0)*at ,
    currentPosX,0,currentPosY, 0,1,0);

//reverses direction of angle
516, MUTATED(O): gluLookAt(
    currentPosX-sin(-direction*PI/180.0)*at,at,currentPosY-cos(-direction*PI/180.0)*at ,
    currentPosX,0,currentPosY, 0,1,0);

//stops multiplication of eye coordinate with at variable
517, MUTATED(O): gluLookAt(
    currentPosX-sin(-direction*PI/180.0)*at,at,currentPosY-cos(-direction*PI/180.0) ,
    currentPosX,0,currentPosY, 0,1,0);

//stops multiplication of eye coordinate with at variable
518, MUTATED(O): gluLookAt(
    currentPosX-sin(-direction*PI/180.0),at,currentPosY-cos(-direction*PI/180.0) ,
    currentPosX,0,currentPosY, 0,1,0);

//multiplies eye coordinate with fixed value (20)
519, MUTATED(O): gluLookAt(
    currentPosX-sin(-direction*PI/180.0)*20,at,currentPosY-cos(-direction*PI/180.0)*20 ,
    currentPosX,0,currentPosY, 0,1,0);

//modifies polar coordinate function, swaps sine and cosine
520, MUTATED(O): gluLookAt(
    currentPosX-cos(-direction*PI/180.0)*20,at,currentPosY-sin(-direction*PI/180.0)*20 ,
    currentPosX,0,currentPosY, 0,1,0);

//reverses direction of angle
521, MUTATED(O): gluLookAt(
    currentPosX-cos(direction*PI/180.0)*20,at,currentPosY-sin(direction*PI/180.0)*20 ,
    currentPosX,0,currentPosY, 0,1,0);

```

```

//multiplies eye coordinate with at variable
522, MUTATED(O): gluLookAt(
    currentPosX-cos(direction*PI/180.0)*at,at,currentPosY-sin(direction*PI/180.0)*at ,
    currentPosX,0,currentPosY, 0,1,0);

//multiplies eye by further camIN value to produce zoom, but incorrectly excludes y from multiplication
524, MUTATED(O): gluLookAt(
    currentPosX-cos(direction*PI/180.0)*at*camIN,at,currentPosY-sin(direction*PI/180.0)*at*camIN ,
    currentPosX,0,currentPosY, 0,1,0);

//adds y to multiplication
527, MUTATED(O): gluLookAt(
    currentPosX-cos(direction*PI/180.0)*at*camIN,at*camIN,currentPosY-sin(direction*PI/180.0)*at*camIN ,
    currentPosX,0,currentPosY, 0,1,0);

//tweaks lookAt point upward along y-axis
528, MUTATED(O): gluLookAt(
    currentPosX-cos(direction*PI/180.0)*at*camIN,at*camIN,currentPosY-sin(direction*PI/180.0)*at*camIN ,
    currentPosX,4,currentPosY, 0,1,0);

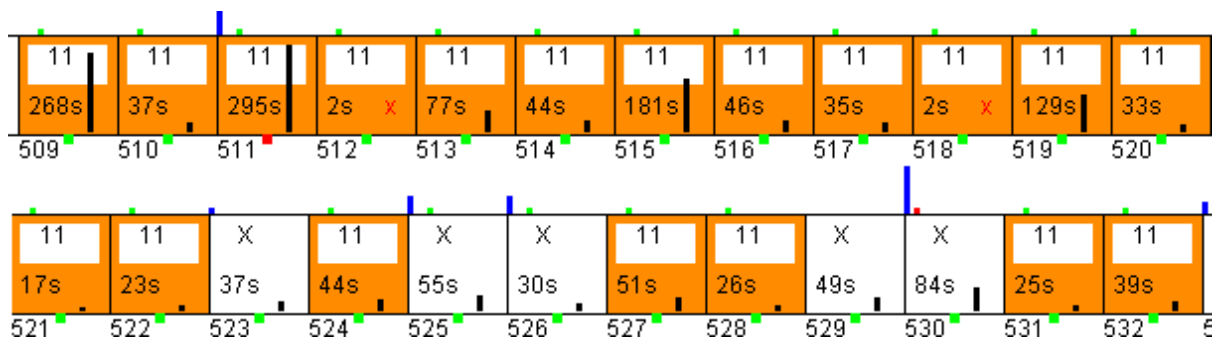
//adds another direction variable camDir to allow orbiting of view, but forgets converting to radian
531, MUTATED(O): gluLookAt(
    currentPosX-cos(camDIR+direction*PI/180.0)*at*camIN,at*camIN,currentPosY-
    sin(camDIR+direction*PI/180.0)*at*camIN ,
    currentPosX,4,currentPosY, 0,1,0);

//converts to radian
532, MUTATED(O): gluLookAt(
    currentPosX-cos(camDIR*PI/180.0+direction*PI/180.0)*at*camIN,at*camIN,currentPosY-
    sin(camDIR*PI/180.0+direction*PI/180.0)*at*camIN ,
    currentPosX,4,currentPosY, 0,1,0);

```

**Figure 324: Line History for the lookAt function implementing Thomas's third-person View**

The student begins implementation of the view by setting the eye value of the gluLookAt call to equal the avatar's position at (509) but does not modify the lookAt point, leading to the camera being located at the avatar's position but stuck looking at the origin. As can be seen in the timeline in Figure 325, the Change involves a moderately long thinking phase of 268 seconds.



**Figure 325: Timeline showing Changes for the implementation of the third-person View**

After considering his next action for 295 seconds, the student adds spherical coordinates to the eye position (the avatar's position) at (511) together with a direction variable to enable the camera to 'look around'. However, the student still does not realise that the `lookAt` variable must also be modified, so the camera can only look from the avatar's position (slightly offset by the spherical coordinates) to the origin.

The student removes the (correct) spherical coordinates at (513-514) replacing them by a static variable (called `at`) deducted from the `x` and `z` axes of the eye point. The student also sets the `lookAt` point to equal the avatar's position, thereby creating a working camera view looking from (`currentPosX-at,at,currentPosY-at`) towards (`currentPosX,0,currentPosY`). However because the student does not use spherical coordinates it lacks the ability to orbit or rotate with the avatar's head.

The student then reintroduces spherical coordinates at (515) to enable rotation of the camera. The modification is preceded by a thinking pause of 181 seconds. The student spends Changes (515-522) fixing the formulae used to calculate the eye coordinates, including swapping the order of the sine and cosine functions and reversing the direction of rotation. As the timeline shows, after (519) the student does not spend much time on each Change, suggesting the following Changes are relatively trivial.

At (524) the student adds an additional variable to the eye's coordinates to enable 'zooming' away from or towards the avatar, but he initially applies this variable only to the `x` and `z` coordinates which produces an incorrect zoom effect. This error is corrected at (528) by also using the variable to modify the `y`-coordinate.

The student adds another variable which stores an angle added to the avatar's direction angle used in the calculation of spherical coordinates. This allows the manual orbiting of the view at (531-532), producing a third-person view that follows the avatar's position and head rotation, can be zoomed

towards or away from the avatar and can orbit around the avatar, thereby fulfilling all goals for the third-person view.

### **First-person View**

Thomas next implements the first-person View in Changes (533-539). The associated Line History is shown in Figure 326. The state of the View in selected Changes is shown via screen-capture in Figure 327.

```
(2135): (533-539 : total = 7)
//Copies from top-down view, replaces at and camIn variable multiplication with static multiplication
by 0.1 for x and z coordinates; should apply polar coordinates to lookAt instead of eye
533, MUTATED(O): gluLookAt(
posX-cos(PI/180.0+direction*PI/180.0)*0.1,at*camIn,posY-sin(PI/180.0+direction*PI/180.0)*0.1 ,
posX,4,posY, 0,1,0);

//View looks upward almost vertically, since eye=(-0.1, 3.2, -0.1) and lookAt (0.0, 4.0, 0.0) produces a
view vector with a very large y-component
534, MUTATED(O): gluLookAt(
posX-cos(PI/180.0+direction*PI/180.0)*0.1,4,posY-sin(PI/180.0+direction*PI/180.0)*0.1 ,
posX,4,posY, 0,1,0);

//Student tweaks y-value to correct the problem
535, MUTATED(O): gluLookAt(
posX-cos(PI/180.0+direction*PI/180.0)*0.1,3,posY-sin(PI/180.0+direction*PI/180.0)*0.1 ,
posX,4,posY, 0,1,0);

536, MUTATED(O): gluLookAt(
posX-cos(PI/180.0+direction*PI/180.0)*0.1,4,posY-sin(PI/180.0+direction*PI/180.0)*0.1 ,
posX,4,posY, 0,1,0);

537, MUTATED(O): gluLookAt(
posX-cos(PI/180.0+direction*PI/180.0)*0.1,5,posY-sin(PI/180.0+direction*PI/180.0)*0.1 ,
posX,4,posY, 0,1,0);

538, MUTATED(O): gluLookAt(
```

```
posX-cos(direction*PI/180.0)*0.1,3.2,posY-sin(direction*PI/180.0)*0.1 , posX,4,posY, 0,1,0);

//Tweaks eye's x and z values instead, increasing them so the vector's y-component is less significant
539, MUTATED(O): gluLookAt(
posX-cos(direction*PI/180.0)*3.2,3.2,posY-sin(direction*PI/180.0)*3.2 , posX,4,posY, 0,1,0);
```

**Figure 326: Line History showing the implementation of the lookAt call for the first-person View**



**Figure 327: The first-person View at (533, 534-536, 537, 538, 539)**

The initial `gluLookAt` call is copied from the third-person view previously implemented, but the student removes the `camIn` variable that was used for zooming in the third-person view with a multiplication by a static variable.

Copying the `gluLookAt` statement is not optimal, as the third-person camera looked onto the avatar from above and behind and therefore offset the eye point from the avatar's position (with the `lookAt` point equalling the avatar's position), whereas the first-person view should offset the `lookAt` point from the avatar's position and have the eye point equal the avatar's head position.

The students' attempt at (534) in fact produces a good first-person view (though the eye is offset behind the avatar, rather than the `lookAt` point being offset to in front of the avatar). However as the second panel in Figure 327 shows, the complete lack of any other objects or reference points such as walls makes it impossible to determine from running the application whether the View is working correctly or not at all. As a result, the student continues working on the View.

At (535) the student modifies the eye's `y`-value, setting it to be 1 lower than the `lookAt`'s `y`-value. This causes the view to look upward almost vertically. However, the application's visible output remains the same due to the lack of objects to serve as reference points. Modification of the eye's `y`-value to be higher than the `lookAt` value at (537) makes the view vector point almost vertically downward (middle panel, Figure 327).

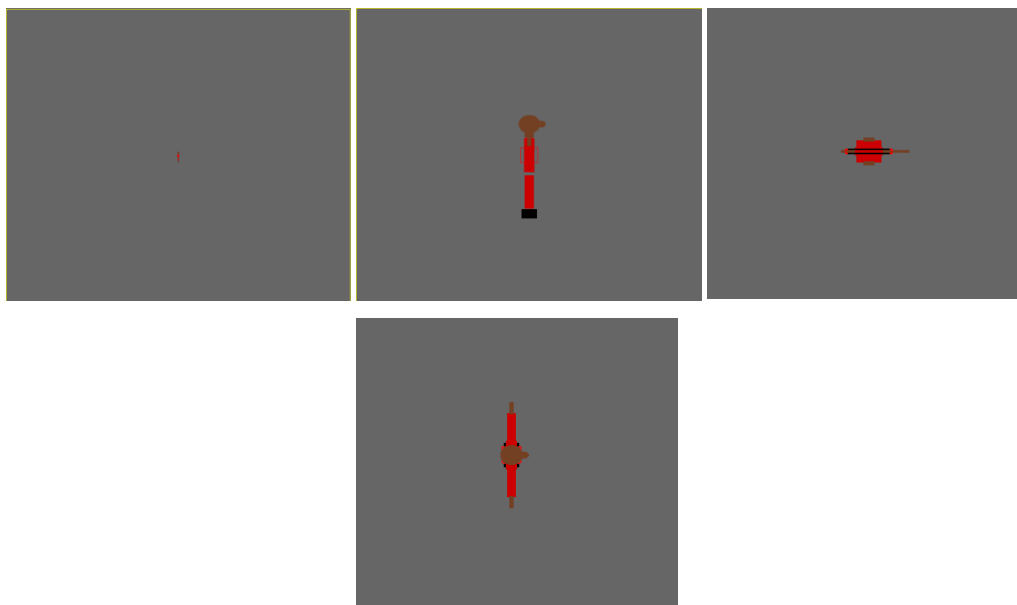
The student addresses this problem at (539) by modifying the eye's `x` and `z` offsets, making them larger to decrease the view vector's angle to the `x` and `z` axes. This positions the camera above and behind the avatar rather than at the avatar's eye level. While the View is in fact not as good a first-

person View as the View produced in (534) it shows part of the avatar's head (compared to the correct View, which showed nothing as the room is empty), thus allowing the student to understand where it is positioned. This is the final implementation of the View. The student adds code to ensure the head is not drawn when in first-person view in order to prevent it from obscuring the View from (540-545), completing the implementation of the first-person camera.

Overall, the camera implementation does meet the assignment requirements; however, the confusion regarding the setting of eye and lookAt variables, probably increased by the lack of visible objects to add at reference points to better understand the way the view is working is an example of how students tend to struggle in trying to understand spatial problems when lacking a functional way to visualise and understand the effect of implemented changes.

### **Ortho Minimap View**

Thomas's implementation of a minimap View is captured in segment [THOMAS\\_A3.VIEW.2."Ortho/Top-down Camera"](#) [20]. However, instead of implementing it as a separate mini-map displayed in the corner of the screen as the assignment specification suggests, the student implements a top-down View. Screen captures of different stages of implementation are shown in Figure 328. Line Histories associated with this work are shown in Figure 329 and Figure 330.



**Figure 328: The top-down View at (550), (554), (556) and the final View at (560) after modification of the rotation call**



(2149): (556, 559, 561 : total = 3)
556, MUTATED(O): glRotatef(90,1,0,0);
559, MUTATED(O): glRotatef(-90,1,0,0);

**Figure 329: Line History showing the rotate call designed to produce a top-down View by rotating the scene**

(2138): (542-543, 546-547, 549, 550-555, 557-558, 560 : total = 14)
542, MUTATED(O): glOrtho(-100,100,-100,100,100,100);
543, MUTATED(O): glOrtho(-1000,1000,-1000,1000,-1000,100);
546, MUTATED(O): glOrtho(-10,10,-10,10,-100,100);
547, MOVED(X): MOVED (glOrtho(-10,10,-10,10,-100,100);)
549, DELETED(O): DELETED (glOrtho(-10,10,-10,10,-100,100);)
550, GHOST(X): gluOrtho2D(-100,100,-100,100);
551, MUTATED(O): gluOrtho2D(-100,100,-100,100);
552, MUTATED(O): gluOrtho2D(-10,10,-10,10);
553, MUTATED(X): gluOrtho(-10,10,-10,10,-10,10);
554, MUTATED(O): glOrtho(-10,10,-10,10,-10,10);
555, MUTATED(O): glOrtho(-10,10,-10,10,-100,100);
557, MUTATED(O): glOrtho(-100,100,-100,100,-100,100);
558, MUTATED(O): glOrtho(-10,10,-10,10,-10,10);
560, MUTATED(O): glOrtho(-20,20,-20,20,-20,20);

**Figure 330: Line History showing the implementation of the top-down View's projection**

The student begins implementing the orthogonal view at (542), adding a glOrtho call. The student also resets the modelview matrix. However, the student does not reset the projection matrix; as a result, the orthogonal projection is multiplied on top of the perspective projection used for the other views, leading to nothing being visible in the view. Misidentifying the problem, the student adds a gluLookAt call at (545) and a glTranslate call at (547) to debug the broken view. The student also tweaks the glOrtho call's parameters at (543, 545, and 546). None of these modifications produces any output on the screen due to the broken projection.

At (550), the student removes the glOrtho call and the code resetting the modelview matrix, replacing it with a gluOrtho2D call and code to reset the projection matrix which addresses the problem introduced at (542). However, due to the settings of the projection the avatar appears very small. The student then tweaks the projection's settings from (551-555). At (554) a good projection is produced (as shown in the middle panel of Figure 328), but at (555) the student skews this projection.

Since the view should be top-down, the student adds a rotation to the scene at (556). The resulting view shows that the projection is skewed, which the student addresses at (557-558). Once the skewing is removed, it becomes apparent that the rotation added rotates the wrong way and is displaying a bottom-up view of the scene, so the student reverses the rotation angle at (559). Finally the student tweaks the projection limits once more at (561).

Overall, the student implemented the orthogonal projection quickly and efficiently, with the only major problem being the OpenGL-related initial problem of resetting the modelview instead of the projection matrix.

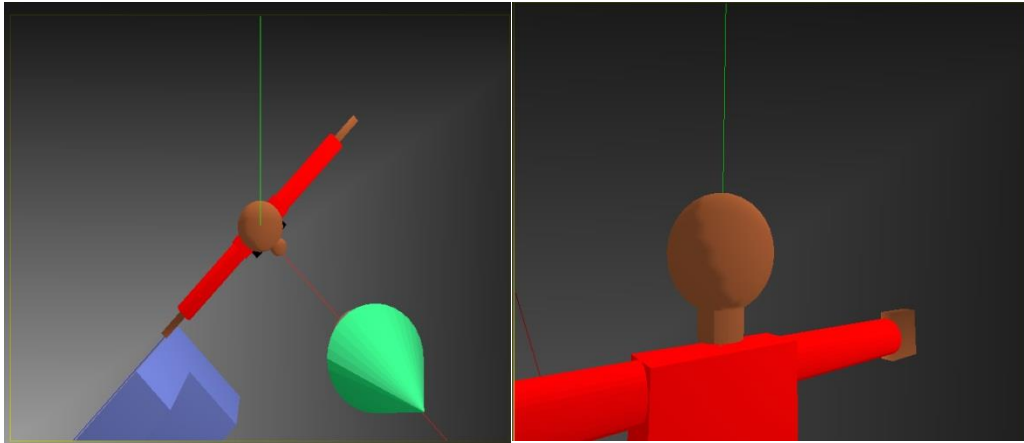
#### 9.7.7.1.1.5.4 Michael

##### Third-Person View

The student works on the implementation of the top-down View in Segment [MICHAEL\\_A3.VIEW.1."View" \[35\]](#). Two states of the view are shown in Figure 332, whereas work on the view is shown in Figure 333. The student uses fixed values for the eye position, leaving the lookAt position unchanged. While working on the view, the student makes axis changes at (117), (122), (125) and (126) with the remainder of Changes being tweak modifications to eye values.

(2144): (113, 115-117, 121-126, 191-193 : total = 13)
113, MUTATED(O): if (view1)gluLookAt(at,at,at , 0,0,0, 0,1,0);
>>> 115, MUTATED(X): if (view2)gluLookAt(0,0,at , 0,0,0, 0,1,0);
>>> 116, MUTATED(X): else if (view2)gluLookAt(0,0,at , 0,0,0, 0,1,0);
117, MUTATED(X): gluLookAt(0,20,0 , 0,0,0, 0,1,0);
121, MUTATED(O): gluLookAt(0,10,0 , 0,0,0, 0,1,0);
122, MUTATED(O): gluLookAt(5,10,5 , 0,0,0, 0,1,0);
123, MUTATED(O): gluLookAt(1,10,1 , 0,0,0, 0,1,0);
124, MUTATED(O): gluLookAt(1,15,1 , 0,0,0, 0,1,0);
125, MUTATED(O): gluLookAt(0,15,0 , 0,0,0, 0,1,0);
126, MUTATED(O): gluLookAt(0.1,15,0.1 , 0,0,0, 0,1,0);
191, MUTATED(O): gluLookAt(0.1,150,0.1 , 0,0,0, 0,1,0);
192, MUTATED(O): gluLookAt(0.1,50,0.1 , 0,0,0, 0,1,0);
193, MUTATED(O): gluLookAt(0.1,15,0.1 , 0,0,0, 0,1,0);

Figure 331: Line History showing development of the Third-person view



**Figure 332: The Third-Person View in (193) and (248)**

Despite the considerable work put into putting the View into the correct position, the final view is fixed. It cannot be rotated and it does not follow the avatar, thereby not fulfilling the assignment specifications.

### **First-Person View**

The student next implements the first-person View in segment [MICHAEL\\_A3.VIEW.1."View"](#) [35], as shown Figure 333.

(2293): (231-233, 235-243, 248 : total = 13)

```
231, MUTATED(O): gluLookAt(headx,heady,headz, 0,0,0, 0,1,0);
232, MUTATED(O): gluLookAt(headx,heady,headz-1, 0,0,0, 0,1,0);
233, MUTATED(O): gluLookAt(headx,heady,headz, 1,5,0, 0,1,0);
235, MUTATED(O): gluLookAt(headx-2,heady,headz, 1,5,0, 0,1,0);
236, MUTATED(O): gluLookAt(headx-2,heady,headz, 1,7,0, 0,1,0);
237, MUTATED(O): gluLookAt(headx-3,heady,headz, 1,7,0, 0,1,0);
238, MUTATED(O): gluLookAt(headx-3.5,heady,headz, 1,7,0, 0,1,0);
239, MUTATED(O): gluLookAt(headx,heady,headz-3.5, 1,7,0, 0,1,0);
240, MUTATED(O): gluLookAt(headx-3.5,heady,headz-3.5, 1,7,0, 0,1,0);
241, MUTATED(O): gluLookAt(headx-3.5,heady,headz-1.5, 1,7,0, 0,1,0);
242, MUTATED(O): gluLookAt(headx-3.5,heady,headz-2.0, 1,7,0, 0,1,0);
243, MUTATED(O): gluLookAt(headx-3.5,heady,headz-2.0, headx+3.5,7,headz+3.5,0,1,0);
248, MUTATED(O): gluLookAt(-3.5,6,-2.0, 3.5,7,3.5,0,1,0);
```

**Figure 333: Line History showing development of the First-person view**

The student utilises the avatar's head position as the eye position, but for most of the development uses a static point as the lookAt position, meaning that the view always looks towards a static point.

At (243) the student adds the head position to the lookAt point. This would produce a partially working view which would move along with the avatar but would be unable to rotate / 'look around'. However, the student removes the head's position from both the eye and the lookAt point at (248).

The result is another static view which neither moves with the avatar nor is able to rotate or orbit, therefore failing the assignment specification. This result comes despite the 13 Changes the student spent modifying the view, all but one of which were used in changing the static coordinates for the lookAt point or the values added to the eye point.

### **Manually Moveable View**

The final view implemented at (117, 204-205, 414, 417-418, 425) as shown in Figure 334 is the student's only effort at implementing a non-static view. The student implements a view which has its eye position and lookAt position which are separately moveable via keyboard keys. However, neither position is adjusted with avatar movement, and spherical coordinates are not used to produce a 'look-around' view. The student does not produce a minimap view.

(2171): (117, 204-205, 414, 417-418, 425 : total = 7)
117, ADDED(X): gluLookAt(at,at,at , 0,0,0, 0,1,0);
204, MUTATED(O): gluLookAt(at,10,10 , 0,0,0, 0,1,0);
205, MUTATED(O): gluLookAt(at,at,at , 0,0,0, 0,1,0);
414, MUTATED(X): gluLookAt(atx,aty,atz , 0,0,0, 0,1,0);
417, MUTATED(O): gluLookAt(atx,aty,atz , 0,0,0+atz, 0,1,0);
418, MUTATED(O): gluLookAt(atx,aty,atz , 0,0,0+lookatz, 0,1,0);
425, MUTATED(O): gluLookAt(atx,aty,atz , 0+lookatz,0,0+lookatz, 0,1,0);

**Figure 334: Line History showing the development of a View that is manually moveable via the keyboard**

Despite spending many Changes on implementing the three views, the student does not develop a view that moves along with the avatar or a view that is able to 'look around', thereby failing the core assignment specifications for viewing; two views are completely static, with the third only being modifiable via manual input.

### **9.7.7.1.1.5.5 Christopher**

#### **First-Person View**

Christopher implements the first-person View in segment [CHRISTOPHER\\_A3.VIEW.1."FP Camera" \[47\]](#) and [CHRISTOPHER\\_A3.VIEW.3."View Spherical Coordinates" \[14\]](#). The associated Line History is shown in Figure 335.

(1690): (608, 611-612, 614, 653-654, 688, 712, 975, 977-982, 985-986, 1141-1142 : total = 19)

608, : gluLookAt(,at,at , 0,0,0, 0,1,0);

611, : gluLookAt(,at,at , fromX,fromY,fromZ, 0,1,0);

(GP) 612, : gluLookAt(,at,at , fromX,fromY,fromZ, 0,1,0);

(GP) 614, : gluLookAt(at,at,at , fromX,fromY,fromZ, 0,1,0);

653, : gluLookAt(fromX+1,fromY,fromZ, at,at,at , 0,1,0);

654, : gluLookAt(fromX,fromY,fromZ, fromX+1,fromY,fromZ, 0,1,0);

688, : gluLookAt(fromX+1,fromY,fromZ, fromX,fromY,fromZ, 0,1,0);

692 – break added

712, : gluLookAt(fromX,fromY,fromZ, fromX+1,fromY,fromZ, 0,1,0);

975, : gluLookAt(

fromX,fromY,fromZ,

(cos (scene->getNodeByName("Torso")->rotation.y\*PI/180)\*fromX)+1,fromY,-sin (scene->getNodeByName("Torso")->rotation.y\*PI/180)\*fromZ, 0,1,0);

977, : gluLookAt(

fromX,fromY,fromZ,

(-cos (scene->getNodeByName("Torso")->rotation.y\*PI/180)\*fromX)+1,fromY,-sin (scene->getNodeByName("Torso")->rotation.y\*PI/180)\*fromZ, 0,1,0);

(GP) 978, : gluLookAt(

fromX,fromY,fromZ,

(-cos (scene->getNodeByName("Torso")->rotation.y\*PI/180)\*fromX)+1,fromY,+sin (scene->getNodeByName("Torso")->rotation.y\*PI/180)\*fromZ, 0,1,0);

(GP) 979, : gluLookAt(

fromX,fromY,fromZ,

(-cos (scene->getNodeByName("Torso")->rotation.y\*PI/180)\*fromX)+1,fromY,sin (scene->getNodeByName("Torso")->rotation.y\*PI/180)\*fromZ, 0,1,0);

(GP) 980, : gluLookAt(

fromX,fromY,fromZ,

(-cos (scene->getNodeByName("Torso")->rotation.y\*PI/180)+1,fromY,sin (scene->getNodeByName("Torso")->rotation.y\*PI/180), 0,1,0);

981, : gluLookAt(

fromX,fromY,fromZ,

-cos (scene->getNodeByName("Torso")->rotation.y\*PI/180)+1,fromY,sin (scene->getNodeByName("Torso")->rotation.y\*PI/180), 0,1,0);

```

982, : gluLookAt(
    fromX,fromY,fromZ,
    fromX+cos (scene->getNodeByName("Torso")->rotation.y*PI/180),fromY,fromZ+sin (scene-
>getNodeByName("Torso")->rotation.y*PI/180), 0,1,0);
985, : gluLookAt(
    fromX,fromY,fromZ,
    fromX-cos (scene->getNodeByName("Torso")->rotation.y*PI/180W),fromY,fromZ-sin (scene-
>getNodeByName("Torso")->rotation.y*PI/180), 0,1,0);
986, : gluLookAt(
    fromX,fromY,fromZ,
    fromX+cos (scene->getNodeByName("Torso")->rotation.y*PI/180),fromY,fromZ-sin (scene-
>getNodeByName("Torso")->rotation.y*PI/180), 0,1,0);
(GP) 1141,: gluLookAt(
    fromX,fromY,fromZ,
    fromX+cos (scene->getNodeByName("Torso")->rotation.y*PI/180),fromY,fromZ-sin (scene-
>getNodeByName("Torso")->rotation.y*PI/180), 0,1,0);
(GP) 1142, : gluLookAt(
    fromX,fromY,fromZ,
    fromX+cos (torsoRot),fromY,fromZ-sin (torsoRot), 0,1,0);

```

**Figure 335: Line History for the implementation of the First-Person View**

The student adds the first `gluLookAt` command to implement viewing at (608-614) and begins work on a first-person view. The first implementation looks from the static point (10,10,10) to the avatar's head (the position of which is stored in the `fromX`, `fromY` and `fromZ` variables as shown in Figure 336). However, a bug in the student's selection of views (a missing break statement in a switch loop) makes it impossible to switch to the first person view. The student modifies the first person's `gluLookAt` statement several times at (653), (654) and (688), presumably confused by the fact that these changes have no effect (due to the view's case statement falling through). The remainder of the Changes between (614) and (688) are spent implementing the top-down view and modifying the scene graph implementation (which may in fact be an attempt to fix the problem with the first-person view).

```

case FP_CAM:
    float x, y, z;
    fromX=scene->getNodeByName("man")->translate.x
        + scene->getNodeByName("head")->translate.x;
    fromY=scene->getNodeByName("man")->translate.y
        + scene->getNodeByName("head")->translate.y;
    fromZ=scene->getNodeByName("man")->translate.z
        + scene->getNodeByName("head")->translate.z;
    gluLookAt(, at, at , fromX, fromY, fromZ, 0, 1, 0);
    break;

```

**Figure 336: Implementation of 'from' variables to store the avatar's (x,y,z) head position at (612)**

While modifying the `gluLookAt` statement in this period, the student modifies it to look from the avatar's head position to the avatar's head position plus an offset at (654) and from the offset head position to the head position at (688). This suggests a naïve approach of using the avatar's position as the eye position and then adding a fixed value to 'look ahead of' the avatar, an approach which will not allow for the first-person view to be rotated to 'look around'. Since the view does not work before the fix of the switch statement at (692), most of the experimentation with fixed eye and `lookAt` values takes place in the implementation of the top-down view described later.

After fixing the switch statement at (692) by adding a `break` statement to the first-person View's case block, the View becomes functional but since the eye is positioned inside the avatar's head only the inside of the avatar is visible (see Figure 337). Moving the offset from the `lookAt` to the eye coordinate in (712) does not provide enough offset to move the view out of the chest area, leaving the View apparently unchanged.



**Figure 337: The first-person View showing only the inside of the avatar's chest at (692, 712)**

The student is apparently unable to make the conceptual leap to producing a rotatable View since he suspends work on the first-person View until (975). It is likely that he has used the intervening time to research solution strategies, as at (975) he moves directly to the correct approach of using polar coordinates to produce a `lookAt` point 'in front of' the avatar's head and in the avatar's view direction. Despite having discovered the correct approach the student requires modifications at (977, 979, 981, 982, 985, and 986) to correctly implement the correct trigonometric equations for

the View's spherical coordinates. Initially the student multiplies the head's position by the polar coordinate. He then removes the head's position altogether at (981) before correctly adding the polar coordinate to the avatar's head position from (982). However, even at this point the student still requires until (986) to correct the signs for the sine and cosine functions used.

### Third-Person View

The student's implementation of the third-person View is captured in Segment [CHRISTOPHER\\_A3.VIEW.1."FP Camera" \[47\]](#) and [CHRISTOPHER\\_A3.VIEW.2."TP Camera" \[15\]](#). The associated Line History is shown in Figure 338. As discussed in the implementation of the first-person View, a bug in the View-changing switch statement had rendered the first-person View inoperable. As a result, experimentation with a naïve fixed View occurs during the implementation of the third-person View.

```
(1699): (616, 653, 655, 688-691, 713, 741-747, 987-993, 1142 : total = 23)
616, : gluLookAt(at,at,at , fromX,fromY,fromZ, 0,1,0);
653, : gluLookAt(fromX,fromY,fromZ, at,at,at , 0,1,0);
655, : gluLookAt(fromX,fromY,fromZ, fromX+2,fromY+2,fromZ+2, 0,1,0);
688, : gluLookAt(fromX+2,fromY+2,fromZ+2, fromX,fromY,fromZ, 0,1,0);
689, : gluLookAt(fromX+2,fromY,fromZ+2, fromX,fromY,fromZ, 0,1,0);
690, : gluLookAt(fromX,fromY,fromZ+2, fromX,fromY,fromZ, 0,1,0);
691, : gluLookAt(fromX-2,fromY,fromZ-2, fromX,fromY,fromZ, 0,1,0);
713, : gluLookAt( fromX,fromY,fromZ, fromX-2,fromY,fromZ-2, 0,1,0);
741, : gluLookAt( fromX,fromY,fromZ, fromX-2,fromY-2,fromZ, 0,1,0);
742, : gluLookAt( fromX,fromY,fromZ, fromX-5,fromY-5,fromZ, 0,1,0);
743, : gluLookAt( fromX,fromY,fromZ, fromX-15,fromY-5,fromZ, 0,1,0);
744, : gluLookAt( fromX,fromY,fromZ, fromX-15,fromY+5,fromZ, 0,1,0);
745, : gluLookAt( fromX,fromY,fromZ, fromX-1,fromY+1,fromZ, 0,1,0);
746, : gluLookAt( fromX,fromY,fromZ, fromX-4,fromY+1,fromZ, 0,1,0);
747, : gluLookAt( fromX-4,fromY+1,fromZ, fromX,fromY,fromZ, 0,1,0);
987, : gluLookAt(
    fromX+fromX*cos (scene->getNodeByName("Torso")->rotation.y*PI/180)-4,fromY+1,fromZ-
    sin (scene->getNodeByName("Torso")->rotation.y*PI/180),
    fromX,fromY,fromZ, 0,1,0);
988, : gluLookAt(
```



```

        fromX+cos      (scene->getNodeByName("Torso")->rotation.y*PI/180)-4,fromY+1,fromZ-sin
(scene->getNodeByName("Torso")->rotation.y*PI/180),
        fromX,fromY,fromZ,  0,1,0);
989, : gluLookAt(
        fromX+cos      (scene->getNodeByName("Torso")->rotation.y*PI/180)-4,fromY+1,fromZ+sin
(scene->getNodeByName("Torso")->rotation.y*PI/180),
        fromX,fromY,fromZ,  0,1,0);
990, : gluLookAt(
        fromX+cos      (scene->getNodeByName("Torso")->rotation.y*PI/180),fromY+1,fromZ+sin
(scene->getNodeByName("Torso")->rotation.y*PI/180),
        fromX,fromY,fromZ,  0,1,0);
991, : gluLookAt(
        fromX-cos      (scene->getNodeByName("Torso")->rotation.y*PI/180),fromY+1,fromZ+sin
(scene->getNodeByName("Torso")->rotation.y*PI/180),
        fromX,fromY,fromZ,  0,1,0);
992, : gluLookAt(
        fromX-5*cos     (scene->getNodeByName("Torso")->rotation.y*PI/180),fromY+1,fromZ+sin
(scene->getNodeByName("Torso")->rotation.y*PI/180),
        fromX,fromY,fromZ,  0,1,0);
993, : gluLookAt(
        fromX-5*cos     (scene->getNodeByName("Torso")->rotation.y*PI/180),fromY+1,fromZ+5*sin
(scene->getNodeByName("Torso")->rotation.y*PI/180),
        fromX,fromY,fromZ,  0,1,0);

```

**Figure 338: Line History implementing the Third-Person View**

The student begins work on the third-person view at (653). The from variables (fromX, fromY,fromZ) hold the avatar's head position as was the case with the first-person View. The student first modifies the view at (655) to look from the avatar's head position to the avatar's head position plus an offset. After correcting an error in his scene-graph implementation from (656-687) the student spends from (713-747) attempting to implement a working top-down view using the head position and offsets of which five are major changes involving the addition or removal of offsets to axes and three are tweaks. The final view achieved at (747) is well-positioned, but since it uses static offsets it does not rotate with the avatar as it turns.

The student suspends work on the third-person View until (987-993) where he applies the spherical-coordinate approach first used with the first-person View to the third-person View to achieve View

rotation as the avatar is rotated. The student requires six major modifications and one tweak, changing the sign of the sine and cosine functions used as well as the way in which the camera is offset from the body. The student initially attempts to add polar coordinates to static offsets before discovering the correct solution of multiplying the offset by the polar coordinate to achieve a functional third-person view at (992).

However, the student does not implement the zooming 'orbiting' camera as required by the assignment specification.

### **Minimap View**

The minimap view is implemented at Changes (619,621, 686-687), during which the student initially incorrectly adds a 2D orthogonal projection at (619) but corrects this at (621) and then finally corrects the minimap's look-at from (686-687) to show the room from a top-down rather than a side view as shown in Figure 339. Considering the speed of the implementation, the student either very quickly understood the problem and solved it or discovered a similar solution in the study material and applied it.

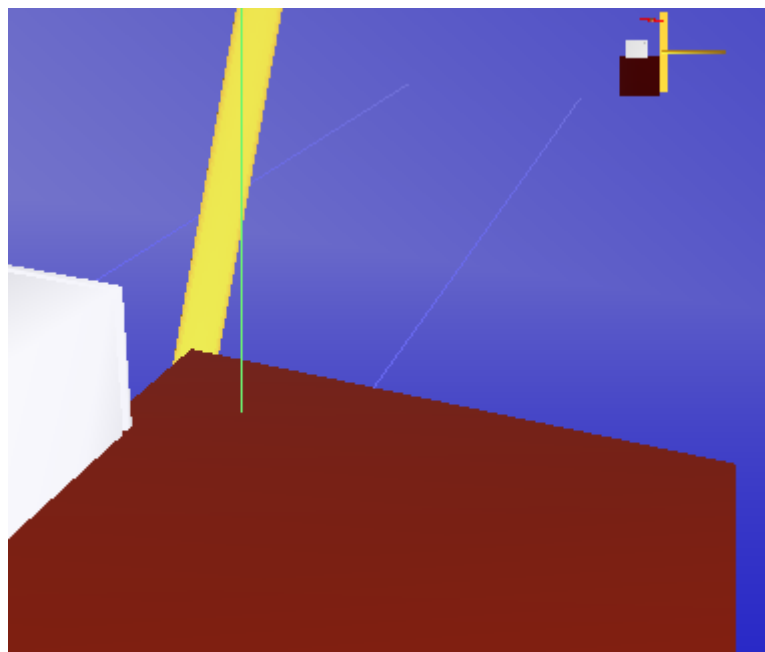


Figure 339: The application's output at (688) with the minimap shown in the top-right corner

### ***9.7.7.1.2 Qualitative Analysis of Classification Category Segments***

#### **9.7.7.1.2.1 Two-Dimensional Spatial Tasks**

In most 2D spatial programming problems, even a trial-and-error approach would quickly yield correct results and the spatial reasoning required is exceedingly simple given that there are only two 'choices' when moving a coordinate.

Most of the 2D errors occur in the first assignment, as the first assignment included many 2D problems in the assignment specification, including the creation of user interface icons, the implementation of hit code functionality, the implementation of user interface clipping and the implementation of object movement and rotation about an object's parent

All students made a number of 2D spatial errors while solving spatial problems, but most of the associated problem segments are on the small end of the scale. As the list of problems is very long, the reader is referred to Section 6.3.4 for details. Because of the large number of problem segments related to two-dimensional spatial programming, only a subset will be presented in detail to illustrate the general nature of student errors during two-dimensional spatial programming. Also, two-dimensional spatial problems relating to the more complex interaction or manipulation of two-dimensional coordinates which occurred during implementation of parent-child rotation is described in detail in a separate section (Section 9.7.7.1.1.1).

While there are a large number of segments involving two-dimensional spatial problems, a significant amount of two-dimensional spatial programming (by John, for example) is carried out very rapidly and without error; such work is often accompanied by large time gaps between Changes often lasting several minutes indicating that the student is probably working out the problem on paper beforehand, followed by the implementation of an icon in one or two Changes. It seems likely that other students may also have used pen-and-paper as a tool to plan and support their 2D spatial programming (and bypassing their visio-spatial sketch pad) but measuring the prevalence of such behaviour is not possible in retrospect, though future research could be aimed at investigating the role of pen-and-paper sketches in both 2D and 3D spatial problem solving.

### **General 2D Spatial Problems**

An example is [CHRISTOPHER\\_A1.SP.3."Door Icon" \[20\]](#) in which the student is attempting to implement a door icon which consists of nine points as shown in Figure 340. Most of these points are parallel to other points, meaning the problem is spatially simple. Still, it takes the student 20 changes, 14 of which are major changes, to implement this rather simple shape (several steps are shown in Figure 341).

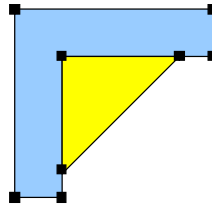


Figure 340: The door icon shape



Figure 341: Three steps of the implementation of the door shape from 342-347

The student later makes further adjustments to the door in order to use it as a furniture object. In doing so, he makes several erroneous changes to determine the correct y-coordinates for the inside triangle despite the fact that the correct y-coordinates are already in use for the outer yellow L-shape.

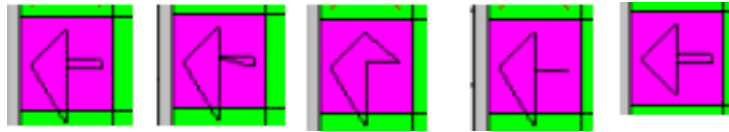
```
glBegin(GL_LINE_LOOP);
glVertex2i(x+width/2,y+height-5);
glVertex2i(x+5,y+height/2);
glVertex2i(x+width/2,y-5);
glVertex2i(x+width/2,y+height/2+2);
glVertex2i(x+width-5,y+height/2+2);
glVertex2i(x+width-5,y+height/2-2);
glVertex2i(x+width/2,y+height/2-2);
glEnd();

glBegin(GL_LINE_LOOP);
glVertex2i(x+width/2,y+height+5);
glVertex2i(x+5,y+height/2);
glVertex2i(x+width/2,y-5);
glVertex2i(x+width/2,y+height/2+2);
glVertex2i(x+width-5,y+height/2+2);
glVertex2i(x+width-5,y+height/2-2);
glVertex2i(x+width/2,y+height/2-2);
glEnd();
```

Figure 342: The initial incorrect code at 1034 and the almost-correct code at 1041 (The error lies in the first line)

Sometimes, students make an error during their development of a two-dimensional spatial shape, which appears to lead to a breakdown in their spatial model, resulting in additional work during which the mental model is corrected. The Segment [THOMAS\\_A1.SP.5."Drawing Arrow" \[20\]](#) is an example of a student's breakdown in spatial understanding. After correctly and quickly creating an arrow drawing in one go (probably through the use of a paper sketch) the student makes a spatial error when mentally "turning" the arrow's coordinates to create a left-pointing arrow from an upward-pointing arrow. The student requires several changes to correct this error, making a total of 4 errors and twice removing parts of the arrow to aid with the debugging process (see Figure 343). This shows that the student could not easily debug the spatial error mentally, despite the underlying error being simple. This shows that while students usually find it easy to build a 2D drawing step by

step, attempting to gain a full understanding of several 2D coordinates from scratch (required to find the error) is a much harder problem.



**Figure 343: Development of an 'arrow' shape at Changes 1034, 1035, 1037, 1039, 1043**

The segment [IDA\\_A1.SP.1."Move Icon" \[21\]](#) is another example of a fairly simple spatial problem presenting undue difficulty; the student is attempting to fashion a triangle to serve as an arrow-head, but the first three changes (395-397) are completely incorrect (see Figure 344) and the student requires many changes and tweaks to finally produce a correct arrowhead at 408.



**Figure 344: Implementation of an arrow at 396, 397, 399, 403, 408, 415. Two completely incorrect changes create no arrow at 396 and 397. Notice start of an arrow at 399)**

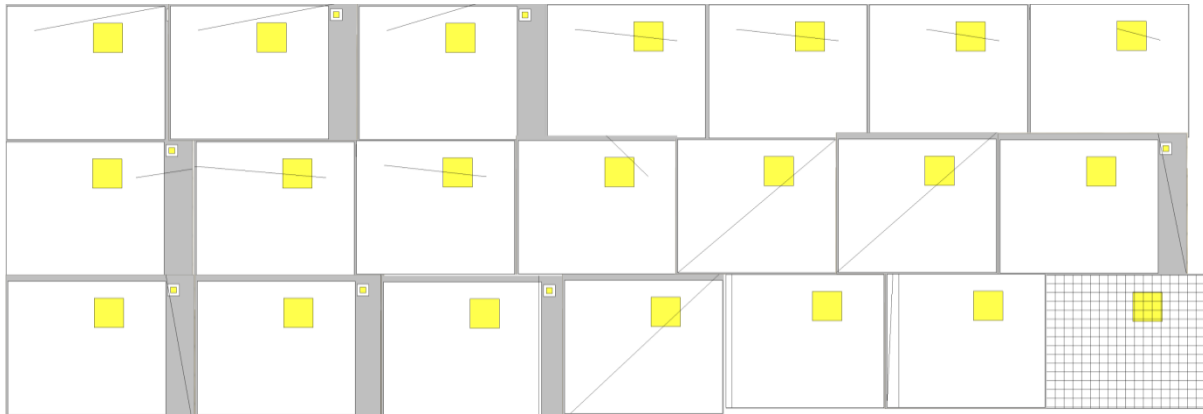
An example from Michael's implementation of the first assignment is captured in segment [MICHAEL\\_A1.SP.15."Delete Icon" \[18\]](#). While developing the delete icon (an x-shaped cross) the student makes a total of seven coordinate errors (see Figure 345) while developing the first half of the shape, a very high number of errors considering the simplicity of the delete icon's shape.



**Figure 345 : Error-prone development of the delete icon (2103-2110, 2115, 2119)**

While most 2D spatial problems were comparatively minor, Segment [MICHAEL\\_A1.SP.1."Grid Drawing" \[36\]](#) (involving 36 Changes) showing how even simple spatial problems can present a significant challenge especially during the student's early exposure to spatial programming.

The student is attempting to implement a 2D grid (simple horizontal and vertical gridlines) and makes 15 errors while performing this extremely simple spatial task, as shown in Figure 346.



**Figure 346: Implementation of a simple two-dimensional grid, Changes Student grid 133-150, 164, and 227**

### **Non-visible Output**

One special category of spatial problems that cause problems when they occur are those that produce non-visible output, such as spatial conditional statements (hit detection, for example). Two such problems are described next.

Relatively simple problems such as the implementation of button hit code (testing whether a point lies inside a button's rectangular area) can also cause problems. For example, segment [CHRISTOPHER\\_A1.SP.2."Hit Code" \[17\]](#) shows a student requiring 17 erroneous changes / debug statements (for a total of four sub-conditions) to correctly implement a correct conditional statement for detecting points inside a rectangle (hit code), some of which are shown in Figure 347. In this instance, part of the reason for the extremely large number of changes required for such a simple problem may be the that the hit coordinates are not drawn to screen, thus preventing the student from visually debugging the problem and forcing the student to instead experimentally debug it via attempting mouse presses and to debug it via cout statements.

