

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering & Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:

Caitlin Kelleher, Chair

Roger Chamberlain

Michael Horn

Tao Ju

Alvitta Ottley

Code Puzzle Completion Problems in Support of
Learning Programming Independently

by

Kyle J. Harms

A dissertation presented to
The Graduate School
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

May 2017
St. Louis, Missouri

© 2017, Kyle J. Harms

Table of Contents

List of Figures	vii
List of Tables	xiv
Acknowledgments	xv
Abstract	xviii
Chapter 1: Introduction	1
1.1 Informal Resources for Learning Programming	3
1.2 Cognitive Load Theory	7
1.2.1 Human Cognitive Architecture	7
1.2.2 Cognitive Load.....	8
1.2.3 Cognitive Load Theory Effects	10
1.3 Approach.....	13
1.3.1 Code Puzzle Completion Problems	14
1.3.2 Hypotheses.....	19
1.4 Contributions	20
1.5 Summary.....	21
Chapter 2: Developing Code Puzzles as Completion Problems	23
2.1 Completion Problems & Programming.....	24
2.2 Formative Evaluation I: Puzzle Interface and Mechanics.....	26
2.2.1 Methods & Lessons Learned.....	28
2.2.2 Interface & Mechanics Overview	39
2.3 Formative Evaluation II: Introductory Curriculum	40
2.3.1 Lessons Learned	41
2.3.2 Curriculum Overview	47

2.4	Revised Code Puzzle Completion Problems.....	47
2.4.1	Revised Puzzle Interface.....	47
2.4.2	Revised Curriculum.....	50
2.5	Discussion.....	52
Chapter 3: Evaluating the Learning Effectiveness of Code Puzzle Completion Problems		54
3.1	Related Work	55
3.1.1	Tutors	56
3.1.2	Tutorials.....	56
3.1.3	Puzzle-Like Systems	57
3.2	Summative Evaluation	57
3.2.1	Participants.....	60
3.2.2	Materials	60
3.2.3	Methods	63
3.3	Analysis	65
3.3.1	Transfer Tasks	65
3.3.2	Task Surveys	65
3.3.3	Task Evaluation Questionnaire	66
3.3.4	Controlling External Effects.....	66
3.4	Results.....	67
3.4.1	Time & Cognitive Load	67
3.4.2	Motivation	69
3.4.3	Transfer Task Performance	70
3.4.4	Instructional Efficiency	72
3.5	Discussion.....	75
3.6	Conclusion.....	76
Chapter 4: The Impact of Errors in Code Puzzle Completion Problems.....		77
4.1	Related Work	79
4.1.1	Learning with Errors	79
4.1.2	Distractors and Errors in Testing	80

4.2	Formative Evaluation	82
4.2.1	Extra Noise is Easy to Ignore	84
4.2.2	Use a Familiar, but Suboptimal Path	85
4.2.3	Allow Only One Solution.....	86
4.3	Summative Evaluation	87
4.3.1	Participants.....	88
4.3.2	Materials	88
4.3.3	Methods	92
4.4	Analysis	95
4.4.1	Training & Transfer Tasks	95
4.4.2	Controlling External Effects.....	97
4.5	Results.....	98
4.5.1	Distractor Usage.....	98
4.5.2	Training Performance	100
4.5.3	Transfer Performance	105
4.5.4	Instructional Efficiency	109
4.6	Threats to Validity	110
4.7	Discussion.....	110
4.8	Conclusion	112
Chapter 5: Learners' Perceived Value of Code Puzzle Completion Problems		113
5.1	Related Work	114
5.1.1	Learners' Perceptions	114
5.1.2	Independent Learning Support	115
5.2	Evaluation	116
5.2.1	Instructional Formats	117
5.2.2	Participants.....	117
5.2.3	Methods	118
5.2.4	Materials	120
5.3	Analysis	124
5.3.1	Task Evaluation Questionnaires	124

5.3.2	Interview Responses	125
5.4	Results.....	126
5.4.1	Decision Rationales	127
5.4.2	Expected Task Difficulty	134
5.4.3	Sources of Ease & Difficulty	137
5.4.4	Experience Outcomes	140
5.5	Threats to Validity	144
5.6	Discussion.....	144
5.7	Conclusion	146
Chapter 6:	Summary & Future Work	148
6.1	Improving the Efficiency & Effectiveness of Code Puzzle Completion Problems	150
6.1.1	Using Worked Examples.....	151
6.1.2	Promoting Self-Explanation	155
6.1.3	Errors for Experienced Learners.....	157
6.2	Extending Code Puzzle Completion Problems	158
6.2.1	Alternative Contexts	158
6.2.2	Curricular Content.....	159
6.2.3	Alternative Completion Approaches.....	161
6.3	Supporting Goal-Oriented Learners	162
6.3.1	Choices for Instructional Formats.....	163
6.3.2	Self-Directed Learning	164
6.3.3	Self-Assessment & Scaffolding	166
References	168
Appendix A:	Initial Puzzle Interface.....	[186]
Appendix B:	Initial Puzzle Curriculum.....	[195]
Appendix C:	Revised Puzzle Interface	[203]
Appendix D:	Revised Puzzle Curriculum	[215]
Appendix E:	Summative Evaluation I Materials	[231]
E.1	Training Phase	[233]

E.1.1	Control Condition: Tutorials	[233]
E.1.2	Experimental Condition: Puzzles	[236]
E.1.3	Training Tasks	[238]
E.2	Transfer Phase.....	[247]
E.2.1	Transfer Tasks	[250]
Appendix F: Summative Evaluation II Materials	[256]
F.1	Training Phase	[260]
F.1.1	Control Training Tasks: No Distractors	[264]
F.1.2	Experimental Training Tasks: Distractors	[271]
F.1.3	Post Training Task Surveys.....	[278]
F.2	Transfer Phase.....	[281]
F.2.1	Transfer Tasks	[289]
F.2.2	Post Transfer Task Surveys.....	[293]
Appendix G: Summative Evaluation III Materials	[296]
G.1	Semi-Structured Interviews.....	[298]
G.1.1	Interview Materials	[298]
G.1.2	Pre-Study Interview	[300]
G.1.3	Post-Task Interview	[300]
G.1.4	Early Termination Interview	[301]
G.1.5	Post-Study Interview	[302]
G.2	Familiarization Tasks	[304]
G.3	Instructional Tasks	[307]
G.4	Post-Study Surveys.....	[324]

List of Figures

Figure 1.1:	Sample worked example	10
Figure 1.2:	Example completion problem	12
Figure 1.3:	The Looking Glass programming environment	15
Figure 1.4:	Puzzle activity selection interface	16
Figure 1.5:	Code puzzle completion problems.....	18
Figure 2.1:	Code scrambles	27
Figure 2.2:	Prototype of reorder-only statements	29
Figure 2.3:	Code puzzle reassembly prototype	30
Figure 2.4:	Movable puzzle play windows	32
Figure 2.5:	Puzzle feedback overlay pane.....	34
Figure 2.6:	Correct puzzle dialog.....	35
Figure 2.7:	Puzzle feedback indicator iterations	36
Figure 2.8:	Overlapping statements.....	37
Figure 2.9:	Quit and success dialogs.....	38
Figure 2.10:	Initial code puzzle completion problem interface.....	39
Figure 2.11:	Early puzzle without scenario	42
Figure 2.12:	Hard to remember puzzle	43
Figure 2.13:	Puzzle with memorable segments	44
Figure 2.14:	Tricky puzzles	45
Figure 2.15:	Initial code puzzle animations	46
Figure 2.16:	Revised puzzle interface	48

Figure 2.17: Puzzle task selection interface	49
Figure 2.18: Revised puzzle animations	51
Figure 3.1: Scratch's step-by-step tutorials	58
Figure 3.2: The tutorial condition.....	59
Figure 3.3: Summative evaluation procedure.....	63
Figure 3.4: Training task completion time	68
Figure 3.5: Training task cognitive load	68
Figure 3.6: Post-training Task Evaluation Questionnaire	69
Figure 3.7: Transfer task performance	71
Figure 3.8: Transfer task completion time	72
Figure 3.9: Transfer task cognitive load	73
Figure 3.10: Instructional efficiency	74
Figure 4.1: Partial suboptimal path distractors	83
Figure 4.2: Example transfer task.....	90
Figure 4.3: Summative evaluation procedure.....	92
Figure 4.4: Percent of participants who used distractors.....	99
Figure 4.5: Percent of time spent with distractors.....	99
Figure 4.6: Average training task completion time.....	101
Figure 4.7: Percent of completed training tasks	102
Figure 4.8: Percent of participants that gave up on the training tasks	102
Figure 4.9: Percent of participants who ran out of time on training tasks.....	103
Figure 4.10: Training task cognitive load	104
Figure 4.11: Training cognitive load types.....	105
Figure 4.12: Mean transfer task completion time.....	106
Figure 4.13: Transfer task performance	107
Figure 4.14: Mean transfer task cognitive load.....	107
Figure 4.15: Transfer cognitive load types.....	108

Figure 4.16: Instructional efficiency	109
Figure 5.1: Tutorial and puzzle instructional tasks	117
Figure 5.2: Overview of the study procedure	118
Figure 5.3: Instructional format selection screen	119
Figure 5.4: Instructional task program/animation selection	122
Figure 5.5: Mean instructional format decisions	127
Figure 5.6: Task Evaluation Questionnaire	130
Figure A.1: Directions pane	[187]
Figure A.2: Initial state	[188]
Figure A.3: Play correct animation	[189]
Figure A.4: Play user's animation	[190]
Figure A.5: Completed puzzle	[191]
Figure A.6: Correct feedback	[192]
Figure A.7: Solved puzzle dialog	[193]
Figure A.8: Quit dialog	[194]
Figure B.1: Puzzle mechanics	[196]
Figure B.2: Do in Order	[197]
Figure B.3: Repeat	[198]
Figure B.4: Do Together	[199]
Figure B.5: Repeat & Do Together	[200]
Figure B.6: Do Together { Repeat }	[201]
Figure B.7: Repeat { Do Together }	[202]
Figure C.1: Puzzle selection pane	[204]
Figure C.2: Puzzle preview pane	[205]
Figure C.3: Directions pane	[206]
Figure C.4: Initial state	[207]

Figure C.5: Play correct animation.....	[208]
Figure C.6: Play user's animation	[209]
Figure C.7: Completed puzzle	[210]
Figure C.8: Correct feedback.....	[211]
Figure C.9: Correct dialog	[212]
Figure C.10: Quit dialog.....	[213]
Figure C.11: Break dialog	[214]
Figure D.1: Do in Order	[217]
Figure D.2: Repeat I	[218]
Figure D.3: Repeat II	[219]
Figure D.4: Do Together I	[220]
Figure D.5: Do Together II	[221]
Figure D.6: Do Together III.....	[222]
Figure D.7: Repeat & Do Together.....	[223]
Figure D.8: Repeat { Repeat }	[224]
Figure D.9: Do Together { Repeat }	[225]
Figure D.10: Repeat { Do Together } II.....	[226]
Figure D.11: Repeat { Do Together } I.....	[227]
Figure D.12: Do Together { Do in Order } I	[228]
Figure D.13: Do Together { Do in Order } II	[229]
Figure D.14: Do Together { Do in Order } III.....	[230]
Figure E.1: Computing history survey.....	[232]
Figure E.2: Control training instruction sheet.....	[233]
Figure E.3: Tutorial steps window.....	[234]
Figure E.4: Control training familiarization task	[235]
Figure E.5: Experimental training instruction sheet	[236]
Figure E.6: Experimental training familiarization task	[237]