```
return ( _p.getX()<lower.getX() && _p.getX()>upper.getX() && _p.getY()>lower.getY() && _p.getY()<upper.getY() );
return ( _p.getX()>lower.getX() && _p.getX()<upper.getX() && _p.getY()>lower.getY() && _p.getY()<upper.getY() );
return ( _p.getX()<lower.getX() && _p.getX()>upper.getX() && _p.getY()<lower.getY() && _p.getY()>upper.getY() );
return ( _p.getX()>lower.getX() && _p.getX()<upper.getX() && _p.getY()<lower.getY() && _p.getY()>upper.getY() );
return ( _p.getX()>lower.getX() && _p.getX()<upper.getX() && _p.getY()>lower.getY() && _p.getY()<upper.getY() );
return ( ( _p.getX()>lower.getX() ) && ( _p.getX()<upper.getX() ) && ( _p.getY()>lower.getY() ) && ( _p.getY()<upper.getY() ) );
```

**Figure 347: Changes to the hit condition (266,267,268,269,271, 275)**

A clear example of the difficulty arising from spatial problems that do not give rise to directly visible output is demonstrated in segment [MICHAEL\\_A1.SP.3.2."Button Spatial Coloring" \[26\]](#). In this problem, the student is attempting to set button colour based on a button's coordinates using an if/else-if/... conditional block with one condition per button position (a needlessly complex approach). The student requires 28 changes, making 18 errors along the way, to partially implement the approach

before giving up and simply setting button colour via an input parameter. The problem is difficult to debug since failure of the conditional simply leads the buttons in question to not change colour since the coordinates are not drawn to screen as they would be if this were a 2D drawing problem (like the implementation of an icon).

### **Tweaking**

Students invested a lot of effort in ‘tweaking’ spatial coordinates to precisely place objects. Some tweaking occurred in most segments containing spatial problems, but in some segments tweaking Changes made up a substantial proportion of total Changes. Since tweaking occurred in both two-dimensional and three-dimensional spatial problems (as well as in Viewing problems), it is discussed in a separate section (Section 9.7.7.1.2.2).

### **Discussion**

Two-dimensional spatial problems did not generally present a significant challenge with the exception of more complex tasks such as the parent-child rotation task discussed in Section 9.7.7.1.1.1 in which additional factors affected outcomes.

However it also appears that students sometimes used pencil and paper to plan out icons to implement as seems to have been the case with John, and this habit may have affected the size of some two-dimensional drawing segments.

It appeared that students only encountered difficulty with simple two-dimensional spatial problems in two situations. The first is the situation in which the student introduces an error into a drawing, which appears to lead to a breakdown of the student’s spatial understanding and a large number of corrective Changes. The second involves problems like hit code that produces no visible output and hence appears to be harder to understand and debug.

Given that simple two-dimensional drawing problems do not generally cause any significant problems for students, there is probably no need to invest a large amount of effort in supporting student work in this area. As students struggled with two-dimensional spatial problems which did not produce visible output, it may help to support students by teaching them ways to use OpenGL to visualize these types of problems; for example in the context of hit code students could draw out the boundaries of the bounding box geometry, after which they could visually identify any errors in the hit code quickly.

#### **9.7.7.1.2.2 Spatial Tweaking**

Spatial tweaking is defined as small changes to coordinates or transformation values that do not include a sign change or axis change. For example, `glTranslatef(0,0,1)` -> `glTranslatef(0,0,2)` would be

considered a tweak, whereas `glTranslatef(0,0,1)` -> `glTranslatef(0,1,0)` would not be. These changes occur for cosmetic effect, to precisely position objects, points or to make an animation look better.

The following segments include a significant amount of spatial tweaking in relation to their total number of Changes (making up at least half of all Changes):

- [MICHAEL\\_A1.SP.4."Line buttons" \[18\]](#)
- [MICHAEL\\_A1.SP.5."Positioning button text" \[26\]](#)
- [MICHAEL\\_A1.SP.7."Positioning status text" \[23\]](#)
- [MICHAEL\\_A1.SP.16."Onto Icon" \[14\]](#)
- [JOHN\\_A3.SP.3."Furniture positioning, tweaking" \[12\]](#)
- [IDA\\_A3.SP.1."Viewport" \[17\]](#)
- [IDA\\_A3.SP.2."GUI Background" \[16\]](#)
- [IDA\\_A3.SP.4.2.2."Teapot Animation" \[14\]](#)

These segments include some periods of tweaking:

- [JOHN\\_A3.SP.4.1."Initial naive assembly" \[26\]](#)
- [JOHN\\_A3.SP.2."Pickup Teapot anim " \[14\]](#)
- [JOHN\\_A3.SP.5.3."Swim Anim" \[120\]](#)
- [MICHAEL\\_A3.SP.1."Initial Assembly" \[47\]](#)
- [THOMAS\\_A3.SP.4."Carrying Lamp" \[36\]](#)
- [THOMAS\\_A3.SP.5.2."Proper Assembly" \[55\]](#)
- [THOMAS\\_A3.SP.6.1."Walk Animation" \[36\]](#)
- [THOMAS\\_A3.SP.6.2."Pickup Animation" \[21\]](#)
- [IDA\\_A3.SP.4.1."The Walk Animation" \[66\]](#)
- [IDA\\_A3.SP.4.2.1."Pickup Animation" \[79\]](#)

As the above list shows, tweaking occurs both in two-dimensional and three-dimensional problems.

The tweaking changes are interesting not because they reveal a breakdown in spatial understanding but rather because they present a potential waste of assignment time, since students are probably not developing spatial or other Computer Graphics-related skills while making these small tweaking changes, instead spending most of that time waiting for the program to compile and execute so they can examine whether the object is precisely placed.



The following examples will serve to illustrate the type of work done by students during periods of 'tweaking'.

### Two-Dimensional Tweaking

In the segment [MICHAEL\\_A1.SP.7."Positioning status text" \[23\]](#) the student is attempting to correctly position text into a status box (see Figure 348) for which he requires 23 changes, 21 of which are tweaks (see Figure 349) in order to position the text pixel-perfect for each of the 13 different strings associated with UI actions.



**Figure 348: Positioning of text in the status indicator**

```
(2634): (1046, 1048-1050, 1053-1057 : total = 9)
1046, ADDED(O): resizetxt->write(resize,
    Point2(application_width*0.86+45+2,application_height*0.85-150+10-12-60-60));
1048, MUTATED(O): resizetxt->write(resize,
    Point2(application_width*0.86,application_height*0.01));
1049, MUTATED(O): resizetxt->write(resize,
    Point2(application_width*0.86+10,application_height*0.01+5));
1050, MUTATED(O): s1txt->write(s1,
    Point2(application_width*0.86+10,application_height*0.01+5));
1053, MUTATED(O): s1txt->write(s1,
    Point2(application_width*0.86+10,application_height*0.01+10));
1054, MUTATED(O): s1txt->write(s1,
    Point2(application_width*0.86+10,application_height*0.01+7));
1055, MUTATED(O): s1txt->write(s1,
    Point2(application_width*0.86+10,application_height*0.01+8));
1056, MUTATED(O): s1txt->write(s1,
    Point2(application_width*0.86+12,application_height*0.01+8));
1057, MUTATED(O): s1txt->write(s1,
    Point2(application_width*0.86+11,application_height*0.01+8));
```

**Figure 349: Tweaking the text for one of the actions (1243, 1248-1250)**

### Three-Dimensional Tweaking

Some segments (usually two-dimensional spatial segments) are comprised almost entirely of tweaks. The segment [IDA\\_A3.SP.1."Viewport" \[17\]](#) involves a student's implementation of a viewport over 17 Changes (see Figure 350 for the Line History encompassing these Changes). It involves small modification to the viewport's position. The Line History shows the student returning to the same x-value of 0.84 three times during the implementation.

```
11 : glViewport(10, 10, 670, 570);
401: glViewport(10, 10, 670, 570);
422: glViewport(10, 10, 670, 565);
423: glViewport(8, 8, 665, 565);
424: glViewport(8, 7, 667, 565);
425: glViewport(8, 6, 669, 565);
426: glViewport(8, 6, 669, 567);
428: glViewport(int(w*0.01), int(w*0.01), int(w*0.85), int(h*0.955));
429: glViewport(int(w*0.01), int(w*0.01), int(w*0.83), int(h*0.855));
430: glViewport(int(w*0.01), int(w*0.01), int(w*0.85), int(h*0.9));
431: glViewport(int(w*0.01), int(h*0.01), int(w*0.87), int(h*0.93));
432: glViewport(int(w*0.01), int(h*0.01), int(w*0.85), int(h*0.95));
433: glViewport(int(w*0.01), int(h*0.01), int(w*0.84), int(h*0.94));
434: glViewport(int(w*0.01), int(h*0.01), int(w*0.835), int(h*0.945));
435: glViewport(int(w*0.01), int(h*0.01), int(w*0.835), int(h*0.9455));
436: glViewport(int(w*0.01), int(h*0.01), int(w*0.835), int(h*0.95));
437: glViewport(int(w*0.01), int(h*0.01), int(w*0.84), int(h*0.9455));
438: glViewport(int(w*0.01), int(h*0.01), int(w*0.8355), int(h*0.9455));
439: glViewport(int(w*0.01), int(h*0.01), int(w*0.84), int(h*0.9455));
```

**Figure 350: Tweaks to the glViewport size for 11, 401, 422-426, 428-439.**

Tweaking also occurs in segments involving three-dimensional spatial programming, though tweaking Changes do not make up the majority (nor in most cases a substantial minority) of Changes in such segments. Segment [JOHN\\_A3.SP.5.3."Swim Anim" \[120\]](#) illustrates how time-consuming such tweaking can be. While implementing a swim animation the student spends a total of 55 of the 120 changes are minor tweaks to coordinates. Figure 351 shows one sequence of such tweak actions from that problem segment.

```
507: guy.partsRot[PERSON_LLA][X] -= INC;
```

```

508: guy.partsRot[PERSON_LLA][X] -= 3*INC/8;
510: guy.partsRot[PERSON_LLA][X] -= 3*INC;
511: guy.partsRot[PERSON_LLA][X] -= 8*INC/3;
512: guy.partsRot[PERSON_LLA][X] -= 3*INC/8;
513: guy.partsRot[PERSON_LLA][X] -= INC;
514: guy.partsRot[PERSON_LLA][X] -= 2*INC;
521: guy.partsRot[PERSON_LLA][X] -= 3*INC;
522: guy.partsRot[PERSON_LLA][Y] -= 3*INC;
523: guy.partsRot[PERSON_LLA][X] -= 3*INC/8;
524: guy.partsRot[PERSON_LLA][X] -= INC;
528: guy.partsRot[PERSON_LLA][X] -= 3*INC/8;
529: guy.partsRot[PERSON_LLA][X] -= 3*INC;
530: guy.partsRot[PERSON_LLA][X] -= ((3*INC)/8);
531: guy.partsRot[PERSON_LLA][X] -= 0.375*INC;
532: guy.partsRot[PERSON_LLA][X] -= 0.8*INC;
533: guy.partsRot[PERSON_LLA][X] -= 2.66*INC;
534: guy.partsRot[PERSON_LLA][X] -= 0.9*INC;
535: guy.partsRot[PERSON_LLA][X] -= 1.4*INC;
537: guy.partsRot[PERSON_LLA][X] -= 9*INC/8;
538: guy.partsRot[PERSON_LLA][X] -= 11*INC/8;
539: guy.partsRot[PERSON_LLA][X] -= 1.355*INC;
540: guy.partsRot[PERSON_LLA][X] -= 1.34*INC;
541: guy.partsRot[PERSON_LLA][X] -= 1.3*INC;
542: guy.partsRot[PERSON_LLA][X] -= 1.28*INC;

```

**Figure 351: Tweaking of the lower arm's x-rotation in the swim animation**

## **DISCUSSION/CONCLUSION**

It seems reasonable to assume that spatial tweaking does not present a significant learning challenge to students. It seems unlikely that time spent on spatial tweaking would improve student spatial ability, as the process involved in pixel-perfectly positioning some item involves an iterative process of comparing visual outcomes basic on a one-dimensional modification of a variable rather than the complex spatial reasoning that students seem to find difficult based on analysis of 3D spatial programming.

Therefore, this type of problem may be extraneous to the learning process since it probably does not help increase student problem-solving ability in core spatial computer graphics concepts.

It may therefore be desirable to provide students with tools and/or programming strategies to avoid the need for extensive spatial tweaking. Such tools/strategies may include providing coordinate-labelled grids, techniques for printing out mouse coordinates (to enable students to precisely determine coordinates on-screen) or even more advanced tools which allow coordinates to be changed in a more interactive manner without the necessity of time-consuming compile actions.

#### **9.7.7.1.2.3 Mathematical**

Tasks implementing the drawing of circles (A1) parent-child rotation (A1) and movement in avatar facing direction (spherical coordinates, A3) involved the use of trigonometry. Ida also utilised trigonometry in the implementation of one animation. Two students attempted to calculate the distance between the avatar and a scene object in the third assignment to determine whether the object was 'in range' for pickup, though neither succeeded ([CHRISTOPHER\\_A3.SP.3.5."Pickup / Drop Object" \[47\]](#), [THOMAS\\_A3.SP.4."Carrying Lamp" \[36\]](#)). All the mathematics tasks in Assignment 1 and Assignment 3 relied on mathematics that students would have learned in the first year of their Computing degrees.

The following segments involved the use of mathematics:

- [JOHN\\_A1.SP.7."Rotation around Parent" \[56 \]](#)
- [JOHN\\_A3.SP.5.3."Swim Anim" \[120\]](#)
- [CHRISTOPHER\\_A3.SP.3.3."Walk In ViewDir" \[25\]](#)
- [CHRISTOPHER\\_A3.SP.3.5."Pickup / Drop Object" \[47\]](#)
- [MICHAEL\\_A1.SP.3.2."Button Spatial Coloring" \[26\]](#)
- [MICHAEL\\_A1.SP.9."Circle Draw" \[26\]](#)
- [MICHAEL\\_A1.SP.10."Circle resize" \[18\]](#)
- [THOMAS\\_A1.SP.1."Button Spatial Modulus" \[20\]](#)
- [THOMAS\\_A1.SP.4.1."Object rotate maths" \[19\]](#)
- [THOMAS\\_A1.SP.4.2."Object rotate angle Math Formula" \[10\]](#)
- [THOMAS\\_A3.SP.2.1."Angle Conversion" \[12\]](#)
- [THOMAS\\_A3.SP.2.2."Avatar Movement" \[36\]](#)
- [THOMAS\\_A3.SP.4."Carrying Lamp" \[36\]](#)
- [IDA\\_A1.GP.7."Child object rotate"\[56\]](#)
- [IDA\\_A1.SP.3."Object rotation" \[23\]](#)

## Spatial Modulus

The problem segment (THOMAS\_A1.SP.1."Button Spatial Modulus" [20]) involves a student attempting to implement a mechanism for automatically creating and positioning buttons based on arithmetic involving the modulus operator. This was not the suggested implementation approach for the implementation of buttons, so no other students have implementations to compare against.

As can be seen in Figure 352, the student makes many errors (9 in total) while implementing the mathematical formulas to correctly position the buttons (See Figure 353 for selected code extracts including the formulae), showing the student's difficulty in developing a spatial understanding of the result of the formulae used.



Figure 352: Positioning of buttons based on modulus mathematics at 137, 138, 139, 141, 142, 144, 146, and 158

```
button(int m){
    type=m;
    width=32;
    height=width;
    x=type*width;
    y=0;
}

button(int m){
    type=m;
    width=32;
    height=32;
    x=400+type%2*32;
    y=type*width;
}

button(int m){
    type=m;
    width=32;
    height=32;
    x=400+type%2*32;
    y=type*height-type%2*32;
    y/=2;
}

button(int m){
    type=m;
    width=50;
    height=width;
    x=550+type%2*width;
    y=200+type*height-type%2*width;
    y/=2;
}
```

Figure 353 : Code for spatial buttons at 137, 139, 146 and correct code at 158

## Circle Rotation

When implementing parent-child rotation in Assignment 1 (IDA\_A1.GP.7."Child object rotate"[56] and IDA\_A1.SP.3."Object rotation" [23]), as discussed in the section relating to the parent-child rotation task (Section 9.7.7.1.1.1.1), Ida used an incorrect rotation method, independently rotating an object's upper and lower points as shown in Figure 354.

```

Point c = epicenter;
double angle = atan((p.y-epicenter.y)/(p.x-epicenter.x))
               - atan((pivot.y-epicenter.y)/(pivot.x-epicenter.x));
int t = int((angle+0.785398163)*2/3.141592654);
lower.x -= c.x;
lower.y -= c.y;
upper.x -= c.x;
upper.y -= c.y;
lower.x = int(lower.x*cos(t*2/3.141592654)
               - lower.y*sin(t*2/3.141592654));
lower.y = int(lower.y*cos(t*2/3.141592654)
               + lower.x*sin(t*2/3.141592654));
upper.x = int(upper.x*cos(t*2/3.141592654)
               - upper.y*sin(t*2/3.141592654));
upper.y = int(upper.y*cos(t*2/3.141592654)
               + upper.x*sin(t*2/3.141592654));
lower.x += c.x;
lower.y += c.y;
upper.x += c.x;
upper.y += c.y;

```

**Figure 354: Ida's approach to rotating child objects about their parent at (1146)**

This incorrect approach hindered the student from developing a working parent-child algorithm, forcing her to instead implement a work-around to hide the problems caused by the incorrect mathematical algorithm. Despite working on parent-child rotation over a long period of time and trying many different solution approaches, the student appears to never have realised this basic flaw, which would have become apparent if she had drawn out the circles about which the two points rotated.

John and Thomas (the other students who implemented parent-child rotation) used correct rotation methods based on one single point on the child object.

### **Calculating Angles**

All students who worked on the rotation-around parent problem had problems with calculating the angles between two objects.

In Ida's implementation of parent-child rotation, the student utilised the trigonometric atan function to calculate the angle between two objects, as shown in Figure 355. This approach is almost correct, but unfortunately the atan function can measure only angles between 0-180 degrees; to be able to detect angles between 0-360 degrees, the student should have switched to the atan2 function.

(848): (1141, 1204-1206, 1212 : total = 5)

```

1141, ADDED(X): double angle = atan((p.y-epicenter.y)/(p.x-epicenter.x)) - atan((pivot.y-epicenter.y)/(pivot.x-epicenter.x));
1204, MUTATED(X): double angle = abs(atan((p.y-epicenter.y)/(p.x-epicenter.x)) - atan((pivot.y-epicenter.y)/(pivot.x-epicenter.x)));
1205, MUTATED(X): double angle = atan((p.y-epicenter.y)/(p.x-epicenter.x)) - atan((pivot.y-epicenter.y)/(pivot.x-epicenter.x));
1206, MUTATED(O): double angle = atan((p.y-epicenter.y)/(p.x-epicenter.x)) - atan((pivot.y-epicenter.y)/(pivot.x-epicenter.x));
1212, MUTATED(X): //double angle = atan((p.y-epicenter.y)/(p.x-epicenter.x)) - atan((pivot.y-epicenter.y)/(pivot.x-epicenter.x));

```

**Figure 355: History Line shownign Ida's implementation of calculation of the angle between parent and child**

As part of John's implementation of parent-child rotation ([JOHN\\_A1.SP.7."Rotation around Parent" \[56\]](#)) John initially incorrectly attempts to calculate the angle between parent and child objects with the asin function as shown in Figure 356. After debugging for 7 Changes, the student moves to using the atan function at (664), and then to using the correct atan2 function at (665). Like other students, the student initially utilised incorrect trigonometric functions to calculate the angle between objects, but after 9 Changes discovers the correct approach.

(833):	(658, 664-667, 673, 693, 699, 710-714, 777 : total = 14)
658, ADDED(O):	float angle = asin(float(newy-centery)/float(newx-centerx));
664, MUTATED(O):	float angle = atan(float(newy-centery)/float(newx-centerx));
665, MUTATED(X):	float angle = atan2(float(newy-centery)/float(newx-centerx));
666, MUTATED(X):	float angle = atan2(double(newy-centery)/double(newx-centerx));
667, MUTATED(O):	float angle = atan(float(newy-centery)/float(newx-centerx));
673, MUTATED(O):	float angle = atan(float(newy-py)/float(newx-px));
693, MUTATED(X):	float newAngle = atan(float(newy-py)/float(newx-px));
699, MUTATED(O):	float offsetAngle = atan(float(newy-py)/float(newx-px));
710, MUTATED(X):	float offsetAngle = atan2(float(newy-py)/float(newx-px));
711, MUTATED(X):	float offsetAngle = atan2(double(newy-py)/double(newx-px));
712, MUTATED(X):	float offsetAngle = std::atan2(double(newy-py)/double(newx-px));
713, MUTATED(X):	float offsetAngle = std::atan2(double(newy-py), double(newx-px));
714, MUTATED(O):	float offsetAngle = atan2(double(newy-py), double(newx-px));
777, MUTATED(X):	float offsetAngle = std::atan2(double(newy-py), double(newx-px));

**Figure 356: John's implementation of parent-child angle calculation**

Unfortunately the student appears to be uncertain about the correctness of the approach. As the Line History shows, the student makes several (11 Changes) modifications to the correct approach as part of attempting to debug other unrelated problems which in fact introduces an additional error rather than contribution to a solution to the underlying problem. If the student had had a way of verifying the correctness of the approach, this additional superfluous work may have been omitted.

In Thomas's implementation of angle calculation (part of parent-child rotation in segment THOMAS\_A1.SP.4.1."Object rotate maths" [19] and THOMAS\_A1.SP.4.2."Object rotate angle Math Formula" [10]) between objects is shown in Figure 357. As with John, the student initially incorrectly uses the asin function. However, unlike John the student requires many Changes to develop a correct solution based on atan2 in Changes (870-872), thus using up time that could have been used on other tasks. In addition, the problem contributes to the overall difficulty the student encounters in implementing parent-child rotation as discussed in Section 9.7.7.1.1.1.3. As can be seen in (852, 854-856) the problem is compounded by the student unsure of what unit the angle is stored in and what unit to use with the C++ trigonometric functions.

```
(1679): (835-837, 844-845, 848, 852, 854-856, 861, 868 : total = 12)
835, ADDED(X): f.angle=asin()
836, MUTATED(X): f.angle=asin((f.y-f.centrey)/f.radius)
837, MUTATED(O): f.angle=asin((f.y-f.centrey)/f.radius);
844, DELETED(O): DELETED (f.angle=asin((f.y-f.centrey)/f.radius);)
845, GHOST(O): GHOST (f.angle=asin((f.y-f.centrey)/f.radius);)
848, MUTATED(O): f.angle=asin((f.y-f.centrey)/f.radius)*(180/3.1415);
852, MUTATED(O): f.angle=asin(((double(f.y)-double(f.centrey))/double(f.radius))*(180/3.1415);
854, MUTATED(O): f.angle=asin(((double(f.y)-double(f.centrey))/double(f.radius)));
855, MUTATED(O): f.angle=asin(((double(f.y)-double(f.centrey))/double(f.radius))*(180/3.1415);
856, MUTATED(O): f.angle=asin(((double(f.y)-double(f.centrey))/double(f.radius)));
861, MUTATED(X): f->angle=asin(((double(f->y)-double(f->centrey))/double(f->radius)));
868, DELETED(O): DELETED (f->angle=asin(((double(f->y)-double(f->centrey))/double(f->radius)));)
870, ADDED(O): angle=atan((x-centrex)/(y-centrey));
872, MUTATED(O): angle=atan2((y-centrey),(x-centrex));
```

**Figure 357: Thomas's implementation of parent-child angle calculation**

### **Midpoint Formula**

During implementation of parent-child rotation, John utilises an incorrect formula for calculating the midpoint between two points and requires several Changes to correct the formula.



### Distance Formula

Two students attempt to implement pick-up functionality based on the proximity of the object to be picked up in Assignment 3. Calculation of the distance of avatar to object should be calculated using the distance formula  $c^2 = a^2 + b^2$  which should be familiar to all students in third-year Computing. However, neither student manages to correctly calculate the distance to object based on this function.

Christopher's work on determining whether an object is in range for pick-up is captured in segment [CHRISTOPHER\\_A3.SP.3.5."Pickup / Drop Object" \[47\]](#). Figure 358 shows the student's final code for determining distance. Instead of using the distance formula, the student separately calculates x and z distances by simply subtracting the man's z/x coordinate from the object's z/x coordinate.

```
if (cur!=man) {
    cout<<abs((man->translate.x) - (cur->translate.x))<<" "
        <<abs((man->translate.z) - (cur->translate.z))<<endl;
    if (abs((man->translate.x) - (cur->translate.x)) < 15) {
        if (abs((man->translate.z) - (cur->translate.z)) < 15) {
            scene->childNodes.erase(scene->childNodes.begin()+i);
            cur->setTranslate(
                cos(scene->getNodeByName("Torso")->rotation.y*PI/180)*20,
                scene->getNodeByName("Torso")->translate.y,
                -sin(man->rotation.y*PI/180)*20);
            scene->getNodeByName("Torso")->childNodes.push_back(cur);
            animatePickUp(0);
            cur=NULL;
            break;
        }
    }
}
```

**Figure 358: The final (incorrect) solution at (1126) which is part of the student's final submission**

The student's development of the final incorrect approach is shown in the Line Histories shown in Figure 359 and Figure 360. The initial approach of simply comparing whether the object's x and z positions are smaller than the avatar's x and z positions (at 998-999) is even less correct than the final solution. When the student realises the new formula of deducting the avatar's position from the object's position (1015) is still not working, he misidentifies the reason as having to do with the scaling of objects, adding a scaling factor at (1021, 1022-1023, 1026, 1029) before removing it again at (1031).

(1977): (998-999, 1003-1004, 1014-1015, 1020-1023, 1026, 1029, 1031, 1055 : total = 14)

998, ADDED(X): if (man->tranlate.x+1<scene->childNodes[i].translate);

```

999, MUTATED(X): if (man->translate.x+1>scene->childNodes[i].translate.x && man-
>translate.x+1<scene->childNodes[i].translate.x ){
1003, MUTATEDa(X): if (man->translate.x+1>cur->translate.x && man->translate.x+1<cur-
>translate.x ){
1004, MUTATED(X): if (man->translate.x+1 > cur->translate.x && man->translate.x+1<cur-
>translate.x ){
1014, MOVED(O): MOVED (if (man->translate.x+1 > cur->translate.x && man->translate.x+1<cur-
>translate.x ){
1015, MUTATED(O): if (abs(man->translate.x - cur->translate.x) < 1) {
1020, MUTATED(O): if (abs(man->translate.x - cur->translate.x) < 5) {
1021, MUTATED(X): if (abs((man->translate.x/mam->scale.x) - (cur->translate.x/cur->scale.x) < 5) {
1022, MUTATED(X): if (abs((man->translate.x/mam->scale.x) - (cur->translate.x/cur->scale.x) < 1) {
1023, MUTATED(X): if (abs((man->translate.x/mam->scale.x) - (cur->translate.x/cur->scale.x)) < 1) {
1026, MUTATED(X): if (abs((man->translate.x/man->scale.x) - (cur->translate.x/cur->scale.x)) < 1) {
1029, MUTATED(O): if (abs((man->translate.x/man->scale.x) - (cur->translate.x/cur->scale.x)) < 10) {
1031, MUTATED(X): if (abs((man->translate.x) - (cur->translate.x)) < 10) {
1055, MUTATED(O): if (abs((man->translate.x) - (cur->translate.x)) < 15) {

```

**Figure 359: History Line for the development of the conditional for testing the x-distance from avatar to object (a separate conditional exists for testing of z-distance)**

```

(1995): (1017-1019, 1024-1028, 1030 : total = 9)
1017, ADDED(X): cout<<abs(man->translate.x - cur->translate.x)<<" "<<abs(man->translate.z - cur-
>translate.z)<<endl
1018, MUTATED(X): cout<<abs(man->translate.x - cur->translate.x)<<" "<<abs(man->translate.z - cur-
>translate.z)<<endl;
1019, MUTATED(O): cout<<abs(man->translate.x - cur->translate.x)<<" "<<abs(man->translate.z -
cur->translate.z)<<endl;
1024, MUTATED(X): cout<<abs((man->translate.x/mam->scale.x) - (cur->translate.x/cur->scale.x))<<"
"<<abs((man->translate.z/mam->scale.z) - (cur->translate.z/cur->scale.z));
1025, MUTATED(X): cout<<abs((man->translate.x/mam->scale.x) - (cur->translate.x/cur->scale.x))<<"
"<<abs((man->translate.z/mam->scale.z) - (cur->translate.z/cur->scale.z))<<endl;
1026, MUTATED(X): cout<<abs((man->translate.x/man->scale.x) - (cur->translate.x/cur->scale.x))<<"
"<<abs((man->translate.z/mam->scale.z) - (cur->translate.z/cur->scale.z))<<endl;
1027, MUTATED(X): cout<<abs((man->translate.x/man->scale.x) - (cur->translate.x/cur->scale.x))<<"
"<<abs((man->translate.z/mas->scale.z) - (cur->translate.z/cur->scale.z))<<endl;

```

```

1028, MUTATED(O): cout<<abs((man->translate.x/man->scale.x) - (cur->translate.x/cur->scale.x))<<"
"<<abs((man->translate.z/man->scale.z) - (cur->translate.z/cur->scale.z))<<endl;
1030, MUTATED(O): cout<<abs((man->translate.x) - (cur->translate.x))<<" "<<abs((man->translate.z)
- (cur->translate.z))<<endl;

```

**Figure 360: Line History showing the debugging using cout statements of the conditional statement testing distance between avatar and pickup object**

Thomas's attempt at testing the distance to object before pickup is captured in segment THOMAS\_A3.SP.4."Carrying Lamp" [36]. As can be seen in the Line History shown in Figure 361, the first implementation of the distance formula at (790) is  $\text{dist}^2 = (\text{xo}^2 - \text{xa}^2) + (\text{yo}^2 - \text{ya}^2)$  rather than the correct  $\text{dist}^2 = (\text{xo} - \text{xa})^2 + (\text{yo} - \text{ya})^2$ . As the student realises the conditional does not provide the desired effect he modifies it by deducting values from the lamp position at 809, which introduces another error into the formula. Interestingly, the student correctly applied the distance formula in Assignment 1 at (832) in measuring the distance from child to parent object.

```

(2303): (788-790, 798-799, 808-809, 811, 818 : total = 9)
788, ADDED(O): if(true){//trig
789, MUTATED(O): if(true){//check if lamp is in picking up range
790,          MUTATED(X):          if((lampX*lampX-currentPosX*currentPosX)+(lampY*lampY-
currentPosY*currentPosY)<2)
798,          MUTATED(X):          if((lampX*lampX-currentPosX*currentPosX)+(lampY*lampY-
currentPosY*currentPosY)<2){
799,          MUTATED(X):          if((lampX*lampX-currentPosX*currentPosX)+(lampY*lampY-
currentPosY*currentPosY)<2)
808,          MUTATED(O):          if(sqrt((lampX*lampX-currentPosX*currentPosX)+(lampY*lampY-
currentPosY*currentPosY))<2)
809,  MUTATED(X):  if(sqrt(((lampX-4)*(lampX-4)-currentPosX*currentPosX)+((lampY-2.5)*(lampY-
2.5)-currentPosY*currentPosY))<2)
811, MUTATED(X): if(sqrt(((lampX-4)*(lampX-4)-(currentPosX)*(currentPosX))+((lampY-2.5)*(lampY-
2.5)-(currentPosY)*(currentPosY)))<2)
818, DELETED(O): DELETED  (if(sqrt(((lampX-4)*(lampX-4)-(currentPosX)*(currentPosX))+((lampY-
2.5)*(lampY-2.5)-(currentPosY)*(currentPosY)))<2))

```

**Figure 361: Line History showing Thomas's implementation of the conditional for testing the distance between the avatar and the pickup object**

The reason why the student failed to realise the error with the distance formula may be due to the other conditional implemented by the student to test whether the object is 'in front of' the avatar by calculating the angle from object to avatar, as shown in Line History Figure 362. The initial code is a good first start, though the test of whether the object's angle to the avatar is less than the avatar's direction angle - 25 is incorrect; the student should be testing whether the object's angle to the avatar's direction angle is within a certain bound by using the absolute value. However, the student does not correctly resolve this problem, instead changing the conditional to testing whether it is within the direction angle + 25 at (810), which means that the object will essentially be selected only if it is not in front of the avatar. The student continues to incorrectly modify the conditional, including deducting values from the lamp position as he does in the distance conditional at (809).

(2305): (791-794, 801-802, 809-818 : total = 16)

```
791, ADDED(X): if((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / PI)<direction-25))
792, MUTATED(X): if((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / PI)<direction-25)
793, MUTATED(X): if(((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / PI)>direction-
25)%((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / PI)<direction+25))
794, MUTATED(X): if(((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / PI)>direction-
25)&&((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / PI)<direction+25))
801, MUTATED(O): if(((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / 3.1415)>direction-
25)&&((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / 3.1415)<direction+25))
802, MUTATED(X): if(((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / 3.1415)>direction-
25)&&((atan2(lampY - currentPosY, lampX - currentPosX) * 180 / 3.1415)<direction+25)){
809, MUTATED(X): if(((atan2((lampY-2.5) - currentPosY, (lampX-4) - currentPosX) * 180 /
3.1415)>direction-25)&&((atan2((lampY-2.5) - currentPosY, (lampX-4) - currentPosX) * 180 /
3.1415)<direction+25)){
810, MUTATED(X): if(((atan2((lampY-2.5) - currentPosY, (lampX-4) - currentPosX) * 180 /
3.1415)>direction-25)&&((atan2((lampY-2.5) - currentPosY, ((lampX-4) - currentPosX) * 180 /
3.1415)<direction+25)){
811, MUTATED(X): if(atan2((lampY-2.5) - currentPosY, (lampX-4) - currentPosX) * 180 /
3.1415)>direction-25)
812, MUTATED(X): if(atan2((lampY-2.5) - currentPosY, (lampX-4) - currentPosX) * 180 /
3.1415)>direction-25)
813, MUTATED(X): if((atan2((lampY-2.5)- currentPosY,(lampX-4) - currentPosX ) * 180 /
3.1415)>direction-25)
```

```

814, MUTATED(X): if((atan2((lampY-2.5)- (currentPosY),(lampX-4) - (currentPosX) ) * 180 /
3.1415)>direction-25)
815, MUTATED(O): //if((atan2((lampY-2.5)- (currentPosY),(lampX-4) - (currentPosX) ) * 180 /
3.1415)>direction-25)
816, MUTATED(X): (((atan2((lampY-2.5) - currentPosY, ((lampX-4) - currentPosX) * 180 /
3.1415)>direction-25)))
817, MUTATED(O): if(((atan2((lampY-2.5) - currentPosY, ((lampX-4) - currentPosX) * 180 /
3.1415)>direction-25)))
818, DELETED(O): DELETED (if(((atan2((lampY-2.5) - currentPosY, ((lampX-4) - currentPosX) * 180 /
3.1415)>direction-25))))

```

**Figure 362: Line History showing Thomas's implementation of the conditional for testing the angle between the avatar and the pickup object**

Given that the problem involving the distance formula was exceedingly simple, it appears the student's failure to resolve the problem may be to do with the two simultaneous errors making debugging of one of the conditionals in isolation difficult (the student only once for a single Change comments out one of the conditionals) as well as with the lack of visual feedback from the conditional statements. It should be noted that this student was the highest-achieving student enrolled in the course, so this was not an instance of a poor student with insufficient mathematical background being baffled by a relatively simple problem.

The student's final attempt at (817) is shown in Figure 363. Both the angle-testing and distance-testing conditionals are wrong, leading to poor and difficult-to-interpret results when testing the conditionals by picking up objects. The student removes the approach entirely at (818) and reverts to simply picking up objects wherever they are on screen.

```

if(sqrt(((lampX-4)*(lampX-4)-(currentPosX)*(currentPosX))
+((lampY-2.5)*(lampY-2.5)-(currentPosY)*(currentPosY))<2)
if((atan2((lampY-2.5) - currentPosY,
((lampX-4) - currentPosX) * 180 / 3.1415)>direction-25)))
    if((atan2((lampY-2.5) - currentPosY,
((lampX-4) - currentPosX) * 180 / 3.1415)<direction+25)))
    {
        holding=true;
    }

```

**Figure 363: Final attempt at using the distance and angle to object to determine whether to pick up an object at (817); removed in the next Change**

## **Conclusion**

Neither of the programming assignments involved complex mathematics. However, in most instances where students applied simple mathematical concepts (such as trigonometric functions for rotation or angle calculation, or the calculation of distances) students initially produced errors.

Students often appeared unable to correctly identify the source of problems as being rooted in their incorrect mathematical formulas as Ida's example of attempting to implement object rotation using a wrong rotation approach showed. In cases like this, considerable work and effort was wasted on attempting incorrect solution approaches. Another example is the work of Thomas and Christopher on limiting pick-up to objects near the avatar. They debugged their incorrect distance formulas for a long time without being able to identify the simple errors. On the other hand, student uncertainty often led students to work on and attempt to fix their mathematical formulas when the underlying problem was unrelated. For example, John's work on a formula to calculate a midpoint saw him initially use an incorrect formula  $(upper-lower)/2$ ; after correcting this to  $upper-(upper-lower)/2$ , the student reintroduced errors into the formula due to an unrelated problem to do with loss of precision.

When mathematical problems intersect with other problems, students tend to misidentify the source of the problem, either incorrectly overlooking an error in their mathematical formulas or incorrectly modifying them to fix non-existent problems. Such issues could be resolved by creating tasks which would allow students to test the correctness of their mathematical approach independent of other assignment tasks (such as event-driven programming). This may be undesirable however, as students are likely to experience such confluences of different problems in real-world programming.

A better approach might be to equip students with tools and strategies to debug mathematical problems visually. None of the students ever implemented any visual aids (such as drawing out the 'circle' around which an object is rotated, or drawing out the values produced by their incorrect distance formulas) which would have helped them quickly identify the source of their problems. By teaching students approaches to visualize their mathematical work using their OpenGL programs as visualization tools, students may learn how to better solve such problems and develop a greater appreciation of Computer Graphics.

Finally, the types of problems faced by students show that even for talented third-year students, relatively simple first-year mathematics problems can present a significant challenge and that challenge should not be underestimated when designing assignment tasks.

#### **9.7.7.1.2.4 General Programming**

## **Introduction**

The General Programming category of segments includes any segments that mainly involve application of / implementation of general computer science concepts such as implementation of algorithms or data structures, as well as segments that involve the fixing of syntax or semantic errors , errors relating to concepts such as Object-Orientation or errors relating to the particulars of the programming language and environment (in this case C++), such as errors involving the language's memory management implementation.

As the analysis of Coding results showed, General Programming made up a large number of total changes for all students' assignments. The percentage of those changes that were errors was relatively small, but was still a significant proportion of total errors for most students.

Many of these errors were minor syntax errors which occurred very regularly but took only one or two changes to fix (and hence are not included in analysis since they did not exceed the 10 Change thresholds). Such minor errors may be considered par for the course for a programming assignment, but tools or development environment support for the prevention or rapid addressing of such errors may reduce student time spent on dealing with them, hence increasing the time students can devote to core tasks and learning goals. However, the detailed examination of the role of minor errors falls outside the scope of this thesis.

Apart from the frequent minor syntax errors several more specific and more significant general programming errors were observed. Some students encountered General Programming errors that in fact were (by number of Changes required to solve the underlying problem) the most significant problems in those assignments. In some cases, it may be that these difficulties contributed to poor overall outcomes in assignments. For example Christopher did not manage to complete large parts of the third Assignment, which may in part be due to the fact that he spent more than 10% of his total assignment time on a problem relating to a bug in an algorithm (see [CHRISTOPHER\\_A3.GP.2."SceneTree traversal" \[121\]](#)) which was not directly related to any assignment task; this bug further interfered with his work on other assignment tasks relating to the implementation of avatar animation and assembly, since it was used to retrieve limbs for application of transformations. Whether students encountered such significant problems with General Programming depended heavily on the details of the way in which they chose to implement their assignments, meaning that some students encountered these difficult problems and were forced to invest significant effort to solve them while other perhaps luckier students did not encounter any very significant (30+ Changes) General Programming problems and could hence spend more time completing core assignment tasks.

## General Syntax and Semantic

All students at some point experienced syntax/semantic errors which required  $\geq 10$  Changes to fix.

Three segments ([MICHAEL\\_A1.GP.1."= /=="](#) [14], [THOMAS\\_A1.GP.1."= instead of =="](#) [10], [THOMAS\\_A1.GP.3."= instead of == \(2\)"](#) [14]) involved the confusion of the C++ assignment operator '=' and the C++ equality operator '==', with students accidentally utilising the assignment operator in conditional statements, leading them to always evaluate as 'true'. Several smaller problem segments ( $< 10$  Changes to fix) were also observed.

Three students encountered errors while dealing with C++ arrays in segments ([JOHN\\_A1.GP.2."Data type for RGB"](#) [10], [IDA\\_A3.GP.1."Passing Light W. Array"](#) [11], [MICHAEL\\_A1.GP.4."2D array syntax"](#) [11]). In the first two cases these problems relate to discovering the correct type to use with arrays. The third problem involves the usage of C++ syntax for two-dimensional arrays.

The GLUT API allows for the binding of user-implemented callbacks for various GLUT tasks; for example, the function designated to render the scene is registered via `glutDisplayFunc(...)`, whereas `glutTimerFunc(...)` calls a user-defined function after an interval specified by the user and can be used to implement time-driven behaviour. The callback mechanism uses C++ function pointers which are passed to the registration function. Two students encounter problems with function pointer syntax while attempting to register function pointers for timing-related tasks in segments ([JOHN\\_A3.GP.1."Static function pointer"](#) [13]) and ([CHRISTOPHER\\_A3.GP.3."glutTimerFunc function pointer"](#) [25]).

Students also encounter problems relating to a wide variety of different syntax/semantic issues. Ida ([IDA\\_A1.GP.4."Static functions"](#) [21]) encounters problems with both the syntax and the semantics of static functions while attempting to implement a hit code method for Diagram Objects.

While attempting to calculate the distance between two points, Michael ([MICHAEL\\_A1.GP.3."Bitwise operator"](#) [12]) utilises the `^` operator, attempting to calculate the square of a value. This produces an incorrect result as the `^` operator in C++ is a bitwise operator and does not perform the expected power-of-two operation, which the student realises after some debugging, after which he replaces the operator.

In the third assignment, Thomas ([THOMAS\\_A3.GP.3."Accidental Octal"](#) [14]) attempts to indent numeric literals used in his animation by using additional prefix zeroes to make them match up visually in the array in which they are stored. Unfortunately in C++ a leading 0 specifies that a literal



should be treated as an octal number, resulting in his animation not working as expected until the problem is addressed. The full scope of the problem is not captured in the associated segment, since much of the preceding work on the animation can likely also be attributed to problems caused by this accidental use of octal numbers.

Christopher's poor grasp of C++ syntax relating to pointers and referencing of objects causes him to struggle in the assignment of a value or NULL value to an object in segment (CHRISTOPHER\_A3.GP.1."Setting NULL / This keyword" [13]).

In segment (IDA\_A1.GP.2."Switch statement" [19]) Ida incorrectly sets up a switch statement, utilising the wrong syntax for cases and also omitting break statements which causes the switch statement to 'fall through' to all cases below the first case executed.

While creating debugging cout statements, Christopher (CHRISTOPHER\_A1.GP.3."C++ String" [22]) requires over twenty Changes to determine the correct syntax for concatenating an integer to a string, the syntax for which is somewhat terse in C++.

All general syntax/semantic problems discussed so far involved between 10-25 Changes, falling into the minor-to-moderate category of problem segments. Only a single general syntax/semantic problem segment falls into the category of severe problems. Christopher's use of macros to enumerate user interface actions in Assignment 1 requires 47 Changes to address (CHRISTOPHER\_A1.GP.4."Macro" [47]). As can be seen in Figure 364 the student adds a semi-colon after two macro statements (MOVE\_MODE and DELETE\_MODE) but not after three others. As a result, the switch statement shown below that fails to compile when DELETE\_MODE is used, since this effectively produces the statement `case 20;:` (notice the semicolon) which produces a compilation error. The student requires an extremely large number of Changes to identify and resolve this very simple problem.

```
37 #define PLACE_TABLE_MODE 1
38 #define PLACE_FLOWER_POT_MODE 2
39 #define PLACE_DOOR_MODE 3
40
41 #define MOVE_MODE 10;
42 #define DELETE_MODE 20;
```

```

141     case PLACE_DOOR_MODE:
142         vDoors.push_back(new Doc
143         break;
144
145     case DELETE_MODE:
146         for (int i=0;i<vTables.s
147             if (vTables[i]->hit
148                 vTables.erase
149             } //end if

```

**Figure 364: A semicolon after the DELETE\_MODE macro breaks the switch statement in (371)**

One final interesting case occurs in the same assignment ([CHRISTOPHER\\_A1.GP.7."Confusing function bracket error" \[25\]](#)) when the student misplaces a function's brackets, causing the error message: "error: insufficient contextual information to determine type". Without any real indication of where the error lies, the student requires many Changes to identify and fix it. While some of the other syntax/semantic errors were probably also made more difficult by less-than-helpful error messages, C++ error messages such as this are bound to cause significant confusion for students.

### **Student Oversights**

Student oversights also caused several problems: ([IDA\\_A1.GP.1."Variable naming" \[21\]](#), [IDA\\_A1.GP.3."Colour buttons" \[18\]](#), [CHRISTOPHER\\_A1.GP.6."General Semantics" \[10\]](#), [THOMAS\\_A3.GP.2."Bracketing" \[8\]](#), [THOMAS\\_A1.GP.5."Circular Parent-Child" \[54\]](#)). For example, in the case of [IDA\\_A1.GP.1."Variable naming" \[21\]](#) the student creates two different variables, `r` and `_r`, which she then confuses, mistakenly assigning to the wrong variable.

The largest segment involving an oversight is [THOMAS\\_A1.GP.5."Circular Parent-Child" \[54\]](#). The student is working on Assignment 1, implementing an algorithm to create programmatic parent-child links between objects (one of the assignment tasks).

The student's code implementing the functionality is shown in the first two panels of Figure 365. The `isChild(...)` function is implemented correctly and will detect whether the object whose id is input is a child of the object on which the method is called.

*The recursive function which tests whether an object is the child of another object*

```
126 bool isChild(int _id){
127     for(unsigned int i=0;i<children.size();i++){
128         if(children.at(i)->isChild(_id)){
129             return false;
130         }
131     }
132     return id==_id;
133 }
```

*The function which creates parent-child relationships.*

```
251 void parentFurniture(int _x, int _y){
252     if (selected==-1){//no parent
253         selectFurniture(_x,_y);
254     }else{
255         int parent=selected;
256         selectFurniture(_x,_y);
257         if (selected==-1){//no child
258             selected=parent;//revert changes
259             //NOTE: infinite loop if child =parent somewhere
260         }else{
261             if(!(furn.at(parent).isChild(selected)))//cannot
262                 furn.at(parent).append(furn.at(selected));
263         }
264     }
265 }
```

*The correction to the parentFurniture function which would have resolved the problem*

```
261         if(!(furn.at(parent).isChild(furn.at(selected))))
262             furn.at(parent).append(furn.at(selected));
```

**Figure 365: The student code for creating parent-child relationships and the fix that would have resolved the bug in the student's code**

However, the parentFurniture(...) code contains a bug. Instead of testing the parent item's id to check whether it is a child of the item it is to become parent of, it tests the parent item's position in the vector. To test the id, the student should have instead retrieved the object and then used its id with the statement furn.at(selected) instead of using selected, which is not the object's id but rather its position in the furn vector. The correction is shown in the third panel. This correction would have fixed the algorithm and the segment would have been complete after 14 Changes. Instead, the student removes the correct isChild(...) function and starts over, requiring 40 more Changes until he develops a working but inferior approach.

Overall, it is perhaps surprising that so few problems were caused by student oversight, though that may in part be because some errors that could have been classified as oversights (such as errors relating to event-driven programming discussed in Section 9.7.7.1.2.5) were classified as more

specific errors. On the other hand, it may also be due to third-year students making few such errors or being able to usually identify and correct them rapidly.

### **C++ Pointers and Pass-by-Value/Pass-by-Reference**

The students Christopher and Thomas encountered problems with C++ pointer syntax as well as the related concept of pass-by-value/pass-by-reference. These problems are captured in the segments:

- [CHRISTOPHER\\_A1.GP.1."this' keyword" \[10\]](#)
- [CHRISTOPHER\\_A1.GP.2." C++ this. syntax / static functions" \[12\]](#)
- [CHRISTOPHER\\_A3.GP.4."C++ Pointer/Direct syntax" \[12\]](#)
- [THOMAS\\_A1.GP.2."Child-Parenting Algorithm" \[114\]](#)
- [THOMAS\\_A1.GP.4."Pointer syntax" \[20\]](#)

The most significant problem related to C++ pointer semantics occurred in segment [THOMAS\\_A1.GP.2."Child-Parenting Algorithm" \[114\]](#). At 114 Changes, it is one of the largest segments discovered during analysis, indicating that this problem presented a significant challenge to the student.

The problem occurred while the student was attempting to implement functionality to programmatically create parent-child links between objects (the same task for which the student also later encountered the problem described earlier, captured in segment [THOMAS\\_A1.GP.5."Circular Parent-Child" \[54\]](#)). Code for the implementation is shown in Figure 366.

*The furniture vector to which furniture objects are added*

```
97 | vector<furniture> furn;
```

*The function for attaching a child to a parent:*

```
87 | //attaching a child
88 | void append(furniture f){
89 |     children.push_back(f);
90 | }
```

*Code creating a new furniture object, adding it to the furn vector and parenting it to a different furniture object*

```
276 | furniture a1(350, 150, 50, 50, T_SQUARE);
277 | furn.push_back(a1);
278 | a1.id=furn.size()-1;
279 |
280 | a.append(a1);
```

*The fix to the function for attaching children at (325)*

```

109 //attaching a child
110 void append(furniture &f){
111     children.push_back(&f);
112 }

```

Figure 366: Source code implementing the parent-child functionality at (166); last panel shows the fix at (325)

The student directly initialises objects as shown in the third panel. He then attempts to store these objects in a global vector called `furn`, which was to contain all furniture objects. Objects from this vector are drawn to screen in the `display` function. The objects can then be parented to other objects via the `append` function, shown in the first panel. The `append` function was intended to store the child object in a vector in the parent object. However, the student did not realise that since he does not use pointers, objects would be passed by value. This means that the `furn` vector stores a copy of the created furniture. The `append` function also receives a copy of the object. This means that the object in the `furn` vector is not the same as the child object stored by a parent. When a child object is moved by a parent (according to assignment specifications, the parent should move its child objects when it is moved), the parent changes the coordinates of a copy of the object instead of the object that is stored in the `furn` vector and displayed on screen.

The student spends well over a hundred Changes debugging this problem with `cout` statements, testing to see whether various parts of the source code have been executed and printing the id of objects to standard output before he finally produces the simple fix shown in the last panel in Figure 366.

### **C++ Memory:**

One of the assignment tasks was the implementation of a method of deleting diagram objects. Four of the five students simply removed deleted objects from the global data structures containing all objects, thereby introducing memory leaks. This was not a major issue since the final submission programs did not use a lot of memory and would have required a long period of ongoing use before generating a memory fault. However, John did attempt to free memory associated with deleted objects in segment [JOHN\\_A1.GP.3."Delete Mode Memory Problems" \[20\]](#). His attempts produced segmentation faults and he was unable to correct the error in his implementation to properly free memory.

### **C++ Object-Oriented Programming**

The assignment skeleton for Assignment 1 is object-oriented. Students were asked, though not required, to extend this skeleton using object-oriented programming to implement additional room

objects. Extension mainly involved the copying and adapting of existing objects. Nevertheless, three of the four students who implemented object-oriented code encountered problems which required more than ten Changes to address:

- [JOHN\\_A1.GP.1."Virtual function and Constructor" \[11\]](#)
- [JOHN\\_A3.GP.3."Object-Oriented" \[23\]](#)
- [IDA\\_A1.GP.5."Hit code and methods" \[17\]](#)
- [CHRISTOPHER\\_A1.GP.5."Virtual Keyword" \[31\]](#)
- [CHRISTOPHER\\_A1.GP.8."Virtual keyword 2" \[15\]](#)

In all segments except ([IDA\\_A1.GP.5."Hit code and methods" \[17\]](#)) the root problem involves the concept of ‘virtual’ methods.

Virtual methods work as follows. A Parent class implements a method called `getName()`. A subclass inheriting from this parent class, called Child, also implements a method called `getName()`, overriding the parent’s implementation. Given an object of type Child, by default C++ will call the Child `getName()` method if the Child object is stored in a variable of type Child, and the parent `getName()` method if the Child object is stored in a variable of type Parent. If the ‘virtual’ keyword precedes the `getName()` method declaration in the Parent Class, then the Child `getName()` function will be called whether it is stored in a variable of type Child or Parent. These semantics are different to languages such as Java in which an overriding method in a subclass object is always called instead of the parent object’s method. The fact that several students struggled with unexpected behaviour caused by the omission of the virtual keyword shows that the C++ semantics for implementing overriding functions proved confusing and were a hindrance in their implementation of an object-oriented programming model.

### **Loss of precision**

Some problems appeared to occur because of the way the assignment was structured. One such case was the ‘Loss of Precision’ problem.

The assignment skeleton included a Point class which represented screen coordinates with two Integer values. Students appear to have used this class as the basis for their own coordinate storage methods, using Integer variables to store coordinates for their on-screen objects.

As a result, the floating-point result of trigonometric calculation ‘rotating’ the child object’s centre coordinate around its parent was stored in an integer variable which led to loss of precision, with the

decimal part of the result being lost. This led to the child object moving away from or toward the parent object during rotation.

The problem is described in more detail in Section 9.7.7.1.1.1. In short, students for a long time failed to identify the Loss-of-Precision error, instead looking for the error in their event-driven and mathematical work, making the rotation problem much more difficult to solve. All of the three students that implemented child-parent rotate functionality (two failed to implement it) used integer variables to store coordinates, resulting in the following problem segments involving loss of precision:

- [IDA\\_A1.GP.7."Child object rotate"\[56\]](#)
- [THOMAS\\_A1.GP.6."Loss of Precision" \[31\]](#)
- [THOMAS\\_A1.GP.7."Unintentional Rounding" \[43\]](#)
- [JOHN\\_A1.SP.7."Rotation around Parent" \[56 \]](#)

As this example shows, care must be taken to structure assignments in ways which do not create unintended, hard to understand problems. In this case the problem was made difficult by the interaction between different ‘problem types’. This particular problem with the assignment specification would likely have gone unnoticed if not for Project History analysis, since both students largely addressed the problem in their final submission. To prevent similar problems from occurring in the future, a problem similar to the one encountered by students in this case could be made into a practical exercise, requiring students to identify and address it in a controlled environment.

### **WaveFrontImporter:**

Another category of General Programming problem was also related to the assignment specification. In Assignment 3, students were required to import models they had created in a 3D modelling environment into their OpenGL programs. To do so, they were required to use the WaveFrontImporter library, which was implemented by this research project’s main researcher who was also part of the teaching staff for the unit. Three students encountered problems with using the WaveFrontImporter library:

- [JOHN\\_A3.GP.2."WaveFrontImporter" \[12\]](#)
- [IDA\\_A3.GP.2."WaveFront importer" \[21\]](#)
- [THOMAS\\_A3.GP.1."Importer Problems" \[39\]](#)

This suggests that better documentation or examples should be developed for future use of the WaveFrontImporter.

### **General Algorithms and data structures**

General Computer Science:

- [IDA\\_A1.GP.6."Loop error" \[20\]](#)
- [CHRISTOPHER\\_A3.GP.2."SceneTree traversal" \[121\]](#)
- [MICHAEL\\_A1.GP.2."Inefficient render" \[24\]](#)
- [MICHAEL\\_A3.GP.1."Inefficient Limb Selection" \[23\]](#)
- [MICHAEL\\_A3.GP.2."Inefficient Limb Rotation" \[28\]](#)

The assignment specifications for Assignment 1 and Assignment 3 intentionally did not include any tasks that involved the implementation of general Computer Science algorithms or data structures since the focus of the assignments was to be on content related to Computer Graphics programming. However, in two instances students did encounter problems with general Computer Science algorithms during their assignment work.

The first ([IDA\\_A1.GP.6."Loop error" \[20\]](#)) involved the student incorrectly returning a 'false' value (-1) inside rather than outside a loop (see Figure 367), causing it to always terminate after the first iteration.