Figure E.7: Training task survey	[238]
Figure E.8: Training task 1	[239]
Figure E.9: Training task 2	[240]
Figure E.10: Training task 3	[241]
Figure E.11: Training task 4	[242]
Figure E.12: Training task 5	[243]
Figure E.13: Training task 6	[244]
Figure E.14: Training task time notice	[245]
Figure E.15: Task Evaluation Questionnaire	[246]
Figure E.16: Transfer instruction sheet	[247]
Figure E.17: Transfer task instruction window	[248]
Figure E.18: Transfer familiarization task	[249]
Figure E.19: Transfer task survey	[250]
Figure E.20: <i>Repeat</i> transfer task	[251]
Figure E.21: <i>Do Together</i> transfer task	[252]
Figure E.22: <i>Repeat { Do Together }</i> transfer task	[253]
Figure E.23: <i>Do Together { Repeat }</i> transfer task	[254]
Figure E.24: Transfer task time notice	[255]
Figure F.1: Computing history survey	[257]
Figure F.2: Follow-up computing history survey	[258]
Figure F.3: Programming list	[259]
Figure F.4: Training instruction sheet	[260]
Figure F.5: Training familiarization task	[261]
Figure F.6: Training task survey	[262]
Figure F.7: Training task time notice	[263]
Figure F.8: control training task 1	[265]
Figure F.9: control training task 2	[266]

Figure F.10: control training task 3	[267]
Figure F.11: control training task 4	[268]
Figure F.12: control training task 5	[269]
Figure F.13: control training task 6	[270]
Figure F.14: Distractors training task 1	[272]
Figure F.15: Distractors training task 2	[273]
Figure F.16: Distractors training task 3	[274]
Figure F.17: Distractors training task 4	[275]
Figure F.18: Distractors training task 5	[276]
Figure F.19: Distractors training task 6	[277]
Figure F.20: Training task reminder sheet	[278]
Figure F.21: Training overall cognitive load survey.....	[279]
Figure F.22: Training CS CLCS	[280]
Figure F.23: Transfer mechanics instruction sheet	[281]
Figure F.24: Transfer mechanics familiarization task	[282]
Figure F.25: Transfer mechanics familiarization task solution	[283]
Figure F.26: Transfer task instruction sheet.....	[284]
Figure F.27: Transfer instructions window.....	[285]
Figure F.28: Transfer instructions familiarization task	[286]
Figure F.29: Transfer task survey	[287]
Figure F.30: Transfer task time notice.....	[288]
Figure F.31: <i>Do Together { Repeat }</i> transfer task	[290]
Figure F.32: <i>Repeat { Do Together }</i> transfer task	[291]
Figure F.33: <i>Repeat { Repeat }</i> transfer task.....	[292]
Figure F.34: Transfer task reminder sheet	[293]
Figure F.35: Transfer overall cognitive load survey.....	[294]
Figure F.36: Transfer CS CLCS	[295]

Figure G.1: Computing history survey.....	[297]
Figure G.2: Instructional task reminder sheet	[298]
Figure G.3: Semi-structured interview question scales.....	[299]
Figure G.4: Familiarization task 1	[305]
Figure G.5: Familiarization task 2	[306]
Figure G.6: Instructional task format selection.....	[307]
Figure G.7: Animation selection.....	[308]
Figure G.8: List of instructional tasks	[309]
Figure G.9: Instructional task 1	[310]
Figure G.10: Instructional task 2	[311]
Figure G.11: Instructional task 3	[312]
Figure G.12: Instructional task 4	[313]
Figure G.13: Instructional task 5	[314]
Figure G.14: Instructional task 6	[315]
Figure G.15: Instructional task 7	[316]
Figure G.16: Instructional task 8	[317]
Figure G.17: Instructional task 9	[318]
Figure G.18: Instructional task 10.....	[319]
Figure G.19: Instructional task 11.....	[320]
Figure G.20: Instructional task 12.....	[321]
Figure G.21: Instructional task 13.....	[322]
Figure G.22: Instructional task 14.....	[323]
Figure G.23: Tutorial Task Evaluation Questionnaire.....	[325]
Figure G.24: Puzzle Task Evaluation Questionnaire.....	[326]

List of Tables

Table 1.1:	Approaches' support for independent learning.....	3
Table 2.1:	Initial code puzzle curriculum.....	46
Table 2.2:	Revised code puzzle curriculum	50
Table 3.1:	Training tasks	61
Table 4.1:	Training task constructs and distractors.	89
Table 5.1:	Instructional tasks	121
Table 5.2:	Sublabel interrater agreement.....	126
Table 5.3:	Decision rationale response themes.	129
Table 5.4:	Expected task difficulty response themes.....	135
Table 5.5:	Sources of ease & difficulty response themes.	138
Table 5.6:	Experience outcomes response themes.	141
Table B.1:	Initial puzzle curriculum	[195]
Table D.1:	Revised puzzle curriculum	[216]

Acknowledgments

I want to acknowledge and thank everyone who helped and encouraged me throughout this journey.

I especially want to thank my best friend, partner, and husband, Todd Brennan. Without you, I would never have made it this far. Your kind words, support, and encouragement gave me the courage to succeed. Thank you for your patience, your understanding, and keeping me sane. I love you.

I want to thank my BFF, Doddy. You go by many names: Becky, Becky Dodd, Rebecca Dodd, but you'll always be Doddy to me. No one can ever match the fun, the drama, and the excitement of being your friend. Thank you for your support and eating lots of food with me. You will always be my best friend.

Thank you to my parents, Jim and Debbie Harms. I don't tell you enough how much I appreciate what you have done and continue to do for me. I'm so lucky to have parents that support my decisions and encourage me to push forward and succeed. Thank you for everything.

I want to thank my advisor, Caitlin Kelleher. You believed in me. You supported me. You stuck with me and pushed me when I didn't think I could do it. I cannot thank you enough

for the limitless time, understanding, and patience you gave me. I feel so lucky to have been mentored by you.

A special thanks to Michelle Ichinco. You are a great friend and colleague. Thank you for listening and giving advice when I didn't know what to do.

Dennis Cosgrove, thank you for all the hard work you put into helping Looking Glass and my research succeed. Thank you for all of the programming support, brainstorming sessions, and making work fun.

I would like to acknowledge all of the undergraduate researchers who worked with me on this project: Jordana Kerr, Shannon Gray, Noah Rowlett, Jason Chen, and Danielle Clemons. Your hard work, dedication, and ideas made this research possible.

Anne Bracy, I want to thank you for the teaching guidance and helping me become a better educator. I also want to thank Ron Cytron for supporting my teaching and for giving valuable life and teaching advice.

A special thanks to all the staff in the Computer Science & Engineering Department. Thank you Monét Demming, Kelli Eckman, Myrna Harbison, Sharon Matlock, Cheryl Newman, Jayme Moehle, Lauren Huffman, and Madeline Hawkins. Whether it was a user study, keys, pay checks, paper towel dispensers, or paperwork, you were always there to help.