```
90 int DiagramObject::buttonHit(Point _p){
91
92     for(int j=0; j<4; j++){
93         if (cornerButton[j]->hit())
94             return j;
95         return -1;
96     }
97 }
```

**Figure 367: Incorrectly returning -1 inside the loop body**

A much more significant problem related to algorithms occurs for Christopher ([CHRISTOPHER\\_A3.GP.2."SceneTree traversal" \[121\]](#)). In lectures, students were taught about the concept of a Scene Graph, a graph which is used to store objects such as graphics primitives. It is in essence a regular graph, with visit functions applying transformations to objects as they visit nodes in the graph, and connections from one node to the next representing parent-child relationships which can be used to produce hierarchical models. One student decided to implement a Scene



Graph to implement the hierarchical construction of the avatar. Excerpts from his code relating to the function used to retrieve limbs are shown in Figure 368. The `getNodeByName(...)` function is intended to return an object stored in the Scene Graph using a recursive search starting at the node on which the call is made, and then recursively searching the node's descendants. The initial implementation at (104) shown in the first panel omits the 'return' keyword in front of the recursive `getNodeByName(...)` call, meaning that the function will always fall through to return NULL at line 102. The student fixes this error at (137) (second panel) by preceding the `getNodebyName(...)` call with a return statement. However, this means that the algorithm will always return the search result for the first child, even if the node was not found when recursing into the first child and its descendants, in which case it returns nothing (essentially equivalent to returning a NULL value). The student corrects this error at (211) by testing whether the value returned by a recursive `getNodeByName(...)` call is NULL before returning it, and not returning the result if it is NULL. This simple algorithmic error takes the student 121 Changes to fix, making it the largest single segment found during analysis of all assignments. It involves over 10% of all Changes made by the student for that assignment.

*The recursive limb-retrieval function at (104) does not return the value of its recursive call:*

```

92 SceneTreeNode* getNodeByName (string _name) {
93     if (name==_name) {
94         return this;
95     }
96     else
97     {
98         for (int i =0; i<childNodes.size();i++) {
99             childNodes[i]->getNodeByName (_name);
100         }//end for
101     }
102     return NULL;

```

*The return statement is added at (137), but it returns even if the function does not find the limb:*

```

98 SceneTreeNode* getNodeByName (string _name) {
99     cout << "SEARCHING: " << name << endl;
100     if (name==_name) {
101         return this;
102     }
103     else
104     {
105         for (int i =0; i<childNodes.size();i++) {
106             return(childNodes[i]->getNodeByName (_name));
107         }//end for
108     }
109 }

```

The correction at (211) checks whether the recursive function found the limb before returning the result:

```

103 SceneTreeNode* getNodeByName (string _name) {
104     cout << name << ":" << _name << " -> " ;
105     if (name==_name) {
106         cout << "Search found: " << name << endl;
107         return this;
108     }
109     else
110     {
111         if (childNodes.size()==0) {
112             return NULL;
113         }
114         for (int i =0; i<childNodes.size();i++) {
115             if (childNodes[i]->getNodeByName(_name)!=NULL) {
116                 return (childNodes[i]->getNodeByName(_name));
117             }
118         } //end for
119     }
120 }

```

Figure 368: Implementation of a Scene Graph traversal function

Student Michael encountered a different problem in his implementation of both Assignments. The student produced an algorithm which negatively impacted on the performance of his program in Assignment 1 ([MICHAEL\\_A1.GP.2."Inefficient render" \[24\]](#)) and implemented a very verbose and ineffective mechanism for selecting and rotating limbs in Assignment 3 ([MICHAEL\\_A3.GP.1."Inefficient Limb Selection" \[23\]](#), [MICHAEL\\_A3.GP.2."Inefficient Limb Rotation" \[28\]](#)). The implementation of the algorithm for selection of a limb ([MICHAEL\\_A3.GP.1."Inefficient Limb Selection" \[23\]](#)) is shown in Figure 369. The student could utilise a simple variable which stores an integer identifying the currently selected limb; he could then switch between the limbs by incrementing or decrementing this value. Instead, the student produces a Boolean variable for every limb, and then in the keyboard handler produces a block containing a conditional statement for every limb; in each block, the Boolean value identifying the previous limb is set to false, and the Boolean variable identifying the current limb is set to true. This is unnecessarily verbose and takes a long time to implement. In addition, it means that when he subsequently works on rotating the selected limb ([MICHAEL\\_A3.GP.2."Inefficient Limb Rotation" \[28\]](#)) he must again produce a block of conditional statements for every limb. Had he used a single Integer variable to hold the id / vector position of the currently selected limb, this could have been achieved with a single statement. This is clearly a case of the student not having mastered some basic Computer Science concepts, which leads him to spend more time than necessary on non-core tasks. It is especially significant in the context of the assignment in which it occurs, since the student fails to implement many of the

required tasks; part of the reason may be that he spent nearly 10% of Changes on this very inefficient limb selection algorithm which also caused all tasks related to limb selection to become unnecessarily complex. The student never implements proper animations, perhaps in part because of the inadequacy of his method of limb selection.

```

648 else if(key == '['){
649
650     if (slthead){
651         cout << "RightUpperLeg" << endl;
652         slthead=FALSE;
653         sltrul=TRUE;
654     }
655     else if (sltrul){
656         cout << "RightLowerLeg" << endl;
657         sltrll=TRUE;
658         sltrul=FALSE;
659     }
660     else if (sltrll){
661         cout << "RightFoot" << endl;
662         sltrll=FALSE;
663         sltrfoot=TRUE;
664     }
665     else if (sltrfoot){
666         cout << "RightPalm" << endl;
667         sltrfoot=FALSE;
668         sltrpalm=TRUE;
669     }
670     else if (sltrpalm){
49 bool slthead=TRUE;
50 bool slttorso=FALSE;
51 bool sltlua=FALSE;
52 bool sltlla=FALSE;
53 bool sltlpalm=FALSE;
54 bool sltlfoot=FALSE;
55 bool sltlll=FALSE;
56 bool sltlul=FALSE;
57 bool sltbody=FALSE;
58 bool sltrua=FALSE;
59 bool sltrla=FALSE;
60 bool sltrpalm=FALSE;
61 bool sltrfoot=FALSE;
62 bool sltrll=FALSE;
63 bool sltrul=FALSE;

```

Figure 369: The inefficient algorithm for switching between limbs

## Conclusion

All students at some point encountered problems to do with programming language syntax or semantics. Some were of a general nature and would probably have occurred in most modern programming languages, like students using the assignment operator (=) instead of the equality operator (==) but others such as problems relating to object-orientation (which occurred for several students) seemed to occur specifically because of the rather complex and unintuitive C++ semantics.

One of the most significant problem segments found in any of the ten Project Histories was segment THOMAS\_A1.GP.2."Child-Parenting Algorithm" [114] in which the student's implementation of programmatically storing child objects inside parent objects failed because the objects were

accidentally passed 'by value' instead of 'by reference', yet another C++ semantic feature which is unintuitive to students.

Since no data on the use of a different programming language exist there is no comparison point, but it may be useful to experiment with a different programming language / environment such as Java, for which JOGL provides a full OpenGL API implementation. Indeed, several students expressed their preference for Java in their reflection question answers submitted along with the third assignment which are disseminated in Appendix Section 9.3.5.

Whereas problems relating to core assignment tasks generally occurred fairly predictably as intended for all students as was apparent when analysing major tasks, General Programming errors occurred largely based on the particulars of a student's implementation, since students using other implementation approaches would never be in a situation to make that type of error. For example, Christopher's decision to implement a Scene Graph in Assignment 3 ([CHRISTOPHER\\_A3.GP.2."SceneTree traversal" \[121\]](#)) was the largest problem faced by any student in terms of number of Changes. His decision to use macros rather than some other method to store identifiers in Assignment 1 ([CHRISTOPHER\\_A1.GP.4."Macro" \[47\]](#)) led to his spending a considerable amount of time on debugging C++ macro syntax issues.

When such problems occur, they force the student to invest effort into non-core tasks and hence to have less time for the core tasks designed to help students achieve learning goals. Furthermore, since they occur in an unpredictable manner based on students' design decisions and previous knowledge of certain semantic issues, the work required differs significantly between students, creating an advantage for students who do not encounter such problems, which is often a matter of chance and may in fact penalise the more exploration-minded students interested in creating richer, more real-world programs. This raises the question as to whether assignments should be structured more rigidly to prevent such problems from affecting a subset of students, or whether this issue is offset by the increased interest and external validity created by a more open assignment task specification.

Assignment task specification can also lead to unintended but predictable challenges for students. The loss-of-precision problem turned out to be a significant problem for all students who implemented the child-parent rotation functionality (see Section 9.7.7.1.1.1), but it was a side-effect of how the assignment skeleton was implemented rather than an intentional problem to be solved. Through analysis of Project Histories such problems occurring during assignment development can be identified and addressed in subsequent iterations of the unit, either by designing assignments to

lessen the likelihood of the problem occurring (by using floating-point values in the assignment skeleton, for example) or by presenting students with material regarding the problem to equip them to identify and solve it when it does occur. Materials such as the WaveFrontImporter library used in the third assignment can also be evaluated by analysis of the Project History. In the case of the WaveFrontImporter library, student difficulties with its use([JOHN\\_A3.GP.2."WaveFrontImporter" \[12\]](#), [IDA\\_A3.GP.2."WaveFront importer" \[21\]](#), [THOMAS\\_A3.GP.1."Importer Problems" \[39\]](#)) indicate that the API and documentation should be improved for future iterations of the unit.

Finally, a lack of programming fundamentals can hinder students in achieving learning objectives. This was the case for Michael, whose poor implementation of supporting data structures and mechanisms for selection of limbs in Assignment 3 cost him a significant amount of time, and presumably also made work on core tasks more difficult because source code to implement assembly or animations as a result needed to be much more verbose and was also less comprehensible. While addressing such weaknesses lies outside the scope of Computer Graphics education, the identification of students struggling with basic computer science skills and the provision of remedial support to these students may be the best way to improve their learning outcomes in specialised areas such as Computer Graphics.

#### **9.7.7.1.2.5 Event-Driven Programming (Program Flow) and Pipeline**

**Introduction** In this section errors relating to event-driven programming and program flow will be discussed together with pipeline errors since these error types often co-occurred. Lighting problems also included pipeline-related issues, but these are described in the Lighting section (Section 9.7.7.1.2.7).

##### **Program Flow**

**Event-drive program flow** Event-driven program flow problems involve errors such as a student setting the wrong variable or overwriting a variable's value at the wrong location, leading to the variable containing the wrong value. It can also include the event-driven model being in the wrong state during an action, or losing information relating to an action because of an incorrect sequence of instructions. All students Project Histories for the first assignment contain at least one Segment larger than 10 Changes involving an event-driven program flow problem, several of which involved a large number of Changes ([JOHN\\_A1.ED.2."Deleting Children" \[11\]](#), [JOHN\\_A1.ED.3."GLUT Menu" \[23\]](#), [IDA\\_A1.ED.2."Dynamic Object Storing" \[19\]](#), [IDA\\_A1.ED.4."Flower pot move" \[15\]](#), [IDA\\_A1.ED.5."Selection" \[22\]](#), [CHRISTOPHER\\_A1.ED.5."ED program flow" \[11\]](#), [CHRISTOPHER\\_A1.ED.6."ED Program flow 2" \[12\]](#), [MICHAEL\\_A1.ED.1."Button highlight" \[45\]](#),

MICHAEL\_A1.ED.2."Select II > ur" [30], MICHAEL\_A1.ED.3."GLUT Menu" [28], MICHAEL\_A1.ED.4."Resizing circle" [10], THOMAS\_A1.ED.1."Not Updating Variable" [20]). Such problems tended to occur towards the middle or end of the assignment as event-driven code became more complex.

A description of the segment (IDA\_A1.ED.5."Selection" [22]) will illustrate the difficulty students encountered when implementing event-driven functionality. The student is attempting to implement a new user interface functionality to allow the user to change the colour of the selected object. In doing so, the student breaks the event-driven mechanism, since the selection algorithm to select a different object becomes unreachable once an object has been selected (the mode variable is no longer "none"). As shown in Figure 370, the second and third else-if clauses which deselect and select objects become inaccessible due to the addition of the first if-clause which is always triggered once an object has been selected. This makes deselection or selection of a different object impossible. The student then requires from (586-614) to correct the program-flow issues and produce a working event-driven algorithm which allows for deselection and reselection of objects. Wrong solution attempts include moving the deselection/mode-reset statement to the 'else' clause where it never gets triggered (at 594), making the reset action always trigger (therefore always unselecting any selected object immediately and always switching back into "None" mode) at (597-598) and removing the "selected" mode altogether at 609.

```

if(action == "selected"){
    for(int i=0; i<28; i++){
        if(colorButton[i]->hit(clickDown)==true){
            color = i;
            action = "color";
            for(int j=0; j<tableCount; j++){
                if(newPieceOfFurniture[i]->isSelected()){
                    newPieceOfFurniture[i]->setColor(color);
                }
            }
        }
    }
}

else if(clickDown.x>int(application_width*0.85)){
    action = "none";
    draw = "none";
}

else if(action=="none")    //No buttons clicked and no action
{
    draw = "none";
    for(int i=0; i<tableCount; i++){
        if(newPieceOfFurniture[i]->hit(clickDown)){
            newPieceOfFurniture[i]->setSelected(true);
            newPieceOfFurniture[i]->render();
            action = "selected";
        }
        else{
            newPieceOfFurniture[i]->setSelected(false);
            newPieceOfFurniture[i]->render();
        }
    }
}
}

```

**Figure 370: The 'if' clause is always triggered, making deselection of an object impossible.**

The correct solution shown in Figure 371 is developed from (612-614) when the selection algorithm is placed into its own separate if clause that is always called when the event-driven mode is “select” or “selected” (and therefore allows deselection or reselection).

```

if(action == "select" or action == "selected"){
    draw = "none";
    for(int i=0; i<tableCount; i++){
        if(newPieceOfFurniture[i]->hit(clickDown)){
            newPieceOfFurniture[i]->setSelected(true);
            newPieceOfFurniture[i]->render();
            action = "selected";
        }
        else{
            newPieceOfFurniture[i]->setSelected(false);
            newPieceOfFurniture[i]->render();
        }
    }
}
if(action == "selected"){

```

**Figure 371: The correct solution always executes the selection algorithm**

**Lower-Left > Upper-Right** The segment previously discussed involved the student losing track of the program flow of event-driven code, leading to functionality being inaccessible. Another common cause for errors is the incorrect modification of variables, leading to unexpected behaviour. An example of this type of problem is the segment ([MICHAEL\\_A1.ED.2."Select ll > ur" \[30\]](#)). Michael is attempting to implement event-driven furniture select functionality; however, because the student does not ensure that a furniture's lower-left coordinate is smaller in its x and y components than an upper-right coordinate, the furniture hit test does not work for objects where this is the case, which means that such objects cannot be selected. To debug and fix the problem, the student uses cout statements (1133-1139), modifies loops used to iterate through objects (1140-1142) and modifies the call to the display handler (1144-1145) before correctly identifying the source of the error and fixing it. As happens in other instances of this problem type, a large part of the problem-solving process involves the correct identification of the source problem.

**GLUT event-handling** Some problems are also caused by the GLUT event-handling mechanism acting in ways not anticipated by the student. A common problem involved students not realising that mouse presses cause two mouse events (one mouse up and one mouse down), leading to two instead of one application event being created (such as adding two objects instead of one). For instance, in segment ([JOHN\\_A1.ED.3."GLUT Menu" \[23\]](#)) John is attempting to create a GLUT popup menu which is then closed after a mouse press and the selected value stored. However, because GLUT generates two mouse events (a mouse-up and a mouse-down event), the final value is read after the menu is already closed. This results in the returned value always being the default value. The student spends several changes debugging this problem.

### **Non-storage of graphics primitives**



**Draw-not-store** While the event-driven program flow issues discussed earlier involved students losing track of the workings of their event-driven code, some problems were due to conceptual misunderstandings of the handling of information in OpenGL. Such ‘draw-not-store’ problems involved students drawing out visual elements at the time of an event-driven action meant to create a new object (drawing it in the mouse or keyboard handler, for example) without storing data related to the object to be created. This error will mean that the event-driven action’s effects will be removed when the screen is cleared after the next `glClear` call, which usually occurs whenever the screen is redrawn (as the result of the next mouse/keyboard action for example). This problem occurred for two students in segments ([IDA\\_A1.ED.2."Dynamic Object Storing" \[19\]](#), [MICHAEL\\_A1.PI.1."Init render" \[19\]](#), [MICHAEL\\_A1.PI.4."Highlight not stored" \[24\]](#), [MICHAEL\\_A1.PI.5."Not storing objects" \[45\]](#), [MICHAEL\\_A1.PI.8.1."Wall Drawn Not Stored" \[65\]](#)). Problems of this type proved especially challenging to Michael.

The problem segment ([MICHAEL\\_A1.PI.5."Not storing objects" \[45\]](#)) is an example of this type of problem. To implement event-driven functionality to add new furniture objects to the scene, the student attempts to render newly created objects inside the mouse handler immediately without storing them rather than storing them and rendering them when the display handler is called (see Figure 372). This means the object is cleared immediately, as the mouse handler calls `glutPostRedisplay()` and the display function clears the screen. The student attempts to fix this problem by adding more (superfluous) `glutPostRedisplay()` and `display()` calls (e.g. 816-817, 830-835), drawing various shapes in the mouse handler directly to see whether the object’s render function is buggy (it’s not) at (826-831), using `cout` statements to debug whether the object’s render statement inside the mouse handler is being reached at (818, 820), debugging the event-driven code (which is not at fault) from (820-835) and changing the colour of the furniture object from (826-831) before correctly rendering the furniture object in the display function from (837-838). It is also interesting to note that the student first attempts (correctly) to render the object in the display handler at (807), but due to a general programming error that leads to a (silent) segmentation fault the student gives up on the correct solution approach and only returns to it many changes later.

```

static void processMouse(int button, int state, int x, int y)
{
    ... code removed...

    //drawing objects or executing actions
    if(mode ==3 && x > application_width*0.01 && 480-y > application_height*0.01 && x < app]
    {
        // Point windowLower = Point(int(application_width*0.01), int(application_height*0.01))
        // Point windowUpper = Point(int(application_width*0.85), int(application_height*0.955))
        (new DiagramObject("Piece of Furniture2", Point(x,y), Point(x+50,y+50)))->render();
        glutPostRedisplay();
    }
    ... code removed...
    glutPostRedisplay();
}

```

**Figure 372: Student attempt to add objects by initialising and rendering them in the mouse handler**

This stretch displays a definite confusion with how the pipeline works and how draw actions are sequenced, and also showcases the most common behaviours when such a problem is encountered: useless colour changes, debugging of event-driven code and superfluous addition of redisplay or flush calls.

**Wall not Stored** Segment ([MICHAEL\\_A1.PI.8.1."Wall Drawn Not Stored" \[65\]](#)) by the same student is another example of drawing instead of storing a new object, suggesting that the student still has not developed a proper model of the OpenGL pipeline and how to use it to create persistent objects. While implementing an algorithm to draw walls by specifying corner points the student uses a hanging glBegin() call, making further glVertex() calls when the mouse is pressed in order to add further vertices to the wall (see Code Excerpt Figure 373 from 1903). This approach shows the student's misunderstanding of the glBegin-glVertex-glEnd mechanism, and spends considerable time attempting to implement this incorrect approach. Furthermore, the problem is also another case of a *Draw-not-store* error, as the student is attempting to draw the wall to the screen as it is being drawn rather than storing it and drawing it in the display handler.

```

Entering Wall-Drawing mode: Hanging glBegin
438  if (mode==1) {
439      glColor3f(0.0,1.0,0.0);
440      glBegin(GL_LINE_LOOP);
441
442  }

Mouse-down: Adding vertices to hanging glBegin
342  void drawWalls(int x,int y)
343  {
344      glColor3f(1.0,0.0,0.0);
345      glBegin(GL_POINTS);
346      glVertex2i(x, y);
347      glPointSize(10.0);
348      glEnd();
349  }

```

**Figure 373: Attempting to draw a wall using a hanging glBegin statement (1903)**

### **Logical Operations**

In implementing rubber shapes for the creation / resizing of objects (the rubber shape shows the final size of the object while it is being created), students were given the choice to either simply redraw the screen with the new 'shape' or to utilise logical operations, specifically the XOR operation, to draw and erase the rubber shape without a redraw operation. Of the students analysed, only Ida attempted to utilise logical operations to implement rubber shapes. She encountered significant problems during implementation, captured in Segment [IDA\\_A1.ED.3."Logic Ops"](#) [37]. The student's implementation erroneously changes the colour to black (r=0, g=0, b=0) before drawing the 'rubber shape'. Since 0 xor'ed with any value x will simply produce the value x unchanged, the drawing of a black 'rubber shape' will not draw anything at all. The student spends 37 Changes until she correctly identifies the source of the problem; in the meanwhile she misidentifies it as a problem with the event-driven code launching the action, modifying variables holding the dimensions of the rubber shape, as well as enabling and disabling the use of logical operations at various different points in the program flow. In doing so, she introduces new errors which make solving of the problem more difficult.

This problem is an example of a student lacking the tools or debugging strategies necessary to promptly identify such an error. For example the student could have accessed the red/green/blue values before and after the XOR operation to trace the exact numerical (non-)effect of the XOR operation on pixels. Solutions to such learning problems might include teaching students techniques for debugging OpenGL problems more effectively and/or creating a list of such common student misconceptions and oversights to allow students to quickly identify their source.

### **Stacking of calls**

John and Thomas both encountered significant pipeline issues when attempting to implement their minimap Views ([JOHN\\_A3.VIEW.1."Minimap" \[37\]](#), [THOMAS\\_A3.VIEW.2."Ortho/Top-down Camera" \[20\]](#)). These problems were caused largely by pipeline issues relating to stacking of projection or view calls because students failed to load the identity matrix into the View / Projection matrix before loading a new top-down View/projection.

### **Discussion**

All students encountered event-driven program flow problems. A solution to this would be to provide students with better tools and strategies to debug program flow problems such as breakpoint debugging. These tools and strategies should also be designed to be used in an OpenGL environment, giving students access to the state of the OpenGL graphics pipeline and state machine. Giving students the ability to visually represent coordinates may also prove helpful. In the segment ([MICHAEL\\_A1.ED.2."Select II > ur" \[30\]](#)) in which the student accidentally makes the 'lower-left' coordinate value larger than the 'upper-right' value, leading to hit code not working, the ability to draw lower-left and upper-right coordinates to screen would probably have helped the student resolve the problem quickly. In general, the analysed event-driven and pipeline segments show that a large part of the challenge with this type of problem is its hidden nature; methods to visualise or break down such problems may aid students in their learning and increase their confidence when working on such problems by allowing them to exclude potential problems.

As the segment ([JOHN\\_A1.ED.3."GLUT Menu" \[23\]](#)) showed, the OpenGL event-handling mechanism can also lead to student errors when it operates counter to student expectations. Debugging and visualization techniques would also help address these types of issues. In addition, the inner workings of the OpenGL event-handling mechanism should be explained to students and learning material to be consulted in the case of problems should be available.

#### **9.7.7.1.2.6 General OpenGL and OpenGL Syntax**

##### **General OpenGL**

When first learning programming, many students struggle with program language syntax, which has led educators to using scaffolding by providing specially designed programming languages and/or IDEs. While some students still struggle with some of the peculiarities of the C++ programming language (see Section 9.7.7.1.2.4), students generally did not have problems with using the OpenGL API.

General OpenGL syntax and semantic problems not falling into any more specific category did occur for most students, but rarely presented a significant hurdle, often requiring only a single change to fix.

An OpenGL syntax problem that did pose a moderate hurdle to one student ([JOHN\\_A1.GP.2."Data type for RGB"](#)) was the use of the `glColor3f` function; the student required several changes to correctly match up the `glColor3ub` function call with the `GLubyte` data type to store object colour.

The most common general OpenGL syntax problem involved the use of incorrect OpenGL enumerated types. For example, Thomas required eight changes (Assignment 1, 432-440) to identify the erroneous use of `GL_POINT` enumeration instead of the correct `GL_POINTS` enumeration in a `glBegin(...)` call; this led to nothing being drawn by the affected `glBegin(...)` call, and required the student to spend several changes to identify the problem source.

One student repeatedly experimented with different ways of drawing primitives. In Assignment 1 Thomas several times ((991, 1016, 1018-1021), (1164-1167, 1170), (432, 440, 493, 495), (590, 593-595), (625-627, 633)) changed between `GL_LINE_LOOP`, `GL_POINTS`, `GL_POLYGON`, `GL_TRIANGLE_STRIP` and `GL_TRIANGLE_FAN` draw modes in drawing 2D shapes. However, these instances appear to have arisen from experimentation rather than any real underlying problem, and the most significant instance consists of only 6 Changes.

Problems relating to OpenGL syntax as described above generally took few changes to fix and did not exceed the ten-Change boundary above which segments were included for detailed analysis, meaning that they were not real problems requiring further consideration.

An instance of a misunderstanding of OpenGL semantics relating to OpenGL enumerations occurs in [MICHAEL\\_A1.GL.1."Drawing lines" \[14\]](#) when the student first experiments with creating 2D primitives. A list of milestones is shown in Figure 374.

*Change 74:*

```
122 void grid (void)
123 {
124     Point windowLower = Point(int(application_width*0.01),
125     Point windowUpper = Point(int(application_width*0.85),
126     GL_LINES();
127 }
```

*Change 77:*

```

124 |     Point windowLower = Point(int(application_width*0.01),
125 |     Point windowUpper = Point(int(application_width*0.85),
126 |     drawLine()

```

Change 81:

```

124 |     line (int(application_width*0.01),

```

Change 83:

```

121 | glBegin(GL_WHATEVERTHESHAPE)
122 | {
123 | glVertex2i(somex, somey);
124 | /* lots more vertices */
125 | }
126 | glEnd();

```

Change 113:

```

81 |     glBegin( GL_LINE); /*
82 |     glVertex2i(200, 125);
83 |     glVertex2i(100, 375);
84 |     glVertex2i(300, 375);
85 |     glEnd(); /* OpenGL dra
86 |     glFlush(); /* Complete

```

Change 114:

```

81 | glBegin( GL_LINE); /* draw filled triangle */
82 | Point(int(application_width*0.86), int(applicat
83 |     Point(int(application_width*0.86+40),
84 | glEnd(); /* OpenGL draws the filled triangle */

```

Change 124:

```

83 | glBegin( GL_LINE); /*
84 | glVertex2i(200, 125);
85 | glVertex2i(100, 375);

```

Change 129:

```

83 | glBegin(GL_LINES); /* draw filled triangle */
84 | glVertex2i(application_width*0.86,application_height*0.85);
85 | glVertex2i(100, 375);

```

**Figure 374: The student's first attempt at drawing lines; different approaches at 74, 77, 81, and 83**

In (74) the student attempts to use the `GL_LINES` enumeration as a function, calling the non-existent `GL_LINES()` function. In (77) the student attempts to use another non-existent function, `drawLine()`. At (81) he experiments with another non-existent function `line(int,int)` which he provides with two parameters, presumably hoping it will draw a line between the points. At (83) he introduces pseudo-code from the lecture notes which he then experiments with for several Changes. At (113) he attempts to use this code to draw lines, but uses `GL_LINE` instead of

`GL_LINES` with the `glBegin(...)` function, resulting in the function not producing any output since `GL_LINE` is not meant to be used as an input parameter to `glBegin(...)`. As a result the student abandons the correct approach at (114), replacing the correct `glVertex(...)` calls with calls to construct `Point` objects (which are not part of OpenGL and have no effect on the OpenGL state machine). Between (114) and (128) the student experiments with various solution approaches, including changing the drawing colour at (118) (presumably because he thinks the drawing may be blending in with the background) before correcting the error at (129) by utilising `GL_LINES` instead of `GL_LINE`, resulting in correct line rendering.

All in all, the student spends considerable time on the problem (unfortunately the segment only captures part of the related Changes). However, other students do not encounter difficulty of anything near this magnitude, generally needing only a handful of Changes to draw their first OpenGL primitive. This suggests that other students developed a good grasp of the concepts involved based on practical and tutorial exercises while this student did not, resulting in considerable extra work for him in completing the assignment.

### **Screen-Window conversion**

In OpenGL, the drawing area's coordinate system in 2D has its origin in the top-left corner. On the other hand, the GLUT mouse event coordinate system starts in the bottom-left corner. As a result, drawing and mouse coordinates are inverted. To address this problem, the programmer must invert either drawing or mouse y-coordinates manually ( $y = \text{height} - y$ ).

Despite students being made aware of this inconsistency during lectures and in tutorials on OpenGL event handling, all students at some point did not convert from screen to window coordinates. Most of these oversights were addressed relatively quickly in 2-9 changes, but there were several instances, usually when students first encountered the mouse-coordinate window-coordinate discrepancy, that took 10 or more changes to address ([CHRISTOPHER\\_A1.ED.1."Screen-Window" \[9\]](#), [JOHN\\_A1.ED.1."Screen-Window convo" \[15\]](#), [MICHAEL\\_A1.SP.10."Circle resize" \[18\]](#), [IDA\\_A1.ED.1."Hit Code" \[11\]](#), [THOMAS\\_A3.ED.2."Forgets screen-window conversion" \[21\]](#)).

For instance while implementing code to detect button clicks in Assignment 3, Thomas ([THOMAS\\_A3.ED.2."Forgets screen-window conversion" \[21\]](#)) mistakenly identifies the button hit code as the reason why the buttons are not registering mouse clicks, and spends most of the 21 changes attempting to fix the hit code before realising the problem lies with the mouse coordinate-screen coordinate conversion, upon which he fixes the problem in a single Change.

Most students encountered problems related to mouse coordinate-screen coordinate conversion more than once, though in most cases subsequent problems were quickly addressed as students were able to immediately identify the source of the problem.

### **Degree-Radian Inconsistency**

The inconsistency between OpenGL and C++ in trigonometric functions (OpenGL uses degrees, C++ uses radians) causes several students to incorrectly use degrees rather than radians with C++ trigonometry functions (see [IDA\\_A1.SP.2."Rotate Icon" \[41\]](#), [IDA\\_A1.SP.3."Object rotation" \[23\]](#), [THOMAS\\_A3.SP.2.2."Avatar Movement" \[36\]](#), [THOMAS\\_A3.SP.2.1."Angle Conversion" \[12\]](#)). These segments do not encompass all the times students incorrectly mix up radians and angles, but other instances do not reach significance at 10 or more Changes.

For example, incorrect use of degrees and radians contributed to the difficulty in implementing parent-child rotation for Ida in [IDA\\_A1.SP.3."Object rotation" \[23\]](#) as is described in more detail in Section 9.7.7.1.1.1.1. Students attempted to convert between radians and degrees in situations where this conversion was incorrect (converting to degrees for use with C++ maths library functions, for example) in an attempt to solve unrelated problems because of their misidentification of the unit of measurement as a potential problem source.

### **Conclusion**

The OpenGL/GLUT APIs were mastered quickly by students and did not appear to present a major challenge. Problems associated with general OpenGL syntax were few, and those that did occur were usually solved very quickly. OpenGL appears to be a good API for teaching Computer Graphics, not placing a large overhead on students during their learning. Michael did encounter a problem relating to the syntax for drawing OpenGL primitives but then addressed the problem by use of lecture notes. This combined with the observation that no other student faced similar difficulties indicates that practical and tutorial materials were generally effective at conveying the basics of OpenGL syntax and semantics. The best approach to helping students such as Michael is probably to ensure that learning materials are available and structured in a coherent manner which allows students to access solutions to common problems and tasks.

Two inconsistencies in the OpenGL API, the inconsistency between screen and window coordinates and the inconsistent use of units (degrees versus radians) in OpenGL and C++, were also identified as issues for several students. In most cases such problem segments did not exceed the 10-Change cut-off value and were hence not analysed, but even such little problems will add up to a lot of wasted time over many instances. More importantly, several instances of larger segments related to



problems with screen-window or degree-radian inconsistency were also found. In these segments, the inconsistency often occurs alongside other problems like in Thomas's implementation of Avatar movement in segment [THOMAS\\_A3.SP.2.2."Avatar Movement" \[36\]](#) or implementation of parent-child rotation ([IDA\\_A1.SP.2."Rotate Icon" \[41\]](#), [IDA\\_A1.SP.3."Object rotation" \[23\]](#), [THOMAS\\_A3.SP.2.2."Avatar Movement" \[36\]](#), [THOMAS\\_A3.SP.2.1."Angle Conversion" \[12\]](#)) and makes those problems much more difficult to solve. Furthermore, it leads to student uncertainty; for example, Ida ([IDA\\_A1.SP.3."Object rotation" \[23\]](#)) switches between degrees and radians incorrectly because she had previously encountered problems related to this inconsistency, thereby introducing a further error into the problem.

This shows that educators should seek to avoid inconsistencies. In cases where they cannot be avoided (the OpenGL API being an example) it would be beneficial either to provide students with additional scaffolding (such as utility classes resolving the inconsistency) or to explicitly discuss the problems in class and practical problems and to ensure that learning materials contain instructions on common problems and pitfalls.

Given these approaches, syntax and semantic errors should not present a major hurdle to most students who properly utilise available learning materials and engage in practical exercises.

#### **9.7.7.1.2.7 Lighting**

One of the Assignment 3 tasks was to implement simple OpenGL lighting (without use of the OpenGL Shader language). The details were left up to the student, and the task was worth only a small number of marks reflecting its status as a minor task in the assignment. Students had also implemented simple lighting in a previous practical exercise, and the use of that implementation would have been sufficient to score full marks for the lighting task.

Thomas, John and Christopher implemented lighting with few difficulties. The segment [THOMAS\\_A3.LIGHTING.1."Simple Lighting" \[13\]](#) involves Thomas's experimentation with lighting settings. John and Christopher both required fewer than ten Changes to implement simple lighting.

Ida and Michael experienced more significant problems while implementing lighting in their assignment projects.

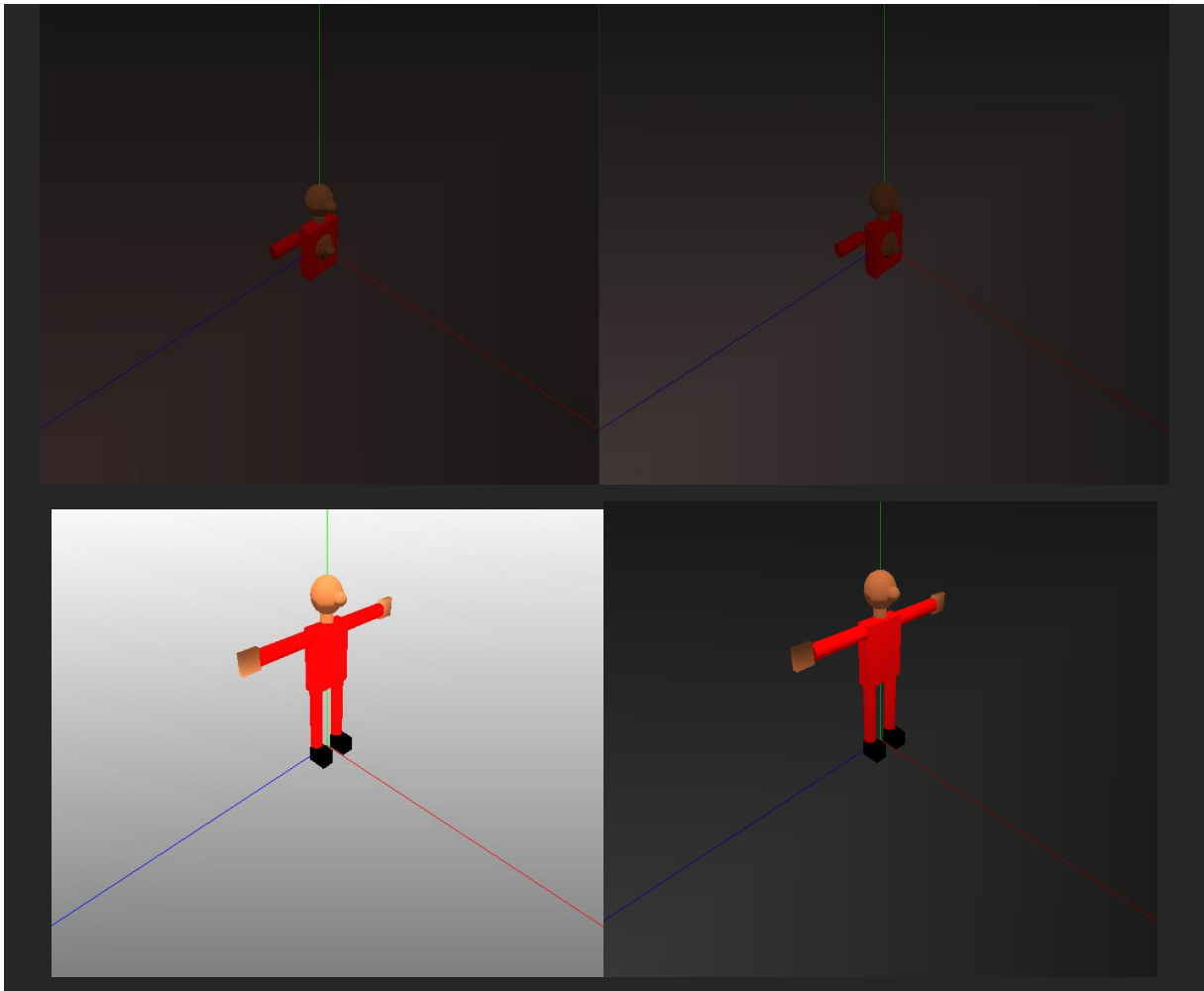
In Ida's case, in the initial implementation of lighting in segment [IDA\\_A3.LIGHTING.1."Lighting" \[26\]](#) the student correctly activates lighting in the state machine but fails to actually add a light to the scene with the `glLight(...)` command, meaning that the scene is unlit (black) when she turns on lighting. She spends most of the 26 Changes experimenting with the state machine commands for activating and deactivating lighting and materials before correctly adding the `glLight(...)` call in the

final Change of the segment, producing satisfactory lighting. The student returns to the lighting task in segment [IDA\\_A3.LIGHTING.2."Lighting 2" \[21\]](#), experimenting with the different types of light, light attenuation and 'shininess'.

The most significant issues concerning lighting occur in Michael's assignment implementation. The student added lighting in (8) as shown in Figure 375, probably copying the code from the internet or a practical exercise. The implementation produced working lighting as shown in Figure 376 top-left, but the scene was darkly lit due to the large attenuation value used by the student. The student then spent 60 Changes ([MICHAEL\\_A3.LIGHTING.1."Lighting Attenuation" \[60\]](#)) experimenting with the light's position as well as with pipeline lighting commands in order to make it brighter. However, moving the light, as can be seen in the top-right, only made the lighting even darker.

```
82 | lightingSetup() {
83 |     glEnable( GL_LIGHTING);
84 |     glEnable( GL_COLOR_MATERIAL);
85 |     glEnable( GL_LIGHT0);
86 |     glShadeModel(GL_SMOOTH);
87 |     glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
88 |
89 |     //This is the light color. Note that values of > 1.0 DO make a difference
90 |     //as opposed to glColor calls which are capped at 1.0
91 |     float lightColor[] = { 0.5, 0.5, 0.5 };
92 |     glLightfv(GL_LIGHT0, GL_AMBIENT, lightColor);
93 |     glLightfv(GL_LIGHT0, GL_DIFFUSE, lightColor);
94 |
95 |     //Light position.
96 |     float xPosition = 0;
97 |     float yPosition = 4.0;
98 |     float zPosition = 0;
99 |
100 |     float lightPosition[] = { xPosition, yPosition, zPosition, 1.0f };
101 |     glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
102 |
103 |     glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 0.5);
```

Figure 375: The source code of the initial lighting implementation



**Figure 376: Lighting with large attenuation at (112) and (111) top-left and top-right, negative attenuation at (221) bottom-left and proper attenuation at (220) bottom-right**

The student then suspended the unsuccessful work on lighting before returning to the implementation at (210) in segment [MICHAEL\\_A3.LIGHTING.2."Final Lighting" \[19\]](#). The first Change involves the modification of the attenuation value, suggesting he may have consulted a tutor or the internet to find possible problems with his lighting implementation. He first implements a negative attenuation as shown in the bottom-left, making the light grow brighter as it emanates from the light source, before correctly utilising a small attenuation value to produce the 'good' lighting shown in the bottom-right. All in all, the student required 79 Changes or around 13% of all Changes produced for that assignment for the implementation of simple lighting (which was not worth many marks). The problem was clearly one of identifying the underlying problem; the student did not have an appropriate debugging technique, leading him to tweak light position values and experiment with different OpenGL lighting pipeline commands unrelated to the source of the problem.

### **Discussion**

Three students implement very simple lighting based on a practical task and face no significant issues with their implementation.

However, Ida and Michael both encounter significant problems with their relatively simple lighting implementations. In Ida's case, this is largely due to issues with the OpenGL pipeline, since she fails to add a light and then spends many Changes activating and deactivating different OpenGL states.

Michael utilises a large attenuation value, causing his light source's light to quickly be eliminated, and he spends a significant percentage of his third assignment debugging this problem.

In both cases, it would be useful to teach students techniques to better visualize light sources and the mathematical effect of different commands and states on the 'light' at any given point. For example, students could be given access to a library method or assignment skeleton feature allowing them to see the precise value of light at any given point, and the mathematical equations used to produce that value.

More widely, problems encountered during the implementation were an example of the type of problem caused by students' black-box understanding of OpenGL. In such circumstances, students should be given the tools or techniques to gain insight into the inner workings of the 'black box' to allow them to correctly identify the underlying source of the problem.

## 9.7.8 Analysis of Segment Features

### 9.7.8.1 Analysis of Segment Features Introduction

The previous section presented results of the primarily qualitative analysis of Segment contents. This form of analysis makes good use of the richness and depth of source-code level data. This section will complement the analysis of Segment contents with a mainly quantitative analysis a Segment feature, the difference in average time spent on Segment Changes belonging to different classification categories (*Segment-Change Average Time*).

The time data for Segments' Changes can be automatically calculated by the SCORE Analyser and output to Excel spreadsheet files. The data was statistically evaluated using the R statistics package.

Section 9.7.8.2 provides a detailed description of how data analysed in this section were produced. Section 9.7.8.3 delves into the relationship between 'Segment-Change Average Time' and Category and the likely factors underlying observed differences. Section 9.7.8.4 tests a hypothesis based on findings of the analysis of 'Segment-Change Average Time' through a detailed examination of several Segments involving the creation of animations, focussing on periods of low Time per Change.

Descriptive analysis of individual Change data as well as Segment data was also conducted. Since the hypotheses raised by this analysis were not followed up, results are located in the appendix. Appendix Section 9.7.8.1 lists results of the quantitative analysis of Change time data, whereas Appendix Section 9.7.8.7 presents descriptive statistics on Segment time data.

### 9.7.8.2 Preparation of Data for Quantitative Analysis

This section will discuss the generation of data for the analysis presented in this chapter. Data is generated by the SCORE Analyser. The SCORE Analyser produces Excel spreadsheets containing data for both individual Changes as well as data for Segments. Segment data is derived by calculating average and total values for Change metrics such as time or number of modifications (only time data is analysed).

The complete set of Segments does not necessarily (or usually) include all Changes that are part of a Project History, only those that form sets containing ten or more related Changes since only such sets are used to produce Segments. Changes that are not part of a Segment are not included in any analysis of Segment metrics. These Changes are included in the analysis of individual Change metrics presented in Appendix Section 9.7.8.1.

The mechanism used for generation of this data is described in technical detail in Section 6.4.1.2. As is discussed in that section, the calculation of Change time involves pitfalls relating to student work patterns, since there is no way to be sure whether a student was working on the project at any given time between two Changes. The heuristic algorithm designed to circumvent these issues is described in Section 6.4.1.2. As the calculation of time data involves heuristics the time metric is an estimate and not a precise measurement.

Sections 9.7.8.7 and 9.7.8.3 investigate the relationship between different Segment classification categories. In the Segment classification scheme used to categorise Segments relating to spatial programming are captured in the same 'Spatial' category. In the following quantitative analysis 'Spatial' Segments were split into two sub-categories, 'Transform Spatial' and 'Non-Transform Spatial'. The 'Transform Spatial' category includes the Segments belonging to the 'Order of Transformations' and 'Three-dimensional transformations' category whereas the 'Non-Transform Spatial' category includes Segments belonging to all other 'Spatial' sub-categories (most of them involving two-dimensional spatial programming) as described in Section 6.3.4. 'Transform Spatial' and 'Non-Transform Spatial' Segments were analysed separately because of an intuition that two-dimensional spatial problems may present a different challenge level and involve different problem-solving approaches than three-dimensional spatial problems. Classification categories which include

less than ten Segments are excluded from the ANOVA analysis presented in Section 9.7.8.3.2 due to the low sample size; the excluded categories are 'GL', 'Lighting' and 'Animation'.

### 9.7.8.3 Analysis of Segment Average Time per Change by Classification Category

#### 9.7.8.3.1 Descriptive Statistics for Segment Average Time per Change by Classification Category

Descriptive statistics for Average Time for different classification categories are shown in Table 91. They are also shown as boxplots across the different categories in Figure 377 and as a means plot in Figure 378. Pipeline and Spatial Segments have the shortest Average Time with 57.55 and 60.61 seconds respectively, with 'General Programming' and 'TransformationSpatial' Segments having means close to the global mean at 64.77 and 69.58 seconds respectively. 'Event-Driven' Segments have a high average 'Average Time' of 80.82 seconds. Segments of type 'View' have the highest average 'Average Time' of 92.28 seconds, 25.01 seconds above the global mean. This means that assuming that time spent between Changes involves the student thinking about the problem being solved, 'View' Segments and (to a lesser extent) 'Event-Driven' Segments seem to involve the most cognitive effort applied between Changes, while 'Pipeline' Segments and 'Spatial' Segments seem to frequently be solved more often using trial-and-error approaches with shorter thinking breaks.

**Table 91: Descriptive Statistics for Segment Average Time per Change broken down by Category**

	N	mean	diff. mean	Sd	Median	Range	Skew	kurtosis	se
Event-Driven	22	80.82	13.55	39.55	79.06	181.29	1.64	3.27	8.43
GeneralProg	42	64.77	-2.5	32.59	61.29	129.36	0.46	-0.75	5.03
Pipeline	10	57.55	-9.72	14.07	56.94	49.31	-0.24	-0.78	4.45
Spatial	59	60.61	-6.66	32.72	50.85	152.99	1.34	1.63	4.26
TransSpatial	20	69.58	2.31	30.14	74.07	110.69	0.41	-0.83	6.74
View	10	92.28	25.01	34.71	80.72	94.21	0.27	-1.77	10.98

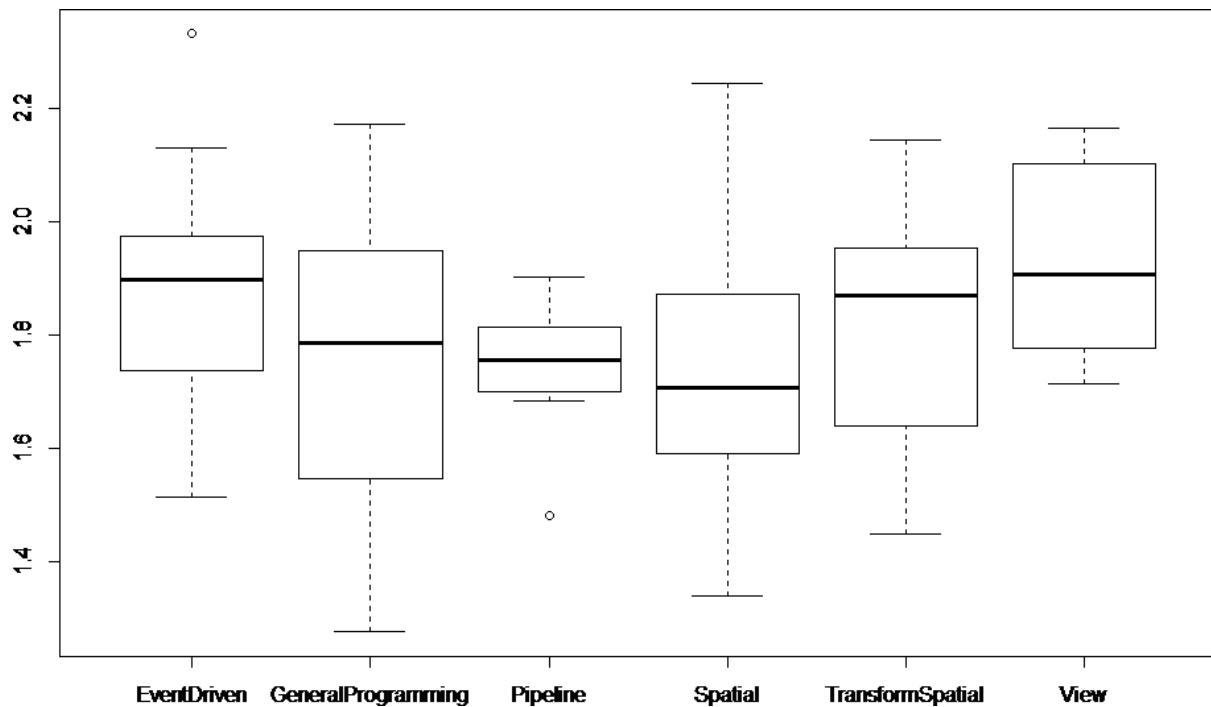


Figure 377: BoxPlot for Segment Log-Average Time per Change broken down by Category

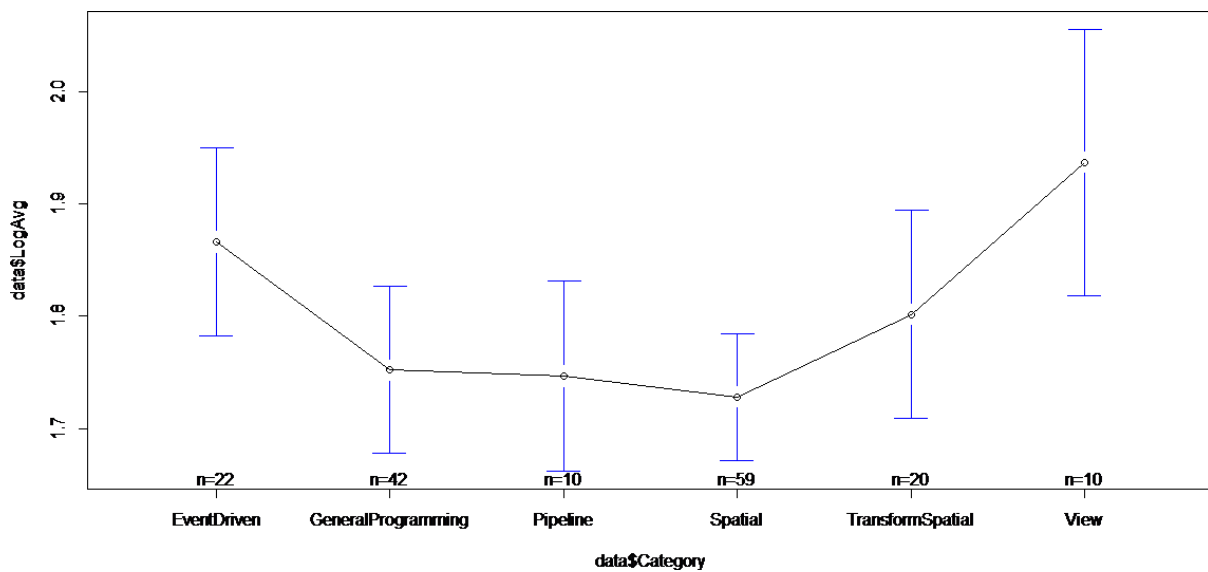


Figure 378: LogAvgTime ~ Category Means Plot

While there seems to be a fairly large difference in means, standard deviation is also quite high for most categories. This may suggest that the classification categories are not precise enough or that problems of the same classification category may be solved using different problem-solving strategies, perhaps based on the difficulty of the problem or the student's preferred strategy or a combination of such factors.

To discover whether the difference in mean Average Times between Segments of different classification categories is significant, Analysis of Variance comparing the means of Average Times between categories was conducted. Since analysis of Segment Average Time per Change was shown to be logarithmically distributed (the full analysis is presented in the appendix, Section 9.7.8.7.1) a logarithmic transformation was applied to the data since normal distribution is a prerequisite for the application of the ANOVA test. The transformed data shown in Table 92 was used for the Analysis of Variance presented in the next section.

**Table 92: Descriptive Statistics for Segment Log-Average Time per Change broken down by Category**

	N	Mean	diff. mean	Sd	median	Min	Max	Skew	Kurtosis	se
Event-Driven	22	1.87	0.09	0.19	1.9	1.51	2.33	0.32	-0.25	0.04
GeneralProg	42	1.75	-0.03	0.24	1.79	1.28	2.17	-0.26	-1.09	0.04
Pipeline	10	1.75	-0.03	0.12	1.76	1.48	1.9	-0.81	-0.01	0.04
Spatial	59	1.73	-0.05	0.22	1.71	1.34	2.24	0.25	-0.58	0.03
TransSpatial	20	1.8	0.02	0.2	1.87	1.45	2.14	-0.14	-1.37	0.04
View	10	1.94	0.16	0.17	1.91	1.71	2.16	0.06	-1.8	0.05

### 9.7.8.3.2 Analysis of Variance of Segment Average Time per Change

Analysis of Variance between Segment classification categories yields a p-value of 0.018 as shown in Table 95. Residuals are presented in Table 93, whereas t and p-values for all categories are presented in Table 94. Since the p-value is < 0.05, the null hypothesis that there is no difference in Average Time means between different classification categories is rejected. The adjusted  $r^2 = 0.0532$ , meaning the effect is rather small and much of the variance in Average Times is not explained by Segment classification category. This indicates that there are other factors involved in causing differences in Average Time. These factors should be identified via detailed analysis of Segments grouped according to Average Time in order detect whether similarities exist between Segments of similar Average Time that can explain more of the variance. Some of these factors were identified during the detailed analysis of Segment contents described as issues in Section 6.6.6.

**Table 93: ANOVA Residuals**

Residuals:				
Min	1Q	Median	3Q	Max
-0.47598	-0.14114	0.00192	0.14597	0.5144



**Table 94: ANOVA t and p-values**

	Estimate	Std. Error	t-value	Pr(> t )
(Intercept)	1.86631	0.04464	41.804	> 2E-16
General Programming	-0.11412	0.05511	-2.071	0.04002
Pipeline	-0.11958	0.07986	-1.497	0.1363
Non-Transform Spatial	-0.13812	0.05231	-2.64	0.00911
TransformSpatial	-0.06479	0.0647	-1.001	0.31814
View	0.07064	0.07986	0.885	0.37777

Residual standard error: 0.2094 on 157 degrees of freedom  
Multiple R-squared: 0.08247  
Adjusted R-squared: 0.05325  
F-statistic: 2.822 on 5 and 157 DF  
p-value: 0.01806

**Table 95: ANOVA r-squared and p-value**

While the correlation is weak it is nevertheless statistically significant which means that there is a difference in the means of Average Time based on Segment classification category.

To determine for which category 'Segment-Change Average Time' means differ significantly from other categories a Tukey's Range Test (aka Tukey's Honest Significant Difference test) was performed on the data. Results are presented in Table 96. The results show that the mean of View Segments is significantly different to (greater than) that of Spatial Segments (at  $p \leq 0.05$ ). While the p-value of 0.13 for between 'View' and 'General Programming' is not significant, it is fairly small and hence given that ANOVA has already shown that Average Time varies by Segment classification category it is likely that 'View' Segments on average have a higher mean Average Time than 'General Programming' Segments. The only other near-significant difference ( $p=0.09$ ) is between 'Event-Driven' Segments (second-largest mean Average Time) and 'Non-Transform Spatial' Segments (smallest mean Average Time); again, while not reaching significance given that there is a significant difference between means based on ANOVA the small p-value is at least indicative that Event-Driven Segments have a higher Average Time on average than Spatial Segments.

**Table 96: Tukey's Range Test (aka Tukey's Honest Significance Difference)**

	Event-Driven	GeneralProg	Pipeline	Spatial	TransSpatial
General Programming	0.31	-	1.00	0.99	0.95
Pipeline	0.67	1.00	-	1.00	0.98
Spatial	0.09	0.99	1.00	-	0.75
Non-Transform Spatial	0.92	0.95	0.98	0.75	-
View	0.95	0.13	0.33	0.05	0.55

In summary it seems that View Segments (and probably Event-Driven Segments) involve a more deep-thinking solution approach, while 'Non-Transform Spatial' Segments (and probably 'General Programming' Segments) involve a somewhat more trial-and-error approach, with 'Pipeline' and 'TransformSpatial' Segments involving more balanced approaches.

This leaves open the question as to why 'View' Segments have a significantly lower mean than 'Non-Transform Spatial' Segments while 'Transform Spatial' Segments do not, given that both involve three-dimensional spatial programming. To determine whether students apply a more deep-thinking problem-solving approach to the solving of 'View' problems compared to 'Transform Spatial' problems the next section will present a qualitative analysis of several 'Transform Spatial' and 'View' Segments. The ability to apply detailed analysis when initial quantitative analysis fails to provide a full picture or fails to yield statistical significance is one of the strengths of the analysis method presented in this thesis.

#### **9.7.8.4 Detailed Investigation of differences in Time between Changes for View and Transformation problems**

##### **9.7.8.4.1 Detailed Investigation Analysis Method**

Analysis of Variance of Average Time did not provide a clear picture on whether 'View' Segments involve a different problem-solving approach than 'Transform Spatial' Segments, this section presents an investigation of Time spent on Changes in 'View' and 'TransformSpatial' Segments utilising a Detailed Analysis approach, evaluating Segments falling into these categories at the source code level.

The analysis will focus on the frequency of 'Long' and 'Short' Changes in 'View' Segments compared to 'Transform Spatial' Segments. Qualitative analysis of these Changes as well as quantitative analysis of their prevalence will be presented.

Based on the lower mean Average Time of 'TransformSpatial' Segments, it is expected that the 'TransformSpatial' Segments will include many more 'Short' Segments indicating that students are more prone to utilising a trial-and-error approach when working on 'TransformSpatial' problems compared to 'View' problems

Since evaluation of all Segments falling into the 'TransformSpatial' or 'View' category would be very time-consuming, one 'View' and one 'TransformSpatial' Segment are analysed per student. The Segment involving the first serious attempt at implementation of a 'View' and the first Segment implementing an 'Animation' were selected for analysis.

One approach to classifying Changes as 'Long' or 'Short' one approach would have been to use the global mean or quartiles for Time spent per Change and then classifying Changes falling into the upper/lower extremes as Long/Short. However given the significant differences in mean and median between students such an approach would have seen very large differences in Changes identified as 'Short' or 'Long' between students. Instead quartiles of Change time were calculated for each student and Changes smaller or equal to the 0.25 Quartile were classified as 'Short' whereas Changes larger or equal to the 0.75 quartile were classified as 'Long'. This means a 'Long' Change is one of the longest 25% of Changes of that assignment, and a 'Short' Change is one of the shortest 25%, with other Changes falling into the middle 50%. In an 'average' Segment 25% of Changes would be 'Long' and 25% would be 'Short'. The 0.25, 0.5 and 0.75 quartiles for all students are shown in Table 97.

**Table 97: Quartiles for Assignment 3**

Quartile/ Student	0.25	0.5	0.75
Christopher	17	34	60
John	35	63.5	129.25
Michael	16.25	27	52.75
Thomas	21	38	67
Ida	32	56	114

For each student's third assignment a quantitative comparison of the rate of occurrence of Short/Long Changes in 'View' versus 'TransformSpatial' Segments will be presented in the following section complemented by a qualitative analysis of Short and Long Changes in one 'View' and one 'TransformSpatial' Segment. For brevity's sake only a summary of the qualitative analysis is

presented; the full analysis accompanied by excerpts of modification summaries showing the source code associated with Long and Short modifications can be found in the appendix, Section 9.7.8.8.

This in-depth examination will serve to test the hypothesis that ‘View’ problems involve deeper spatial thinking and visualization than ‘TransformSpatial’ problems such as implementation of Animations or Avatar assembly.

#### 9.7.8.4.2 Quantitative Analysis of View and Animation Segments

The percentage of Changes that fall into the ‘Short’ and ‘Long’ quantiles for Animation and View Segments is shown in Table 98. Table 99 shows the ratio of Long/Short Changes between the analysed Animation and View Segments.

**Table 98: Percentage of Short and Long Changes for the analysed Animation (left) and View (right) Segments**

Anim	Short	Long	View	Short	Long
Christopher	0.21	0.08	Christopher	0.29	0.19
John	0.34	0.22	John	0.10	0.33
Michael	0.24	0.22	Michael	0.04	0.32
Thomas	0.12	0.33	Thomas	0.05	0.33
Ida	0.12	0.21	Ida	0.12	0.41

**Table 99: Ratio of Short and Long Changes of Anim / View**

	AnimShort/ ViewShort	AnimLong/ ViewLong
Christopher	0.72	0.43
John	3.62	0.67
Michael	6.69	0.70
Thomas	2.55	1.00
Ida	1.03	0.52
	2.92	0.66

As Table 98 shows, both ‘View’ and ‘TransformSpatial’ Segments include fewer than expected ‘Short’ Changes for 8/10 analysed Segments. The exceptions are John’s Animation implementation which involved significant experimentation with transformations after the student encountered Gimbal Lock, and Christopher’s incomplete View implementation. The low number of ‘Short’ Changes for both View and Animation problems in most assignments suggests that students tend to engage in

deeper thinking while solving three-dimensional spatial problems than when solving other problems and that their problem-solving approach is not primarily trial-and-error for either type of Segment.

Examination of the ratio of 'Long'/'Short' Changes between the two types of problem presented in Table 99 shows that for all students except Christopher Animation Segments involve proportionally more 'Short' Changes than 'View' Segments, with an average ratio of 2.92 times more 'Short' Changes occurring in Animation than in View Segments. In addition, for all students except Thomas there are proportionally more Long Changes in 'View' than in 'TransformSpatial' Segments, with 'View' Segments on average containing 1.51 times more 'Long' Changes than 'TransformSpatial' Segments.

These results support the hypothesis investigated in this section that students utilise a more deep-thinking spatial visualization approach when working on View problems than when working on Animation problems. However as the numbers also show, students generally produce fewer Short Changes for both View and Animation problems than would be expected, which means that the many axis errors students make not a result of students simply trying different axes in a pure trial-and-error approach.

#### ***9.7.8.4.3 Qualitative Analysis of View and Animation Segments***

Qualitative analysis of data relating to the nature of Long/Short Changes in Animation and View Segments provides further insight into the difference between programming approaches for Animation and View problems.

##### **9.7.8.4.3.1 Christopher**

Christopher appears to utilise OpenGL as a visual aid for both the examined View and Animation problem. While he appears to engage in deeper spatial thinking while working on the View problem as measured by the number of Large Changes, he is also the only student who produces more Short than Long Changes while implementing the View. When examining the text of these modifications it seems the student is trying to implement the View by using his application as a visualization aid, trying to manipulate the View to gain an understanding of how it works. When that approach does not yield the required insight the student gives up and does not implement the View properly. The failure of OpenGL as a visualization tool combined with his inability or unwillingness to expend additional effort on generating a mental spatial understanding of Viewing appears to lead to his not reaching the learning goals associated with Viewing and hence not completing the assignment task. A more detailed description is given in Section 9.7.8.8.1 of the appendix.

##### **9.7.8.4.3.2 John**

John appears to use a deep-thinking approach for implementation of the View. He implements the View in a relatively small number of Changes but of those Changes many are 'Long' Changes. This indicates that he seeks to develop a spatial understanding which is then implemented rather than developing this understanding during the implementation.

During implementation of the animation John initially produces few Long Changes and many Short Changes, indicating that he is utilising a different strategy, externalising the visualization process via his application as much as possible. He then encounters Gimbal Lock, a difficult spatial problem. He initially produces many Short Changes, again using his program as a visualization tool to attempt to develop an understanding of the underlying problem. When that does not succeed he produces several Long Changes. In two Long Changes parts of the Animation are removed to study the effect of different transformations. In two other Long Changes, the student attempts to augment the visualization capabilities of his program by adding grid lines to different limbs and adding statements to print coordinates to standard output. This approach is complemented by continued use of his program as a visualization aid making small and 'Short' Changes to transformation axes and observing the effect. This approach appears to be of limited utility as the process continues for a considerable time until the student realises through trial and error that the effect can be prevented by limiting the rotation about the x-axis. This appears to be an example of the student's problem-solving strategies becoming exhausted, leaving the student with no other choice but to go through a lengthy experimental process. It seems unlikely that this process ultimately resulted in a real understanding of the underlying problem. A more detailed description is given in Section 9.7.8.8.2 of the appendix.

#### **9.7.8.4.3.3 Michael**

Michael produces no real 'Short' Changes and several 'Long' Changes during implementation of the View. This could mean either that the student feels more comfortable mentally visualizing the View or that the student doesn't know how to utilise his program as an external visualization tool to aid in the implementation. The second seems the likely answer, as the student gives up on a correct implementation of Views, apparently unsure as to how to proceed.

When implementing the avatar assembly (since the student did not produce any animations, an assembly Segment was analysed instead) the student produces a mix of Long and Short Changes, indicating that the student is attempting to get a spatial understanding of the problem and then relies on his program to support his spatial reasoning. However, despite working on the assembly problem for many Changes the student never understands how to properly composite transformations. The pattern seems similar to the case of John in which the student attempts to

utilise the OpenGL drawing surface as an external visualization aid to develop a spatial understanding of the compositing transformation concept, but finds insufficient visualization support to aid his spatial reasoning. The student appears to run out of ideas for further problem-solving approaches and gives up on understanding the concept and implementing the related functionality in his assignment. Like with John, this appears to be because of his application's limited utility as external sketchpad. See Appendix Section 9.7.8.8.3 for more detailed descriptions.

#### **9.7.8.4.3.4 Thomas**

Thomas appears to not rely heavily on OpenGL as a visualization tool. He produces very few Short Changes for either View or Animation implementation, yet produces a functional View and Animation quickly and effectively. His hesitation to use OpenGL as a visualization tool during animation implementation may in part be because of the verbosity of his frame-based animation mechanism which requires many modifications to produce a single transformation. The student also produces a comparatively simple animation, which may make such an approach less necessary. The detailed description of his work is contained in the appendix Section 9.7.8.8.4.

#### **9.7.8.4.3.5 Ida**

The student appears not to utilise OpenGL as an external visualization aid during the implementation of either the View or the Animation, instead performing a large number of Long Changes for both.

Ida produces both a very good view and a very good animation and does so without many Short Changes for either implementation. This indicates she is comfortable with visualizing her spatial actions and hence does not need to rely as heavily on her application as an external aid.

While she produces many Long Changes for both View and Animation implementation she produces many more Long Changes for the implementation of the View. This suggests that developing a spatial understanding of the View may have proved more challenging than maintaining a spatial understanding of the animation, or that she found it necessary to develop a full understanding of Viewing before producing the solution, while she was happy with a step-by-step approach while implementing the Animation. A detailed examination is given in the appendix, Section 9.7.8.8.5.

#### **9.7.8.4.3.6 Overview**

In summary, all students except for Thomas showed a deeper-thinking approach for implementing Views than when developing Animations (Thomas used a deep-thinking approach for both). This may be because it is harder to use OpenGL to visualize Viewing than to visualize transformations of objects in Animations. This may be the reason some students like Michael give up on views quickly while working considerably longer (though ultimately also unsuccessfully) on transformation-related spatial problems.

Problems like Animation implementation which encourage use of OpenGL as an external visualization aid may help support a student's spatial visualization ability. However, it may also encourage students to develop a shallower spatial understanding which they complement with the external visualization tool (their program). For example, John appeared to utilise a shallow approach initially, producing many Short Changes during the early phase of his Animation implementation. This in turn may limit the development of a student's spatial ability. Furthermore, reliance on OpenGL as an external visualization tool may lead to a breakdown in problem-solving when the student's program offers insufficient visualization support to allow the student to understand the underlying concept, as could be observed with John (Animation), Michael (Animation and View) and Christopher (View).

#### 9.7.8.5 Analysis of Segment Features Discussion

This section explored the differences in students' problem solving approaches to Segments through an examination of differences in Average Time per Change between Segments falling into different classification categories. Quantitative analysis of the data indicated that 'View' and 'Event-Driven' Segments involved high 'Segment-Change Average Time', with 'Pipeline' and 'Non-Transform Spatial' Segments involving low 'Segment-Change Average Time'. 'General Programming' and 'Transform-Spatial' Segments fell in the middle.

Analysis of Variance was performed to test whether the variance of Average Change Time per Segment significantly differed based on the Segment's classification category. The ANOVA procedure did detect a significant difference based on classification category, but a follow-up Tukey's Range Test showed that the only significant difference occurred between View Segments and non-transform Spatial Segments, with View Segments having significantly higher Average Times per Segment than non-transform Spatial Segments. This indicates that View Segments involve an unusually high Average Time per Change, whereas Transform Spatial Segments do not despite both being similar kinds of problems involving three-dimensional transformation of the ModelView matrix.

To investigate whether there is an actual difference between View and Transform-Spatial Segments, a detailed investigation comparing the first View and the first Animation Segment for each student was performed. Results (presented in Section 9.7.8.4) show that the View Segments included far more 'long' Changes falling into the largest 25% of Changes and far fewer 'short' Changes falling into the smallest 25% of Changes than the (Transform-Spatial) Animation Segments. This shows that students engage in more in-depth planning and visualization activity when working on the View tasks than on the Animation tasks.



Results of the qualitative analysis of the ‘View’ and ‘TransformSpatial’ Segments suggest that this is because it is simpler to utilise OpenGL as an external visualization aid for problems involving the transformation of objects rather than the transformation of the camera.

Research in the field of the use of Visualizations for learning (Höffler, 2010) found that students with weak spatial ability benefit from dynamic visualizations which allow them to externally manipulate the visualization, while students with strong spatial skills benefit more from static visualizations which they manipulate mentally. This raises the question of to what degree use of an external visualization aid is desirable in Computer Graphics Education. On the one hand, students may be forced to engage in deep spatial thinking when denied the ability to utilise external visualization methods which may result in the development of spatial ability. On the other hand, if students are unable to visualize a spatial problem sufficiently to understand the underlying concept they may give up and not solve the problem or learn the underlying concept. In that case external visualization support enabling them to complete the task and learn the concept would be very beneficial to support their learning. It may be that good scaffolding for weaker students may turn into a liability for stronger students, though it may be that students self-select the degree to which they require use of OpenGL as a visualization aid, as Ida and Thomas apparently did judging by the relatively low number of Short Changes produced by both.

A useful approach to scaffolding Computer Graphics assignments in general may be to set early tasks to involve three-dimensional spatial tasks that are easy to visualize externally using OpenGL such as simple animations. This could then be followed to problems that are harder to visualize externally such as view problems. Final tasks would be the hardest to visualize, involve the simultaneous transformation of the camera, the avatar and scene objects in relation to one another.

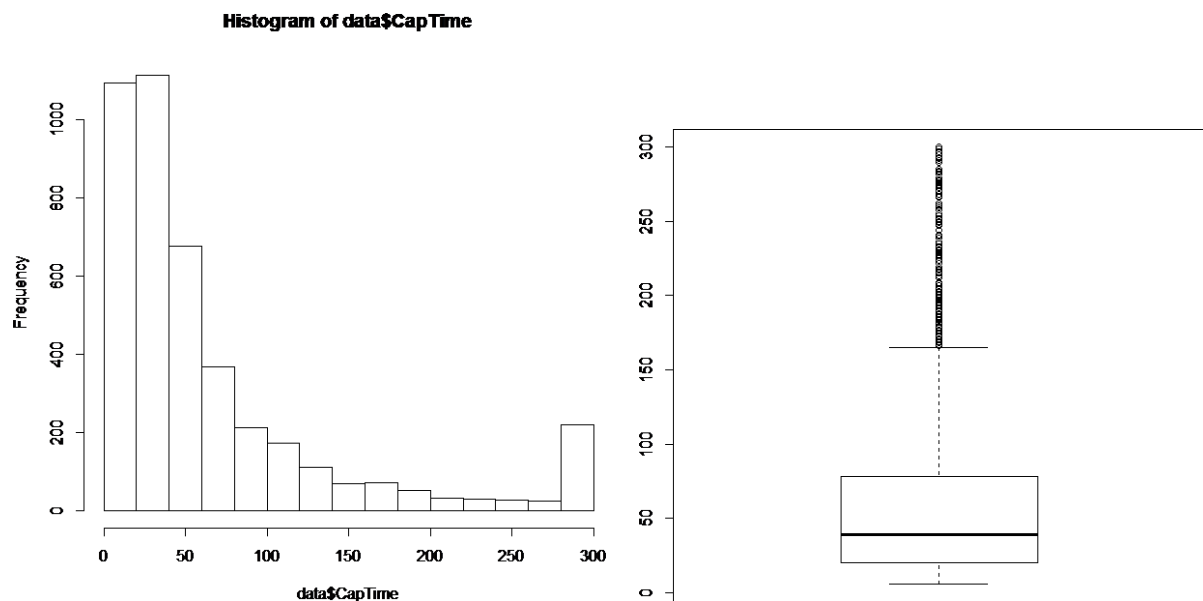
Future work should focus on other features of Segments such as differences in the amount of modifications of different types that occur based on the Segment classification category. For example, do Segments of a certain category include more mutations than other category Segments? The research should also focus on the role of ‘Long’ Changes to explore in detail what role such Segments play in the problem-solving process.

### **9.7.8.6 Quantitative Analysis of Change Data**

#### **9.7.8.6.1 Overall**

Table 100 presents descriptive statistics for Time spent per Change (capped in the fashion described in 9.7.8.2). Figure 379 shows a histogram and a boxplot for this data. Mean time per Change is 66.44 seconds. The median is substantially lower at 39 seconds. The reason for this can be seen in the

histogram, with an unusually high number of Changes at the maximum value of 300 seconds. Standard deviation is very high; at 73.26 seconds, it is higher than the mean.



**Figure 379: Histogram and BoxPlot for Time Spent per Change**

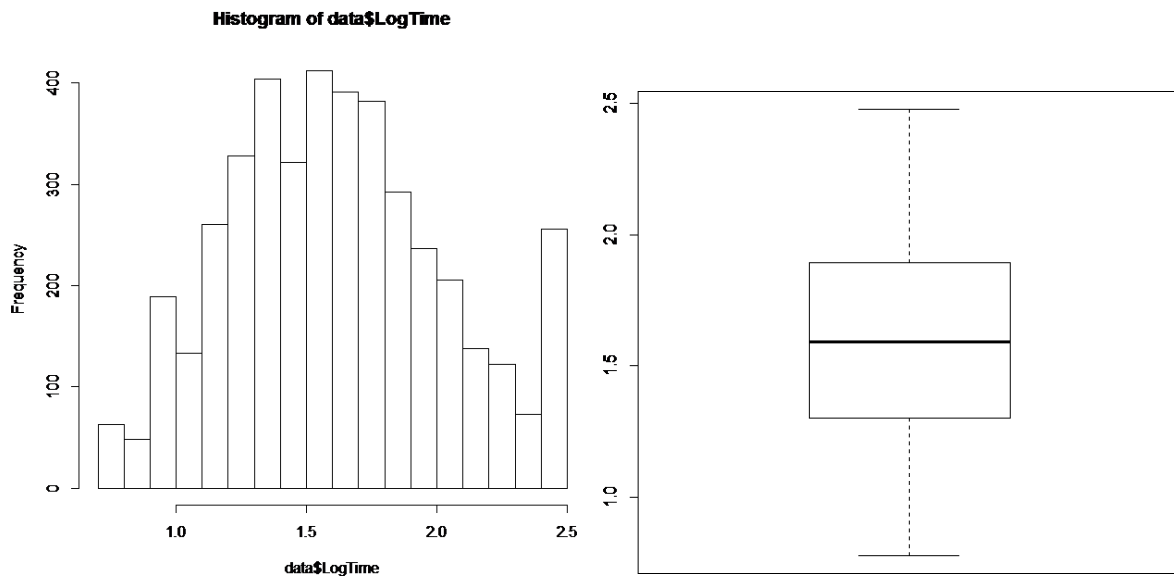
**Table 100: Descriptive Statistics for Time Spent per Change**

N	mean	Sd	median	trimmed	Mad	min	max	range	skew	Kurtosis	Se
4255	66.44	73.23	39	49.9	34.1	6	300	294	2.02	3.43	1.12

The positive skew of 2.02 indicates that the distribution has a long right tail and that most values lie on the left-hand side of the distribution, which the histogram confirms. The kurtosis of 3.43 suggests that the distribution is far more ‘peaked’ than a normal distribution.

Based on the histogram, the distribution seems to be distributed logarithmically rather than normally, with the exception of the smaller-than-expected number of Changes falling into the lowest basket and the larger-than-expected number of Changes falling into the largest basket.

Applying a logarithmic transformation to the Time metric produces the data shown in Table 101. Skew and kurtosis are low, indicating that the data is now approximately normally distributed. The box plot (Figure 380, right) indicates almost perfect normal distribution. The histogram (Figure 380, left) also shows the data to be largely normally distributed, with the exception of a peak in large values (the largest basket) maintained from the non-transformed data. While the data is still not perfectly normally distributed, the distribution post-logarithmic-transformation is normally distributed enough for the application of tests such as ANOVA which are robust against moderate violations of their prerequisite conditions.



**Figure 380: Histogram and BoxPlot for the Log of Time Spent per Change**

**Table 101: Descriptive Statistics for Log Time Spent per Change**

N	mean	sd	Median	trimmed	mad	min	max	range	Skew	kurtosis	se
4255	1.62	0.42	1.59	1.6	0.43	0.78	2.48	1.7	0.25	-0.55	0.01

The large number of large Time value outliers is interesting. It may in part be an artefact of the capping process (since there would otherwise be several more baskets on the right tail) which brings up the question of whether future analysis should utilise a capping method which produces a ‘normal’ tail. It may also be interesting to analyse these outliers to see whether they appear (based on the content of associated Changes) to be outliers involving significant planning/thinking work (preceding the start of a new problem-solving approach) or whether they appear to be randomly distributed. This would shed some light on whether the heuristics used to produce Time data were successful in producing a metric measuring a real process or attribute of the student programming process.

#### 9.7.8.6.2 Per-Student

The total time spent by students on the assignment is shown in Table 102. For the first assignment John, Ida and Thomas who all submit good assignments invest the second, third and fourth-most time respectively. Interestingly, students Michael and Christopher who submit assignments falling short of the core requirements invest the most and the least time respectively.

**Table 102: Total Time spent on A1 and A3**

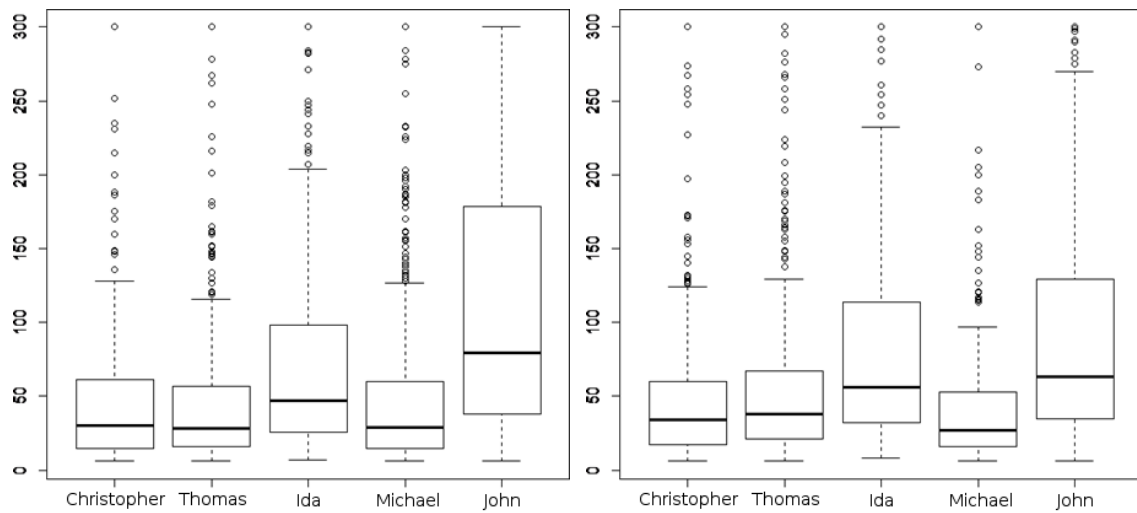
Assignment 1			Assignment 3		
Name	TotalTime (seconds)	TotalTime (hours)	Name	TotalTime (seconds)	TotalTime (hours)
Ida	96085.76	26.69	Ida	67575.69	18.77
John	74725.74	20.76	John	75436.9	20.95
Michael	125567.2	34.88	Michael	29411.76	8.17
Thomas	58851.12	16.35	Thomas	47886.14	13.3
Christopher	42699.09	11.86	Christopher	56190.25	15.61

**Table 103: Quantiles for Average Time per student for A1, A3 and both assignments combined**

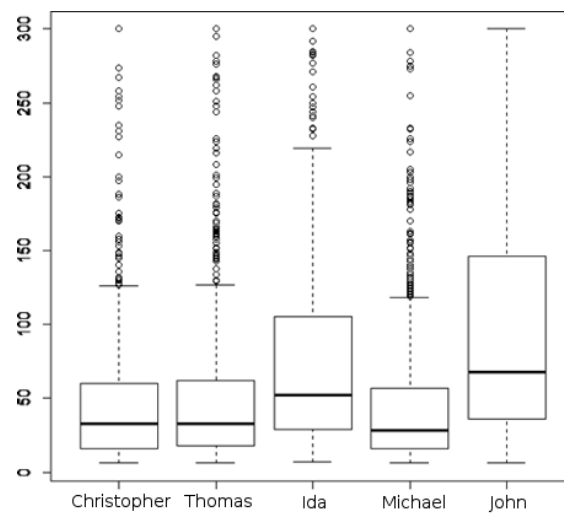
A1	0.25	0.5	0.75	A3	0.25	0.5	0.75	A1+A3	0.25	0.5	0.75
Christopher	14.75	30	61.25		17	34	60		16	33	60
John	38	79.5	178.25		35	63.5	129.25		36	68	146
Michael	15.25	29	60		16.25	27	52.75		16	28	57
Thomas	16	28.5	57		21	38	67		18	33	62
Ida	26	47	98		32	56	114		29	52	105.5

For the third assignment, John and Ida invest the most and second-most time respectively and produce the best results. Christopher fails to implement core tasks despite investing the third-most time, whereas Thomas implements all major tasks despite investing less time. Michael invests much less time than other students (less than half as much as John) and produces the poorest result.

Table 103 shows quantiles for Average Time per Change for all students in A1, A3 and for both assignments combined. The data is also resented using boxplots in Figure 381 and Figure 382. As the data shows, Ida and John invest a relatively high time per Change on average, whereas Thomas, Christopher and Michael invest a relatively low time per Change. Ida and John both perform well in both assignments and Christopher and Michael both perform poorly, with Thomas being an outlier in that he performs well despite investing a low time per Change.

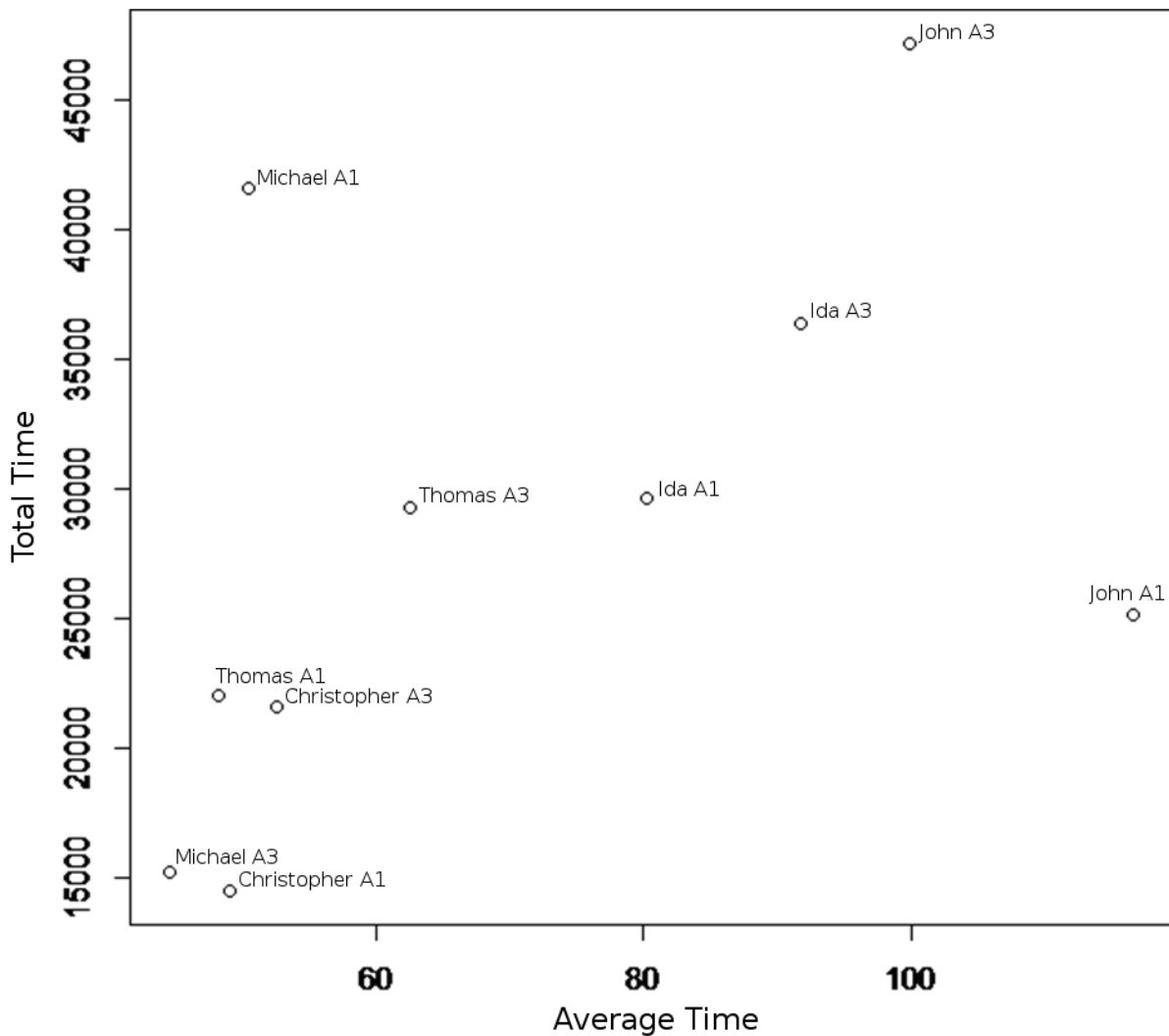


**Figure 381: Box plots for Average Time per Student for A1, A2**



**Figure 382: Box plots for Average Time per Student for A1 and A2 combined**

Figure 383 shows the relationship between Total Time spent on Changes and Average Time spent on Changes (for Changes involved in Segments only, Changes not part of any Segment are excluded). Visual examination shows what appears to be a linear relationship with two large outliers. Michael's A1 implementation has a high Total Time but a low Average Time, whereas John's A1 implementation has a high Average Time but a comparatively low total time.



**Figure 383: Scatterplot showing the relationship between Average Time and Total time spent for individual students**

Previous work (Mierle et al., 2005) has demonstrated Total Time as one of the only useful abstract metrics for predicting student performance. Future research should explore whether average time per Change is also a useful metric, and whether the outliers observed in the apparently linear relationship between Total Time and Average Time represent a meaningful trend. For example, it may be that students who utilise a shallow-thinking approach spending very little time per Change may be more likely to become stuck. However the sample size analysed in this research project is insufficient to make any firm conclusions about the precise relationship between Total Time, Average Time and performance.

#### 9.7.8.7 Quantitative Analysis of Segment Data

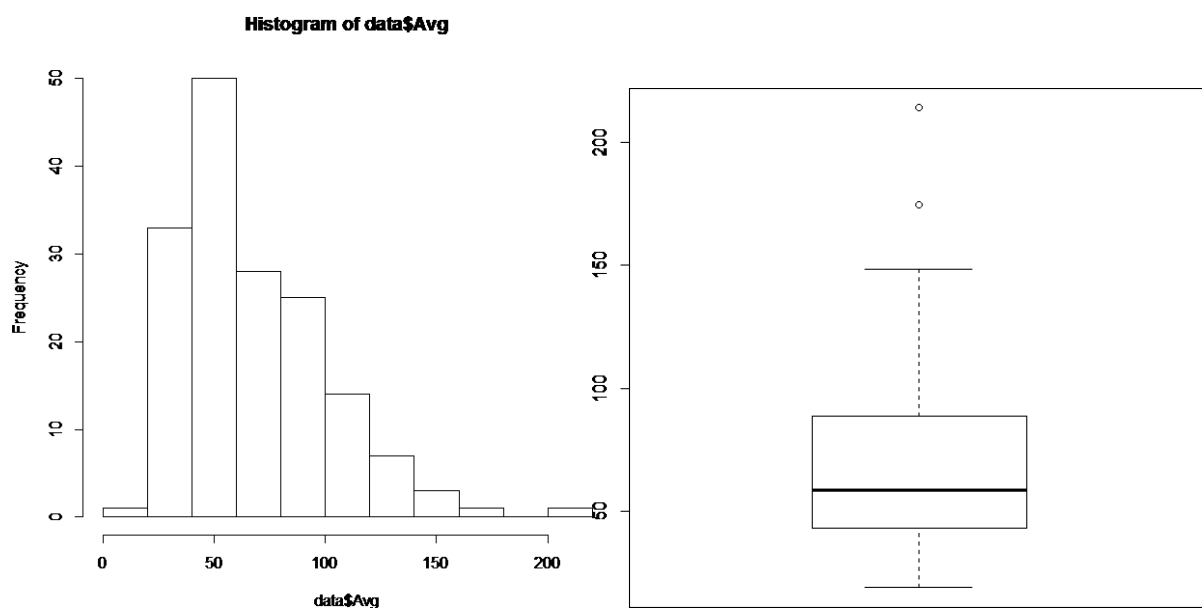
Having explored Time per Change as it relates to individual Changes in the last section, this section will analyse metrics relating to Segments.

### 9.7.8.7.1 Average Time per Segment

Table 104 shows descriptive data for Average Time for segments. Mean Average Time is 67.27 seconds, slightly lower than the 71.6 second average for individual Changes. While standard deviation of segment averages is still high at 33.58 seconds (half the mean), it is much lower than the standard deviation of times for individual Changes. The distribution is skewed to the right (skew 1.11) with a long right tail and most values falling on the left-hand side of the distribution. The kurtosis value of 1.67 indicates that the distribution is more peaked (has a faster drop-off than expected) than a normal distribution. The histogram (Figure 384, left) confirms this. The boxplot also shows a somewhat skewed distribution with two outliers.

**Table 104: Descriptive Statistics for Segment Average Time per Change**

N	Mean	Sd	median	trimmed	Min	Max	Range	skew	kurtosis	Se
163	67.27	33.58	58.6	63.83	18.89	214	195.11	1.11	1.67	2.63

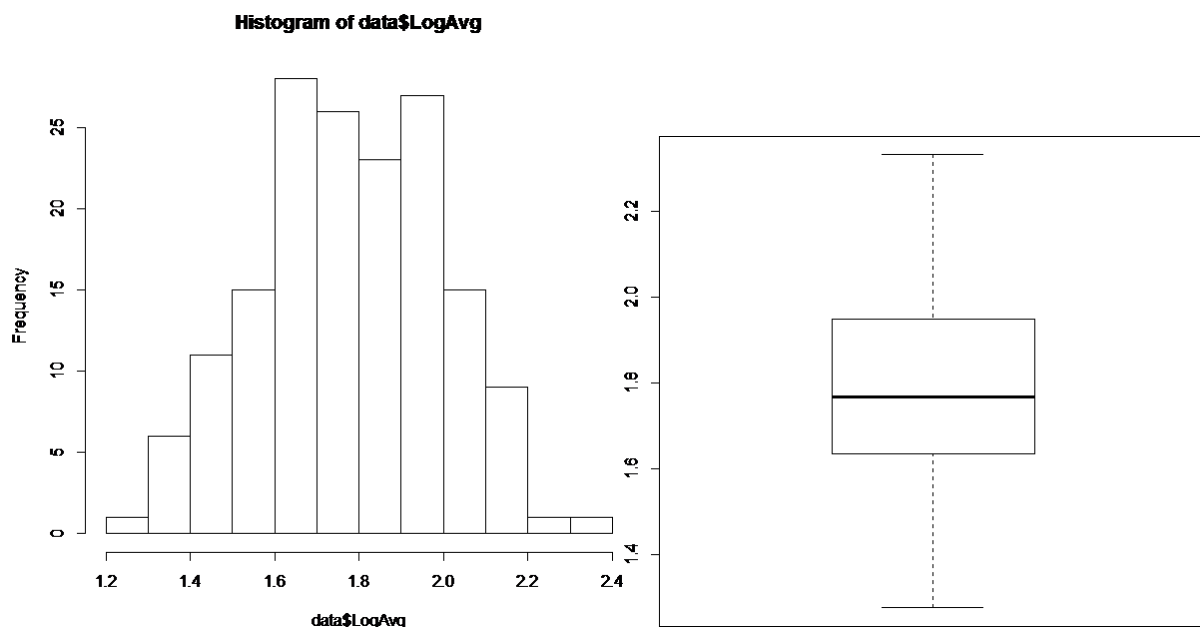


**Figure 384: Histogram and BoxPlot for Segment Average Time per Change**

Since Average Time data will be used for ANOVA, a more normal distribution is desirable. Application of a logarithmic transformation<sup>54</sup> of Average Time data produces the statistics shown in Table 105. The skew value of -0.07 indicates that there is almost no skew at all. Examining the histogram (Figure 385, left) confirms this. There is no unusually large number of Segments with large Average Times. This is in contrast to Time values for individual Changes, which showed an unusually large number of

<sup>54</sup> A logarithmic transformation involves taking the logarithm of each value in the data set

Changes with large values. The kurtosis value of -0.59 suggests a slightly flatter than normal distribution. The log-normal transformed data is considerably nearer to normal distribution than the non-transformed data. Both skew and kurtosis values are close enough to normal to make ANOVA of Average Time feasible.



**Figure 385: Histogram and BoxPlot for Segment Log-Average Time per Change**

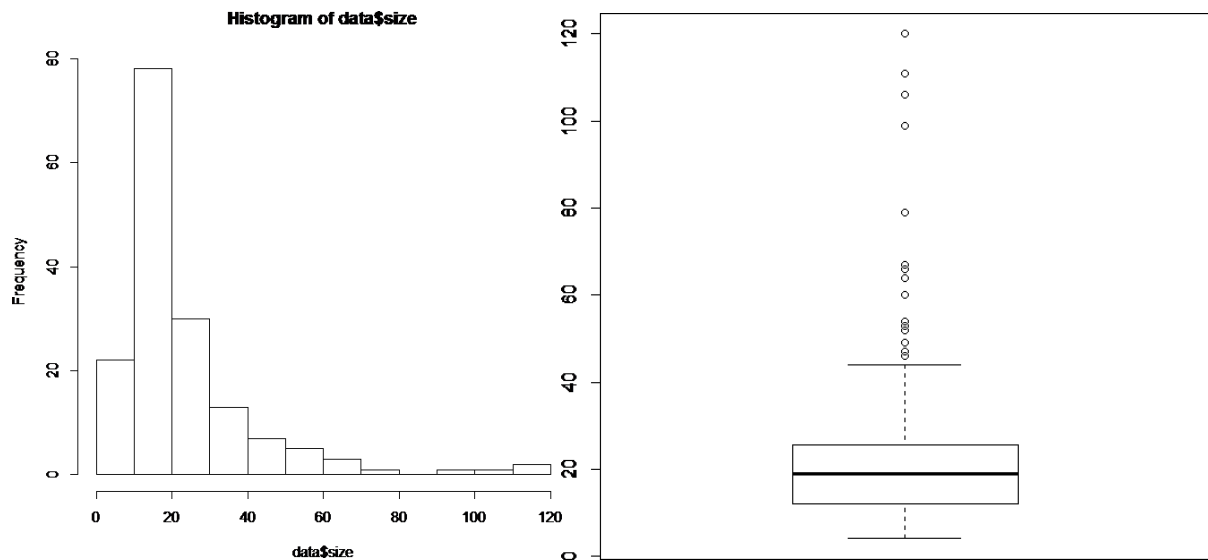
**Table 105: Descriptive Statistics for Segment Log-Average Time per Change**

N	Mean	Sd	Median	Trimmed	mad	Min	Max	range	Skew	Kurtosis	Se
163	1.78	0.22	1.77	1.78	0.22	1.28	2.33	1.05	-0.07	-0.59	0.02

#### 9.7.8.7.2 Size

Segment size statistics are presented in Table 106. Average Segment size is 23.64 Changes, with a large standard deviation of 19.11 Changes. The kurtosis value of 8.5 indicates an extremely steep distribution. The histogram (Figure 386, left) confirms this, with the majority of segments falling into the 10-20 Changes basket and the number of segments in baskets dropping off rapidly towards the right-hand side of the distribution. The boxplot (Figure 386, right) shows that most of the segments on the right-hand side of the distribution are outliers.





**Figure 386: Histogram and BoxPlot for Segment Logarithm-transformed Time per Change**

**Table 106: Descriptive Statistics for Segment Size**

N	mean	sd	Median	trimmed	Mad	min	max	range	skew	kurtosis	se
163	23.64	19.11	19	19.79	10.38	4	120	116	2.67	8.5	1.5

#### 9.7.8.7.3 Correlation between Avg Time and Size

It is possible that Segment Size and Average Time measure the same underlying problem attribute(s) such as problem difficulty. In this case, Size and Average Time should show a strong correlation. In order to test the hypothesis that Size and Average Time measure the same problem attributes, a linear regression analysis testing the correlation between Size and Average Segment Time was performed using R. The data used in this analysis can be found in the electronic appendix.

The linear regression produces a p-value of  $p=0.92$  (see Table 109) which is greater than 0.05, hence supporting the null hypothesis that Size and AvgTime are not linearly correlated. The scatter plot (see Figure 387) of the relationship between AvgTime and Size confirms this result, with the linear regression line being almost horizontal, indicating no linear correlation. The scatter plot also indicates that there is no other (non-linear) correlation between the two metrics.

**Table 107: Residuals for Linear Regression Avg ~ size**

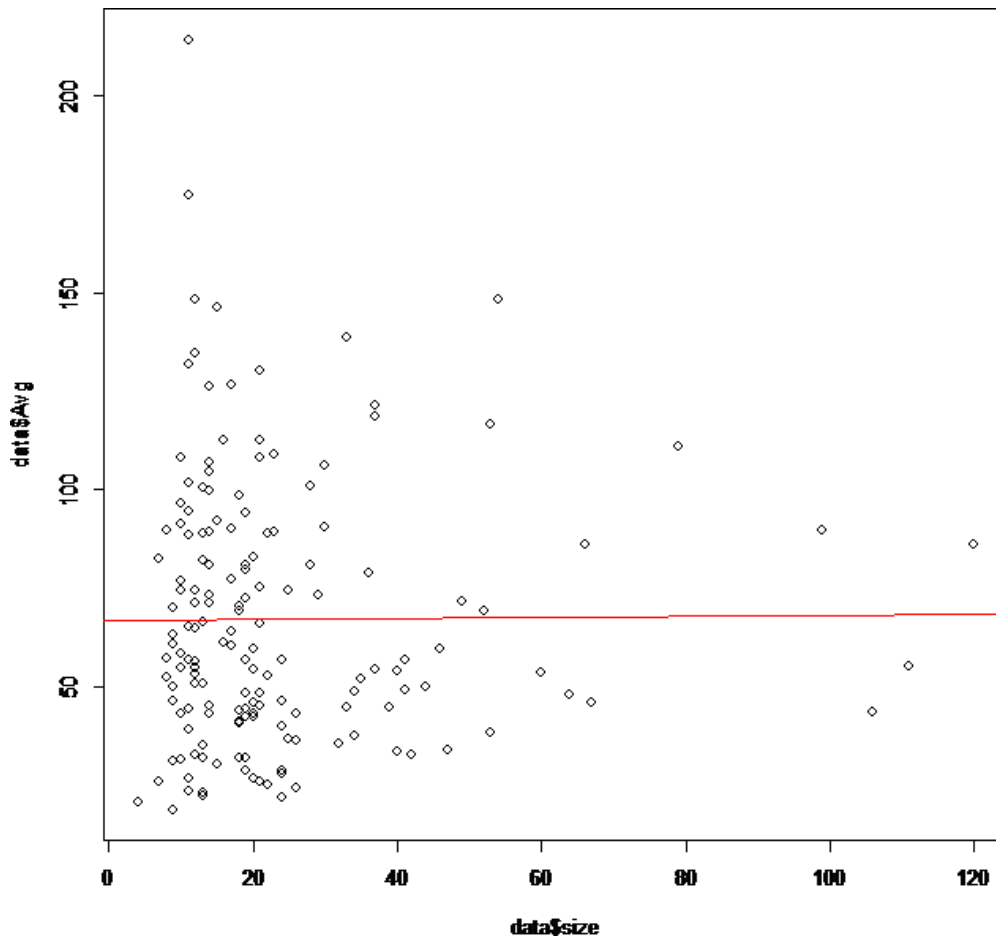
Min	1Q	Median	3Q	Max
-48.184	-24.032	-8.486	21.578	146.9

**Table 108: Coefficients for Linear Regression Avg ~ size**

	Estimate	Std. Error	t value	Pr(> t )	
(Intercept)	66.9544	4.2045	15.925	<2e-16	***
size	0.0132	0.1385	0.095	0.924	

**Table 109: Result of Linear Regression Analysis of AvgTime ~ Size**

Residual standard error: 33.68 on 161 degrees of freedom  
Multiple R-squared: 5.645e-05,  
Adjusted R-squared: -0.006154  
F-statistic: 0.00909 on 1 and 161 DF, p-value: 0.9242



**Figure 387: Scatterplot with Linear Regression Line**

This result leads to the conclusion that Segment Average Time and Size are not correlated, and hence measure different problem attributes.

## 9.7.8.8 Detailed Investigation of differences in Time between Changes for View and Transformation problems

### 9.7.8.8.1 Christopher

#### 9.7.8.8.1.1 Christopher View

Implementation of the student's first view, a first-person View, is contained in Segment [CHRISTOPHER\\_A3.VIEW.1."FP Camera"](#) [47]. As Table 110 shows, the Segment implanting the view contains more short than long Changes. Short Changes include three Changes involving tweaking of view coordinates (737-739), experimenting with view coordinates to develop a spatial understanding of the `gluLookAt` function (743-746) as well as the implementation of spherical view coordinates through trial-and-error in Change 978; this modification follows several other modifications of the same line as can be seen in the line's history (see Figure 388). Short Changes (shown in Figure 389) also include the fixing of a syntax error (981) and addressing of an unrelated OpenGL misunderstanding in 983.

**Table 110: Count of Long and Short Changes**

	Short	Long	Normal	Total
View	9	6	16	31