I want to thank my current and past members of my committee: Caitlin Kelleher, Roger Chamberlain, Michael Horn, Tao Ju, Alvitta Ottley, and Robert Pless. I appreciate all of the time and advice you invested in me.

I want to thank all of my friends and family, including the Dodds. You were patient and understanding when I had to work. You were excited for my successes and pushed me through my setbacks. Thank you for making life fun and enjoyable. You all are great!

The gym has been my refuge from stress throughout this entire process; weightlifting helped me maintain my sanity. Thanks to my gym buddies: Jason Franklin, Henry Barry, Paul Northrop, and Seth Alms. A special thanks to Nathan Williams who through it all, taught me Olympic weightlifting and then coached me through my first competition.

Lastly, I want to thank our dogs, Berkeley and Gregory. You are fluffy and crazy doodies and I love it!

This material is based upon work supported by the National Science Foundation under Grants No. 1054587 and 1440996.

Kyle J. Harms

Washington University in Saint Louis

May 2017

ABSTRACT OF THE DISSERTATION

Code Puzzle Completion Problems in Support of
Learning Programming Independently

by

Kyle J. Harms

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2017

Professor Caitlin Kelleher, Chair

Middle school children often lack access to formal educational opportunities to learn computer programming. One way to help these children may be to provide tools that enable them to learn programming on their own independently. However, in order for these tools to be effective they must help learners acquire programming knowledge and also be motivating in independent contexts. I explore the design space of using motivating code puzzles with a method known to support independent learning: completion problems. Through this exploration, I developed *code puzzle completion problems* and an introductory curriculum introducing novice programmers to basic programming constructs. Through several evaluations, I demonstrate that code puzzle completion problems can motivate learners to acquire new programming knowledge independently. Specifically, I found that code puzzle completion problems are more effective and efficient for learning programming constructs independently compared to tutorials. Puzzle users performed 33% better on transfer tasks compared to tutorial users, while taking 21% less time to complete the learning materials. Additionally, I present evidence that children are motivated to choose to use the code puzzles because they find the experience enjoyable, challenging, and valuable towards developing their programming skills. Given the choice between using tutorials and puzzles, only 10% of participants opted to use more tutorials than puzzles. Further, 80% of participants also stated a preference towards the puzzles because they simply enjoyed the experience of using puzzles more than

the tutorials. The results suggest that code puzzle completion problems are a promising approach for motivating and supporting independent learning of programming.

Chapter 1

Introduction

Computing has fundamentally impacted every aspect of our world, including every facet of our economy. Yet, even though our economy cannot function without the skills of computer scientists, there are many computing jobs that go unfulfilled due to a lack of qualified workers. Of the Science, technology, engineering, and mathematics (STEM) related jobs added to the U.S. economy between 2009 and 2015, 80% were computing related jobs [49]. Further, the U.S. Bureau of Labor Statistics projects that over the next decade nearly 500,000 computing jobs will be added to the U.S. economy [49]. This situation is not unique to the United States, a similar trend is also expected across Europe [81]. While this is great news to the 8% of STEM graduates with computer science degrees [200], many of these jobs are still expected to go unfulfilled due to a shortage of computer science degree earners [31, 206]. Given the importance of computing to our economy, additional computer science graduates are absolutely necessary in order to meet this demand.

Because computer science, as a skill, is necessary for the future success of our economy, our K-12 educational policies should prioritize its place in the classroom. Unfortunately, computer

science is frequently absent from required lists of “core academic subjects” like English, reading, mathematics, and history [103, 200]. Due to its non-primary status in educational standards, many school districts neglect or fail to offer any computer science related education [103, 203]. In fact, 60% of K-12 schools do not offer any formal opportunities to learn computer science [200] and in the schools that do, computer science is often misrepresented as using office productivity software. Without the opportunity to learn these fundamental computing skills, many students may simply be left behind in the global economy.

Given the lack of formal resources and opportunities in K-12 schools to learn programming, informal learning resources may provide an alternative for students. Informal resources would enable students to learn programming on their own, independently without the support of teachers. However, in order for these resources to be effective, there are several goals we should consider: 1) the resources need to support learning in an informal setting, 2) the resources must improve learners’ programming knowledge, and 3) learners should be motivated to use the resources while working towards their own goals.

In this work, I investigate how example-inspired code puzzles, called *code puzzle completion problems*, can meet these goals in supporting middle school children to learn programming on their own. In this chapter, I describe the related work which inspired my approach for developing code puzzle completion problems: informal resources for learning programming and cognitive load theory. In the following section, I first explore other resources for informally learning programming and how they can meet some of my goals. Next, I provide a background into cognitive load theory and how it can be used to provide effective learning in independent contexts, thereby addressing my remaining goals. Finally, I describe my approach for integrating these existing techniques into a resource that children can use to learn programming on their own.

Table 1.1: Approaches that are known to support the goals listed above.

	Supports Learners in Informal Settings	Efficiently Develops Programming Knowledge	Motivating for Goal-Oriented Users
Intelligent Tutoring Systems		✓	
Massive Open Online Courses	✓		
Minimalist & Motivating Programming Environments	✓		✓
Programming Tutorials & Puzzles	✓		✓
Task Specific Example Usage			✓

1.1 Informal Resources for Learning Programming

There currently exist several different types of resources for novices to learn programming in informal settings that satisfy some of the above goals. Novices can leverage tutoring systems, online courses, specialized programming environments, tutorials, puzzles, and examples to develop their programming knowledge. In this section and in Table 1.1, I highlight how these approaches are to known to directly support some of my goals.

Intelligent Tutoring Systems Tutoring environments typically present students with a set of practice exercising following a classroom lecture. Based on the student’s progress, the intelligent tutoring system provides supportive feedback or additional exercises for additional practice. This approach has been shown to aid students learning programming as they work through the material on their own [12, 128, 130, 156]. However, tutoring systems are typically designed to supplement traditional classroom instruction [1, 3]. Without the classroom component, the tutors do not provide sufficient support to help individuals learn on their own. Further, students are typically required to use the tutor as part of their

coursework. Without the classroom requirement learners may lack the motivation to use these systems independently. This is reflected in Table 1.1; intelligent tutoring systems do support development of programming knowledge, but they are not designed to motivate goal-oriented learners in informal settings.