```
(1973): (975, 977-982, 985-986, 1141-1142 : total = 11)
975, ADDED(0): gluLookAt( fromX,fromY,fromZ, (cos (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromX)+1,fromY,-sin (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromZ, 0,1,0);
977, MUTATED(0): gluLookAt( fromX,fromY,fromZ, (-cos (scene-
>getNodeByName("Torso")->rotation.y*PI/180)*fromX)+1,fromY,-sin (scene-
>getNodeByName("Torso")->rotation.y*PI/180)*fromZ, 0,1,0);
978, MUTATED(0): gluLookAt( fromX,fromY,fromZ, (-cos (scene-
>getNodeByName("Torso")->rotation.y*PI/180)*fromX)+1,fromY,+sin (scene-
>getNodeByName("Torso")->rotation.y*PI/180)*fromZ, 0,1,0);
```

**Figure 388: Line History for the line modified in 978**

```
739, Main.cpp, 00h:00m:03s -----
* (451:) fromY=scene->getNodeByName("man")->translate.y + scene-
>getNodeByName("head")->translate.y+10;
```

```

-> (451:) fromY=scene->getNodeByName("man")->translate.y + scene->
getNodeByName("head")->translate.y+7;

746, Main.cpp, 00h:00m:17s -----
* (469:) gluLookAt( fromX,fromY,fromZ, fromX-1,fromY+1,fromZ, 0,1,0);
-> (469:) gluLookAt( fromX,fromY,fromZ, fromX-4,fromY+1,fromZ, 0,1,0);

978, Main.cpp, 00h:00m:04s -----
* (289:) gluLookAt( fromX,fromY,fromZ, (-cos (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromX)+1,fromY,-sin (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromZ, 0,1,0);
-> (289:) gluLookAt( fromX,fromY,fromZ, (-cos (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromX)+1,fromY,+sin (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromZ, 0,1,0);

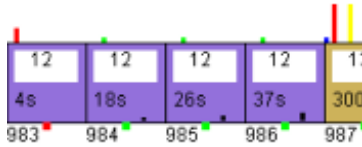
981, Main.cpp, 00h:00m:13s -----
* (289:) gluLookAt( fromX,fromY,fromZ, (-cos (scene->getNodeByName("Torso")-
>rotation.y*PI/180)+1,fromY,sin (scene->getNodeByName("Torso")-
>rotation.y*PI/180), 0,1,0);
-> (289:) gluLookAt( fromX,fromY,fromZ, -cos (scene->getNodeByName("Torso")-
>rotation.y*PI/180)+1,fromY,sin (scene->getNodeByName("Torso")-
>rotation.y*PI/180), 0,1,0);

```

Figure 389: List of Short View Changes

The large Changes (see Figure 391) appear to all involve spatial thinking about how to develop the View. In implementing the View, the student appears to have attempted to develop a spatial understanding of the problem at hand, using experimentation (743-746) to visualize the workings of the `gluLookAt` function and trial-and-error (978) as a short-cut to having to produce an actual understanding of the trigonometric functions and their effects when implementing a view based on spherical coordinates.





Long/Short items: +712, +731, +734, -737-739, -743, -746, +975, -976, -978, +980, -981, +982, -983

**Figure 390: Christopher segment timeline for implementation of a first-person View**

```

712, Main.cpp, 00h:04m:08s -----
* (440:) gluLookAt(fromX+1,fromY,fromZ, fromX,fromY,fromZ, 0,1,0);
-> (440:) gluLookAt( fromX,fromY,fromZ, fromX+1,fromY,fromZ, 0,1,0);
Note:

731, Main.cpp, 05h:21m:10s -----
+ (454:) fromX*=scene->getNodeByName("man")->scale.x + scene->getNodeByName("head")-
>scale.x;
Note:

975, Main.cpp, 00h:01m:16s -----
+ (292:) gluLookAt( fromX,fromY,fromZ, (cos (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromX)+1,fromY,-sin (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromZ, 0,1,0);
- (290:) -sin (scene->getNodeByName("Torso")->rotation.y*PI/180);
- (292:) gluLookAt( fromX,fromY,fromZ, fromX+1,fromY,fromZ, 0,1,0);

980, Main.cpp, 00h:01m:13s -----
* (289:) gluLookAt( fromX,fromY,fromZ, (-cos (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromX)+1,fromY,sin (scene->getNodeByName("Torso")-
>rotation.y*PI/180)*fromZ, 0,1,0);
-> (289:) gluLookAt( fromX,fromY,fromZ, (-cos (scene->getNodeByName("Torso")-
>rotation.y*PI/180)+1,fromY,sin (scene->getNodeByName("Torso")->rotation.y*PI/180),
0,1,0);

982, Main.cpp, 00h:07m:51s -----
+ (289:) cos (scene->getNodeByName("Torso")->rotation.y*PI/180);
+ (290:) -sin (scene->getNodeByName("Torso")->rotation.y*PI/180);
* (285:) fromX/=scene->getNodeByName("man")->scale.x + scene->getNodeByName("head")-
>scale.x;
-> (285:) fromX/=scene->getNodeByName("man")->scale.x + scene->getNodeByName("head")-
>scale.x;
* (289:) gluLookAt( fromX,fromY,fromZ, -cos (scene->getNodeByName("Torso")-
>rotation.y*PI/180)+1,fromY,sin (scene->getNodeByName("Torso")->rotation.y*PI/180),
0,1,0);

```

```
-> (292:) gluLookAt( fromX,fromY,fromZ, fromX+cos (scene->getNodeByName("Torso")->rotation.y*PI/180),fromY,fromZ+sin (scene->getNodeByName("Torso")->rotation.y*PI/180),0,1,0);
```

Figure 391: List of Long View Changes

#### 9.7.8.8.1.2 Christopher Anim



Long/Short items: -950, -955, -958, -960, +966, -968, +974, +1191

Table 111: Segment timeline for the implementation of the pickup animation

The student never managed to correctly implement the avatar assembly and hence could also not properly implement animations. The number of Long and Short Changes is shown in Table 112.

Table 112: Number of Long and Short Changes for the implementation of the pickup animation

	Short	Long	Normal	Total
Anim	5	2	17	24

The animation presented here is the student's simple pick-up animation as contained in Segment [CHRISTOPHER\\_A3.SP.2.1."Pickup Anim" \[24\]](#). One of the Long Changes occurring during the implementation (Figure 392) involves the removal of part of the animation to observe the effect, and the other appears to be a fairly simple modification. Neither involves the construction of a new part of the animation.

```
966, Main.cpp, 00h:01m:09s -----
* (613:) scene->getNodeByName("RightUpperLeg")->rotation.z=60*sin(i*PI/180);
-> (613:) //scene->getNodeByName("RightUpperLeg")->rotation.z=60*sin(i*PI/180);
* (614:) scene->getNodeByName("LeftUpperLeg")->rotation.z=60*sin(i*PI/180);
-> (614:) //scene->getNodeByName("LeftUpperLeg")->rotation.z=60*sin(i*PI/180);
Note:
```

Testing other stuff (remove to see)

1191, Main.cpp, 00h:42m:40s -----

```
* (266:) scene->getNodeByName("RightUpperArm")->rotation.z=60*sin(i*PI/180);  
-> (266:) scene->getNodeByName("RightUpperArm")->rotation.z=-60*sin(i*PI/180);
```

Note:

UpperArm

INCORRECT DIR CORRECTION

**Figure 392: Long Changes for the implementation of the pickup animation**

Short changes (Figure 393) include a tweak at 950, copying of part of an assignment at 955 and trial-and-error implementation of the animation at 958, 960, 968.

950, Main.cpp, 00h:00m:06s -----

```
* (620:) if (i<360){  
-> (620:) if (i<180){
```

Note:

X

955, Main.cpp, 00h:00m:07s -----

```
+ (617:) scene->getNodeByName("Man")->rotation.y=60*sin(i*PI/180);
```

Note:

Transform COPY

958, Main.cpp, 00h:00m:17s -----

```
* (613:) scene->getNodeByName("man")->rotation.z=60*sin(i*PI/180);  
-> (613:) scene->getNodeByName("man")->rotation.z=-60*sin(i*PI/180);
```

Note:

(This line will eventually be changed to leg)

Man

Correct Direction CORRECTION

X

960, Main.cpp, 00h:00m:10s -----

```
* (613:) scene->getNodeByName("man")->rotation.z=-60*sin(i*PI/180);  
-> (613:) scene->getNodeByName("man")->rotation.z=60*cos(i*PI/180);  
* (617:) scene->getNodeByName("man")->rotation.z=-60*sin(i*PI/180);  
-> (617:) scene->getNodeByName("man")->rotation.z=60*cos(i*PI/180);
```

Note:

(This line will eventually be changed to leg)

X

968, Main.cpp, 00h:00m:15s -----

```
* (612:) scene->getNodeByName("man")->rotation.z=60*sin(i*PI/180);  
-> (612:) scene->getNodeByName("man")->rotation.z=-60*sin(i*PI/180);
```

Note:

Body

CORRECT DIRECTION CORRECTION

Z

Figure 393: Short Changes for the implementation of the pickup animation

## 9.7.8.8.2 John

### 9.7.8.8.2.1 John View



Short/Long Changes: +210-211, +219, +222, +225, +688,689, -700-701

Figure 394: Segment timeline for the implementation of John's first-person View

700, Main.cpp, 00h:00m:33s -----

```
* (353:) guy.x+cos(rotAngle), guy.y+8, guy.z-sin(rotAngle),  
-> (353:) guy.x-cos(rotAngle), guy.y+8, guy.z-sin(rotAngle),
```

701, Main.cpp, 00h:00m:18s -----

```
* (353:) guy.x-cos(rotAngle), guy.y+8, guy.z-sin(rotAngle),  
-> (353:) guy.x+cos(rotAngle), guy.y+8, guy.z-sin(rotAngle),  
* (365:) gluLookAt(guy.x+zoom*cos(rotAngle), guy.y+12, guy.z-zoom*sin(rotAngle),  
-> (365:) gluLookAt(guy.x-zoom*cos(rotAngle), guy.y+12, guy.z-zoom*sin(rotAngle),
```



**Figure 395: Short Changes for the implementation of John's first-person View**

John completed all view-related tasks. The Problem Segment shown here is one his implementation of the first-person View. The number of Short and Long Changes is shown in Table 113.

**Table 113: Number of Long and Short Changes for the implementation of John's first-person View**

	Short	Long	Normal	Total
View	2	7	12	21

The first of the Long Changes (Figure 397) involves initial implementation of the View, indicating the student spent some time thinking about the problem spatially. The second Long Change involves what appears to be a small modification (reversing of a sign) , but that modification is in fact part of a large Line History (Figure 396) and may form part of the student's thinking process on how to develop that line. However, while the student makes many modifications to the line, none of the modifications are Short. This suggests that while the student is in fact using OpenGL as a visualization tool, he is doing so while thinking spatially about the problem. He is not using OpenGL as an alternative to developing a spatial model, but as a visual aid. The subsequent three large Changes involve projection angles, suggesting the student is attempting to understand the properties of the View Frustum.

```
(912): (197, 210-213, 216, 218, 220-221, 224, 226, 410, 413, 417, 619, 688 :  
total = 16)  
197, ADDED(X): gluLookAt(guy.x,guy.y,guy.z , 0,0,0, 0,1,0);  
210, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z, guy.x,guy.y+8,guy.z+10, 0,1,0);  
211, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z, guy.x,guy.y+8,guy.z-10, 0,1,0);  
212, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z, guy.x,guy.y+8,guy.z+10, 0,1,0);  
213, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z, guy.x+10,guy.y+8,guy.z, 0,1,0);  
216, MUTATED(O): gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+10,guy.y+8,guy.z, 0,1,0);  
218, MUTATED(O): gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+1000,guy.y+8,guy.z,  
0,1,0);  
220, MUTATED(O): gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+5,guy.y+8,guy.z, 0,1,0);  
221, MUTATED(O): gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+3,guy.y+8,guy.z, 0,1,0);  
224, MUTATED(O): gluLookAt(guy.x+2,guy.y+8,guy.z, guy.x+100,guy.y+8,guy.z, 0,1,0);  
226, MUTATED(O): gluLookAt(guy.x+1,guy.y+8,guy.z, guy.x+3,guy.y+8,guy.z, 0,1,0);  
410, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z, guy.x + tox,guy.y+8,guy.z + toz,  
0,1,0);  
413, MUTATED(O): gluLookAt(guy.x,guy.y+8,guy.z, guy.x + tox,guy.y+8,guy.z - toz,  
0,1,0);
```

```

417, MUTATED(0): gluLookAt(guy.x,guy.y+8,guy.z, guy.x+tox,guy.y+8,guy.z-toz,
0,1,0);
619, MUTATED(0): gluLookAt(guy.x,      guy.y+8, guy.z,
688, MUTATED(0): gluLookAt(guy.x,      guy.y+8, guy.z,

```

Figure 396: Line History for the line modified in (211)

```

210, Main.cpp, 00h:11m:47s -----
- (271:) glRotatef(yrot, 0, 1, 0);
- (272:) glRotatef(xrot, 1, 0, 0);
* (270:) gluLookAt(guy.x,guy.y,guy.z , 0,0,0, 0,1,0);
-> (270:) gluLookAt(guy.x,guy.y+8,guy.z, guy.x,guy.y+8,guy.z+10, 0,1,0);
Note:
Error in projection, angle 180 is TOO LARGE.

211, Main.cpp, 00h:04m:50s -----
* (270:) gluLookAt(guy.x,guy.y+8,guy.z, guy.x,guy.y+8,guy.z+10, 0,1,0);
-> (270:) gluLookAt(guy.x,guy.y+8,guy.z, guy.x,guy.y+8,guy.z-10, 0,1,0);

219, Main.cpp, 00h:02m:11s -----
* (266:) gluPerspective(120, WIDTH/HEIGHT, 0.1, 100.0);
-> (266:) gluPerspective(170, WIDTH/HEIGHT, 0.1, 100.0);

222, Main.cpp, 00h:05m:16s -----
* (266:) gluPerspective(170, WIDTH/HEIGHT, 0.1, 100.0);
-> (266:) gluPerspective(170, WIDTH/HEIGHT, 0.1, 1000.0);

225, Main.cpp, 00h:04m:01s -----
* (266:) gluPerspective(170, WIDTH/HEIGHT, 0.1, 100.0);
-> (266:) gluPerspective(60, WIDTH/HEIGHT, 0.1, 100.0);

```

Figure 397: Long Changes for the implementation of John's first-person View

The only Short Changes involve the student attempting to correct the zoom functionality.

#### 9.7.8.8.2.2 John Animation



Long/Short Changes: +259, -260, -262, +264, -268-270, -274, -276-277, -279, -281, -283, -290, -292, +293, -294, +297, -298, +300-301, -302-303, -305-306, +307, -311, +313-318

**Figure 398: Segment timeline for the implementation of the Pickup animation**

Long Changes implemented during John's work on the first animation are shown in Figure 399, whereas Short Changes are shown in Figure 400. Table 114 summarises the number of Long and Short Changes in the Segment.

264, Main.cpp, 30h:46m:57s -----

```
- (673:) guy.partsRot[PERSON_LLA][X]+=50*(INC/35);
* (672:) guy.partsRot[PERSON_LUA][X]-=50*(INC/100);
-> (672:) guy.partsRot[PERSON_LUA][X]-=INC;
* (675:) if(guy.partsRot[PERSON_LLA][X]>0)
-> (674:) if(guy.partsRot[PERSON_LUA][X]<-90)
```

Note:

Tweaks UpperArm X speed.

Changes condiitonal back from being dependent on lower arm to upper arm.

293, Main.cpp, 00h:02m:09s -----

```
* (676:) if(guy.partsRot[PERSON_LUA][X]<-85)
-> (676:) if(guy.partsRot[PERSON_LUA][X]<-90)
```

Note:

Gimbal Lock

Changing from -85 to -90, this changes the y-transform from 100 to 110, which is less gimbal locked (but it's not locked at the moment anyways)

297, Main.cpp, 00h:02m:50s -----

```
+ (672:) if(guy.partsRot[PERSON_LUA][X]>-90)
* (676:) if(guy.partsRot[PERSON_LUA][X]<-90)
-> (677:) if(guy.partsRot[PERSON_LLA][Y]<-90)
```

Note:

Gimbal Lock

Here changes the conditional for frame 1 to `lla_y < -90`, which leads to total gimbal lock (but not yet, as frame still has a lla y transform)

300, Main.cpp, 00h:05m:30s -----

```
- (672:) if(guy.partsRot[PERSON_LUA][X]>-90)
- (678:) counter++;
- (679:) break;
- (680:) }
- (681:) case 2: {
- (682:) guy.partsRot[PERSON_LUA][Y]-=INC;
- (683:) guy.partsRot[PERSON_LLA][X]-=INC2;
- (684:) //guy.partsRot[PERSON_LLA][Y]+=INC;
- (685:) glutPostRedisplay();
- (686:) if(guy.partsRot[PERSON_LUA][Y]<-30)
```

Note:

Gimbal Lock

Removes the gimbal-locked second frame (but of course the gimbal lock occurs in frame 1, so any new frame will be gimbal-locked)

301, Main.cpp, 00h:19m:51s -----

```
+ (680:) case 2: {
+ (681:) guy.partsRot[PERSON_LLA][X]+=INC;
+ (682:) glutPostRedisplay();
+ (683:) if(guy.partsRot[PERSON_LLA][X] > 0)
+ (684:) counter++;
+ (685:) break;
+ (686:) }
```

Note:

Gimbal Lock

Adds a new second frame with a lla x rotate, gimbal locked of course.

307, Main.cpp, 00h:02m:26s -----

```
- (641:) static const float INC = 0.1;
```

Note:

Probably is thinking the static declare may be influencing something else.

313, Main.cpp, 00h:09m:02s -----

```
+ (517:) glBegin(GL_LINES); {  
+ (518:) glColor3ub(255, 0, 0);  
+ (519:) glVertex3i(0,0,0);  
+ (520:) glVertex3i(1,0,0);  
+ (521:) glColor3ub(0, 255, 0);  
+ (522:) glVertex3i(0,0,0);  
+ (523:) glVertex3i(0,1,0);  
+ (524:) glColor3ub(0, 0, 255);  
+ (525:) glVertex3i(0,0,0);  
+ (526:) glVertex3i(0,0,1);  
+ (527:) } glEnd();
```

Note:

Gimbal Lock

Adds a second set of orientation lines to upper arm.

314, Main.cpp, 00h:28m:17s -----

```
+ (700:) }/*  
+ (707:) */  
- (700:) }  
- (707:) }  
* (454:) glutInitWindowPosition(100, 150);  
-> (454:) //glutInitWindowPosition(100, 150);
```

Note:

Gimbal Lock

Removes second frame from anim.

Also Stops initing window position

315, Main.cpp, 00h:05m:45s -----

Note:

Gimbal Lock

Removes the first frame from anim (by moving /\*)

316, Main.cpp, 00h:03m:47s -----

Note:

### Gimbal Lock

Adds first frame back.

317, Main.cpp, 00h:03m:42s -----

```
+      (693:)   cout   <<   guy.partsRot[PERSON_LUA][X]   <<   '   '   <<
guy.partsRot[PERSON_LUA][Y] << ' ' << guy.partsRot[PERSON_LUA][Z] << endl;
+      (694:)   cout   <<   guy.partsRot[PERSON_LLA][X]   <<   '   '   <<
guy.partsRot[PERSON_LLA][Y] << ' ' << guy.partsRot[PERSON_LLA][Z] << endl;
```

Note:

### Gimbal Lock

Couting the upper and lower arm's rotation values during anim.

318, Main.cpp, 00h:02m:03s -----

```
+ (387:) case '1': {
+      (389:)   cout   <<   guy.partsRot[PERSON_LLA][X]   <<   '   '   <<
guy.partsRot[PERSON_LLA][Y] << ' ' << guy.partsRot[PERSON_LLA][Z] << endl;
+ (390:) break;
+ (391:) }
+ (392:) case '2': {
+      (394:)   cout   <<   guy.partsRot[PERSON_LLA][X]   <<   '   '   <<
guy.partsRot[PERSON_LLA][Y] << ' ' << guy.partsRot[PERSON_LLA][Z] << endl;
+ (395:) break;
+ (396:) }
* (387:) case '1': guy.partsRot[guy.activeLimb][X] +=1; break;
-> (388:) guy.partsRot[guy.activeLimb][X] +=1;
* (388:) case '2': guy.partsRot[guy.activeLimb][X] -=1; break;
-> (393:) guy.partsRot[guy.activeLimb][X] -=1;
```

Note:

### Gimbal Lock

Adds the same output code to key handler when rotation keys are pressed, this will allow him to see the exact value where gimbal lock occurs.

**Figure 399: Long Changes occurring during the implementation of the Pickup animation**

260, Main.cpp, 00h:00m:24s -----

```
* (675:) if(guy.partsRot[PERSON_RUA][Z]>15)
-> (675:) if(guy.partsRot[PERSON_LUA][X]==0)
```

Note:

Modifies the test for going to the next frame to use left upper arm rotation (since right upper arm rotation isn't used anymore.

262, Main.cpp, 00h:00m:27s -----

```
* (675:) if(guy.partsRot[PERSON_LUA][X]==0)
```

```
-> (675:) if(guy.partsRot[PERSON_LUA][X]>0)
```

Note:

Tweaks the conditional for going to next frame.

268, Main.cpp, 00h:00m:20s -----

```
* (673:) guy.partsRot[PERSON_LLA][Z]+=10*INC;
```

```
-> (673:) guy.partsRot[PERSON_LLA][Y]+=10*INC;
```

Note:

LLA Z->Y (Error, Changed in 269)

269, Main.cpp, 00h:00m:23s -----

```
* (673:) guy.partsRot[PERSON_LLA][Y]+=10*INC;
```

```
-> (673:) guy.partsRot[PERSON_LLA][Y]-=INC;
```

Note:

LLA Y -> -Y (Changed in 336)

270, Main.cpp, 00h:00m:31s -----

```
* (673:) guy.partsRot[PERSON_LLA][Y]-=INC;
```

```
-> (673:) guy.partsRot[PERSON_LLA][Y]-=2*INC;
```

Note:

Tweaks lower arm Y

274, Main.cpp, 00h:00m:18s -----

```
* (681:) guy.partsRot[PERSON_LUA][Y]+=INC;
```

```
-> (681:) guy.partsRot[PERSON_LUA][Y]-=INC;
```

Note:

Changes upper-arm +Y to -Y

(Changed in 291)

276, Main.cpp, 00h:00m:26s -----

```
* (682:) guy.partsRot[PERSON_LLA][X]+=INC;
```

```
-> (682:) guy.partsRot[PERSON_LLA][X]-=INC;
```

Note:

Here and the next change is presumably where he realises he's run into a problem (it's gimbal lock, but the student has problems identifying this). He changes from an X to a Z rotation, but due to Gimbal Lock the X-axis and Z-axis overlie each other.

Even though the student knows about gimbal lock, he doesn't identify this as gimbal lock.

Finishes working on it temporarily at 318.

The gimbal lock isn't total yet, because the y-value is at -100 (not -90 which would be total lock) but rotation is already limited.

Changes lower arm +X to -X

Error, changed in 277.

277, Main.cpp, 00h:00m:33s -----

```
* (682:) guy.partsRot[PERSON_LLA][X]==INC;
```

```
-> (682:) guy.partsRot[PERSON_LLA][Z]==INC;
```

Note:

Gimbal Lock

Changes lower arm -X to -Z

Error, changed in 278

279, Main.cpp, 00h:00m:27s -----

```
* (682:) guy.partsRot[PERSON_LLA][Y]==INC;
```

```
-> (682:) guy.partsRot[PERSON_LLA][X]==INC;
```

Note:

Gimbal Lock

Changes lower arm -Y to lower arm -X

Error, changed in 280

281, Main.cpp, 00h:00m:26s -----

```
* (683:) guy.partsRot[PERSON_LLA][Y]+=INC2;
```

```
-> (683:) //guy.partsRot[PERSON_LLA][Y]+=INC2;
```

Note:

Gimbal Lock

Removes the lower arm's y-rotation, again creating gimbal lock.

283, Main.cpp, 00h:00m:34s -----

```
* (682:) //guy.partsRot[PERSON_LLA][X]+=INC;
```

```
-> (682:) /uy.partsRot[PERSON_LLA][X]==INC;
```

Note:

Gimbal Lock

Adds the lower arm's rotation back and changes from +x to -x, still gimbal locked and this won't change it.

GP Syntax error



290, Main.cpp, 00h:00m:28s -----

```
* (682:) guy.partsRot[PERSON_LLA][Z]==INC2;
```

```
-> (682:) guy.partsRot[PERSON_LLA][X]==INC2;
```

Note:

Gimbal Lock

Changes z-transform to x-transform, both of which are gimbal locked.

292, Main.cpp, 00h:00m:30s -----

```
* (682:) guy.partsRot[PERSON_LLA][X]==INC2;
```

```
-> (682:) guy.partsRot[PERSON_LLA][Y]==INC2;
```

Note:

Gimbal Lock

Changes the lower arm's transform from x to y, which resolves the gimbal lock (but no x or z transform, so it's not apparent w.out using keys)

294, Main.cpp, 00h:00m:33s -----

```
* (682:) guy.partsRot[PERSON_LLA][Y]==INC2;
```

```
-> (682:) guy.partsRot[PERSON_LLA][X]==INC2;
```

Note:

Gimbal Lock

Changes the second frame's y-rotate back to x-rotate, leading to semi-gimbal lock.

298, Main.cpp, 00h:00m:25s -----

```
* (683:) guy.partsRot[PERSON_LLA][Y]==INC2;
```

```
-> (683:) //guy.partsRot[PERSON_LLA][Y]==INC2;
```

Note:

Gimbal Lock

Removes the second frame's lla y transform, leading to gimbal lock.

302, Main.cpp, 00h:00m:28s -----

```
* (683:) if(guy.partsRot[PERSON_LLA][X] > 0)
```

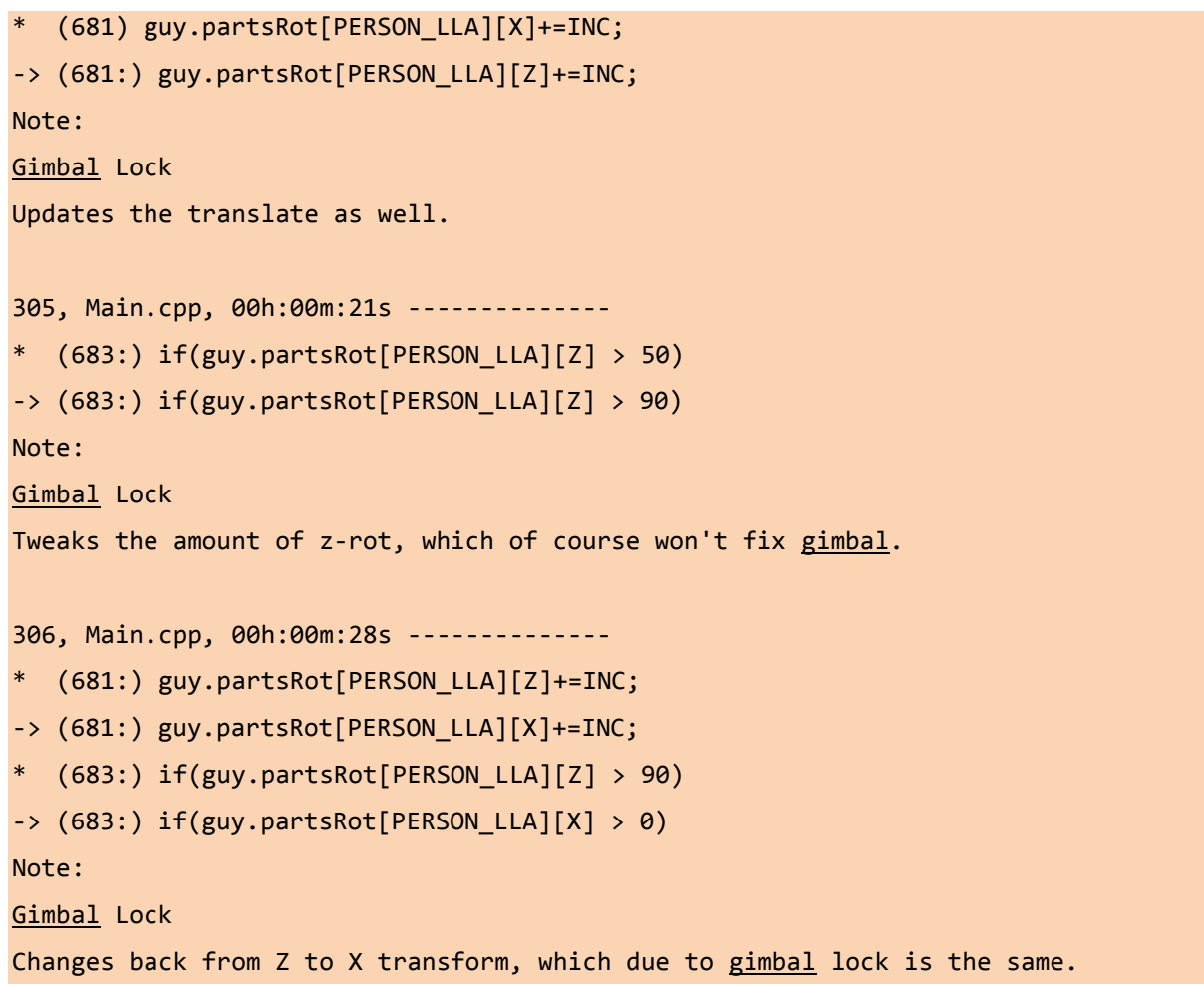
```
-> (683:) if(guy.partsRot[PERSON_LLA][Z] > 0)
```

Note:

Gimbal Lock

Instead of rotating around x, rotates around z (but forgets to update the translate together w. conditional, so infinite), which again shows the gimbal lock.

303, Main.cpp, 00h:00m:29s -----



**Figure 400: Short Changes occurring during the implementation of the Pickup animation**

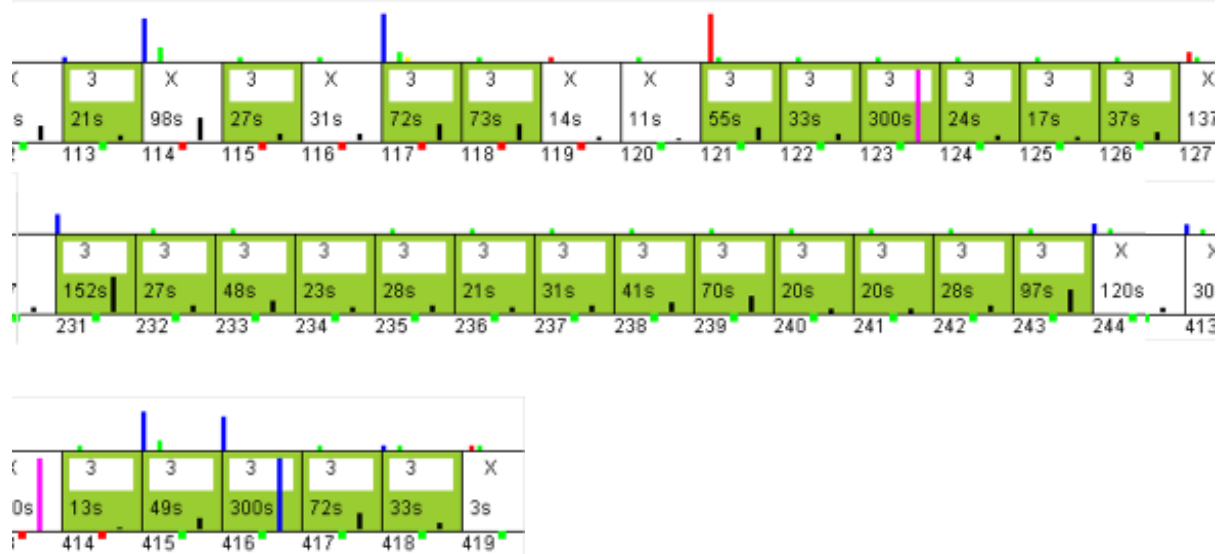
**Table 114: Number of Long and Short Changes occurring during the implementation of the Pickup animation**

	Short	Long	Normal	Total
Anim	20	13	25	58

The student initially produces few Long Changes but several Short Changes to fix spatial problems, suggesting that perhaps he is relying on OpenGL to offload as much of the required spatial visualization as possible onto his program. At Change 276 the student encounters Gimbal Lock. After unsuccessfully attempting to resolve the problem through trial and error with multiple Short Changes (277, 279, 281, 283, 290, 292) the student switches to a reasoning approach, producing Long Changes involving removal of parts of the animation at (300,301), implementation of local coordinate grid line to serve as visual guides at (313) and implementation of cout statements to print out coordinates in (317,318), interspersed with additional Short Changes at (302-303, 305-306). It seems that the student utilises OpenGL as a visualization aid, complementing that approach with more in-depth spatial reasoning when he fails to progress via a trial-and-error approach from Changes (276-301).

### 9.7.8.8.3 Michael

#### 9.7.8.8.3.1 Michael View



Long/Short: +231, +239, +243, +117-118, +121, +123, -414, +416-417

**Figure 401: Segment Timeline for the implementation of Views**

Student Michael only produces a partial implementation of Views. Long Changes are shown in Figure 402, whereas Table 115 summarises the number of Short and Long Changes.

```
117, Main.cpp, 00h:01m:12s -----
+ (256:) if (view1){
+ (257:) glViewport(10, 10, 670, 570);
+ (258:) // sets up projection transformation
+ (259:) glMatrixMode(GL_PROJECTION);
+ (260:) glLoadIdentity();
+ (261:) gluPerspective(60, 800.0/600.0, 0.1, 100.0);
+ (262:) //Viewing Transformation
+ (263:) glMatrixMode(GL_MODELVIEW);
+ (264:) glLoadIdentity();
+ (265:) gluLookAt(at,at,at , 0,0,0, 0,1,0);
+ (266:) }
+ (268:) else if (view2){
+ (269:) glViewport(10, 10, 670, 570);
+ (270:) // sets up projection transformation
+ (271:) glMatrixMode(GL_PROJECTION);
+ (272:) glLoadIdentity();
+ (273:) gluPerspective(60, 800.0/600.0, 0.1, 100.0);
+ (274:) //Viewing Transformation
```

```

+ (275:) glMatrixMode(GL_MODELVIEW);
+ (276:) glLoadIdentity();
+ (278:) }
* (255:) if (view1)gluLookAt(at,at,at , 0,0,0, 0,1,0);
-> (252:) gluLookAt(at,at,at , 0,0,0, 0,1,0);
* (256:) else if (view2)gluLookAt(0,0,at , 0,0,0, 0,1,0);
-> (277:) gluLookAt(0,20,0 , 0,0,0, 0,1,0);

118, Main.cpp, 00h:01m:13s -----
* (252:) gluLookAt(at,at,at , 0,0,0, 0,1,0);
-> (252:) //gluLookAt(at,at,at , 0,0,0, 0,1,0);

123, Main.cpp, 00h:60m:30s -----
* (260:) gluLookAt(5,10,5 , 0,0,0, 0,1,0);
-> (260:) gluLookAt(1,10,1 , 0,0,0, 0,1,0);

231, Main.cpp, 00h:02m:32s -----
+ (270:) else if (view3){
+ (271:) gluLookAt(headx,heady,headz, 0,0,0, 0,1,0);
+ (272:) }

239, Main.cpp, 00h:01m:10s -----
* (271:) gluLookAt(headx-3.5,heady,headz, 1,7,0, 0,1,0);
-> (271:) gluLookAt(headx,heady,headz-3.5, 1,7,0, 0,1,0);

243, Main.cpp, 00h:01m:37s -----
* (271:) gluLookAt(headx-3.5,heady,headz-2.0, 1,7,0, 0,1,0);
-> (271:) gluLookAt(headx-3.5,heady,headz-2.0, headx+3.5,7,headz+3.5,0,1,0);

417, Main.cpp, 00h:01m:12s -----
* (304:) gluLookAt(atx,aty,atz , 0,0,0, 0,1,0);
-> (304:) gluLookAt(atx,aty,atz , 0,0,0+atz, 0,1,0);

```

**Figure 402: Long Changes occurring during the implementation of Views**

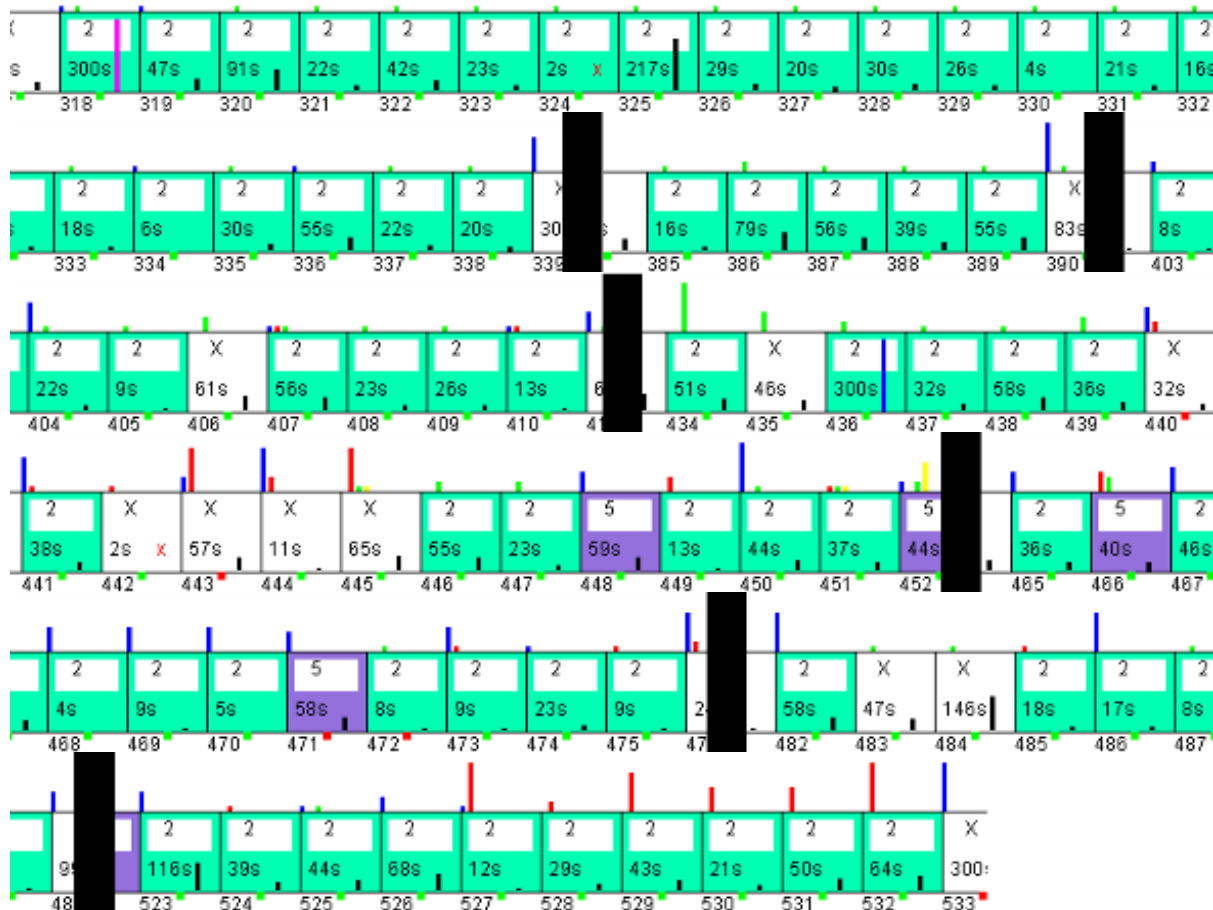
**Table 115: Number of Long and Short Changes in the implementation of Views**

	Short	Long	Normal	Total
View	1	9	18	28

While implementing the Views, the student produces no interesting Short Changes, but several Long Changes. One Long Change involves solving a tangential problem (118 removes stacking of two

Views), but the other Changes involve implementation of the View. The large number of Long Changes and the lack of Short Changes indicates that the student is engaging in deep thinking and visualizing.

#### 9.7.8.8.3.2 Michael Animation



Long/Short Changes: +318, +320, +325, -330, -332, -334, +336, -385, +386-387, +389, -403, -405, +407, -410, +436, +438, +446, -449, -(468-470, 472-473), -475, +482, -487, +523, +526, -527, +532

Figure 403: Segment timeline for the implementation of avatar assembly

The Long Changes in Michael's animation implementation are shown in Figure 404, whereas Figure 405 shows the Short Changes. The total number of Short and Long Changes is shown in Table 116.

```
320, Main.cpp, 00h:01m:31s -----
* (314:) glTranslatef(0,5.9,0);
-> (314:) glTranslatef(0,2,0);
Note:
Makes the second (after rotate) glTranslate call smaller, so now they add up to
roughly the same value as the head's translate was before.
```

Of course, he is still rotating about the wrong point (0, 4, 0) instead of where the head should be.

(Better, but still ERROR)

325, Main.cpp, 00h:03m:37s -----

```
* (310:) glTranslatef(0.0, 4.0, 0.0);
```

```
-> (310:) glTranslatef(0.0, 0.0, 0.0);
```

Note:

Turns the first translate from (0,4,0) to (0,0,0). meanign the head will be rotated about the origin (wronger, ERROR)

386, Main.cpp, 00h:01m:19s -----

```
* (368:) glTranslatef(0.0, 4.9, -3.7);
```

```
-> (368:) //glTranslatef(0.0, 4.9, -3.7);
```

```
* (372:) //glTranslatef(0,0.75,0);
```

```
-> (372:) glTranslatef(0,0.75,0);
```

Note:

Adds the second transform back, but it transforms y instead of z (Error, has a totemistic view of things)

Removes the first transform.

436, Main.cpp, 00h:07m:29s -----

```
* (353:) glTranslatef(0,0.75,0);
```

```
-> (353:) //glTranslatef(0,0.75,0);
```

```
* (354:) //glTranslatef(0,5.9,0);
```

```
-> (354:) glTranslatef(0,5.9,0);
```

Note:

2nd transform from (0,0.75,0) -> (0, 5.9, 0) (ERROR)

Why?? This is really unclear to me. It was \*sorta\* working...

He is trying to implement a body transform in addition to the limb transforms. This should in fact work fine as a stacked rotation, but he doesn't seem to think so...

It may be that here he is trying to make the translation 'independent' as a prelude to removing the movex and movez translate (done in the next step), perhaps as a prelude to adding a translation to the whole body.

(If that's the case it's a gross misunderstanding, since adding a translate would work perfectly well iff it had the rest of the body in the same push/pop block)

523, Main.cpp, 00h:01m:56s -----

```
+ (356:) glRotatef(headx, 1, 0, 0);
```

```
+ (357:) glRotatef(heady, 0, 1, 0);
+ (358:) glRotatef(headz, 0, 0, 1);
+ (359:) glTranslatef(0,0.75,0);
```

Note:

Here tries to combine body transform with limb transform, but still doesn't composite transforms...

The point here is to allow the head to orient while at the same time allowing whole-body orientation.

He just copies across the existing head block to the body, but since the body translate combines the head's pre/post translate, the additional post call translates the head up too far.

**Figure 404: Long Changes occurring during implementation of avatar assembly**

```
330, Main.cpp, 00h:00m:04s -----
```

```
* (314:) glTranslatef(0,2,0);
-> (314:) glTranslatef(0,1,0);
```

Note:

2nd translate (0,2,0) -> (0,1,0)

He is actually moving the head to the origin to examine how it reacts to the transform (for all of the head tweaks below)

```
332, Main.cpp, 00h:00m:16s -----
```

```
* (314:) glTranslatef(0,0.5,0);
-> (314:) glTranslatef(0,0.8,0);
```

Note:

2nd translate (0,0.5,0) -> (0,0.8,0)

```
334, Main.cpp, 00h:00m:06s -----
```

```
+ (315:) glTranslatef(0,5.9,0);
```

Note:

Adds another translate to move head to its proper position just in front of the 2nd translate.

These will add up, of course, and the head again simply rotates about the origin from the top. (ERROR)

```
385, Main.cpp, 00h:00m:16s -----
```

```
* (372:) glTranslatef(0,0.75,0);
-> (372:) //glTranslatef(0,0.75,0);
```

Note:

Removes the palm's second transform, which of course means the palm rotates about its own centre (Error, didn't learn his lesson properly)

405, Main.cpp, 00h:00m:09s -----

\* (424:) glTranslatef(0,0.4,-0.4);

-> (424:) glTranslatef(0,0,0);

Note:

Removes the transform AFTER the foot tranform stack, leaving it at the origin (ERROR)

410, Main.cpp, 00h:00m:13s -----

+ (420:) glTranslatef(0.05, 0.0, 0.45);

Note:

Tweaks

487, Main.cpp, 00h:00m:08s -----

\* (510:) glTranslatef(0.3+movex, 0.0, 0.65+movez);

-> (510:) glTranslatef(0.3+movex, 0.0, -0.65+movez);

Note:

Changes z-sign.

**Figure 405: Short Changes occurring during implementation of avatar assembly Short Changes**

**Table 116: Number of Long and Short Changes occurring during implementation of avatar assembly**

	Short	Long	Normal	Total
Assembly	16	15	36	67

Since the student failed to implement a working Animation mechanism and hence could not implement any proper Animations, the student's avatar assembly (see Segment [MICHAEL\\_A3.SP.1."Initial Assembly" \[47\]](#)) is analysed instead. The student produces a partially working avatar assembly.

A similar number of Long (see Figure 404) and Short (see Figure 405) Changes were produced during the assembly. It appears the student did in fact engage in spatial reasoning during the assembly as can be seen by the Long Changes, many of which like happens at (436) involve implementing different approaches to assembly (using different orders of stacked transformations).

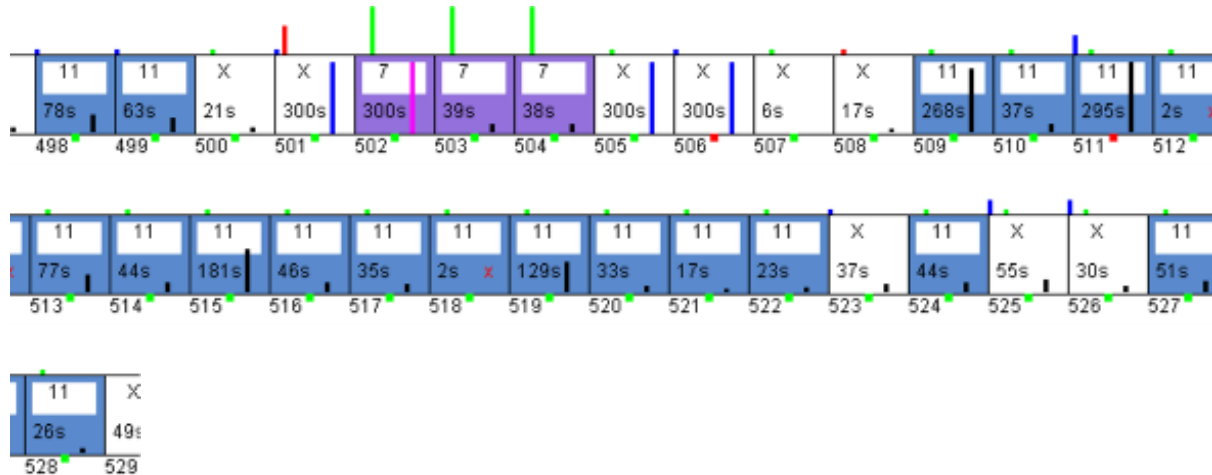
The student also utilised OpenGL as a visualization tool, experimenting with different transformations in the Short Changes as occurs in (385) where a transform is completely removed to



observe the transformation's effect. As with John it seems that reliance on OpenGL for visualization failed to produce the required breakthrough in spatial understanding.

#### 9.7.8.8.4 Thomas

##### 9.7.8.8.4.1 Thomas View



*Long/Short Changes: +498, +502, +509, +511, +513, +515, +519, -521*

**Figure 406: Segment timeline for the implementation of the third-person View**

The student implements a third-person View including all required functionality in segment THOMAS\_A3.VIEW.1."Third-Person Camera" [21]. In the segment the student produces 7 Long (see Figure 407) and only 1 Short Change (see Figure 408). The Long Changes (498, 509) involve initial work on and reasoning about how to implement the View, with Changes (511,513,515,519) involving the development of a spherical, zoomable View. Only a single Change (512) is Short, involving the changing of the direction in which the first-person View is rotated. This suggests that in implementing the View the student produces a mental spatial model of the View in relation to the avatar and the scene and engages in deep spatial thought. Table 117 shows the total number of Long and Short Changes.

```
498, Main.cpp, 00h:01m:18s -----
+ (542:) glTranslatef(currentPosX,0.0,currentPosY);

509, Main.cpp, 00h:04m:28s -----
* (535:) gluLookAt(at,at,at , 0,0,0, 0,1,0);
-> (535:) gluLookAt(currentPosX,at,currentPosY , 0,0,0, 0,1,0);

511, Main.cpp, 00h:04m:55s -----
+ (528:) /*int direction=0;
```

```

+ (529:) float dist=0.0;
+ (530:) float currentPosX=0.0;
+ (531:) float currentPosY=0.0;*/
* (535:) gluLookAt(at,at,at , 0,0,0, 0,1,0);
-> (539:) gluLookAt(currentPosX-cos(direction*0.0174532925 )*at,at,currentPosY-
sin(direction*0.0174532925 )*at , 0,0,0, 0,1,0);

513, Main.cpp, 00h:01m:17s -----
* (539:) gluLookAt(currentPosX-cos(direction*0.0174532925 )*at,at,currentPosY-
sin(direction*0.0174532925 )*at , 0,0,0, 0,1,0);
-> (539:) gluLookAt(at,at,at , currentPosX,0,currentPosY, 0,1,0);

515, Main.cpp, 00h:03m:01s -----
* (539:) gluLookAt(currentPosX-at,at,currentPosY-at , currentPosX,0,currentPosY,
0,1,0);
-> (539:) gluLookAt(currentPosX-sin(direction*0.0174532925)*at,at,currentPosY-
cos(direction*0.0174532925)*at , currentPosX,0,currentPosY, 0,1,0);

519, Main.cpp, 00h:02m:09s -----
* (539:) gluLookAt(currentPosX-sin(-direction*0.0174532925),at,currentPosY-cos(-
direction*0.0174532925) , currentPosX,0,currentPosY, 0,1,0);
-> (539:) gluLookAt(currentPosX-sin(-direction*0.0174532925)*20,at,currentPosY-
cos(-direction*0.0174532925)*20 , currentPosX,0,currentPosY, 0,1,0);

- 498, 509, 511, 513, 515, 519 are all core developments of the view, getting it to look at avatar (498,
509) and then getting an orbital from-behind camera (511, 513, 515, 519)

```

**Figure 407: Long Changes occurring in the implementation of a third-person View**

```

521, Main.cpp, 00h:00m:17s -----
* (539:) gluLookAt(currentPosX-cos(-direction*0.0174532925)*20,at,currentPosY-
sin(-direction*0.0174532925)*20 , currentPosX,0,currentPosY, 0,1,0);
-> (539:) gluLookAt(currentPosX-cos(direction*0.0174532925)*20,at,currentPosY-
sin(direction*0.0174532925)*20 , currentPosX,0,currentPosY, 0,1,0);

```

**Figure 408: Short Changes occurring in the implementation of a third-person View**

**Table 117: Number of Long and Short Changes in the implementation of a third-person View**

	Short	Long	Normal	Total
View	1	7	13	21

The figure consists of three horizontal tracks representing a 1200 bp DNA fragment. The top track shows the full sequence from position 366 to 380. The middle track shows a zoomed-in view of the 381-400 bp region. The bottom track shows a zoomed-in view of the 401-412 bp region. Each track contains a series of boxes representing different features, with labels indicating sequence positions, GC content, and various features like SNPs, indels, and repeats. The tracks are color-coded: purple for GC content, green for SNPs, and red for indels. The bottom track also includes a blue bar representing a specific feature.

**Figure 409: Segment timeline for the implementation of a walk animation**

[illegible]

```
-> (89:) {{ 0, 0, 0},{ 0, 0, 0},{ -60, 0, 0},{ 90, 0, 0},{ 0, 0, 0},{ 0, 0, 0},{ 0, 0, 0},{ 0, 0, 0},{ 0, 0, 20},{ 0, 0, -10},{ 0, 0, 0},{ 0, 0, 0},{ 60, -20, 0},{ 0, 0, 0},{ 0, 0, 0}},//frame7
```

**Figure 410: An example of a Long Change (remaining Changes are not shown as the source code is too verbose)**

In implementing the animation, the student produces many long Changes, indicating that he is utilising deep spatial thinking, starting with the first iteration of the animation at (366). These Changes are not always (or even usually) error free (see 366, 367, 400, 404), but the fact that they are Long Changes indicates that these errors derive from errors in spatial thinking, rather than from a mere trial-and-error process of trying different axes until one works. The student performs only three Small Changes, two involving the reversing of direction for an existing transformation and only one involving an axis modification. Overall this suggests that the animation is the product of a deep spatial reasoning process, with the student preferring mental visualization to utilisation of OpenGL as a visualization aid. This may also have to do with the student's implementation of the animation mechanism which is very verbose and hence requires a large number of Changes to modify a transformation. It may be that the student relies less on OpenGL for visualization because of the effort involved with implementing modifications to transformations. The animation is also fairly simple, which may be why the student found it less necessary to experiment or externally visualize transformations. The total number of Long and Short Changes is shown in Table 118.

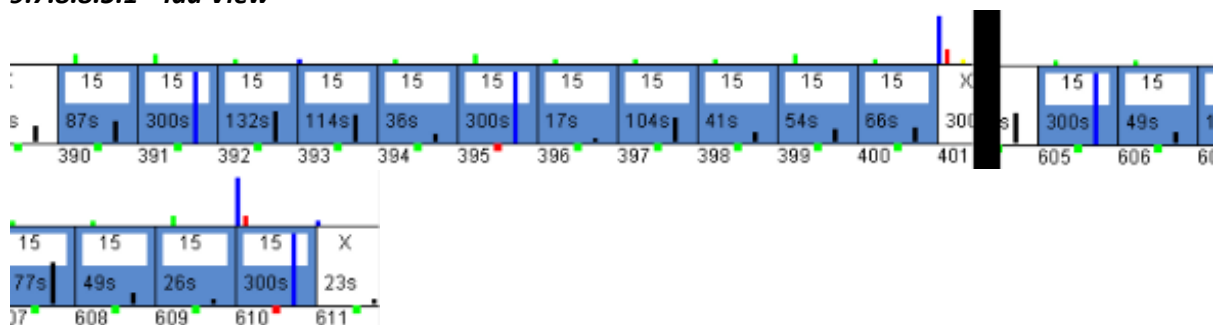
**Table 118: Long and Short Changes occurring during the implementation of a walk animation**

	Short	Long	Normal	Total
Anim	4	11	18	33

#### 9.7.8.8.5

##### Ida

#### 9.7.8.8.5.1 Ida View



*Long/Short Changes: +391-393, +395, -396, +605, +607, -609, +610*

**Figure 411: Segment timeline for the implementation of the first-person View**

Implementation of Views involves 7 Long and 2 Short Changes as shown in Table 119. Long Changes are shown in Figure 412, Short Changes are shown in Figure 413. The large number of Long Changes

suggest that the student is engaging in spatial reasoning while implementing the View, first from 391-395 in implementing a static view and then implementing a spherical view at (605, 607). One of the Short Changes is in fact a correction of a syntax error (and hence irrelevant) while the other involves a tweaking of the camera distance from the avatar.

**Table 119: Number of Long and Short Changes for the implementation of the first-person View**

	Short	Long	Normal	Total
View	2	7	8	17

```

391, Main.cpp, 00h:05m:21s -----
*                                                                    (269:)
gluLookAt(position[xc],position[yc]+7,position[zc],position[xc]+cos(angle),positio
n[yc]+7,position[zc]+sin(angle), 0,1,0);
->                                                                    (269:)
gluLookAt(position[xc],position[yc]+7,position[zc],position[xc]+cos(angle),positio
n[yc]+7,position[zc]-sin(angle), 0,1,0);
* (590:) glTranslatef(position[0],position[1],position[2]);
-> (590:) glTranslatef(position[xc],position[yc],position[zc]);

392, Main.cpp, 00h:02m:12s -----
*                                                                    (269:)
gluLookAt(position[xc],position[yc]+7,position[zc],position[xc]+cos(angle),positio
n[yc]+7,position[zc]-sin(angle), 0,1,0);
->                                                                    (269:)
gluLookAt(position[xc],position[yc],position[zc],position[xc]+cos(angle),position[
yc],position[zc]-sin(angle), 0,1,0);

393, Main.cpp, 00h:01m:54s -----
+ (269:) cout << "x = " << position[xc] << " y = " << position[yc] << " z = " <<
position[zc] << endl;

395, Main.cpp, 00h:09m:36s -----
* (279:) gluPerspective(60, 800.0/600.0, 0.1, 100.0);
-> (279:) glOrtho(-30, 30, -30, 30, 0, 60);
* (283:) gluLookAt(at,at,at,0,0,0, 0,1,0);
-> (283:) gluLookAt(position[x], 10, position[z], 0, 0, 0, 0, 1, 0);

605, Main.cpp, 00h:52m:07s -----

```

```

* (308:) gluLookAt(at,at,at,0,0,0,0,1,0);
-> (308:) gluLookAt(-radius*cos(yrot*3.14/180), 11, radius*sin(yrot*3.14/180), 0,
7, 0,0,1,0);

607, Main.cpp, 00h:02m:57s -----
* (308:) gluLookAt(-radius*cos(yrot*3.14/180), 11, radius*sin(yrot*3.14/180), 0,
0, 0 , 0 , 1 , 0);
-> (308:) gluLookAt(-radius*cos(yrot*3.14/180), 15, radius*sin(yrot*3.14/180), 0,
0, 0 , 0 , 1 , 0);

```

Figure 412: Long Changes occurring during the implementation of the first-person View

```

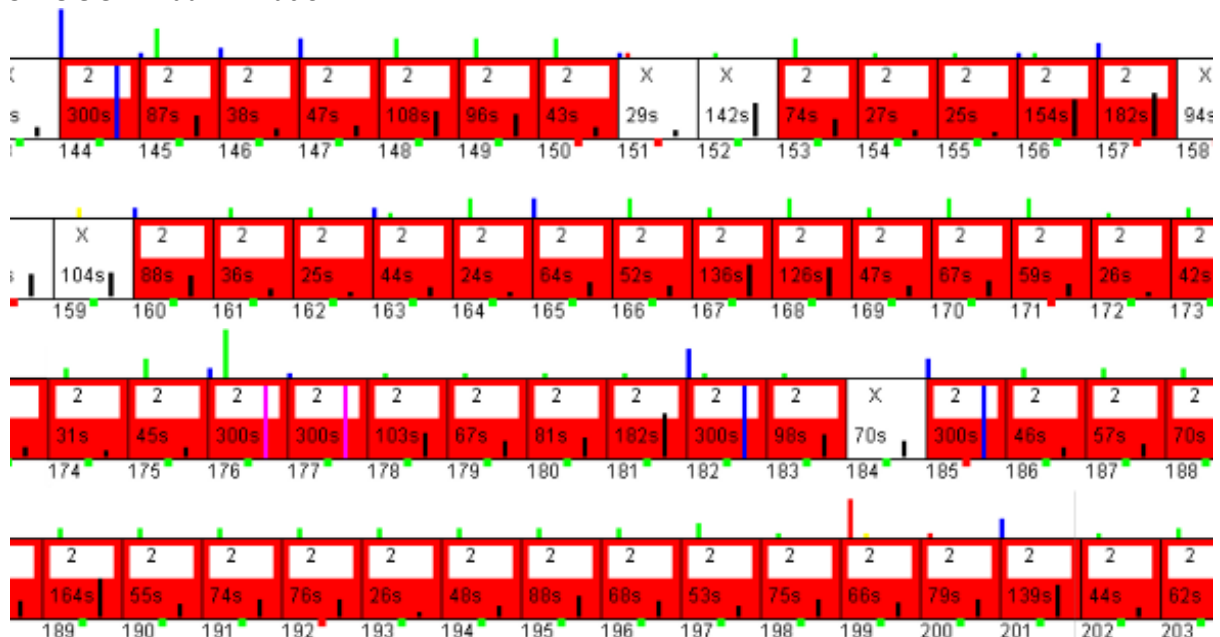
396, Main.cpp, 00h:00m:17s -----
* (283:) gluLookAt(position[x], 10, position[z], 0, 0, 0, 0, 1, 0);
-> (283:) gluLookAt(position[xc], 10, position[zc], 0, 0, 0, 0, 1, 0);

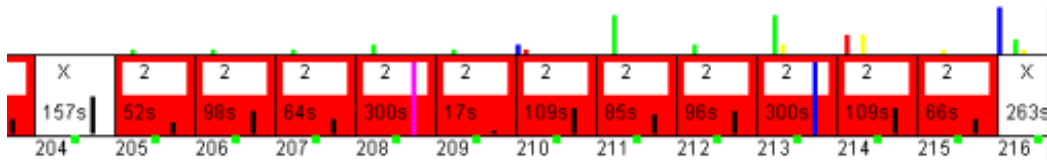
609, Main.cpp, 00h:00m:26s -----
* (126:) float radius = 1;
-> (126:) float radius = 5;
* (308:) gluLookAt(-2*radius*cos(yrot*3.14/180), 15, 2*radius*sin(yrot*3.14/180),
0, 0, 0 , 0 , 1 , 0);
-> (308:) gluLookAt(-radius*cos(yrot*3.14/180), 15, radius*sin(yrot*3.14/180), 0,
0, 0 , 0 , 1 , 0);

```

Figure 413: Short Changes occurring during the implementation of the first-person View

#### 9.7.8.8.5.2 Ida Animation





Long/Short: +144, -154-155, +156-157, -162, -164, +167-168, -172, -174, +176-177, +181-182, +185, +189, -193, +201, +208, -209, +213

**Figure 414: Segment timeline for the implementation of the walk animation**

The student's implementation of a walk animation (captured in Segment [IDA\\_A3.SP.4.1."The Walk Animation"](#) [66]) involves a large number of Long Changes (see Table 120), some of which are shown in Figure 415. While some of these Changes are irrelevant (156 involves modification of the statements encapsulated by one of the animation's if-statements, 176 involves the replacement of a constant with a variable for many lines of code) most Long Changes implement new transformations or stages of the animation.

**Table 120: Number of Long and Short Changes in the implementation of the walk animation**

	Short	Long	Normal	Total
Anim	8	14	44	66

```

144, Main.cpp, 00h:08m:10s -----
+ (71:) bool fwd = true;
+ (531:) if(fwd and rotation[lu_leg][xc]<=30){
+ (532:) rotation[lu_leg][xc]+=0.1;
+ (533:) if(rotation[lu_leg][xc]==30)
+ (534:) fwd = false;
+ (535:) }
+ (536:) if(!fwd and rotation[lu_leg][xc]>=-30){
+ (537:) rotation[lu_leg][xc]-=0.1;
+ (538:) if(rotation[lu_leg][xc]==-30)
+ (539:) fwd = true;
+ (540:) }

156, Main.cpp, 00h:02m:34s -----
+ (547:) }
* (529:) if(position[xc]<=50 and !stopAnimation)
-> (529:) if(position[xc]<=50 and !stopAnimation){

157, Main.cpp, 00h:03m:02s -----
+ (73:) rotation[lu_arm][xc] = -85;

```

```

+ (74:) rotation[ru_arm][xc] = 85;

176, Main.cpp, 08h:33m:22s -----
+ (530:) float inc = 0.05;
+ (531:) float k = 0.5;
* (533:) position[xc]+=0.001;
-> (535:) position[xc]+=inc/10;
* (535:) rotation[lu_leg][zc]+=0.05;
-> (537:) rotation[lu_leg][zc]+=inc;
* (536:) rotation[ru_leg][zc]-=0.05;
-> (538:) rotation[ru_leg][zc]-=inc;
* (537:) rotation[ll_leg][zc]=-0.5*abs(rotation[lu_leg][zc]);
-> (539:) rotation[ll_leg][zc]=-k*abs(rotation[lu_leg][zc]);
* (538:) rotation[r1_leg][zc]=-0.5*abs(rotation[ru_leg][zc]);
-> (540:) rotation[r1_leg][zc]=-k*abs(rotation[ru_leg][zc]);
* (539:) rotation[lu_arm][yc]=0.5*rotation[lu_leg][zc];
-> (541:) rotation[lu_arm][yc]=k*rotation[lu_leg][zc];
* (540:) rotation[ru_arm][yc]=-0.5*rotation[ru_leg][zc];
-> (542:) rotation[ru_arm][yc]=-k*rotation[ru_leg][zc];
* (547:) rotation[lu_leg][zc]-=0.05;
-> (549:) rotation[lu_leg][zc]-=inc;
* (548:) rotation[ru_leg][zc]+=0.05;
-> (550:) rotation[ru_leg][zc]+=inc;
* (549:) rotation[ll_leg][zc]=-0.5*abs(rotation[lu_leg][zc]);
-> (551:) rotation[ll_leg][zc]=-k*abs(rotation[lu_leg][zc]);
* (550:) rotation[r1_leg][zc]=-0.5*abs(rotation[ru_leg][zc]);
-> (552:) rotation[r1_leg][zc]=-k*abs(rotation[ru_leg][zc]);
* (551:) rotation[lu_arm][yc]=0.5*rotation[lu_leg][zc];
-> (553:) rotation[lu_arm][yc]=k*rotation[lu_leg][zc];
* (552:) rotation[ru_arm][yc]=-0.5*rotation[ru_leg][zc];
-> (554:) rotation[ru_arm][yc]=-k*rotation[ru_leg][zc];

189, Main.cpp, 00h:02m:44s -----
* (548:) position[xc] += sin(rotation[ru_leg][zc]*3.14/180)*inc;
-> (548:) position[xc] += abs(sin(rotation[ru_leg][zc]*3.14/180)*inc);
* (561:) position[xc] += sin(rotation[lu_leg][zc]*3.14/180)*inc;
-> (561:) position[xc] += abs(sin(rotation[lu_leg][zc]*3.14/180)*inc);

```

Figure 415: Long Changes in the implementation of the walk animation



Short Changes shown in Figure 416 include tweaks at (154-155), irrelevant syntax error fixes at (172, 193) and the reversal of a transformation's direction modification (164) occurring two Changes earlier. None of these Changes indicate a trial-and-error method or even a utilisation of OpenGL as a visualization aid, suggesting that the student engages in mental visualization of spatial actions rather than attempting to externalise them.

```

154, Main.cpp, 00h:00m:27s -----
* (539:) if(!fwd and rotation[lu_leg][zc]>=-30){
-> (539:) if(!fwd and rotation[lu_leg][zc]>=-20){

155, Main.cpp, 00h:00m:25s -----
* (544:) if(rotation[lu_leg][zc]<=-30)
-> (544:) if(rotation[lu_leg][zc]<=-20)

164, Main.cpp, 00h:00m:24s -----
* (539:) rotation[lu_arm][yc]=-0.5*rotation[lu_leg][zc];
-> (539:) rotation[lu_arm][yc]=0.5*rotation[lu_leg][zc];
* (540:) rotation[ru_arm][yc]=0.5*rotation[ru_leg][zc];
-> (540:) rotation[ru_arm][yc]=-0.5*rotation[ru_leg][zc];
* (549:) rotation[lu_arm][yc]=-0.5*rotation[lu_leg][zc];
-> (549:) rotation[lu_arm][yc]=0.5*rotation[lu_leg][zc];
* (550:) rotation[ru_arm][yc]=0.5*rotation[ru_leg][zc];
-> (550:) rotation[ru_arm][yc]=-0.5*rotation[ru_leg][zc];

172, Main.cpp, 00h:00m:26s -----
* (541:) rotation[l1_arm][yc]=abs(rotation[lu_arm][yc];
-> (541:) rotation[l1_arm][yc]=abs(rotation[lu_arm][yc]);

193, Main.cpp, 00h:00m:26s -----
* (548:) position[xc] += abs(pow(sin(rotation[ru_leg][zc]*3.14/180,2))*inc/2);
-> (548:) position[xc] += abs(pow(sin(rotation[ru_leg][zc]*3.14/180),2)*inc/2);
* (561:) position[xc] += abs(pow(sin(rotation[lu_leg][zc]*3.14/180,2))*inc/2);
-> (561:) position[xc] += abs(pow(sin(rotation[lu_leg][zc]*3.14/180),2)*inc/2);

```

Figure 416: Short Changes in the implementation of the walk animation

## 9.8 Machine Segment Generation

### 9.8.1 Description of data used in Evaluation

Table 121 and

Table 122 describe statistics for Assignment 1 (A1) and Assignment 3 (A3). The most significant difference between A1 and A3 is the very low expected precision in A1 compared to A3. This is due to more of A3's Changes having been classified as 'interesting' by the researcher as a proportion of total Changes.

**Table 121: Description of Project History Change data from Assignment 1**

Student	Selected	Interesting	Total	E.Prec	Max. A/E
John	200	200	797	0.251	3.985
Christopher	281	280	1122	0.25	3.993
Michael	552	807	2206	0.366	2.734
Ida	309	373	1234	0.302	3.308
Thomas	306	469	1222	0.384	2.606
	329.6	425.8	1316.2	0.31	3.325

**Table 122: Description of Project History Change data from Assignment 3**

Student	Selected	Interesting	Total	E.Prec	Max. A/E
John	178	463	708	0.654	1.529
Christopher	300	418	1199	0.349	2.868
Michael	151	349	603	0.579	1.728
Ida	172	395	684	0.577	1.732
Thomas	206	473	822	0.575	1.738
	201.4	419.6	803.2	0.547	1.919

Also, A1 on average involved many more Total Changes than did A3. In fact, only a single student (Christopher) spent more Changes on A1 than he did on A3. The combination of a higher number of Total Changes and lower percentage of interesting Changes may indicate that A1 was less effective at engaging students with relevant material. However, while it does indicate that students may have

spent more time on A1 than on A3, it would be important to also analyse timestamps to verify this since Changes in A3 may on average have taken a longer time to complete. Also, some students such as Michael did not complete several core tasks for A3 which explains why he has a very low number of total Changes in A3.

Since A1 tends to involve a smaller percentage of interesting Changes, the Maximum A/E ratio is also almost twice as high in A1 (3.325) than in A3 (1.919). Since larger A/E ratios are achievable in A1 than in A3, well-performing algorithms will produce better A/E ratios for A1 than they will for A3. As an example, if 75% of changes are interesting, correctly identifying 90% of Target Changes produces a relatively low ratio of  $90/75=1.2$  whereas given 10% interesting Changes, correctly identifying 20% of interesting Changes produces a much higher ratio of  $20/10=2.0$ , despite the first example producing an improvement of 15% over expected Precision and the second example producing an improvement of only 10%. This means that a random algorithm would be likelier to achieve higher A/E ratios in A1 than in A3. This means that lower A/E ratios will be sufficient to produce significant results for A3.

The relatively low percentage of Target Changes for A1 also means that for most students in A1 each correctly identified Change is associated in a larger increase in A/E ratios than in A3. For example, for John A1 with 200 Target Changes, 797 Total Changes and 200 Candidate Changes yields

$$\text{Expected precision} = 200 / 797$$

$$\text{Actual precision} = x / 200$$

$$E/A \text{ ratio} = (x / 200) / (200 / 797) = x * 0.020$$

so every additional correctly identified Change increases the A/E ratio by 0.02 whereas for A3 with 178 Target Changes, 708 Total Changes and 463 Candidate Changes

$$\text{Expected precision} = 463 / 708$$

$$\text{Actual precision} = x / 178$$

$$E/A = (x / 178) / (463 / 708) = 0.009$$

So while the increase by one correctly identified Change in A3 produces an increase of 0.09% in the E/A ratio, a correctly identified Change causes the E/A ratio to increase by 2.0% in A1 for John. Since that trend holds for other assignments (though to a lesser degree), A/E ratios are more likely to change significantly due to chance in A1 than they are A3.

### 9.8.2 Deconstructive Approach to discovering good algorithm settings

Some algorithms have a setting for which a number of items can be included or excluded. For example the LH-Graph algorithm can be set to only visit Line History modifications of a certain type. Each modification type (ADDED, DELETED, GHOST, MUTATED, MOVED) can either be included or excluded. To test what combination of settings would produce the best results would require many runs. For example, for a combination of four different items to be included or excluded, there would be one way to include all items, four ways to include three items, six ways to include two items and four ways to include one item, adding up to a total of  $1+4+6+4 = 15$  different settings to try. Since each run requires both the generation of segments and the evaluation of the generation method through random simulation, performing all the runs can be expensive in terms of time, especially considering the many different algorithms tested in this chapter.

Instead of trying each possible combination of items, a deconstructive approach will be used to discover settings likely to be well-performing. An example of the deconstructive approach will be given now using made-up data.

In the first phase of the application of the deconstructive approach the combination of all items as well as the combination of all items excluding one are used for the setting in question. The results in the example (see Table 123) show that the exclusion of Setting 1 and Setting 2 both improve the A/E ratio. Exclusion of Setting 2 provides the best improvement of A/E ratio, whereas exclusion of Setting 1 provides the best fit\*spread value. Since the initial run (including all settings) was not the best-performing run, another phase of experimentation will be carried out.

**Table 123: Example Data for the first phase of the deconstructive approach**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
All	2	0.5	0.7	0.25	2	0
-Setting1	2.5	0.625	0.9	0.15	2	1
-Setting2	3	0.75	0.8	0.1	3	2
-Setting3	1.5	0.375	0.5	0.65	1	0
-Setting4	1	0.25	0.4	0.9	0	0

Since A/E ratio is used as the primary metric in the deconstructive approach, Setting 2 is excluded for the next phase. As shown in Table 124, the next phase has one run excluding only Setting 2, and runs excluding Setting 2 and one of the other settings. As the results show, exclusion of both Setting 1 and Setting 2 provides the best A/E ratio, better even than the ratio of only excluding Setting 2. This means that another phase of experimentation will be carried out.

**Table 124: Example Data for the second phase of the deconstructive approach**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
-Setting2	3	0.75	0.75	0.1	3	2
-Setting1 -Setting2	3.25	0.813	0.8	0.05	4	3
-Setting2 -Setting3	2.5	0.625	0.6	0.2	2	0
-Setting2 -Setting4	1.5	0.375	0.5	0.7	0	0

In the third phase of experimentation, both Setting 1 and Setting 2 are excluded for the initial setting and two additional runs excluding Setting 1 and Setting 2 as well as one of the remaining settings are executed. Results for this phase are shown in Table 125. As the results show, the exclusion of Setting 1 and Setting 2 (the initial setting) produces the best A/E ratio. Exclusion of an additional setting produces a worse A/E ratio. As a result, the process is terminated and the exclusion of Setting 1 and Setting 2 is chosen as the best combination of settings. If the exclusion of a further setting had improved the result, another phase would have been carried out.

**Table 125: Example Data for the third and final phase of the deconstructive approach**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
-Setting1 -Setting2	3.25	0.813	0.8	0.05	4	3
-Setting1 -Setting2 -Setting3	2.75	0.688	0.85	0.15	3	1
-Setting1 -Setting2 -Setting4	1.25	0.313	0.45	0.75	0	0

While the settings associated with the best run found during the deconstructive procedure is chosen as the 'best' setting the approach is a heuristic and it is possible that different settings that were not tried during this approach (such as the exclusion of Setting 1 and Setting 5 as one example) may produce an even better result.

### 9.8.3 Ensuring Consistent Results

#### 9.8.3.1 Eliminating randomness from segment generation

To ensure consistent results, modification had to be made to the algorithms for generating segments and for filtering out duplicate settings. In practice, several segments may be of equal size or one of several nodes may be chosen as a starting-off point for discovery of adjacent nodes. Utilising a

random node (by selection from a 'set' data structure which does not guarantee any particular ordering and will produce a different ordering of members internally on every run, for example) will generate different results. SCORE cannot differentiate between these 'equally-good' results, but the results may include different numbers of correctly-identified changes. Therefore, care was taken to utilise approaches which create reproducible results by ordering and then using nodes (Changes) with larger overall positions (occur later in the project). This means that two runs of the same algorithm will always produce the same machine-generated segments. Estimated p-values will vary slightly since they are produced through simulation rather than calculation.

### 9.8.3.2 Eliminating randomness in selecting from segments of equal size

A further problem is best illustrated by the example in Figure 417. If the evaluation algorithm is seeking to select 18 Changes, it can trivially select the size 10 and size 5 segments. However, for the last 3 Changes it must choose between three segments. To do so randomly will mean that results for correctly chosen Changes will vary between every run if there exist multiple equal-sized segments and not all of these Segments are selected.

Segment 1: size 10, 10 correct
Segment 2: size 5, 4 correct
Segment 3: size 3, 0 correct
Segment 4: size 3, 3 correct
Segment 5: size 3, 0 correct

**Figure 417: Machine-Segmenting result**

To prevent this source of randomness, instead of choosing from equal-sized segments randomly (in which case the example would yield either 14 or 17 correct Changes) the average number of correct Changes for these equal-sized segments is calculated and then that average value is used to calculate the final number of correctly identified Changes.

In the above example, if a total of 17 Changes is to be selected, then the first two segments are chosen (trivial), after which the average number of correct Changes is calculated for segments of size 3 as  $3/3=1$ . Since 2 more Changes are to be selected,  $2/3 * 1 = 2/3$  Changes are selected from the 3-sized segments; the result is rounded yielding 1 Change. Therefore, the number of correctly identified Changes is  $10+4+1 = 15$ .

## 9.8.4 Machine-Generation Algorithm Pseudo-Code

### 9.8.4.1 Random Method Pseudo-Code

```
Set <Segment> segments = ();
```

```
//Generate segments
```

```

void generate()
{
    for(int i = 0; i < totalChanges; i += segSizeToSelect)
    {
        int limit;
        if(i + segSizeToSelect < totalChanges)
            limit = i + segSizeToSelect;
        else
            limit = totalChanges;

        segments.add(Segment(i, limit));
    }
}

//Select Random Segments
Vector <Change> selectRandomChanges(int toSelectNr)
{
    int segSizeToSelect = x;
    Set selectedChanges = ();
    Set notSelectedSegments = createCopy(segments);

    while(selectedNr < toSelectNr)
    {
        Segment curSelected = Random.select(notSelectedSegments);
        notSelectedSegments.remove(curSelected);

        if(curSelected.size() + selectedNr <= segSizeToSelect)
        {
            selectedChanges.addAll(curSelected.getChanges());
        }
        else
        {
            for(int i = 0; i < (segSizeToSelect - curSelected.size()); i++)
            {
                selectedChanges.add(curSelected.getChangeAt(i));
            }
            break;
        }
        selectedNr += curSelected.size();
    }
    return selectedChanges;
}

```

#### 9.8.4.2 FM Method Pseudo-Code

```

Set <Change> calculateSegments(int segNrToSel)
{
    Set <Pair <Change, Float> > valueForChange = new Vector();
    for(FileHistory fileHistory : ProjectHistory.getFileHistories())

```

```

{
    Change prevDoc;
    RollingAverager rollingAverager = RollingAverager();

    for(Change curDoc : fileHistory.getChangesInOrder())
    {
        Diff diff = calculateDiff(prevDoc, curDoc);
        int totalModified = 0;
        if(useDeleted)
            totalModified += diff.getDeletedLines();
        if(useAdded)
            totalModified += diff.getAddedLines();
        Float value = rollingAverager.calcRA(totalModified);
        valueForChange.set(Pair (curDoc, value));
    }
}

/* Sort by the calculated Rolling Average value
 * Either in ascending or descending order depending on setting */
sort(valueForChange);

Set <Change> selectedChanges = ();
int selSoFar = 0;
while(selSoFar < segNrToSelect)
{
    selectedChanges.add(valueForChange.front());
    valueForChange.removeFront();
    selSoFar++;
}

return selectedChanges;
}

```

#### 9.8.4.3 LM Method Pseudo-Code

```

Set <Change> calculateSegments(int segNrToSel)
{

```



```

for(FileHistory fileHistory : ProjectHistory.getFileHistories())
{
    Change prevDoc;
    Vector <Float> prevLineValues;

    for(Change curDoc : fileHistory.getChangesInOrder())
    {
        //Skip the first document
        if(prevDoc == 0) continue;

        //Calculate values for each of the Change's lines measuring recent activity
        Vector <Float> lineValues;
        lineValues = calculateMetrics(prevDoc, curDoc);
        lineValuesForChange.set(Pair (curDoc, lineValues));

        prevLineValues = lineValues;
        prevDoc = curDoc;
    }
}

/* Sort using either the SUM or MAXIMUM of all line values for a
* given Change */
sort(lineValuesForChange);

//Select those Changes with the highest recent activity values
Set <Change> selectedChanges = ();
int selSoFar = 0;
while(selSoFar < segNrToSelect)
{
    selectedChanges.add(valueForChange.front());
    valueForChange.removeFront();
    selSoFar++;
}

return selectedChanges;
}

Vector <Float> calculateLineValues(Change prevDoc, Change curDoc, Vector <Float>
prevLineValues)

```

```

{
    Vector <Float> lineValues = new Vector(curDoc.getLineNr());

    //This step calculates a line mapping using a method like the one used by the Line History Generation
    algorithm to match added, deleted, mutated and moved lines (no ghost lines) from the previous to the
    current document
    LineMapping mapping = calculateLineMapping(prevDoc, curDoc);

    //For each line in the current document, assign it a value based on the value it had in the previous
    document and the modification type
    for(Line line: curDoc.getLines())
    {
        switch(mapping.modificationType(line))
        {
            case ADDED:
                lineValues.set(line.lineNr(), addedValue);
                break;
            case MOVED:
                lineValues.set(line.lineNr(),
prevLineValues.get(mapping.getPrevLineNo(line) + movedValue);
                break;
            case MUTATED:
                lineValues.set(line.lineNr(),
prevLineValues.get(mapping.getPrevLineNo(line) + mutatedValue);
                break;
            case MAINTAINED:
                lineValues.set(line.lineNr(),    mapping.getPrevLineNo(line)    *
decayFactor);
                break;
        }
    }
    return lineValues;
}

```

#### 9.8.4.4 Line History (LH) Method Pseudo-Code

```

Vector <Segment> generateSegments(int toSelect, int modificationFlags)
{

```

```

//Generate Machine Segments by visiting each Line History and creating a
//segment containing all the Changes in which the line is modified.
//The types of modifications used is determined by modificationFlags
Vector <Segment> generatedSegments = ();

for(LineHistory lineHistory : ProjectHistory.getLineHistories())
{
    Segment segment;

    //Add all Changes associated with modifications of type specified in flags
    for(Modification modification :
lineHistory.getModifications(modificationFlags))
        segment.addChange(modification.getAssociatedChange());

    generatedSegments.add(segment);
}

//Ensure that all segments have unique Changes by only keeping the largest
//segment containing any Change.
filterOutSegmentsWithDuplicateChanges(generatedSegments);

//Sort generated segments by size
sort(generatedSegments);

//Select largest segments until we have enough segments
Vector <Segment> selectedSegments = ();
int selectedSoFar = 0;
while(selectedSoFar + selectedSegments.front().size() <= toSelect)
{
    selectedSegments.add(selectedSoFar);
    selectedSoFar += selectedSegments.front().size();
    selectedSegments.removeFront();
}

return selectedSegments;
}

```

#### 9.8.4.5 Line History Graph (LH-GRAPH) Method Pseudo-Code

```
Vector <Segment> generateSegments(  
    Change change, int adj, int maxDist, int ChangeLineFlags, int LineHistoryFlags ,  
    ExtensionSettings extensionSettings)  
{  
    Vector <Segment> generatedSegments = ();  
    //For each Change in turn, calculate the segment based on a graph traversal of Changes rooted  
    //in that Change  
    for(Change change : ProjectHistory.getChangesInOrder())  
    {  
        Set <Change> changes = getSegmentNodes(change, adj, maxDist, int  
ChangeLineFlags, int LineHistoryFlags);  
  
        //Execute any additional algorithms (described later)  
        for(ExtensionAlgorithm algorithm : extensionAlgorithms)  
        {  
            changes.addAll(algorithm.execute(changes), extensionSettings);  
        }  
  
        //Execute any filters (described later)  
        for(Filter filter : filters)  
        {  
            filter.filter(changes, extensionSettings);  
        }  
  
        Segment segment = Segment(changes);  
        generatedSegments.add(segment);  
    }  
  
    //Ensure that all segments have unique Changes by only keeping the largest  
    //segment containing any Change.  
    filterOutSegmentsWithDuplicateChanges(generatedSegments);  
  
    //Sort generated segments by size  
    sort(generatedSegments);  
  
    //Select largest segments until we have enough segments  
    Vector <Segment> selectedSegments = ();
```

```

int selectedSoFar = 0;
while(selectedSoFar + selectedSegments.front().size() <= toSelect)
{
    selectedSegments.add(selectedSoFar);
    selectedSoFar += selectedSegments.front().size();
    selectedSegments.removeFront();
}

return selectedSegments;
}

Set <Change> getSegmentNodes(
Change change, int adj, int maxDist, , int ChangeLineFlags, int LineHistoryFlags)
{
    Set <Change> adjacents = ();
    Stack <Change> toVisit = (change);

    for(int i = 0; i < adj; i++)
    {
        Stack <Change> newToVisit = (change);

        for(Change curChange : toVisit)
        {
            if(adjacents.contains(curChange))
            {
                continue;
            }
            Set <Change> adjs = getAdjacents(curChange, maxDist,
ChangeLineFlags, LineHistoryFlags);
            newToVisit.addAll(adjs);
            adjacents.addAll(adjs);
        }
        toVisit = newToVisit;
    }