Massive Open Online Courses Alternatively, massive open online courses (MOOCs) also support independent learning, but they do so without being tied to traditional classroom instruction. MOOCs, like Codecademy [27] or Khan Academy [97], enable learners to complete programming lessons with follow-up programming exercises entirely online. While this seems like a promising approach for helping children learn programming on their own, online courses generally have lower completion rates than traditional classes [18, 132, 146]. The high attrition rates are often due to an inconsistency between the learners’ specific goals and the traditional style of instruction [6, 169]. When the material fails to address a learner’s concerns specifically, they may lose motivation and drop the course [6, 169]. Because learners often quit online courses, their strength lies mostly with supporting learners in informal settings as shown in Table 1.1.

Minimalist & Motivating Programming Environments While online courses provide formal instruction, they are often too constrained to effectively support goal-oriented learners who are motivated by their own open-ended activities. Providing learners with open-ended and motivating programming environments enables novices to follow their own goals and possibly learn some programming skills along the way. Researchers have developed programming environments that are specifically designed to motivate children by leveraging animation [94, 174], storytelling [96, 117], dance [40, 147], and game authoring [119, 120].

Motivating programming environments have been shown to encourage learners to spend additional time working on their own projects or goals [96]. During the additional time spent

working in the programming environment, learners may pick-up some new programming knowledge. However, the knowledge that they acquire may be somewhat limited. Minimalist or constructivist approaches to learning are often less effective and efficient than the guided instruction seen in structured resources, like online courses [100]. Yet, these programming environments still work well for motivating learners in informal settings as shown in Table 1.1.

Programming Tutorials & Puzzles Many novice programming environments provide a motivating context as well as additional support to help children learn programming. Children’s programming environments predominantly use two instructional formats to support learning: tutorials [77, 129, 174] and code puzzles [55, 77, 112]. However, as reflected in Table 1.1, the evidence that tutorials and puzzles are effective for learning programming independently, is somewhat unclear.

Traditionally, novice programming environments have often attempted to support informal learning using tutorials [7, 71, 72]. These tutorials typically provide step-by-step instructions for inserting statements into the programming environment. For example, a tutorial introducing repetition in the novice programming environment, Scratch [174], shows where to find the repeat block and how to insert the block into the user’s program. Much of the research on tutorials has focused on improving successful completion by guiding users through a series of steps while minimizing errors [71, 95, 151, 198]. Because these tutorials mostly focus on interface mechanics, it is not clear how well novices acquire programming knowledge when using tutorials and how well learning interface mechanics complements the user’s own goals.

More recently, some novice programming resources have begun to use a more puzzle-like approach to facilitate independent learning [8, 30, 45, 60, 107]. Often these systems present some type of challenge with code pieces and encourage users to solve the challenge using the code pieces. Additionally, most of these puzzle-like systems present themselves as games and

so therefore exists outside of a conventional programming environment [30, 41, 55]. While the game may be motivating to the user, it is not clear how well this style of interaction aligns with informal users' goals, especially when no programming environment similar to the game exists. Without a similar programming environment and language, these users may have trouble transferring any acquired programming knowledge to their own projects. Yet, these code puzzles may actually provide an unexpected benefit. Once completed, a puzzle is a valuable code example that demonstrates correct program behavior. Further, the active process of assembling the code example may facilitate learning.

Task Specific Example Usage Programmers frequently use examples informally throughout the process of programming [10, 167]. Commonly, experienced programmers leverage example code to address an issue while they are working towards a specific goal (i.e. fixing a bug) [10, 167]. Through the process of using an example, experienced programmers often indirectly acquire additional knowledge that will further develop their programming skills [166, 167]. Unfortunately, novice programmers typically have difficulty understanding and integrating unfamiliar example code into their own programs [61, 168]. This limits their ability to make effective use of examples while attempting self-driven projects, which further limits the possible knowledge they might have gained from the examples. Table 1.1 reflects this limit by highlighting that task specific example usage is best for motivating goal-oriented learners.

One possible way to utilize these code examples for novices in informal settings is to leverage existing models of learning. One such model, cognitive load theory, can be applied to the instructional design aspects of examples to reduce the burden of learning from these materials [182]. This theory also provides methods that when combined with many of the informal resources mentioned in this section, may increase their potential impact for facilitating effective and independent learning. In the next section, I summarize cognitive

load theory’s main principles and discuss how it can further help contribute to achieving the goals I outlined above.

1.2 Cognitive Load Theory

Cognitive load theory suggests that learning happens best under conditions that closely align with human cognitive architecture [182]. In practice, this means educators must carefully design educational materials to support the cognitive processes that are necessary for learning. This theory has also been shown to be particularly effective for learning in independent contexts [205]. In this section, I introduce cognitive load theory and how we can leverage this model of learning to accomplish our goals of effective independent learning.

1.2.1 Human Cognitive Architecture

In order to understand how to apply cognitive load theory, we must first understand its basic assumptions about human cognitive architecture. This theory assumes that humans have a limited capacity for processing novel information [126]. Roughly, humans have two types of memory: working and long-term memory. Working memory can be thought of as a pass through to long-term memory; all information we process must first pass through working memory before it can be stored in long-term memory [56, 186].

Before information can be passed to our long-term storage, it must first be processed in our working memory. Unfortunately, our working memory is an extremely limited resource that is easily over-whelmed when processing novel information [126, 183]. Working memory capacity is limited to *storing* between five to nine elements of information. [127]. However, we are limited to only *processing* between two and four elements at a time [38].

Our long term memory does not possess this limitation; instead it has seemingly infinite storage [126, 183]. In long-term memory we store complex representations of our knowledge called schema [126, 182]. Schema are the root of our expertise because they combine our previous knowledge with any newly processed information [126, 183, 186]. Later, whenever we process new information in our working memory, we can retrieve our schemas from long-term memory into working memory with minimal cost to aid in processing the new information [126, 142, 183].

Because information must first pass through working memory before we can *learn* it, the capacity of working memory is a bottleneck for knowledge acquisition [141]. It is only when our working memory is not over-whelmed or at capacity that learning or schema construction can actually take place [141]. In order to improve the efficiency of learning, cognitive load theory focuses on developing instructional materials that carefully manage learners' limited working memory resources [126, 182].

1.2.2 Cognitive Load

Whenever an instructional task overwhelms a learner's working memory, that task is said to impose a high cognitive load on the learner. The idea is to construct educational materials that impose an optimum amount of cognitive load on a learner to maximize their learning efficiency [141]. There are three types of cognitive load: intrinsic, extraneous, and germane. Intrinsic and germane cognitive load are beneficial to learning, while extraneous cognitive load is detrimental to learning [126, 182].

Intrinsic Cognitive Load Intrinsic cognitive load is imposed by the number of interacting elements a learner must process in an instructional task [126, 182]. This type of cognitive load is influenced by prior knowledge and the experience of the learner, unlike extraneous

and germane cognitive load [197]. In learners with low prior knowledge, each interacting element will consume a position in working memory, thus imposing a high intrinsic cognitive load. In learners with high prior knowledge, some or all of these interacting elements may already exist in schemas [186]. If a learner has existing schemas, only one element will be consumed in working memory for the entire schema that contains the interacting elements, thus imposing a low intrinsic cognitive load [186]. In short, intrinsic cognitive load decreases with learner expertise, thereby freeing working memory resources for other purposes [197]. If the intrinsic cognitive load is too high for a learner for an instructional task, the only way to lower the cognitive load is to find a simpler task that omits the difficult material [126, 141].

Extraneous Cognitive Load The cognitive load associated with an instructional task is considered extraneous when it is ineffective or detrimental to learning [141, 186]. Extraneous cognitive load is imposed by the manner in which the material is presented to a learner; expertise does not influence extraneous cognitive load [126, 182]. Poorly presented information can often require learners to use their working memory resources just to understand an instructional task. This might be poorly worded directions or a confusingly labeled figure. Any working memory resources used to understand the task cannot then be applied to learning the material which is detrimental to schema construction [141]. When designing instructional materials, the goal is to keep extraneous cognitive load as low as possible. If extraneous cognitive load is too high for an instructional task, the only solution to lower the cognitive load is to change the presentation of the task [141, 182].

Germane Cognitive Load Germane cognitive load is the load that is devoted to processing and constructing schemas [186]. Much like extraneous cognitive load, germane cognitive load is imposed by the design of the instructional materials, but it is beneficial to learning [186]. When creating instructional tasks, designers should always seek to reduce extraneous cognitive

Make a the subject of the equation $(a + b)/c = d$.

Solution

$$\begin{aligned}(a + b)/c &= d \\ a + b &= dc \\ a &= dc - b\end{aligned}$$

Figure 1.1: Sample worked example from *Cognitive Load Theory* [182]. Studying worked examples is known to aid in schema construction [32].

load and find ways foster germane cognitive load. When imposing germane load, designers should seek to encourage learners to engage in the cognitive processes that aid in schema construction [126]. For example, a designer might use a worked example or ask a learner to explain a concept to themselves. Cognitive load theory contains many of these types of *effects*, like learning from examples, which have been shown to be beneficial for learning.

1.2.3 Cognitive Load Theory Effects

The Worked Example Effect

Learning from examples is a well-known, time-tested, popular, and effective technique for knowledge acquisition [4, 32, 150, 185]. Cognitive load theory researchers have found that showing all of the possible steps towards reaching a solution facilitates efficient learning by imposing beneficial cognitive load while also reducing detrimental cognitive load [32, 185]. This is known as the *worked example effect*. An example is a *worked example*, if it provides a step-by-step solution to a problem [182]. See Figure 1.1 for a sample worked example for studying algebra.

Worked examples are designed to be studied by learners. Studying worked examples has been shown to facilitate learning, and thus schema construction, better than problem solving exercises, like those frequently used in homework [32, 185]. Further, worked examples are known to work extremely well for independent learners. A three-year algebra curriculum was completed in only two years by students who studied worked examples independently instead of learning from direct instruction methods, like lectures [205]. Not only did the worked example students complete the material in less time, but they also out-performed their peers who learned in a traditional classroom setting [205]. One of the reasons that worked examples are so effective for learning is that learners often engage in an internal *self-explanation* process in order to understand the example [22].

The Self-Explanation Effect

Self-explanation is “a mental dialogue that learners have when studying a worked example that helps them understand the example and build a schema from it” [24]. In studying a worked example, a learner must internally self-explain the reason for each step when processing it [22]. During self-explanation, the learner is using their free working memory resources to construct a schema from the interacting elements within the example [22, 182].

The self-explanation process can be a passive internalized process or purposefully elicited as part of the instructional task design. Instructional tasks that elicited self-explanation may explicitly ask learners to verbalize their thoughts [158] or prompt the learner to identify the underlying principle using a multiple-choice question [5]. Unfortunately, when self-explanation is a passive process, learners may choose to opt out of the process entirely or fail to devote sufficient working memory resources to develop quality explanations [160]. Because studying worked examples is a passive process, independent learners may lack the motivation to engage in adequate self-explanation [182]. In these independent contexts, using active rather

Make a the subject of the equation $(a + b)/c = d$.

Solution

$$\begin{aligned}(a + b)/c &= d \\ a + b &= dc \\ a &= \underline{\hspace{2cm}}\end{aligned}$$

Figure 1.2: Example completion problem from *Cognitive Load Theory* [182]. The learner is given the first steps to solving the problem and is required to complete the problem by finishing the remaining steps.

than passive interaction with the worked examples helps trigger self-explanation [5, 159]. *Completion problems* are one such adaptation of worked examples that require learners to actively engage with the example thus triggering self-explanation [124, 182].

The Problem Completion Effect

“A *completion problem* is a partial worked example where the learner has to complete some key solution steps” [182]. This is known as the *problem completion effect*. Figure 1.2 shows the same algebra worked example converted to a completion problem. Notice that the first step of the worked example is already provided and that the learner is required to complete the second step. Because learners are required to actively engage with the materials, rather than passively study it, completion problems may actively encourage self-explanation in learners [22, 180]. Further, completion problems have been shown to be as effective as worked examples [24, 138, 196].