    return adjacents;
}

```

```

Set <Change> getAdjacents(Change change, int maxDist, int ChangeLineFlags, int
LineHistoryFlags)
{
    Set <Change> adjs = ();
    for(LineHistory line : change.getLines())
    {
        if(line.getModificationType() & ChangeLineFlags != 0)
        {
            for(Modification modification : line.getModifications())
            {
                int dist = Math.abs(change.getPosition() -
modification.getChange().getPosition());
                if(dist < maxDist && (modification.getModificationType() &
LineHistoryFlags) != 0)
                {
                    adjs.add(modification.getChange());
                }
            }
        }
    }
    return adjs;
}

```

## 9.8.5 Format of Machine-Generation Method Evaluation Data

### 9.8.5.1 Individual Assignment Data

Table 126 and Table 127 present data for a single run of an algorithm for each student. *Selected* is the number of Changes chosen as ‘interesting’ by the algorithm. For purposes of the evaluation this number is always 25% of the total number of Changes. *Interesting* is the number of ‘interesting’ Changes as pre-identified by the researcher. *Overlap* is the number of *Selected* Changes which also belong to the set of ‘interesting’ Changes. *Total* is the number of total Changes in the Project History. *Precision (Prec)* is the precision produced by the algorithm run. *Expected Precision (E. Prec)* is the precision that a random algorithm would be expected to achieve. *Recall (Rec)* and *Expected Recall (E. Rec)* are the actual and expected recall. *A/E* reports the *Actual/Expected ratio* achieved by the run, *Maximum A/E (M. A/E)* reports the maximum possible *A/E* ratio based on the number of interesting and total Changes and *%M.A/E* reports what percentage of the maximum possible ratio was achieved by the algorithm run. The *A/E* ratio items are the primary measures used in evaluating the performance of algorithms in identifying ‘interesting’ Changes. *Spread (Spr)* and *Fit* report the

Spread and Fit metrics for the algorithm run, and *Spr\*Fit* reports the Spread\*Fit metric which is used for evaluating the quality of grouping of Changes achieved by a segmenting algorithm. Finally, *P-value* reports the probability that a random algorithm would achieve an equally good result.

**Table 126: Example of Evaluation Data for a run of the Machine-Segmenting algorithm for A1**

Student	Selected	Overlap	Interesting	Total	Prec	E. Prec	Rec
John	200	91	200	797	0.46	0.25	0.46
Christopher1	281	124	280	1122	0.44	0.25	0.44
Michael	552	333	807	2206	0.60	0.37	0.41
Ida	309	164	373	1234	0.53	0.30	0.44
Thomas	306	197	469	1222	0.64	0.38	0.42
	A/E	M.A/E	%M.A/E	Spr	Fit	Spr*Fit	P-Value
John	1.81	3.99	0.46	0.54	0.84	0.46	0.006
Christopher1	1.77	3.99	0.44	0.56	0.84	0.47	0.002
Michael	1.65	2.73	0.60	0.73	0.76	0.55	0
Ida	1.76	3.31	0.53	0.65	0.74	0.48	0.002
Thomas	1.68	2.61	0.64	0.64	0.78	0.50	0

**Table 127: Example of Evaluation Data for a run of the Machine-Segmenting algorithm for A3**

Student	Selected	Overlap	Interesting	Total	Prec	E. Prec	Rec
Ida	172	145	395	684	0.84	0.58	0.37
Michael	151	114	349	603	0.75	0.58	0.33
Christopher1	300	165	418	1199	0.55	0.35	0.39

John	178	153	463	708	0.86	0.65	0.33
Thomas	206	192	473	822	0.93	0.58	0.41
	A/E	M.A/E	%M.A/E	Spr	Fit	Spr*Fit	P-Value
Ida	1.46	1.73	0.84	0.70	0.87	0.61	0.001
Michael	1.30	1.73	0.75	0.70	0.84	0.59	0.017
Christopher1	1.58	2.87	0.55	0.65	0.75	0.49	0.001
John	1.31	1.53	0.86	0.74	0.79	0.59	0.003
Thomas	1.62	1.74	0.93	0.78	0.81	0.63	0

Expected precision, expected recall, A/E ratios and the Spread and Fit metrics can be calculated as explained in Section 7.4.

Individual-assignment data such as this are very verbose. Each run produces two such tables worth of data (one for each assignment), and evaluation of algorithms usually involve the comparison of several runs using different settings. This makes it difficult to utilise them in the evaluation of algorithms. For this reason this data is not presented as part of the evaluation. This data is available for all algorithm runs presented in this chapter in the electronic appendix.

### 9.8.5.2 Per-Assignment Summary Data

Table 128 shows summary data for the evaluation of an algorithm for the five A1 assignments. Each row contains Average Precision, Accuracy, Actual/Expected ratio, Maximum A/E ratio, Spread, Fit, Spread\*Fit and p-value as well as total number of assignments reaching the significance level of <0.05 and <0.01 for one run of the algorithm. Different rows present data for runs of the algorithm using different algorithm settings. Table 129 shows the summary data for evaluation of the same algorithm for A3.

These tables average all the results presented in the Individual-assignment tables. These data are only presented in cases where an algorithm's best run for assignment A1 produces very poor results for A3 or vice versa.

**Table 128: Summary data from an example Segment generation run for Assignment 1**



Setting	Prec	Acc	A/E	M. A/E	% M. A/E	Spr	Fit	Spr* Fit	PVal	<0.05	<0.01
Setting 1	0.53	0.43	1.73	3.33	0.54	0.62	0.79	0.49	0.002	5	5
Setting 2	0.55	0.45	1.79	3.33	0.55	0.62	0.76	0.48	0.001	5	5
Setting 3	0.53	0.43	1.73	3.33	0.53	0.62	0.75	0.46	0.001	5	5
Setting 4	0.54	0.44	1.77	3.33	0.54	0.63	0.70	0.44	0.001	5	5

**Table 129: Summary data from an example sement generation run for Assignment 3**

Setting	Prec	Acc	A/E	M.A/E	%M. A/E	Spr	Fit	Spr* Fit	PVal	<0.05	<0.01
Setting 1	0.79	0.36	1.46	1.92	0.79	0.71	0.81	0.58	0.004	5	4
Setting 2	0.81	0.38	1.50	1.92	0.81	0.74	0.75	0.55	0.005	5	4
Setting 3	0.79	0.37	1.47	1.92	0.79	0.75	0.71	0.53	0.010	5	3
Setting 4	0.79	0.37	1.48	1.92	0.79	0.77	0.66	0.51	0.006	5	4

### 9.8.5.3 Overall Summary Data

In most cases, only an Overall Summary table (see Table 130) summarising results over both assignments will be presented since it is concise and makes it easy to see which run performs best overall. As is shown in Table 130 algorithm runs that perform well are highlighted. A yellow highlight means that that run produced the best A/E ratio (best performance in identifying 'interesting' Changes). A green highlight means that the run produced the best spread\*fit value (best segmenting performance). A blue highlight means that the run produced both the best A/E ratio and the best spread\*fit ratio.

**Table 130: Ratio, spread/fit and p-value values averaged over all students and both assignments**

Setting	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
Setting 1	1.594	0.662	0.537	0.003	10	9
Setting 2	1.643	0.6778	0.515	0.003	10	9

Setting 3	1.604	0.661	0.497	0.005	10	8
Setting 4	1.624	0.668	0.475	0.004	10	9

Summary data for individual assignments or Individual-assignment data will be presented on an as-needed basis.

### 9.8.6 LH-Graph Distance/Proximity and Modification Settings

Assignment 1 and Assignment 3 are different in that the A1 structure more strongly encourages the development of classes and data structures and puts more focus on user-interface design. Analysis revealed that this led to much more Assignment 1 time being spent on relatively trivial tasks that were each completed quickly. In comparison students tended to work on a small number of large core problems in A3, and much of their effort was spent on these problems. As a result, more Changes in A3 were deemed ‘interesting’ than in A1. In addition, when examining segments of Changes generated for A1 and comparing their size to A3, it is apparent that segments tend to be much larger in A3.

Table 131 and Table 132 present segment size data for A1 and A3 using the core LH-Graph algorithm for segmenting. As the tables show, for segments making up between 20-30% of total Changes, segments tend to be twice as large for A3 as they are for A1. One reason for this may be that the larger number of General Programming / Implementation problems in A1 involved addition and removal of lines rather than modification, and modification of lines tends to be a far more useful modification type for segmenting since it connects lines and creates large Line Histories, whereas it is more difficult to interpret the co-addition or co-removal of lines since such co-change is much less reliable in detecting related segments.

**Table 131: Average Segment Size of segments making up the top % of Changes for A1**

Student/%	John	Christopher	Michael	Ida	Thomas	Average
0.1	14.333	14.875	60.25	32.75	45	33.442
0.2	9.529	10.318	36.917	25.8	27.444	22.002
0.3	7.273	7.37	24.519	18.65	18.55	15.272
0.4	5.732	5.921	17.038	12.744	13.611	11.009

0.5	4.534	4.692	12.849	9.254	9.903	8.246
0.6	3.742	3.835	9.757	6.944	7.567	6.369
0.7	3.016	3.208	7.124	5.18	5.745	4.855
0.8	2.408	2.515	5.401	3.815	4.234	3.6746
0.9	2.081	2.154	3.637	2.908	3.116	2.7792

**Table 132: Average Segment Size of segments making up the top % of Changes for A3**

Student/%	John	Christopher	Michael	Ida	Thomas	
0.1	73	33	46.5	44	35	46.3
0.2	71	26.889	40.667	36.5	33.8	41.771
0.3	54.5	19.316	30.833	30.857	28.444	32.79
0.4	36.125	14.265	24.6	25.091	24.429	24.902
0.5	26.071	10.362	19.188	18.316	20.85	18.957
0.6	20.381	7.423	14.52	12.875	15.968	14.233
0.7	14.676	5.35	10.071	8.404	11.52	10.004
0.8	9.161	3.582	6.722	5.426	7.741	6.526
0.9	5.646	2.784	4.145	3.645	4.805	4.205

The effect of these differences can be observed when comparing different distance/depth settings for the LH-Graph algorithm. For example, Table 133 presents a comparison of A/E ratios for difference/depth settings. For A1 the best A/E ratios are achieved with a search depth of 4 or even 6, while for A3 the best A/E ratio is achieved with a search depth of 1 with the ratio for larger search depths decreasing substantially. The likely reason for this is that the larger search depths produce more linking and hence larger segments. This is useful in cases where segments are small, since this indicates that many links between related Changes are not being detected. On the other hand, when

segments are already capturing most related Changes and are large as in A3, increased search depth deteriorates performance by including more irrelevant Changes or by merging whole unrelated segments.

**Table 133: A/E ratios for different distance and depth settings for A1 and A3**

	A1 Ratio	A3 Ratio	Avg Ratio
d:25, adj:2	1.73	1.52	1.63
d:50, adj:2	1.72	1.52	1.62
d:75, adj:2	1.70	1.54	1.62
d:150, adj:2	1.76	1.55	1.65
d:25, adj:3	1.73	1.46	1.59
d:50, adj:3	1.79	1.50	1.64
d:75, adj:3	1.73	1.47	1.60
d:150, adj:3	1.77	1.48	1.62
d:25, adj:4	1.77	1.41	1.59
d:50, adj:4	1.78	1.46	1.62
d:75, adj:4	1.79	1.41	1.60
d:150, adj:4	1.74	1.42	1.58
d:25, adj:5	1.80	1.40	1.60
d:50, adj:5	1.77	1.44	1.61
d:75, adj:5	1.75	1.40	1.58
d:150, adj:5	1.71	1.40	1.56
d:25, adj:6	1.79	1.37	1.58

d:50, adj:6	1.76	1.42	1.59
d:75, adj:6	1.73	1.40	1.56
d:150, adj:6	1.70	1.40	1.55
d:25, adj:8	1.80	1.38	1.59
d:50, adj:8	1.74	1.41	1.57
d:75, adj:8	1.72	1.39	1.56
d:150, adj:8	1.71	1.40	1.56
d:25, adj:inf	1.79	1.38	1.58
d:50, adj:inf	1.76	1.40	1.58
d:75, adj:inf	1.73	1.40	1.56
d:150, adj:inf	1.73	1.40	1.57

This result suggests that it may be beneficial to utilise larger depth for assignments in which segments are small until a set average segment size is achieved, and to limit depth for assignments for which lower depth settings already generate large segments. This proposal is left for future research since a larger body of assignments to evaluate against would be desirable to test this hypothesis.

When examining different modification types (Table 134) no clear trade-off exists, with the same settings generally performing well for both assignments, and the setting (-HLD:ADDED) performing best for both assignments. Indeed, the importance of excluding Added lines from the set of modified lines visited was apparent early on and seems to be linked to Changes in which many lines are added at the same time, since navigation from such Changes tends to generate very large segments with many unrelated Changes grouped together.

**Table 134: A/E ratios for different modification settings for A1 and A3.**

Setting	A1 Ratio	A3 Ratio	Avg Ratio

All	1.78	1.38	1.58
HLE: -ADDED	1.77	1.37	1.57
HLE: -DELETED	1.77	1.42	1.59
HLE: -MOVED	1.75	1.39	1.57
HLE: -MUTATED	1.19	0.97	1.08
HLE: -GHOST	1.79	1.38	1.59
HDL: -ADDED	1.79	1.50	1.64
HDL: -DELETED	1.78	1.37	1.57
HDL: -MOVED	1.78	1.37	1.57
HDL: -MUTATED	1.70	1.28	1.49
HDL: -GHOST	1.79	1.39	1.59
HDL: -ADDED -DELETED	1.78	1.49	1.63
HDL: -ADDED -MOVED	1.77	1.48	1.63
HDL: -ADDED -MUTANT	1.38	1.22	1.30
HDL: -ADDED -GHOST	1.77	1.49	1.63
HDL: -ADDED HLE: -ADDED	1.74	1.47	1.61
HDL: -ADDED HLE: -DELETED	1.74	1.49	1.62
HDL: -ADDED HLE: -MOVED	1.76	1.45	1.61
HDL: -ADDED HLE: -MUTATED	1.19	1.02	1.10
HDL: -ADDED HLE: -GHOST	1.77	1.48	1.62

## 9.8.7 Machine-Segmenting Algorithm Results

### 9.8.7.1 Random Selection Algorithm Evaluation

The random selection algorithm randomly selects groups of Changes as described in 7.6.1. The different runs presented in Table 135 used different seed values for the random number generator. The Random algorithm performs as expected, with the best run producing an average p-value of 0.31 and a significant result at the  $p < 0.05$  level for one assignment.

**Table 135: Summary Table for runs of the Random algorithm**

Ratio	% of Max	spread*fit	avg p-val	<0.05 (of 10)	<0.01 (of 10)
0.986	41.6%	0.19	0.535	0	0
1.109	47.3%	0.213	0.311	1	0
1.026	44.6%	0.197	0.427	1	1
0.924	40.4%	0.192	0.622	0	0
1.067	42.8%	0.194	0.493	1	1

### 9.8.7.2 File Metrics Algorithm Evaluation

The File Metrics algorithm calculates a rolling average measuring the recent number of modifications to files (additions and deletions) as described in Section 7.6.2.

In the evaluation, three different sizes for the ‘rolling average window’ (the number of Changes which the rolling averager uses to produce the average of recent modifications) were used. In addition, runs were set to choose either the 25% of Changes with the largest number of recent modifications or the 25% of Changes with the smallest number of recent modifications. This results six runs for which the data is presented in Table 136.

**Table 136: Summary Table for runs of the FM algorithm**

Setting	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
ra = 10	0.835	35.1%	N/A	0.662	0	0

ra = 10, useSmallest	1.32	55.3%	N/A	0.123	6	2
ra = 25	0.859	37.1%	N/A	0.628	0	0
ra = 25, useSmallest	1.272	54.6%	N/A	0.15	6	3
ra = 50	0.867	37.0%	N/A	0.672	1	1
ra = 50, useSmallest	1.176	51.1%	N/A	0.241	4	3

The reasoning behind this approach is that a large amount of activity indicates significant student effort, and that as a result Changes with a lot of recent modification would be more likely to be ‘interesting’. In fact, the runs selecting Changes with the largest rolling average all do significantly worse than random, suggesting that the addition and deletion of many lines does not indicate that a Change is significant. The most ready explanation for this is that Changes with many modifications are part of creation of new classes or data structures, and that these activities are often trivial.

Indeed, while run selecting Changes with high rolling averages perform much poorer than chance, those selecting the Changes with the smallest rolling averages perform significantly better than chance. The run (ra=10, useSmallest), utilising the smallest rolling average window size and selecting the smallest Changes achieves significance at  $p < 0.05$  for 6/10 assignments, significantly better than the two that would be expected by chance. The E/A ratio of 1.32 is also significantly above the expected ratio of 1.0. Lastly the average p-value is relatively low, at 0.122. This suggests that Changes which have seen relatively little modification of lines are more likely to be ‘interesting’. This is probably because the low level of modification is due to a small set of lines being worked on as part of a problem being solved, rather than the high level of modification that accompanies trivial tasks such as the creation of new data structures or functions.

Since FM algorithm does not group Changes spread and fit are not discussed.

### 9.8.7.3 Line Metrics Algorithm Evaluation

The Line Metrics algorithm calculates recent activity for each line of source code and is hence aware of not only additions and deletions but also lines moved or modified. For a description of the algorithm, see Section 7.6.3.

The algorithm takes five parameters. Three parameters specify the weight that is given to different modification types (Addition, Mutation, Moved), with a fourth parameter specifying the rate at



which a line's activity value decays if it is not modified (Maintained) in a Change. The final parameter specifies whether Change scores are calculated as the sum of all line activity values or by choosing the maximum line activity value occurring in that Change.

Results for evaluation of different modification type value weightings and sum/maximum line value can be found in the appendix. The best results were found to be produced by using only the maximum value and by positively weighting only Mutations (modifications of line text) and not Moved or Added lines. This suggests that modification of lines plays a key role in student problem solving, and that analysis of such line modifications will be an important part of any approach to detect interesting Changes.

The final setting is the decay setting which specifies how quickly a line value decays. The result of five runs varying the decay setting from (0.1-1.0) is presented in Table 137. The value of a line is multiplied by the decay value if it is not modified, so lower decay values mean faster decay. The best run using a decay setting of 0.5 (meaning that a line's value is halved in any Change where it is not modified) achieves significance at 9/10 assignments at the  $p > 0.05$  level indicating that it performs significantly better than random. The A/E p-value of 1.453 is also significantly above 1.0. The Line Metric approach is a viable approach to detecting 'interesting' Changes. However, like the FM method it does not group Changes.

**Table 137: Summary Table for runs of the FM algorithm**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
decay: 0.1	1.439	60.3%	N/A	0.031	8	5
decay: 0.25	1.452	60.8%	N/A	0.024	9	5
decay: 0.5	1.453	60.9%	N/A	0.018	9	6
decay: 0.75	1.428	59.9%	N/A	0.023	9	4
decay: 1.0	0.903	38.5%	N/A	0.59	0	0

It is also interesting to note that the run using a decay value of 1.0 (no decay) produces a result that is worse than random.

#### 9.8.7.4 Line History Algorithm Evaluation

The Line History algorithm uses the Line History Table (as described in Section 6.2.1) to identify groups of related Changes by visiting Line Histories and selecting Changes in which modifications to the line occur. The algorithm is described in Section 7.6.4.

The algorithm takes parameters specifying which modification Changes should be included. For example, using the parameters (ADDED, MOVED, MUTATED) would then include all the Changes in which the line in question was Added, Moved or Mutated, ignoring those in which it was Deleted or Ghosted.

Since discovering which combination of modification types produces the best result is difficult as there are  $(1 + 5 + 10 + 10 + 5 = 31)$  different combinations of modification types the *Deconstructive Method* described towards the end of Section 7.4.4 was utilised. Results from the final phase of the method's application are shown in Table 138. The two settings that achieve the best result are those utilising only the Mutated and Ghost modification and the setting using only the Mutated modification. However, the difference to the other two settings is quite small. All settings achieve significance for 9/10 assignments at the  $p > 0.05$  level and for 7/10 assignments at the  $p > 0.01$  level. They also achieve an A/E ratio of around 1.54 and an average p-value of 0.010. The only significance between the best runs and the other two is that the best runs produce a better spread\*fit value.

**Table 138: Summary Table for runs of the LH algorithm**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
Mutated, Ghost, Deleted	1.543	64.6%	0.526	0.01	9	7
Mutated, Deleted	1.543	64.6%	0.526	0.01	9	7
Mutated, Ghost	1.545	64.8%	0.582	0.01	9	7
Mutated	1.545	64.8%	0.582	0.01	9	7

The results demonstrate that the LH algorithm is effective at detecting 'interesting' Changes. The fact that the run that produced the best result included only modifications in which lines were Mutated again demonstrates the importance of line modifications to the detection of 'interesting' Changes. The algorithm also achieves a good spread\*fit value (both spread and fit are good) indicating that it is successful in grouping Changes correctly.

### 9.8.7.5 Line History Graph Algorithm Evaluation

The Line History Graph algorithm uses the Line History Table (as described in Section 6.2.1) to produce a graph, utilising Line Histories as junctions which are connected to Changes. The algorithm is described in Section 7.6.5.

The LH-Graph algorithm has several parameters. One set of parameters determines which lines' Line Histories are visited from a Change via selection of modification types. For example, selection of the Mutation modification type means that only lines that have been Mutated in a Change are visited, while Added, Deleted, Ghost and Moved lines are ignored. Another set of parameters determines which of a Line History's modifications are visited, again by selecting a set of modification types. For example, if the mutated modification type were selected for Changes and the Moved modification type were selected for Line Histories then the algorithm would visit all of a Change's Line Histories where the line was Mutated in the current Change. The algorithm would then visit all the Changes associated with modifications in those Line Histories where the modification type equals Moved. Then for those Changes it would again visit all the Line Histories of lines that were mutated, and so on.

Another parameter sets the maximum distance from the current Change at which modifications of Line Histories will be visited. This restricts related Change discovery to Changes in temporal proximity.

The last parameter sets how often the LH-Graph algorithm will recurse. Without recursion, it will visit all Changes associated with Line Histories modified in the initial Change. With a recursion depth of 1, it will then also visit the Line Histories modified in those Changes discovered during the first search. With infinite recursion, the algorithm will keep discovering related Changes until no Line Histories lead to any unmarked Changes.

To discover the best distance and depth settings for the LH-Graph algorithm runs using distances of (25,50,75,150) and depth settings of (0, 1, 5, inf) were conducted. This resulted in the 16 runs presented in Table 139. The best A/E ratio was produced by the setting (depth=0, dist=150) and the best fit\*spread was produced by the setting (depth=0, dist=25). The setting chosen as the 'best' setting was (depth=1, dist=50) which produces an A/E ratio and fit\*spread value close to the best runs.

**Table 139: Summary Table for the evaluation of the LH-Graph algorithm using different distance settings**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01

d:25, adj:2	1.629	68.2%	0.536	0.002	10	9
d:50, adj:2	1.619	67.7%	0.524	0.002	10	9
d:75, adj:2	1.621	67.8%	0.512	0.003	10	10
d:150, adj:2	1.653	68.7%	0.495	0.003	10	9
d:25, adj:3	1.594	66.2%	0.537	0.003	10	9
d:50, adj:3	1.643	67.8%	0.515	0.003	10	9
d:75, adj:3	1.604	66.1%	0.497	0.005	10	8
d:150, adj:3	1.624	66.8%	0.475	0.004	10	9
d:25, adj:6	1.579	64.5%	0.531	0.011	9	8
d:50, adj:6	1.587	65.0%	0.5	0.007	10	8
d:75, adj:6	1.564	64.1%	0.478	0.02	9	8
d:150, adj:6	1.553	63.3%	0.444	0.026	9	7
d:25, adj:inf	1.582	64.7%	0.527	0.01	9	8
d:50, adj:inf	1.578	64.5%	0.494	0.012	8	8
d:75, adj:inf	1.564	64.0%	0.473	0.017	9	8
d:150, adj:inf	2.622	69.4%	0.020	0.04	7	0

Having discovered a good setting for depth and distance, good settings for modification types for navigation from Changes and navigation from Line Histories had to be discovered. Since the total number of combinations for these settings would have been very large, the deconstructive method (see the end of Section 7.4.4) was utilised. Results from the second and final phase are presented in

Table 141 (results from the first phase are shown in Table 140). The best setting produces significance for all 10/10 assignments at the  $p < 0.05$  level and for 9/10 assignments at the  $p < 0.01$  level and has an average p-value of 0.003. The A/E ratio of 1.643 achieves 67.8% of the best possible A/E ratio. The spread\*fit ratio of 0.515 indicating the quality of segmenting is high, but lower than

that produced by the LH algorithm. Overall the LH-Graph algorithm produces very good results and the probability values clearly indicate that it performs significantly better than random. The A/E ratio is also the highest of any of the different algorithmic approaches discussed.

**Table 140: Summary Table for the first phase of the evaluation of the LH-Graph algorithm using different modification settings**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
All	1.579	64.3%	0.518	0.019	9	8
HLE: -ADDED	1.568	64.1%	0.507	0.019	8	8
HLE: -DELETED	1.594	65.3%	0.513	0.009	9	9
HLE: -MOVED	1.572	64.3%	0.514	0.018	9	8
HLE: -MUTATED	1.081	45.0%	0.213	0.388	0	0
HLE: -GHOST	1.585	64.7%	0.517	0.015	9	8
HDL: -ADDED	1.643	67.8%	0.515	0.003	10	9
HDL: -DELETED	1.573	64.1%	0.521	0.0179	9	8
HDL: -MOVED	1.573	64.3%	0.519	0.019	9	8
HDL: -MUTATED	1.491	61.2%	0.462	0.046	7	7
HDL: -GHOST	1.587	64.7%	0.521	0.018	9	8

**Table 141: Summary Table for the second phase of the evaluation of the LH-Graph algorithm using different modification settings**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
-HDL_ADDED	1.643	67.8%	0.515	0.003	10	9
HDL: -ADDED -DELETED	1.631	67.5%	0.505	0.003	10	9
HDL: -ADDED -MOVED	1.628	67.1%	0.514	0.003	10	9

HDL: -ADDED -MUTANT	1.3	55.1%	0.272	0.12	5	3
HDL: -ADDED -GHOST	1.626	67.1%	0.517	0.003	10	9
HDL: -ADDED HLE: -ADDED	1.606	66.6%	0.472	0.004	10	8
HDL: -ADDED HLE: -DELETED	1.616	67.1%	0.513	0.004	10	8
HDL: -ADDED HLE: -MOVED	1.606	66.0%	0.512	0.004	10	9
HDL: -ADDED HLE: -MUTATED	1.102	46.2%	0.203	0.332	0	0
HDL: -ADDED HLE: -GHOST	1.625	67.0%	0.513	0.003	10	9

## 9.8.8 LH-Graph Extension Algorithm Pseudo-Code

### 9.8.8.1 Compile Filter Pseudo-Algorithm

//Remove all Changes in segment of type compileType

```
void filterByCompileType(Segment segment, int compileType)
{
    for(Change change : segment.getChanges)
        if(change.getCompileStatus() == compileType)
            segment.remove(change);
}
```

//Main compile filter function.

//Type is either COMPILE\_ONLY, NONCOMPILE\_ONLY or ADAPTIVE

```
void compileFilter(Segment segment, int type)
{
    if(type == COMPILE_ONLY)
    {
        filterByCompileType(segment, NONCOMPILED);
    }
    else if(type == NONCOMPILE_ONLY)
    {
        filterByCompileType(segment, COMPILED);
    }
    else //Adaptive filter
    {
```

```

int numberOfCompiled = 0;
int numberOfNonCompiled = 0;
for(Change change : segment)
{
    if(change.getCompileStatus() == COMPILED)
        numberOfCompiled++;
    else
        numberOfNonCompiled++;
}
if(numberOfNonCompiled > numberOfCompiled)
    filterByCompileType(segment, COMPILED)
else
    filterByCompileType(segment, NONCOMPILED)
}
}

```

#### 9.8.8.2 Small-Segment Filter Pseudo-Algorithm

Segment smallSegmentFilter(Segment segment, **int** divideDistance, **int** minSize)

```

{
    //divide segment into subsegments. Start a new subsegment whenever
    //the last Change is farther than divideDistance away
    Vector <Segment> subsegments;
    Segment curSegment;
    Change lastChange;
    for(Change change : segment)
    {
        if(lastChange != null
            && change.getPosition() - lastChange.getPosition() >= divideDistance)
        {
            subsegments.add(curSegment);
            curSegment = new Segment();
        }
        curSegment.add(change);
    }
    if(curSegment.size() != 0)
        subsegments.add(curSegment);
}

```

```

segment = new Segment();
//Add all Changes of subsegments that are > minSize
for(Segment subsegment : subsegments)
{
    if(subsegment.size() >= minSize)
        segment.addAll(subsegment.getChanges());
}
return segment;
}

```

### 9.8.8.3 Short-Lifespan Inclusion Pseudo-Algorithm

Vector <Change> shortLifespanInclusion(Segment segment, **int** maxLifespan, **int** maxDistance)

```

{
    Set <Change> shortLifespanChanges = ();

    //Retrieve the Line Histories of all lines modified within maxDistance of one of the
    //segment's Changes
    Set <HistoryLine> proxLineHistories = ();
    for(Change change : segment.getChanges())
    {
        for(int i = 1; i <= maxDistance; i++)
        {
            proxLineHistories.add(ProjectHistory.getChange(change.getPosition() +
i).getModifiedLineHistories());
            proxLineHistories.add(ProjectHistory.getChange(change.getPosition() -
i).getModifiedLineHistories());
        }
    }

    //For each proximate Line History, see whether its lifespan is <= maxLifespan.
    //If it is, include all of its Changes. Otherwise, ignore it.
    for(LineHistory lh : proxLineHistories)
    {
        if(lh.getLifespan() <= maxLifespan)
        {
            for(Modification m : lh.getModifications())

```



```

        shortLifespanChanges.addAll(m.getChange());
    }
}

return shortLifespanChanges;
}

```

#### 9.8.8.4 Line History Friend Pseudo-Algorithm

Set <Change> findFriends(Segment segment, **int** maxDistance, **float** friendBoundary)

```

{
    Set <Change> friendChanges = change.getModifiedLineHistories();

    //
    Map <LineHistory, Set <Modification>> friendlyLHs = ();
    for(Change change : segment.getChanges())
    {
        for(int i = 1; i <= maxChangeDistance; i++)
        {
            Change other = ProjectHistory.getChange(change.getPosition + i);

            for(Line line : other.getModifiedLines())
            {
                friendlyLHs.get(line.getLineHistory()).add(line.getModification());
            }
        }
    }
}

```

```

//For all LineHistories in proximity, see whether the percentage of
//Modifications in range of Changes in the segment exceeds friendBoundary%
//of the LineHistory's modifications
for(LineHistory lineHistory : friendlyLHs.keySet())
{
    int markedModifications = friendlyLHs.get(lineHistory).size();
    int totalModifications = lineHistory.getModifications().size();

    if(markedModifications / totalModifications >= friendBoundary)
    {

```

```

        for(Modification modification : lineHistory.getModifications())
        {
            friendChanges.add(modification.getChange());
        }
    }
}

return similarTextChanges;
}

```

#### 9.8.8.5 Line Proximity Pseudo-Algorithm

```

boolean inProximity(int maxLineDistance, Change change, Change other)
{
    Set <LineHistory> changeLHs = change.getModifiedLineHistories();
    Set <ChangeLine> otherLines = change.getModifiedLines();

    for(ChangeLine otherLine : otherLines)
    {
        for(int i = 1; i < maxLineDistance; i++)
        {
            LineHistory lh
                = other.getLine(otherLine.getLineNr() + i).getLineHistory();
            if(changeLHs.contains(lh))
                return true;
            lh = other.getLine(otherLine.getLineNr() - i).getLineHistory();
            if(changeLHs.contains(lh))
                return true;
        }
    }
}

Vector <Change> lineProximity(
    Segment segment, int maxChangeDistance, int maxLineDistance)
{
    Set <Change> proximityChanges = ();
    for(Change change : segment.getChanges())
    {

```

```

    for(int i = 1; i <= maxChangeDistance; i++)
    {
        Change other = ProjectHistory.getChange(change.getPosition + i);
        if(change.getFile() != other.getFile())
            continue;
        if(inProximity(change, other))
            proximityChanges.addAll(other);
    }
}

return proximityChanges;
}

```

#### 9.8.8.6 Code Parsing Pseudo-Algorithm

```

Set <String> visit(Set <Integer> modifiedLineNrs, ASTNode node)
{
    Set <String> found = ();
    if(node is FunctionCall || node is Variable || node is FunctionDeclaration)
    {
        if(modifiedLineNrs.contains(node.getLineNr()))
        {
            found.add(node.getIdentifierName());
        }
    }
    found.addAll(visit(node.getChildren()));
    return found;
}

```

```

Set <String> getModifiedIdentifiers(Change change)
{
    ASTNode root = CPPParser.parse(change.getSourceText());
    Set <Integer> modifiedLineNumbers = ();
    for(Line line : change.getModifiedLines())
        modifiedLineNumbers.add(line.getLineNumber());
    Set <String> identifiers = visit(modifiedLineNumbers, root);
    return identifiers;
}

```

```

Set <Change> similarParse(int maxDistance)
{
    Set <Change> parseChanges = change.getModifiedLineHistories();

    Set <Change> proximityChanges = ();
    for(Change change : segment.getChanges())
    {
        Set <String> changeIdentifiers = getModifiedIdentifiers(change);
        for(int i = 1; i <= maxChangeDistance; i++)
        {
            Change other = ProjectHistory.getChange(change.getPosition + i);
            Set <String> otherChangeIdentifiers = getModifiedIdentifiers(others);
            if(changeIdentifiers.hasOverlap(otherChangeIdentifiers))
            {
                parseChanges.add(other);
            }
        }
    }

    return parseChanges;
}

```

#### 9.8.8.7 Text Similarity Pseudo-Algorithm

```

boolean hasSimilarModifiedLines(Set <String> linesA, Set <String> linesB)
{
    for(String lineA : linesA)
    {
        for(String lineB : linesB)
        {
            float levDist = calculateLevenstheinDistance(lineA, lineB);
            if(levDist <= levDistBoundary)
                return true;
        }
    }
    return false;
}

```

```

Set <String> getChangedLinesText(Change change)
{
    Set <String> changedLinesText = ();
    for(Line line : change.getModifiedLines())
    {
        mchangedLinesText.add(line.getText());
    }
    return changedLinesText;
}

```

```

Set <Change> similarText(int maxDistance, float levDistBoundary)
{
    Set <Change> similarTextChanges = change.getModifiedLineHistories();

    for(Change change : segment.getChanges())
    {
        Set <String> changedLinesText = getChangedLinesText(change);
        for(int i = 1; i <= maxChangeDistance; i++)
        {
            Change other = ProjectHistory.getChange(change.getPosition + i);
            Set <String> otherChangedLinesText = getChangedLinesText(other);
            if(hasSimilarModifiedLines(
                levDistBoundary, changedLinesText, otherChangedLinesText))
                similarTextChanges.add(other);
        }
    }

    return similarTextChanges;
}

```

#### 9.8.8.8 SimProx Pseudo-Algorithm

```

boolean inProximity(int maxLineDistance, Change change, Change other)
{
    Set <LineHistory> changeLHs = change.getModifiedLineHistories();
    Set <ChangeLine> otherLines = change.getModifiedLines();
}

```

```

for(ChangeLine otherLine : otherLines)
{
    for(int i = 1; i < maxLineDistance; i++)
    {
        LineHistory lh
            = other.getLine(otherLine.getLineNr() + i).getLineHistory();
        if(changeLHs.contains(lh))
            return true;
        lh = other.getLine(otherLine.getLineNr() - i).getLineHistory();
        if(changeLHs.contains(lh))
            return true;
    }
}

```

```

Vector <Change> lineProximity(
    Segment segment, int maxChangeDistance, int maxLineDistance)
{
    Set <Change> proximityChanges = ();
    for(Change change : segment.getChanges())
    {
        for(int i = 1; i <= maxChangeDistance; i++)
        {
            Change other = ProjectHistory.getChange(change.getPosition + i);
            if(inProximity(change, other))
                proximityChanges.addAll(other);
        }
    }

    return proximityChanges;
}

```

```

boolean hasSimilarModifiedLines(float levDistBoundary, Set <String> linesA, Set
<String> linesB)
{
    for(String lineA : linesA)
    {

```

```

        for(String lineB : linesB)
        {
            float levDist = calculateLevenstheinDistance(lineA, lineB);
            if(levDist <= levDistBoundary)
                return true;
        }
    }
    return false;
}

```

```

Set <String> getChangedLinesText(Change change)
{
    Set <String> changedLinesText = ();
    for(Line line : change.getModifiedLines())
    {
        mchangedLinesText.add(line.getText());
    }
    return changedLinesText;
}

```

```

Set <Change> getSimilarProximity(
    int maxDistance, int maxLineDistance, float levDistBoundary)
{
    Set <Change> simProxChanges = change.getModifiedLineHistories();

    for(Change change : segment.getChanges())
    {
        Set <String> changedLinesText = getChangedLinesText(change);
        int maxDistance = maxChangeDistance;
        boolean foundInLast = false;
        int i = 1;
        while(i < maxDistance || (keepAlive && foundInLast))
        {
            foundInLast = false;

            Change other = ProjectHistory.getChange(change.getPosition + i);
            Set <String> otherChangedLinesText = getChangedLinesText(other);

```

```

        if(hasSimilarModifiedLines(
            levDistBoundary, changedLinesText, otherChangedLinesText))
            simProxChanges.add(other);
            foundInLast = true;
            if(resetMaxDistance)
                maxDistance = i + maxChangeDistance;
        }
        else if(inProximity(maxLineDistance, change, other))
        {
            simProxChanges.addAll(other);
            foundInLast = true;
            if(resetMaxDistance)
                maxDistance = i + maxChangeDistance;
        }
        i++;
    }
}

return similarTextChanges;
}

```

### 9.8.8.9 Visit Expressions Pseudo-Algorithm

(modifications to the original LH-Graph algorithm are in **BOLD ORANGE**)

```

Set <Change> getSegmentNodes(
    Change change, int adj, int maxDist,
    DocumentExpression isValidDocument,
    LineExpression isValidLine,
    ModificationExpression isValidModification
)
{
    Set <Change> adjacents = ();
    Stack <Change> toVisit = (change);

    for(int i = 0; i < adj; i++)
    {
        Stack <Change> newToVisit = (change);

        for(Change curChange : toVisit)
        {
            if(adjacents.contains(curChange))
            {
                continue;
            }

```



```

        //Test whether the document should be visited according to the
        //isValidDocument expression
        if(isValidDocument.execute(curChange))
            continue;
        Set <Change> adjs
            = getAdjacents(curChange, maxDist, ChangeLineFlags, LineHistoryFlags);
        newToVisit.addAll(adjs);
        adjacents.addAll(adjs);
    }
    toVisit = newToVisit;
}

return adjacents;
}

Set <Change> getAdjacents(Change change, int maxDist, int ChangeLineFlags, int LineHistoryFlags)
{
    Set <Change> adjs = ();
    for(LineHistory line : change.getLines())
    {
        //Test whether the line should be visited according to the isValidLine expression
        if(isValidLine.execute(line))
        {
            for(Modification modification : line.getModifications())
            {
                int dist = Math.abs(
change.getPosition() - modification.getChange().getPosition());