Completion problems are well suited to complex domains where traditional worked examples would have many steps and interacting elements [182]. The high number of steps in these worked examples would generate high levels of extraneous cognitive load, thereby limiting schema construction [182]. Unfortunately, the complex nature of computer programs would

necessitate worked examples with many steps [182]. However, researchers have shown that completion problems are an effective method for learning programming [192, 193]. Van Merriënboer demonstrated that high school students who learned programming by completing partial programs were later able to construct superior programs than students who learned by writing programs from scratch [192, 193].

Yet, most completion problem research has been conducted in traditional classroom environments with lectured based instruction, including the prior programming research I mentioned above [192, 193]. While I would expect completion problems to be as effective as worked examples in independent contexts for learning programming, I am unaware of any such evidence. In this work, I investigate the use of completion problems as an effective method for novices to learning programming on their own independently.

1.3 Approach

In the introduction, I outlined three goals for developing an approach that encourages children to learn programming independently:

1) Supports Learners in Informal Settings Middle school children often lack access to formal resources, like classrooms with teacher support, to learn programming. Without support from teachers or others, independent learners must be sufficiently supported to encourage similar development of programming knowledge. Any approach designed for learning in informal settings, must then be evaluated with independent learners in order to demonstrate that it sufficiently supports these users.

2) Efficiently Develops Programming Knowledge In informal settings, time spent learning and direct application of material are likely important considerations for goal-oriented

learners. Middle school children should be able to complete learning materials expeditiously. They should also be able to utilize what they learned independently in similar contexts. The approach must be efficient in facilitating the use of programming constructs in near transfer situations.

3) Motivating for Goal-Oriented Users Goal-oriented learners primarily want to spend their time working towards their own projects. A motivating context encourages those users to spend additional time working to carry out their goals. Learners should be motivated to choose to utilize the approach because it is perceived as valuable and useful for accomplishing their specific goals.

In search of meeting these goals, I briefly reviewed resources for learning programming in informal settings in section 1.1. I also reviewed how cognitive load theory can efficiently support independent learning with examples in section 1.2. While none of this prior work directly meets all of the goals individually, it has heavily influenced my design for an approach that may satisfy them all. For my dissertation, I developed *code puzzle completion problems* by offering code puzzles as an activity in a novice programming environment that are also designed to illicit the completion problem effect. In this section, I provide an overview of code puzzle completion problems and I layout my hypotheses for evaluating this approach.

1.3.1 Code Puzzle Completion Problems

Looking Glass I selected the novice programming environment, Looking Glass [117], to build out my approach for code puzzle completion problems. Looking Glass, as shown in Figure 1.3, is an end-user programming environment specifically geared towards encouraging middle school children to program on their own. In Looking Glass, users program their own 3D animations called *worlds* using a block-based approach with drag-and-drop mechanics.

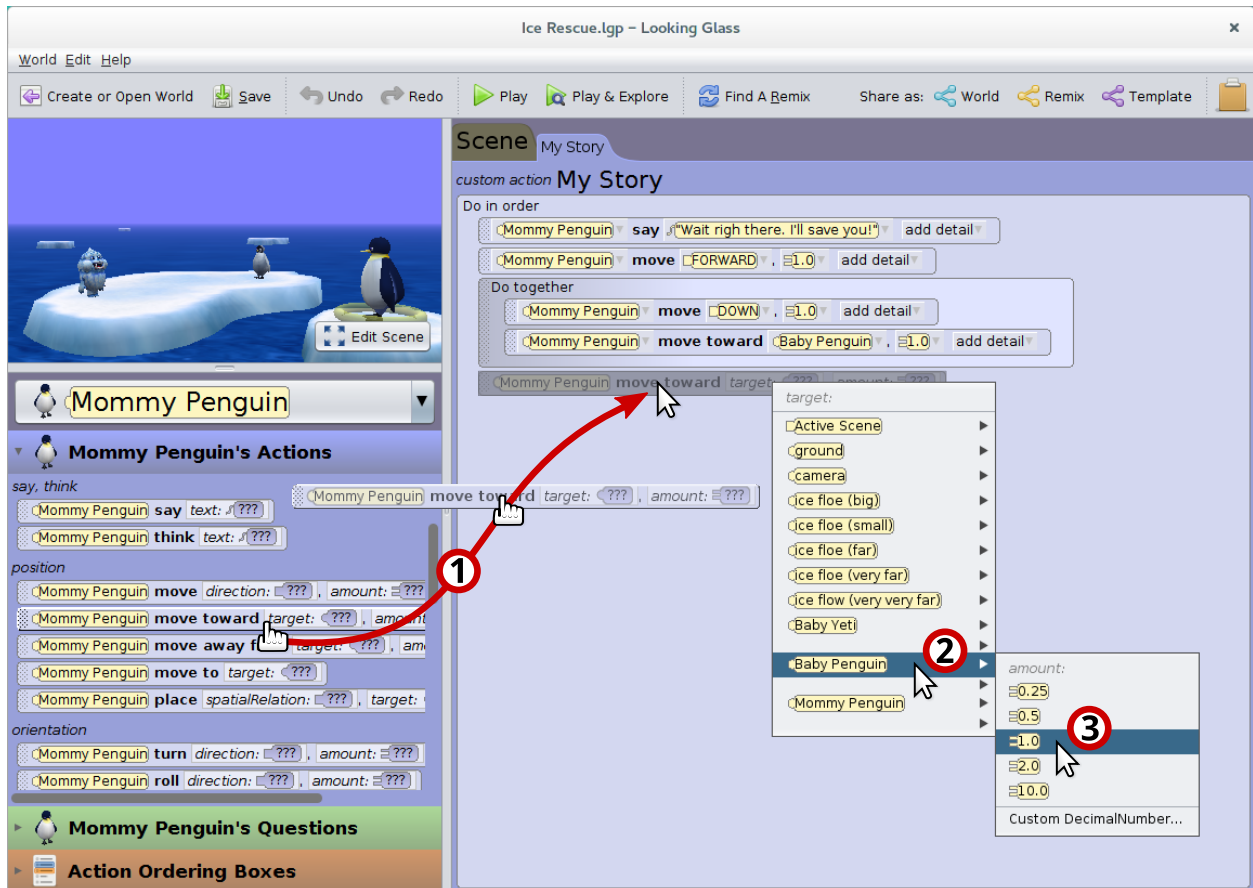


Figure 1.3: The Looking Glass programming environment. Looking Glass users author 3D story-based animations through drag-and-drop programming. Users first (1) drag code statements into the code editor and then select the parameters for each statement using drop-down menus (2-3).