                //Test whether distance to modification's Change is <= maxDist
                and also test whether it passes the isValidModification expression
                if(dist < maxDist &&
                    isValidModification.execute(modification))
                {
                    adjs.add(modification.getChange());
                }
            }
        }
    }
    return adjs;
}

```

### 9.8.9 Evaluation of LH-Graph Extension Algorithms

This section presents results from the evaluation of sub-algorithms designed to improve the performance of the LH-Graph algorithm. As described in Section 7.6.5 these sub-algorithms receive as input segments generated by the core LH-Graph algorithm and then either filter the segment to remove unrelated Changes or attempt to find additional related Changes to include in the segment.

For each algorithm a description of the algorithm and its rationale is followed by details of the algorithm's parameters and the algorithm's pseudo-code. This is followed by the evaluation of the algorithm, with the results of different parameter settings being compared to the results as produced by the core LH-Graph algorithm without the extension algorithm applied.

### 9.8.9.1 Compile Filter Algorithm

Table 142 presents results for use of the compile filter; one run is performed without the filter, one run including only compiled Changes, one run including only non-compiled Changes and one run using the compile filter in adaptive mode.

**Table 142: Overall Summary of Compile-filter algorithm runs**

Setting	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.543	63.5%	0.529	0.007	10	8
Compile	1.617	67.1%	0.476	0.005	10	8
NonCompile	0.852	36.0%	0.125	0.717	0	0
Adaptive	1.643	67.8%	0.515	0.003	10	9

The non-filtered run produces the best spread\*fit ratio. However, both the compile-only and adaptive filters produce better A/E ratios. The adaptive filter produces both a higher spread\*fit value and a higher A/E ratio than the compile-only filter, and also produces the best average p-value. It outperforms the non-filtered run by 10% in terms of A/E ratio. The non-compile only run produces a result that is significantly worse than random. This suggests that fewer non-compiling than compiling Changes are interesting.

Overall, the adaptive compile filter produces significant gains of A/E ratio. Since it produces such good results, the compile filter is used in all other LH-Graph related evaluation; it is always set to adaptive filtering.

### 9.8.9.2 Small-Segment Filter algorithm

Table 143 shows runs for the small-segment filter algorithm utilising division values of (5, 10, 25) and minimum size accept values of (2, 5, 10).

**Table 143: Overall Summary of Small-Segment filter algorithm runs**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.643	67.8%	0.515	0.003	10	9
div=5, acc=2	1.632	67.4%	0.524	0.003	10	9
div=5, acc=5	1.586	65.8%	0.456	0.007	10	8
div=10, acc=2	1.647	67.9%	0.521	0.003	10	9
div=10, acc=5	1.615	66.5%	0.472	0.004	10	9
div=10, acc=10	1.593	66.5%	0.366	0.009	9	8
div=25, acc=2	1.64	67.7%	0.518	0.004	10	9
div=25, acc=5	1.619	66.9%	0.473	0.003	10	9
div=25, acc=10	1.572	65.6%	0.379	0.007	10	8

While the setting (div=10, acc=2) appears to improve A/E ratio and spread\*fit compared to the no-filter run, the improvement is very small and unlikely to represent a significant improvement. The runs using minimum accept sizes of 5 or 10 all perform much more poorly than no filtering at all.

The algorithm does not appear to produce any significant improvement in either identification of interesting Changes or in the grouping of those Changes. This suggests that small outliers are not a significant problem for the core LH-Graph algorithm. However, while it does not produce improvements in isolation, it is possible that it may produce improvements when combined with algorithms which for whatever reason create small outliers.

### 9.8.9.3 Short-Lifespan Inclusion Algorithm

Table 144 shows runs of the Short-Lifespan inclusion algorithm using maximum distance settings of (2,5,10) and lifetime boundaries of (2,5,10). None of the runs performs significantly better than the run not utilising the algorithm. The runs using lifespans larger than span=2 tend to perform slightly worse.

**Table 144: Overall Summary of Short-Lifespan inclusion algorithm runs**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
none	1.643	67.8%	0.515	0.003	10	9
dist=2, size=2	1.647	67.9%	0.517	0.003	10	9
dist=2, size=5	1.632	67.5%	0.517	0.003	10	9
dist=2, size=10	1.64	67.7%	0.518	0.003	10	9
dist=5, size=2	1.639	67.7%	0.515	0.003	10	9
dist=5, size=5	1.625	67.2%	0.514	0.004	10	9
dist=5, size=10	1.635	67.4%	0.507	0.004	10	9
dist=10, size=2	1.643	67.8%	0.51	0.003	10	9
dist=10, size=5	1.632	67.3%	0.501	0.004	10	9
dist=10, size=10	1.64	67.5%	0.493	0.005	10	9

The algorithm does not appear to improve on the core LH-Graph algorithm, so it seems that identification of lines with short lifespans does not help identify problems which involve the addition and deletion of large numbers of lines rather than their modification. This may be because this approach also includes too many incorrect Changes.

#### 9.8.9.4 Line-History Friend Algorithm

The result of runs of the Friend algorithm with search distances of (2,5,10) and selection boundaries of (0.5, 0.75, 1.0) are shown in Table 145.

**Table 145: Overall Summary of Friend algorithm runs**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.643	67.8%	0.515	0.003	5	9
dist=2, bdr=0.5	1.627	66.9%	0.527	0.008	9	9

dist=2, bdr=0.75	1.628	66.9%	0.525	0.008	9	9
dist=2, bdr=1.0	1.643	67.8%	0.515	0.003	10	9
dist=5, bdr=0.5	1.582	65.5%	0.519	0.012	9	8
dist=5, bdr=0.75	1.558	64.9%	0.519	0.015	9	7
dist=5, bdr=1.0	1.643	67.8%	0.515	0.003	10	9
dist=10, bdr=0.5	1.607	65.9%	0.479	0.008	9	9
dist=10, bdr=0.75	1.545	64.2%	0.482	0.015	9	8
dist=10, bdr=1.0	1.643	67.8%	0.515	0.003	10	9

While two runs with a small distance (dist=2, bdr=0.5) and (dist=2, bdr=0.75) slightly improve the spread\*fit ratio, they also decrease the A/E ratio. In fact, all runs either maintain or decrease the A/E ratio, with runs with larger distance values leading to larger drops in A/E ratio.

This result suggests that the Line-History Friend algorithm in its current form is not a useful extension to the core LH-Graph algorithm.

#### 9.8.9.5 Line Proximity Algorithm

Table 146 shows results for the Line Proximity algorithm using search distances of (2,5,10) and line proximities of (2,5,10).

**Table 146: Overall Summary of Line Proximity algorithm runs**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.643	0.678	0.515	0.003	10	9
dist=2, lines=2	1.643	0.677	0.528	0.003	10	9
dist=5, lines=2	1.656	0.68	0.537	0.003	10	9
dist=10, lines=2	1.67	0.685	0.536	0.002	10	9
dist=2, lines=5	1.645	0.677	0.529	0.003	10	9

dist=5, lines=5	1.686	0.689	0.536	0.002	10	9
dist=10, lines=5	1.714	0.698	0.535	0.002	10	9
dist=2, lines=10	1.638	0.674	0.531	0.003	10	9
dist=5, lines=10	1.669	0.684	0.536	0.002	10	9
dist=10, lines=10	1.713	0.7	0.534	0.001	10	10

Smaller distance and line proximity values produce moderate improvements. Larger values provide significant improvements, with (dist=10, lines=5) and (dist=10, lines=10) providing the largest improvement on the base result. The A/E ratio is improved by about 7% and the spread\*fit metric is also slightly improved. The setting (dist=10, lines=10) also produces significant results at the  $p < 0.01$  level for all assignments, compared to the base result of significance of 9/10.

This result indicates that modification of proximate lines in nearby documents is a good way to identify Changes related to a segment and to thus improve the coverage of identified segments. While the quality of segments as measured by spread\*fit is increased slightly, the line proximity algorithm's most significant contribution is an increase in interesting Changes identified.

#### 9.8.9.6 Code Parsing Algorithm

Results from runs of the Code Parsing algorithm are presented in Table 147. Of the three runs using distances of (1, 2, 5), the two runs with smaller distances slightly decrease A/E ratio and slightly increase spread\*fit. The final run with a distance of 5 produces a small improvement in A/E ratio of ~2% and an improvement of spread\*fit of roughly the same amount.

**Table 147: Overall Summary of Code Parsing algorithm runs**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
noParse	1.643	67.8%	0.515	0.003	10	9
dist: 1	1.635	67.4%	0.526	0.003	10	9
dist: 2	1.64	67.5%	0.531	0.003	10	9
dist: 5	1.661	68.3%	0.536	0.002	10	9

While one of the Code Parsing algorithm's runs does improve on the core LH-Graph algorithm's result, the improvement is quite small. As such, the algorithm in its current form is not useful, at least with the data used in this evaluation.

#### 9.8.9.7 Text Similarity Algorithm

The Text Similarity algorithm is run with Levensthein distances of (0.1, 0.25, 0.5) and distances of (1,5,10,20). As shown in Table 148 the algorithm does not produce significant improvements with distances of 1 or 5. It produces a small improvement with (lev=0.25, dist=10) and (lev=0.5, dist=10). The best improvement of ~5% is produced with (lev=0.25, dist=20). This run also produces a small improvement in spread\*fit ratio and increase p<0.01 significance to 10/10 assignments (from 9/10 in the core LH-Graph run).

**Table 148: Overall Summary of Text Similarity algorithm runs**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.643	67.8%	0.515	0.003	10	9
lev=0.1; simDist=1	1.643	67.8%	0.519	0.003	10	9
lev=0.25; simDist=1	1.643	67.8%	0.521	0.003	10	9
lev=0.5; simDist=1	1.646	67.8%	0.522	0.003	10	9
lev=0.1; simDist=5	1.637	67.6%	0.525	0.004	10	9
lev=0.25; simDist=5	1.654	68.2%	0.531	0.002	10	9
lev=0.5; simDist=5	1.653	68.2%	0.528	0.002	10	9
lev=0.1; simDist=10	1.655	68.4%	0.527	0.004	10	9
lev=0.25; simDist=10	1.669	68.9%	0.524	0.002	10	9
lev=0.5; simDist=10	1.667	69.1%	0.527	0.003	10	8
lev=0.1; simDist=20	1.651	68.3%	0.521	0.004	10	8
lev=0.25; simDist=20	1.699	70.0%	0.522	0.001	10	10
lev=0.5; simDist=20	1.654	68.6%	0.523	0.003	10	8

The best run (lev=0.25, dist=20) does produce a significant moderate improvement, but the distance used is fairly large, considering that the algorithm is looking for textually similar lines in all of the Changes in this distance. This may have a large time-cost overhead if Changes have many modifications that need to be compared between (though running time was acceptable for the data used in the evaluation). Further evaluation is also needed with more data to verify whether such a large distance is useful in other assignment contexts as well.

As discussed earlier the Levenstein boundary used to detect Mutants is very lenient, since the lenient setting was found to produce the best performance in segment generation. The leniency of the Mutant detection algorithm may mask the performance of the Text Similarity algorithm. It is possible that a combination of less lenient Mutant generation with a more lenient Text Similarity algorithm run would provide the benefit of finding related lines while not incorrectly linking lines that are not true Mutants, thus increasing Mutant identification precision and recall.

#### 9.8.9.8 SimProx Algorithm

The results of runs of the SimProx algorithm without *keepAlive* active are presented Table 149. Results of the SimProx algorithm with *keepAlive* active are presented in Table 150. Settings using Levenstein distances of (0.1, 0.5, 0.75), line proximities of (5,10) and distances of (1, 5, 10, 15) were conducted. Only some of the runs are shown in the tables for the sake of brevity. Table 151 presents results of the SimProx algorithm with *keepAlive* and *adjacent search* active, utilising maximum adjacent search distances of (10, 25, 50) and (lev=0.75, Prox=10, Dist=1).

**Table 149: Overall Summary of SimProx algorithm runs not using keepAlive**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.643	67.8%	0.515	0.003	10	9
Lev=0.1, Prox=5, Dist=10	1.711	69.7%	0.536	0.002	10	9
Lev=0.1, Prox=10, Dist=10	1.706	69.7%	0.536	0.001	10	9
Lev=0.5, Prox=5, Dist=10	1.713	70.0%	0.539	0.001	10	9
Lev=0.5, Prox=10, Dist=10	1.687	69.2%	0.535	0.001	10	9
Lev=0.75, Prox=5, Dist=10	1.714	70.1%	0.533	0.002	10	9
Lev=0.75, Prox=10, Dist=10	1.688	69.2%	0.529	0.002	10	9



Lev=0.1, Prox=5, Dist=15	1.717	69.9%	0.536	0.003	10	9
Lev=0.1, Prox=10, Dist=15	1.703	69.5%	0.533	0.002	10	9
Lev=0.5, Prox=5, Dist=15	1.706	69.7%	0.535	0.002	10	9
Lev=0.5, Prox=10, Dist=15	1.696	69.4%	0.5306	0.002	10	9
Lev=0.75, Prox=5, Dist=15	1.678	68.6%	0.527	0.005	10	9
Lev=0.75, Prox=10, Dist=15	1.672	68.5%	0.521	0.005	10	9

**Table 150: Overall Summary of SimProx algorithm runs using keepAlive**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.643	67.8%	0.515	0.003	10	9
Lev=0.1, Prox=5, Dist=1	1.682	69.0%	0.533	0.003	10	9
Lev=0.1, Prox=10, Dist=1	1.702	69.8%	0.537	0.001	10	10
Lev=0.5, Prox=5, Dist=1	1.705	70.0%	0.534	0.001	10	10
Lev=0.5, Prox=10, Dist=1	1.718	70.4%	0.535	0.001	10	10
Lev=0.75, Prox=5, Dist=1	1.722	70.6%	0.533	0.001	10	10
Lev=0.75, Prox=10, Dist=1	1.734	70.9%	0.533	0.001	10	10
Lev=0.1, Prox=5, Dist=5	1.71	69.7%	0.538	0.002	10	9
Lev=0.1, Prox=10, Dist=5	1.712	70.0%	0.538	0.002	10	9
Lev=0.5, Prox=5, Dist=5	1.728	70.6%	0.54	0.001	10	10
Lev=0.5, Prox=10, Dist=5	1.7	69.7%	0.539	0.001	10	10
Lev=0.75, Prox=5, Dist=5	1.725	70.5%	0.538	0.001	10	10
Lev=0.75, Prox=10, Dist=5	1.705	70.0%	0.535	0.001	10	10

**Table 151: Overall Summary of SimProx algorithm runs using *keepAlive* and including proximity search for included Changes using the setting (Lev=0.75, Prox=10, Dist=1)**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.643	67.8%	0.515	0.003	10	9
KeepAlive Only	1.734	70.9%	0.533	0.001	10	10
AdjDist=10	1.748	71.6%	0.531	0.001	10	10
AdjDist=25	1.698	70.0%	0.521	0.001	10	10
AdjDist=50	1.677	69.0%	0.517	0.001	10	10

All of the runs of the algorithm without *keepAlive* improve on the standard LH-Graph result, with several providing large improvements in A/E ratio. The best runs (Lev=0.5, Prox=5, Dist=10), (Lev=0.75, Prox=5, Dist=10) and (Lev=0.1, Prox=5, Dist=15) produce improvements of A/E ratios of ~7% and produce significance for 10/10 assignments at the  $p < 0.01$  level, compared to 9/10 for the standard LH-Graph run. There is also a small improvement of fit\*spread ratio. Runs with search distances of 1 and 5 (not shown in the table) produce smaller improvements.

Use of the *keepAlive* setting provides even better results. The run (Lev=0.75, Prox=10, Dist=1) produces an improvement of A/E ratio of ~9%. Using the *keepAlive* setting, better results are achieved for lower distances (the best for the lowest distance). This indicates that the *keepAlive* setting works as intended, keeping alive the algorithm run for an optimal distance.

The final set of runs selecting Changes in proximity of selected Changes is presented in Table 151. The runs use the settings (Lev=0.75, Prox=10, Dist=1, *keepAlive*=true), with adjacent search distances of (10, 25, 50). The best setting is the one using the lowest adjacent search distance of 10. It produces an A/E improvement of ~10% which is slightly higher than the result achieved using *keepAlive* only. However, the improvement is only small which makes it impossible to propose which of the two may work better with a larger selection of data. The SimProx algorithm also outperforms both the Line Proximity algorithm and the Text Similarity algorithm which by themselves provide A/E improvements of ~7% and ~6% respectively.

In summary, the SimProx algorithm provides a significant improvement on the core LH-Graph algorithm.

### 9.8.9.9 Visit Expressions

Since there are many different possible expressions, a small number of expressions were crafted based on intuitions regarding the types of Changes that should be included or excluded during the adjacent Change search.

Two simple expressions were trialled. The first expression is a Document Expression; whenever a modified line is visited, a Document Expression decides whether the associated Line History should be visited. The Document Expression utilised visits Moved, Mutated, Ghost and Deleted lines as usual, but it also visits Added line if the total number of lines modified in the Change is  $< 5$ . The reasoning behind this is that very large numbers of added lines indicates the creation of a new function or data structure and navigating so many Line Histories will generate segments that are too large and combine different non-related segments, but smaller numbers of added lines are more likely to be part of an active problem-solving process. This expression is listed in the table as 'doc\_all  $\leq 5$ '.

The second expression visits only Changes that include less than 10 modifications, hence not visiting any lines from Changes with more than 10 modifications. The rationale is much the same as that behind the first expression, but rather than visiting more Line Histories suspected to be relevant as does the first expression, it makes the algorithm not visit Changes and their Line Histories if it suspects them to be less relevant due to the large number of Changes. The expression is listed in the table as line\_all  $\leq 10$ .

Table 152 lists results for runs using each expression individually, and one run using both expressions together. All runs improve the A/E ratio. However, the combined run improves the A/E ratio by the largest amount (~7%). It also improves the fit\*spread ratio slightly, and causes all assignment results to be relevant at the  $p < 0.01$  level.

**Table 152: Overall Summary of Expression extension runs**

	Ratio	% of Max	spread*fit	avg p-val	$< 0.05$	$< 0.01$
None	1.643	67.8%	0.515	0.003	10	9
line_all $\leq 5$	1.681	69.6%	0.524	0.002	10	10
doc_all $\leq 10$	1.664	68.1%	0.519	0.008	9	9
line_all $\leq 5$ , doc_all $\leq 10$	1.711	70.5%	0.532	0.001	10	10

The result shows that Visit Expressions have the potential to improve LH-Graph segment identification.

### 9.8.9.10 Combination of algorithms results

#### Description

Best results were achieved by the SimProx algorithm and the Visit Expressions extension to the LH-Graph algorithm. Improvements of ~10% and ~7% of A/E ratio respectively were achieved using these approaches.

Since these approaches were individually effective, both algorithms were combined to evaluate whether the combination would provide an even better improvement.

#### Evaluation

Table 153 presents results for the combination of algorithms. Items marked 'line' use a Line Expression of (line\_all <= 5). Items marked with 'doc' use a Document Expression of (line\_all <= 10). Items marked with 'keepAlive' use the SimProx algorithm using (Lev=0.75, Prox=10, Dist=1, keepAlive=true), while items marked 'prox' use the SimProx algorithm with the setting (Lev=0.75, Prox=10, Dist=1, keepAlive=true, adj=10).

**Table 153: Overall Summary of the combination of SimProx and Expression algorithms**

	Ratio	% of Max	spread*fit	avg p-val	<0.05	<0.01
None	1.643	67.8%	0.515	0.003	10	9
line + doc	1.711	70.5%	0.532	>0.001	10	10
keepAlive	1.734	70.9%	0.533	0.001	10	10
keepAlive + line	1.756	71.7%	0.54	>0.001	10	10
keepAlive + doc	1.697	69.4%	0.553	>0.001	10	10
keepAlive + line + doc	1.732	70.4%	0.559	>0.001	10	10
Prox	1.754	71.7%	0.532	0.001	10	10

prox + line	1.743	71.3%	0.537	0.001	10	10
prox + doc	1.688	69.3%	0.551	>0.001	10	10
prox + line + doc	1.721	70.1%	0.559	>0.001	10	10

The two best-performing runs are those using only the SimProx algorithm with adjacent search (Lev=0.75, Prox=10, Dist=1, keepAlive=true, adj=10) without any Visit Expressions and the run using keepAlive only (Lev=0.75, Prox=10, Dist=1, keepAlive=true) with the Line Expression. Both also improve the spread \* fit value.

While the combination of Visit Expression and SimProx algorithm does improve the result slightly when using only keepAlive, Visit Expressions reduce performance for the SimProx algorithm when combined with the addition of adjacent Changes. This may be because the SimProx algorithm finds most of the Changes that are included via the Line Expression and Changes excluded via the Document Expression hamper its performance by excluding Changes from which further relevant Changes could have been detected.

On balance it seems that the combination of the Visit Expressions evaluated in this chapter and the SimProx algorithm are ineffective in producing better performance, at least for the set of test data used in this evaluation.

## 9.9 SGL Parser

### 9.9.1 SGL Parser Manual

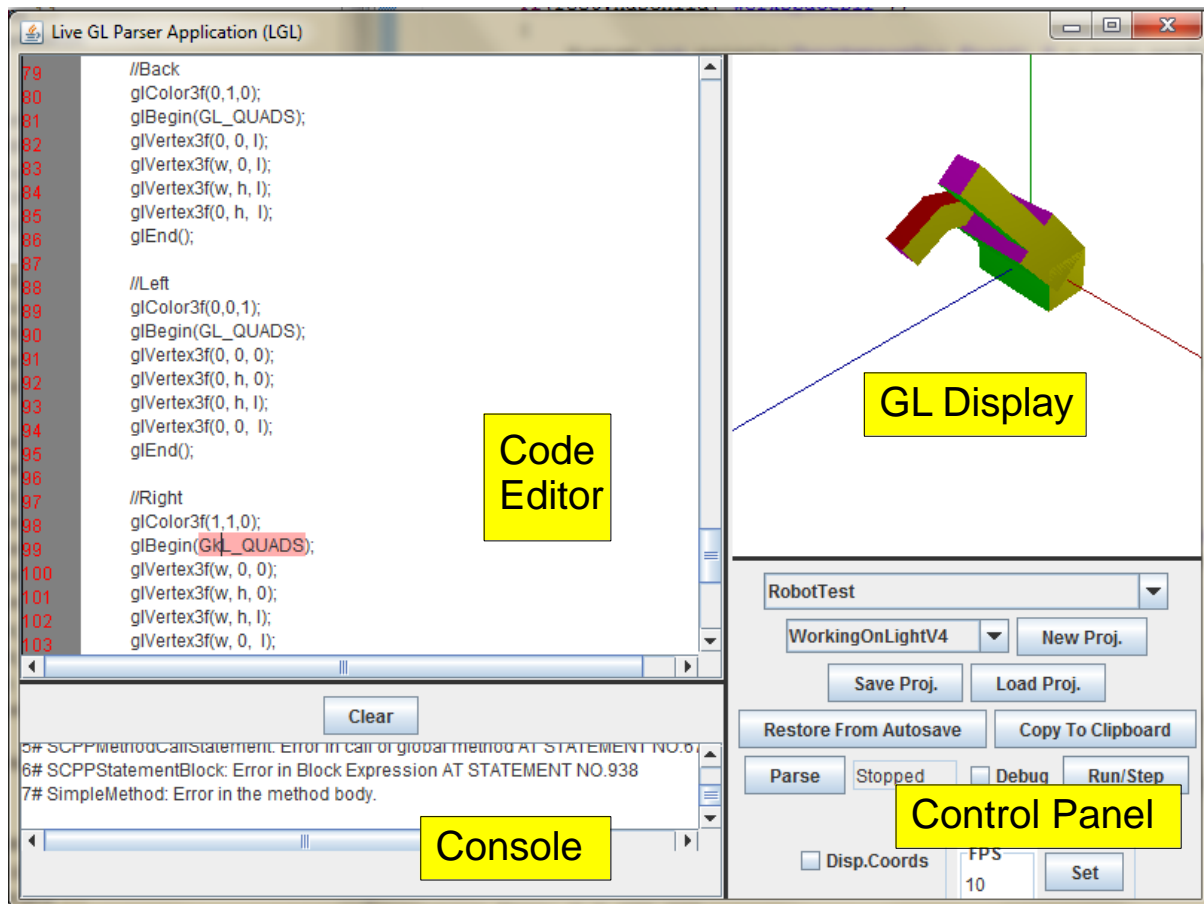


Figure 418: Main SGL Parser Interface

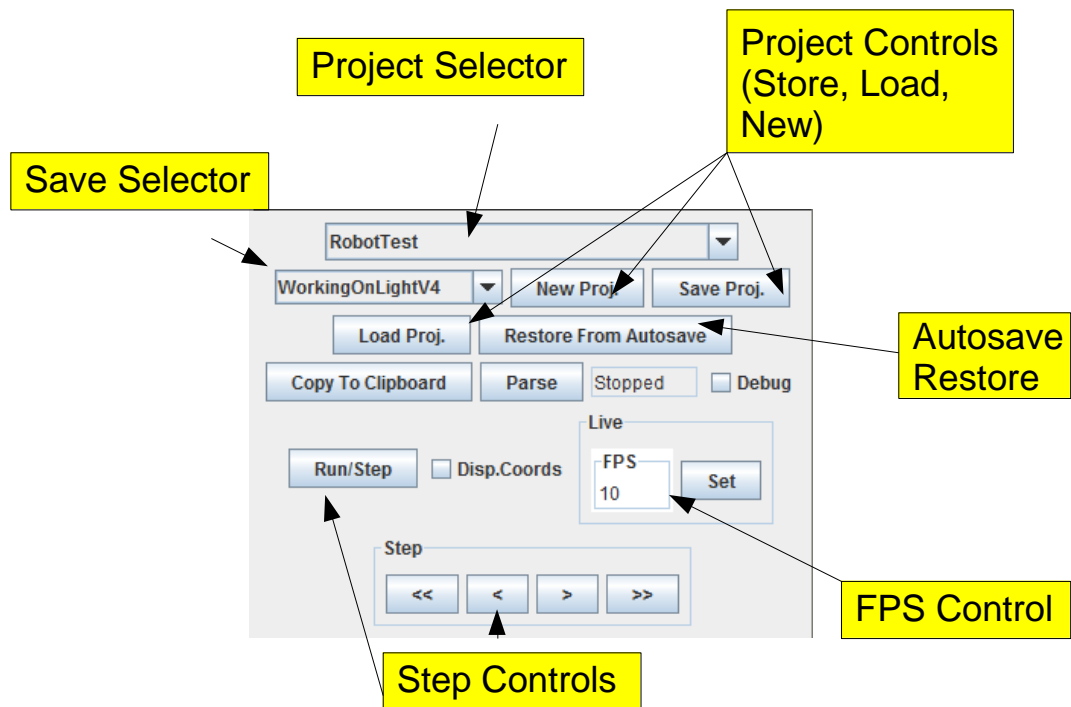


Figure 419: SGL Parser control panel

## Project Controls

-New Project: Creates a new project in your workspace directory (the one you specify when starting up the SGL parser)

-Project Combobox: Allows you to select a saved project. The project's versions will be used to populate the version combobox.

-Version Combobox: Allows you to select a saved version. Its code will automatically be loaded into the code window.

-Save: Saves the current code as a new version for the current project. You can enter the name of the version via the pop-up, and this name will then be used in the version combobox.

-Autosave Restore: Restores the code to the state it was in when "parse" was last pressed for this project. Use this when the parser crashes unexpectedly. Make sure NOT to press parse before pressing autosave, since that will overwrite the autosave. Save your code after using the restore function.

## Run/Step Controls

-Run/Step toggle: Switches between standard run mode and step-by-step execution

- "<<" (Back to Start): Move execution point back to start
- "<" (Back one step): Execute one less GL command
- ">" (Forward one step): Execute one more GL command
- "<<" (Forward to End): Move execution point to last GL command
- FPS field (only active when in run mode): How many times a second the "display" command is called

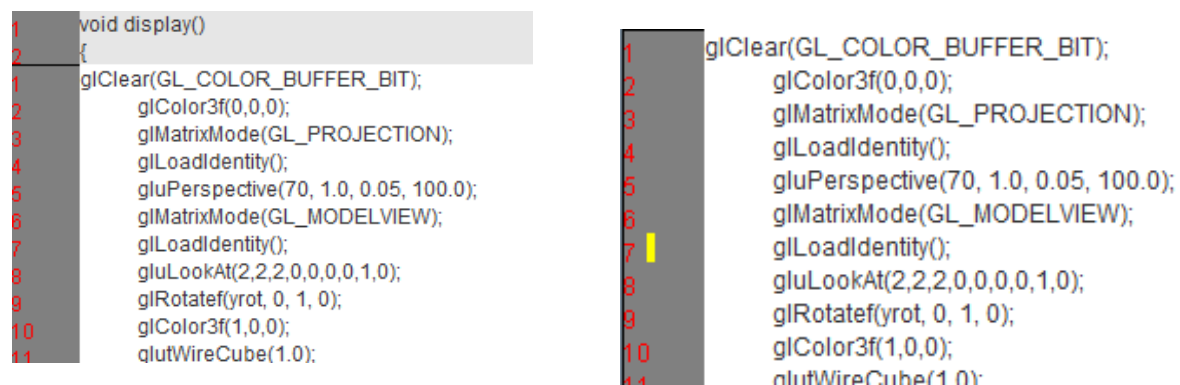
## Parse Control

-Parse button / hotkey: Attempts to run the code. Check console and code window for error messages or highlighting if nothing seems to happen.

### 9.9.1.1 Using the transformation axes functionality

The transformation axes functionality is designed to allow you to understand complex transformations by drawing the coordinate axes at any given point in your program.

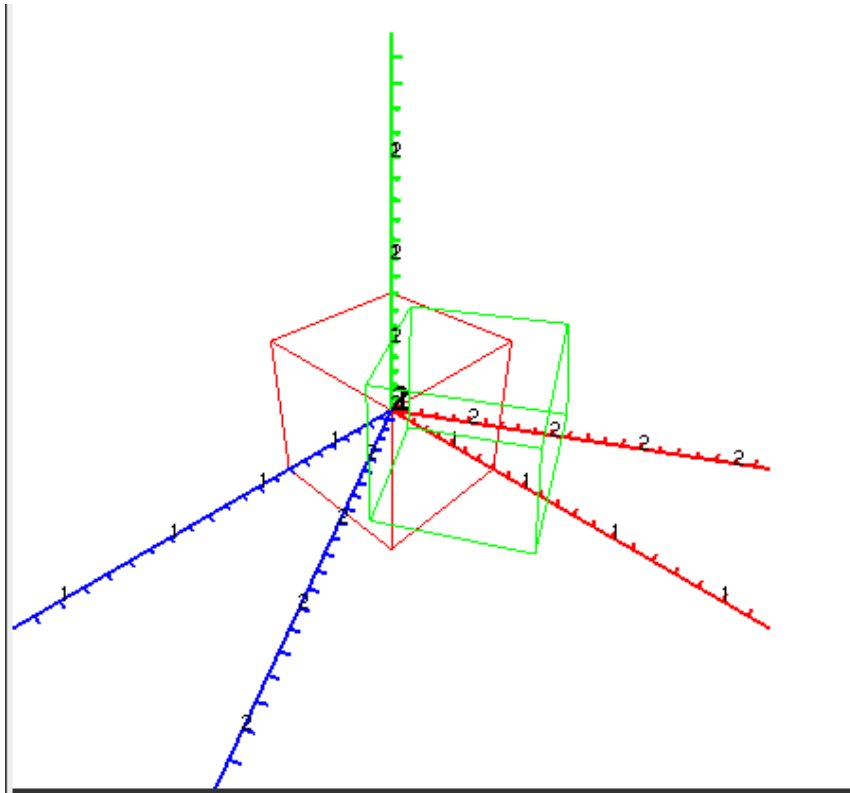
To have the coordinate axes for a given gl command displayed, click on the panel to the left of the code area



**Figure 420: Marking lines; unmarked on the left, marked with a yellow marker on the right**

A yellow marker rectangle should appear next to the command. You can place multiple markers. When you execute your code, the parser will draw coordination system lines pointing to the x-y-z directions AFTER each gl command that is marked (note it only works for OpenGL commands such as glTranslate, glutWireCube, etc..).





**Figure 421: The SGL Parser showing two transformation states of a cube**

You can remove individual markers in the same fashion as adding markers (click, then select toggle in the pop-up menu) or you can press the “Clear Markers” key on the control panel.

Pressing the “Clear Markers” key will remove all markers.

### 9.9.1.2 SGL Features:

#### 1) Instant Parse

Because the SGL parser parses your program without compiling it, it can run your programs almost instantly.

#### 2) Step-By-Step program execution

By pressing the “Step/Run” toggle button, you can set whether you want your program to execute normally or whether you want to run them step by step.

When being run step by step, the parser will execute one GL command each time you press the “step forward” button.

### 3) Transformation Coordinate Axes

You can have the parser draw x-y-z coordinate axes for you at one or several OpenGL commands. This can help you figure out whether you're setting up your transformations correctly.

#### 9.9.1.3 How SCPP is different from C++:

- no bitwise operators ( | or &)
- no compound operators (a += 5;), write it out instead (a = a + 5;)
- no increment / decrement (x++), write it out instead (x = x + 1;)
- no cout (cout << "Hello" << endl), use the printf function instead (printf("Hello");)
- no bracketless if statements [ if(something) printf("Hello") ] instead use bracketing [ if(something) { printf("Hello"); } ]

#### 9.9.1.4 How to debug your SCPP programs:

- errors are highlighted in red
- error messages appear in the console

### 9.9.2 Practical 8 Worksheet – Introduction to the SGLParser

Practical 7 (Week 8)

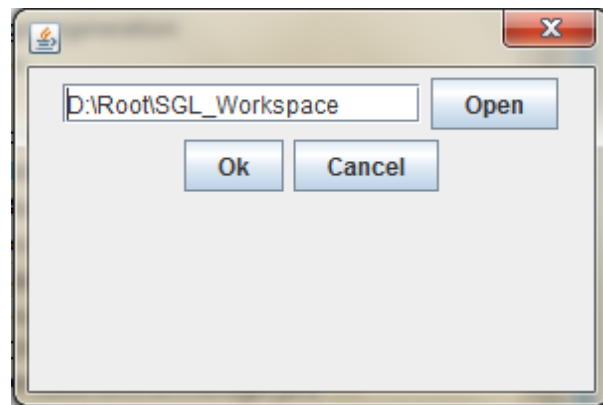
Try using the SCPP application by creating a new project and then running it. (If you run into problems, make sure you raise them during the week's prac/tutorial session).

-Download the application (from the support materials section). Unzip the zip file to a directory of your choice. If you're on a windows machine, it should work as-is.

Note: Linux and Mac users should download the appropriate package for their OS. I've tested the application using Linux, but not with Mac. Let me know if you run into problems.

-Start the application. You can either double-click on the .jar file, or you can run it on the command-line by navigation to the correct directory and typing “java -jar SCPPParser\_VXX.jar” (replace VXX with the version number). If you use the command line, you will get error messages through the console which is useful if anything goes wrong.

Note: If you're a Linux or Mac user, you need to use `java -jar -Djava.library.path=<The directory where you've unzipped the app to> SCPPParser_VXX.jar` instead to set the library path to find the library files bundled with the application. If this approach doesn't work (you get a “no gluegen” error or similar at the command prompt) try installing Jogl through your package management system.

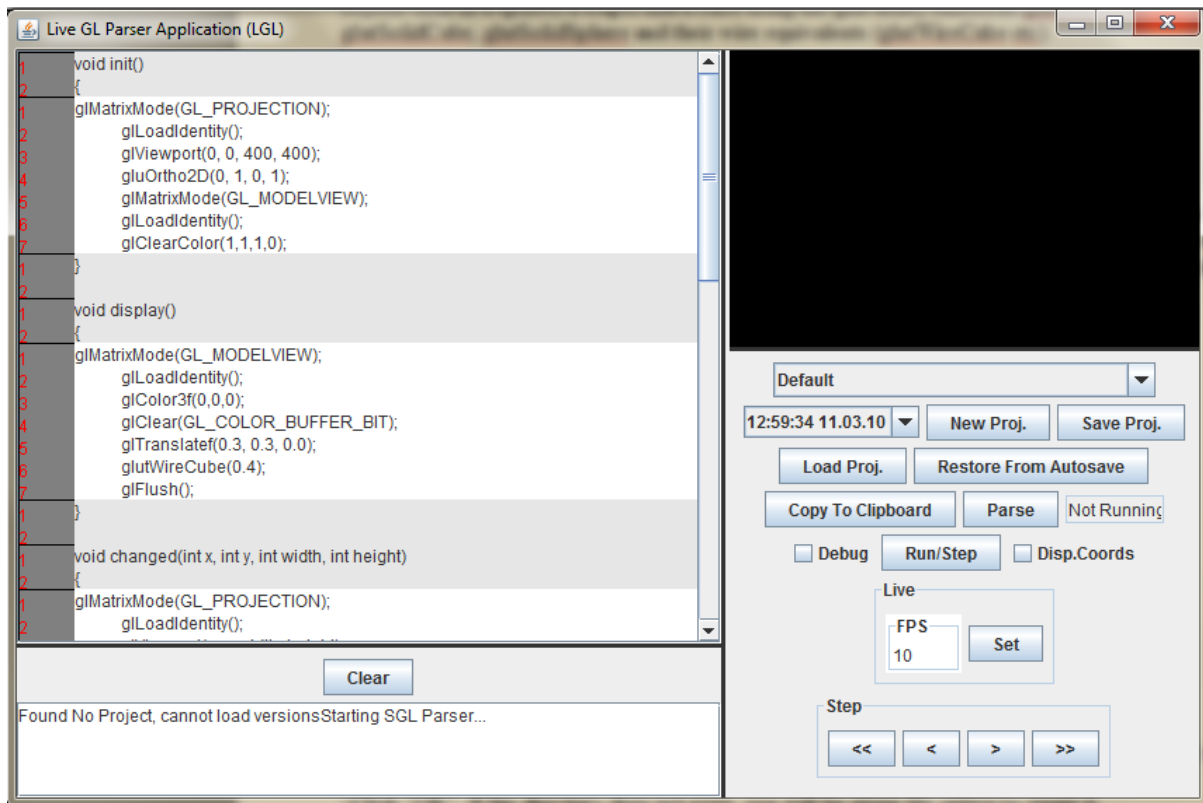


**Figure 422: The workspace selection dialog**

Pressing “Open” opens a file chooser dialog, while “OK” selects the directory. “Cancel” quits the application.

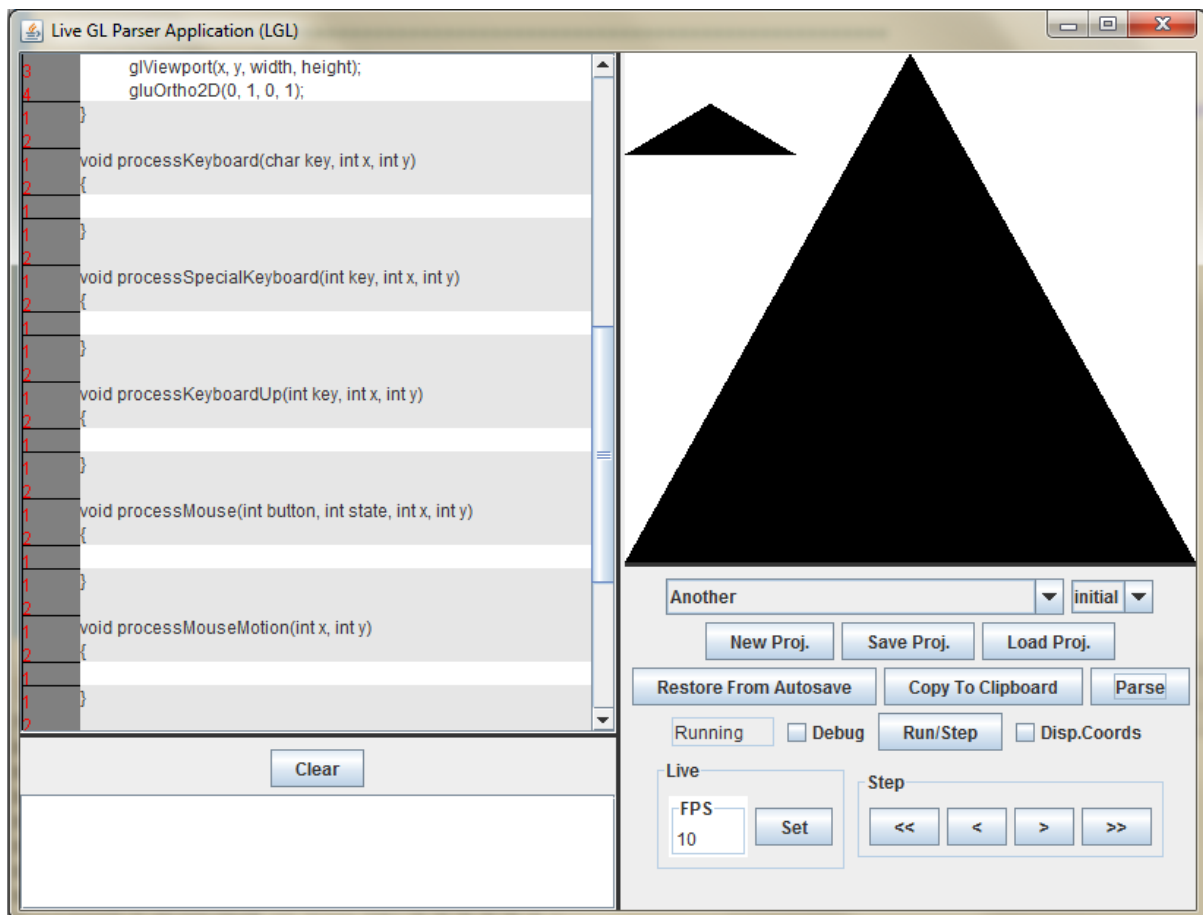
-Pick a directory as the workspace directory. This is similar to your eclipse workspace directory. The application will save projects to this directory. By default, the application comes bundled with a directory called “SQL\_Workspace” which already contains a sample project. Select this directory as the workspace directory.

-Click “OK”. If the directory does not exist, you will be given the option to create it.



**Figure 423: The main window for the SCPP application**

-You should now see the main window as in the screenshot above. Create a new project by pressing “New Proj.”. Enter a name. That name should then appear in the project box (the one containing “default” in the above screenshot). Press the “Parse” button. The black window should now change to show two triangles. Congratulations, you've run your first GL program with the SCPP parser.



**Figure 424: The SGL Parser running a simple program**

-You can save your progress by pressing the “Save Proj.” button. You will then be able to find the project in the “Project selection” combo box which in the above screenshots is filled with a time stamp number (the one below the project combo box). If you enter a name for your save, that is the name that will appear in the version combo box.

If you close the application, you can restore your save the next time by selecting the save from the combobox. Should the application crash, you can also restore your code from the last time you pressed the “parse” button or the parse hotkey by clicking the “Restore from Autosave” button. If the application crashes for some reason, restore from autosave and then save the result to ensure you don't lose your progress should the application crash again.

-To submit your work, just go to the workspace directory and zip the project in question. For this week, just submit the project you've just created. If you encounter difficulties with the application, you will still get a pass for the week's practical if you attend the practical session and work through the problems with me.

-More detail at the practical session

### 9.9.3 Practical 9 Worksheet – Transformations with the SGLParser

#### Step 0) Download

Note: Links to all required materials are also available from the “Practicals” webpage on the Comp330 WebCT website should the links in this document not work.

Download the SGL workspace which includes the project you will be working on here: [http://web.science.mq.edu.au/~mwittman/Comp330/prac9/Prac9\\_SGL\\_Workspace.zip](http://web.science.mq.edu.au/~mwittman/Comp330/prac9/Prac9_SGL_Workspace.zip) . Unzip it to a directory of your choosing.

Download the SGL parser to be used for this practical here: [http://www.comp.mq.edu.au/~mwittman/Comp330/prac9/SGLParser\\_A22.jar](http://www.comp.mq.edu.au/~mwittman/Comp330/prac9/SGLParser_A22.jar) . (If you are not using Windows, you will have to download the Linux or Mac JOGL libraries version 1.1.1a from <https://jogl.dev.java.net/servlets/ProjectDocumentList?folderID=11509&expandFolder=11509&folderID=11508> , unzip them and copy the contents of the “lib” folder [there should be several .jar and .so / .a files] into the same directory you unzip the parser to.)

#### Step 1) Start

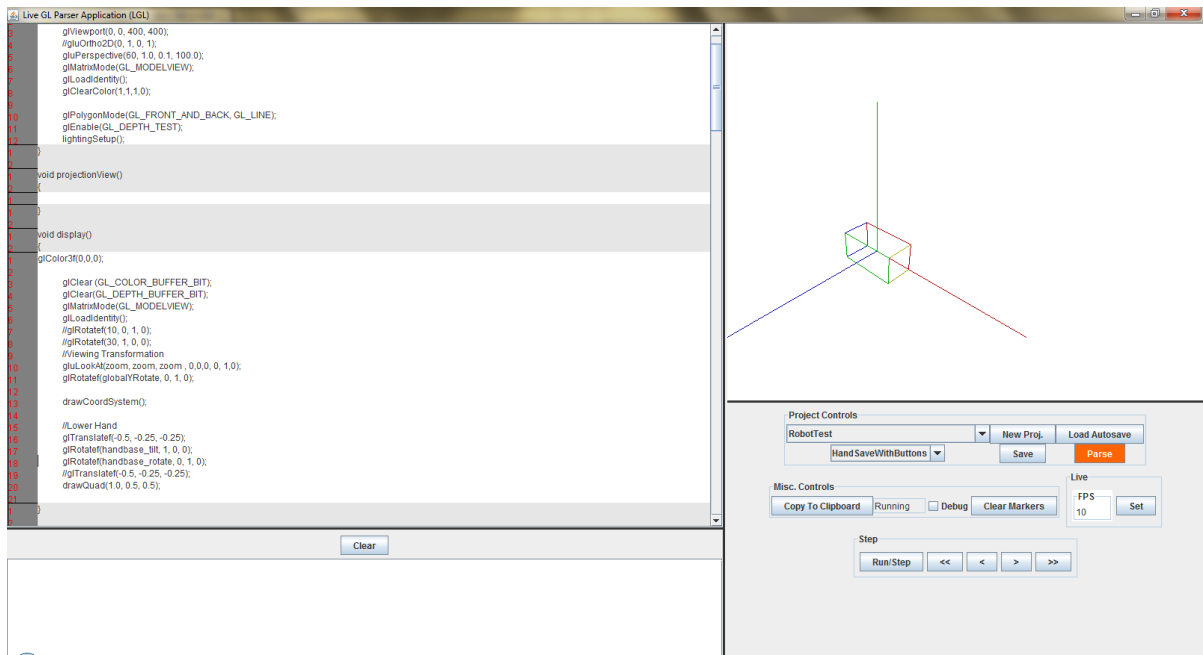
Start the application. You can either double-click on the .jar file, or you can run it on the command-line by navigation to the correct directory and typing “`java -jar SGLParser_A22.jar`” (replace VXX with the version number). If you use the command line, you will get error messages through the console which is useful if anything goes wrong.

Note: If you're a Linux or Mac user, you need to use `java -jar -Djava.library.path=<The directory where you've unzipped the app to> SGLParser_A22.jar` instead to set the library path to find the library files bundled with the application. If this approach doesn't work (you get a “no gluegen” error or similar at the command prompt) try installing Jogl through your package management system. If that still doesn't work, you'll need to use Windows for this and the next practical =( Please let me know about your problems though.

#### Step 2) Load

When the application starts up and opens the file text box, enter the path to the project folder you downloaded and unzipped in Step 0). Press OK.

#### Step 3) Successful Load



**Figure 425: Successfully opened workspace directory**

After opening the provided workspace, the hand project should be loaded (as it's the only project in the workspace). Press the “Parse” button. You should see the base of the hand. (See Figure 425)

```
void display()
{
    glColor3f(0,0,0);

    int handbase_rotate = 0;
    int handbase_tilt = 0;
    int upperhand_tilt = 40;
    int lowerfinger1_twist = 0;
    int lowerfinger1_tilt = 40;
    int upperfinger1_tilt = 40;
    int lowerfinger5_twist = 0;
    int lowerfinger5_tilt = 0;
    int upperfinger5_tilt = 20;
```

**Figure 426: Hand Angles**

#### Step 4) Viewing

In order to help you with this exercise, the 'a' and 's' keys have been set up to rotate the scene about its y-axis. This will allow you to gain a better understanding of what's going on.

In addition, notice the pre-defined angle values (See Figure 426). When changing these values, the fingers or hand parts should move appropriately (so for example, a finger should rotate around the

joint connecting it to the next finger, NOT around the centre of the hand). When your hand is completely coded, it should look like Figure 433. when using the original angle values.

### Step 5) Adding a transformation visualisation aid

Click on the grey panel to the left of the `glTranslatef(-0.5, -0.25, -0.25)` command. Select “Toggle Transform Grid” (See Figure 427). A yellow box should appear next to the command (See Illustration 4.). Press the “Parse” button (or press “F1”).

Now coordinate system lines showing the state of the coordinate system **after** the `glTranslate` call should appear (See Figure 428). You can see that the coordinate system is rooted at the base of the hand after the `glTranslate` call.



Figure 427: A transformation coordinate line marker once it's been placed.



Figure 428: Toggling coordinate lines for an OpenGL command

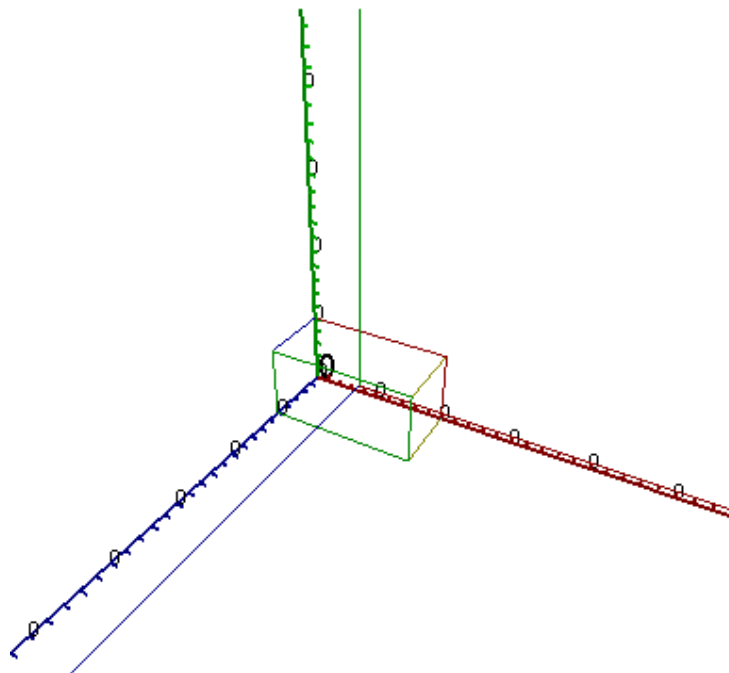


Figure 429: The state of the local coordinate system at the last transformation command for the lower hand. Notice the coordinate system has been transformed to the left of the origin.



```

15
16 //Lower Hand
17 glTranslatef(-0.5, -0.25, -0.25);
18 glRotatef(handbase_tilt, 1, 0, 0);
19 glRotatef(handbase_rotate, 0, 1, 0);
20 drawQuad(1.0, 0.5, 0.5);
21
22 //Upper Hand
23 glTranslatef(0.0, 0.5, 0.0);
24 glRotatef(upperhand_tilt, 1, 0, 0);
25 drawQuad(1.0, 0.75, 0.5);

```

**Figure 430: The marker from the lower hand has been removed, and a marker has been added to the last rotate command of the upper hand.**

### Step 6) Adding the next part of the hand

Now add the code from Figure 430 (also shown below)

```

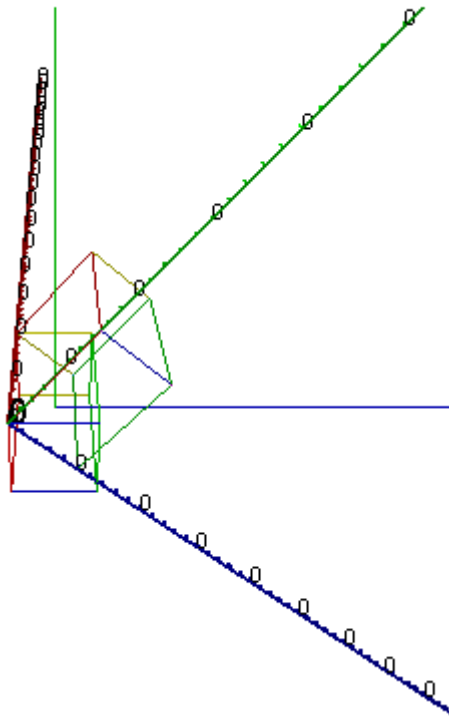
//Upper
glTranslatef(0.0, 0.5, 0.0);
glRotatef(upperhand_tilt, 1, 0, 0);
drawQuad(1.0, 0.75, 0.5);

```

Now remove the first marker you placed (by clicking on it and selecting "Toggle Transform Grid"). Then add a new marker to the glRotatef call (see Figure 430).

Press the parse button or F1. You should now see new coordinate system lines as in Figure 431.

As you can see, the coordinate system is now rotated about the global x axis.



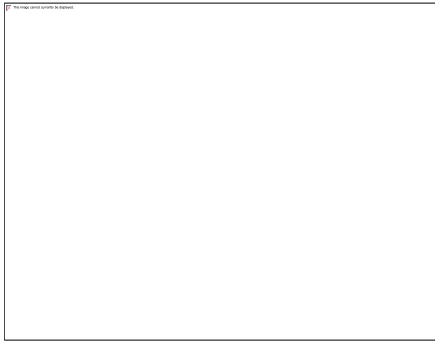
**Figure 431: The local coordinate system axes for the upper hand's `glRotatef` command. Notice it has been rotated about the x axis.**

### **Step 7) Complete coding the hand**

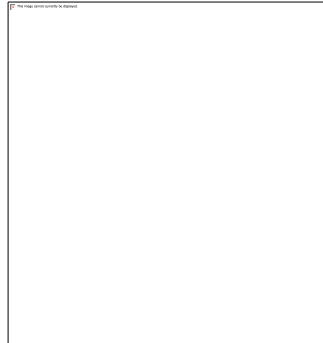
Continue coding your hand until it resembles the hand in Illustrations Figure 433-Figure 432. Add one finger-bone at a time, ensuring you transform it correctly so that it joins the hand.

Notice that your final hand need have only two fingers (one on the left, one on the right of the hand).

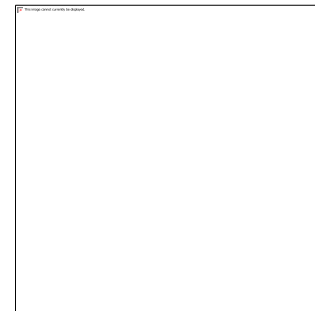
As you are working on your hand, add markers to help you see whether the real grid is currently located. Remember to save often.



**Figure 433: How your final hand should look (side view).**



**Figure 434: The completed hand from behind.**



**Figure 432: The completed hand from the front.**

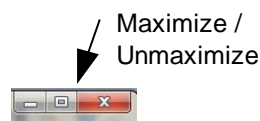
### Step 8) Submission

Once your hand is finished and looks like the hand in Figure 433-Figure 432, zip the workspace directory and submit it as your practical exercises. Also submit the Reflection Questions which follow on the next page, either electronically or on paper.

## 9.9.4 Practical 10 Worksheet – Projections with the SGLParser

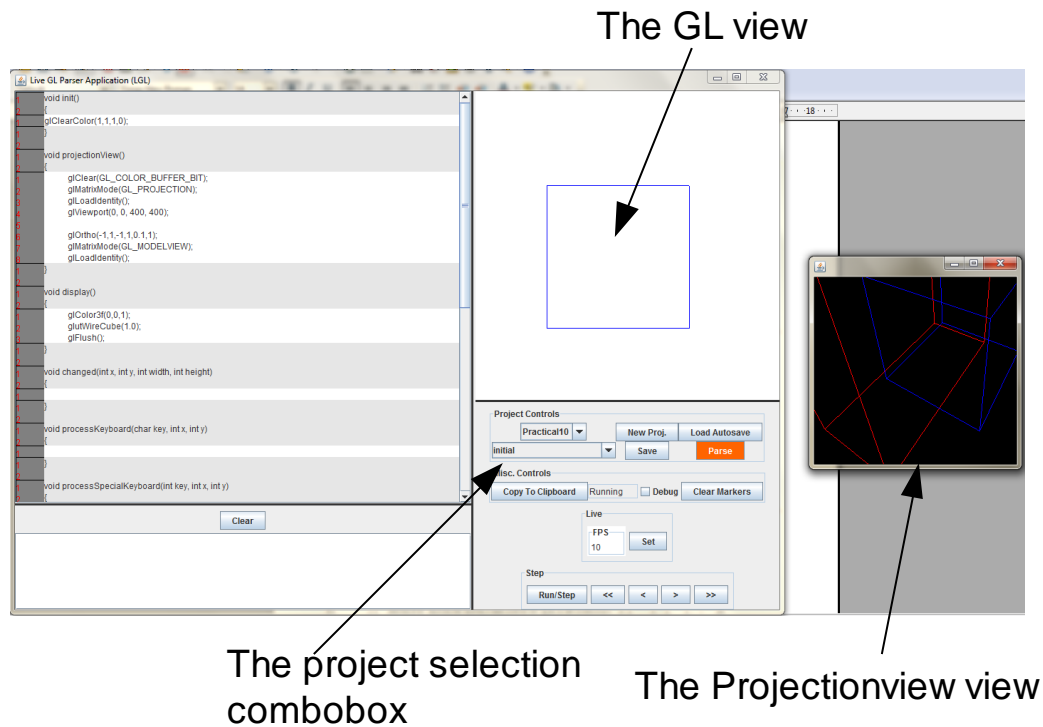
### Getting started )

To get started, download the Practical 10 workspace from the Practicals website and load it; if you're unsure of how to do this, refer to last week's Practical worksheet. DO NOT close the (initially black) window that opens when you start the parser. This is the projectview window (see Figure 436 right) which provides a view of the viewing volume. If you close it, you will need to restart the parser to get it back.



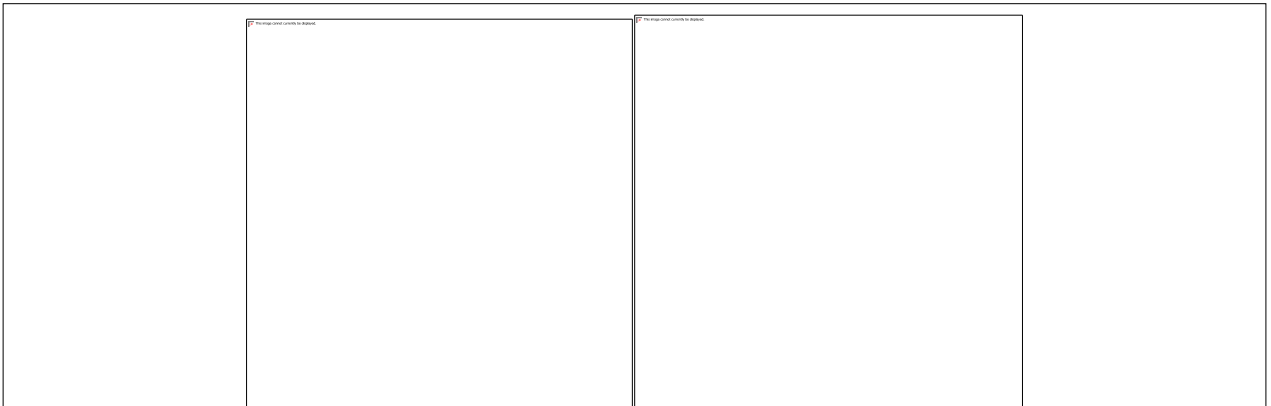
**Figure 435: Unmaximize**

Next, select Q1\_Project from the projects combobox. Then press the "Parse" button. Next, click on the "Maximize/Unmaximize" button (the little box at the top-right hand side of the window, Figure 435) to make the parser's window not take up the entire screen. Select the other window that was opened and arrange both windows so you can see both their contents like in Figure 436.

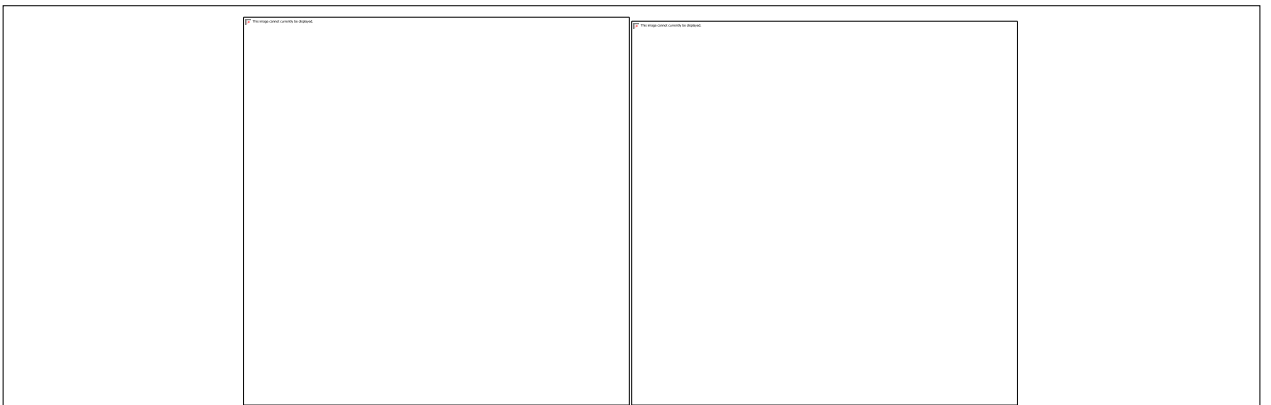


**Figure 436: The SGL Parser interface**

Now click on the smaller window (called the projectionview window, on the RHS in Figure 436). This window shows you the view volume as well as the scene content. Press the a-d keys to rotate the projectionview window's view (Figure 437), and press w-s to zoom-unzoom the projectionview window's view (Figure 438).



**Figure 437: Zooming in and out using the w-s keys. The viewing volume is a green pyramid from the eye to the near plane (items in this part of the viewing volume would NOT be visible) and a blue pyramid from the near to the far plane.**



**Figure 438: Rotating using the a-d keys**

### Question 1)

Load the project Q1\_Project by selecting it from the projects combobox.

**The Scene:** In the display function, you will see the scene's contents defined by:

```
glTranslatef(0.5,0.5,-0.5);  
  
glutWireCube(1.0);
```

**The LookAt:** In the display function, notice that a call to `gluLookAt` positions the eye at (0.5,0.5,2.0) and look at the point (0.5,0.5,0.0) with an up vector of (0, 1, 0):

```
gluLookAt(0.5, 0.5, 2.0, 0.5, 0.5, 0.0, 0.0, 1.0, 0.0);
```

**The Projection:** In the `projectionView` function, you will find the projection set up with

```
gluPerspective(60, 1.0, 0.1, 10.0);
```

Now have a look at the `projectionview` window (Figure 436). Once you zoom out enough, you should be able to see the cube in green, and the view volume shown by a large blue pyramid (extending from the near to the far plane) and a small green pyramid (extending from the eye to the near plane). Geometry (like the cube) which is contained in the blue viewing volume is visible on the screen.

Try using different values for the call to `gluPerspective` and observe the effect on the viewing volume and on the image displayed in the main GL window.

What call to `gluPerspective(...)` will produce an image of the perspective projection of the wire-frame cube in a square viewport such that the projection nearly spans the entire viewport? Write your answer in the space provided below. Press the save button and save your work as `Q1_Answer`.

Your answer here...

## Question 2)

Let's take a different perspective view of the wire-frame cube defined in Question 1.

Load the project `Q2_Project` by selecting it from the projects combobox.

Notice that the eye is now positioned at (2.0, 0.5, 1.0), looking at the point (1.0, 0.5, 0.0)(the centre of the vertical edge of the cube closest to the eye) with an up-vector of (0,1,0).

The call to `gluPerspective` in the `projectionView` function is again initially `gluPerspective(60, 1.0, 0.1, 10.0);`

Again try to estimate suitable arguments for a call to `gluPerspective(...)` which will produce a nice perspective projection of the cube in a square viewport. Write a program to produce such an image. Note that the projection obtained is a TWO-POINT PERSPECTIVE PROJECTION of the cube. Where is the x-axis vanishing point? Where is the z-axis vanishing point?

How could a THREE-POINT PERSPECTIVE PROJECTION of the cube be obtained?

Press the save button and save your work as `Q2_Answer`. Write the call to `gluPerspective` as well as where the x and z vanishing points lie, as well as the way to obtain a three-point perspective projection in the box below.

Your answer here...

### Question 3)

Load the project `Q3_Project` by selecting it from the projects combobox. Notice that the view has been set up with the eye at (2, 0, 0) looking at the origin (0,0,0) with an up vector of (0,1,0).

Also notice the `translate` call moving the cube has been removed, leading to a frontal view of the cube. Press "Parse" and observe the cube in the `gl` window and the `projectionview` window.

What do you expect the effect of increasing the view angle (the first argument to `gluPerspective`) will be?

Try increasing the view angle and pressing "Parse". Observe the effect in the gl window and in the projectionview window. What are the results?

What do you expect the effect of increasing the distance of the eye from the object will be? Try moving the eye farther from the object by changing the eye's x-coordinate in the gluLookAt function (`gluLookAt(eyeX, eyeY, eyeZ, lookAtX, lookAtY, lookAtZ, upX, upY, upZ)`). What are the results?

Press the save button and save your work as Q3\_Answer. Write your answers (observations) into the box below.

Your answers here...

#### Question 4)

Load the project Q4\_Project by selecting it from the projects combobox. Notice that the view remains centered at the origin, and that the projection is orthogonal, given by the opengl call `glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)`. The OpenGL call's last two parameters are -1.0 and 1.0. Would this work for a perspective projection? Why/Why Not? Note your answers in the text area at the end of this question.

Also notice that the transformation call preceding the drawing of the glutWireCube is set to all zeroes initially (`glTranslatef(0.0, 0.0, 0.0);`), and hence has no effect. Therefore the cube is drawn at the origin. Press the "Parse" button and observe the projectionview window. Notice that the viewing pyramid has been replaced by a viewing cube, since we're dealing with an orthographic projection.



Try changing the position of the cube by changing the z-value of the `glTranslate` call preceding the call to `glutWireCube`. Observe the effect in the projectionview window. What happens to the size of the cube in the viewing window? Why? Note your answers in the text area at the end of this question.

Remove the `glTranslate` call so the cube again rests at the centre of the viewing volume. Change the projection call `glOrtho` so that the cube appears twice as wide but half as high compared to the call `glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0)`. Note your answer in the text area at the end of this question.

Your answers here...