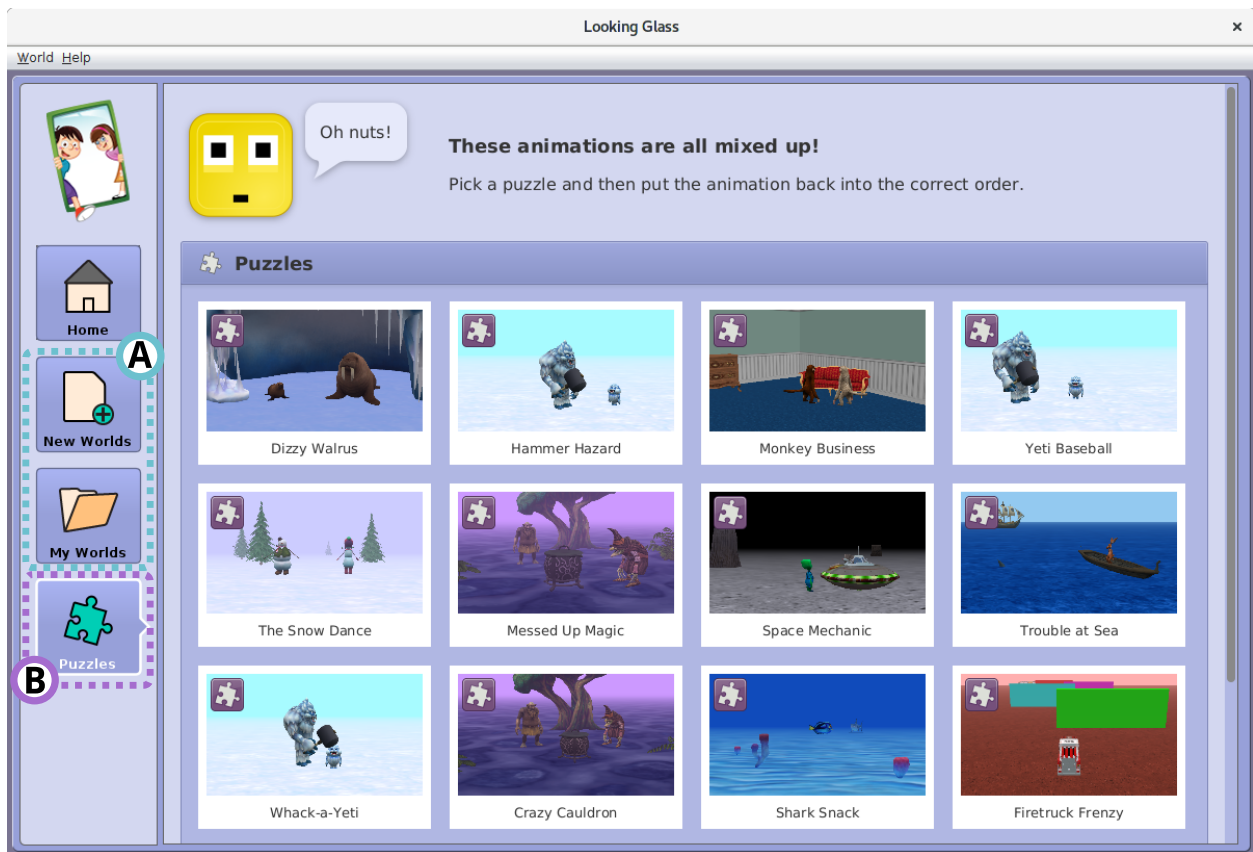


Figure 1.4: The puzzle activity selection interface in Looking Glass. Users can work towards their own goals by opting to create their own animation (A) or work on a code puzzle (B).

Blocks-based programming helps reduce the barriers to programming by removing the need to deal with the complexity of syntax. Further, removing the barrier of syntax may help limit extraneous cognitive load for novice programmers.

One of the key advantages of Looking Glass, is that it uses storytelling as motivating context [96]. Storytelling has been found to be a powerful motivator to encourage middle school children to spend more time programming on their own [96]. For my approach, I utilized the 3D animated stories of Looking Glass for a motivating context in the code puzzle completion problems.

Code Puzzles Looking Glass also has some existing support for helping novices acquire programming knowledge independently. Looking Glass features an example-based code-reuse system called *remixing* that enables novices to incorporate examples into their current project [62]. Remixing is an on-demand activity in Looking Glass; users choose to remix when it aligns with their own personal goals for their project. As an alternative to remixing and programming, code puzzles may provide an additional motivating activity that also helps foster learning [104, 105, 106]. For my approach, I added code puzzles to the activity selection screen in Looking Glass as shown in Figure 1.4. This gives goal-oriented learners the option to pursue programming their own 3D animated stories or solving code puzzles when it best addresses their current needs.

Completion Problems In order for the code puzzle activity to benefit goal-oriented learners, it must help them acquire knowledge that they can then apply to similar situations in their own projects. Many of these puzzle-like activities appear to use similar mechanics to completion problems: an incomplete initial state that users' must solve. This similarity suggests that code puzzles can be adapted into completion problems. Given that completion problems are well-suited to helping learners acquire novel information independently, code puzzles that employ the completion problem effect will likely be both motivating and effective for acquiring new programming knowledge.

In my approach, code puzzle completion problems ask learners to reassemble a compelling animation, as shown in Figure 1.5. Unlike similar code puzzles [148], I have designed these code puzzles with additional supportive scaffolding in an effort to produce completion problems. The supportive scaffolding of the completion problems is designed to reduce extraneous cognitive load and foster germane cognitive load in order to promote efficient and effective independent learning. When reassembling the animation in code puzzle completion problems, learners must internalize the behavior that they are observing and generate a self-explanation

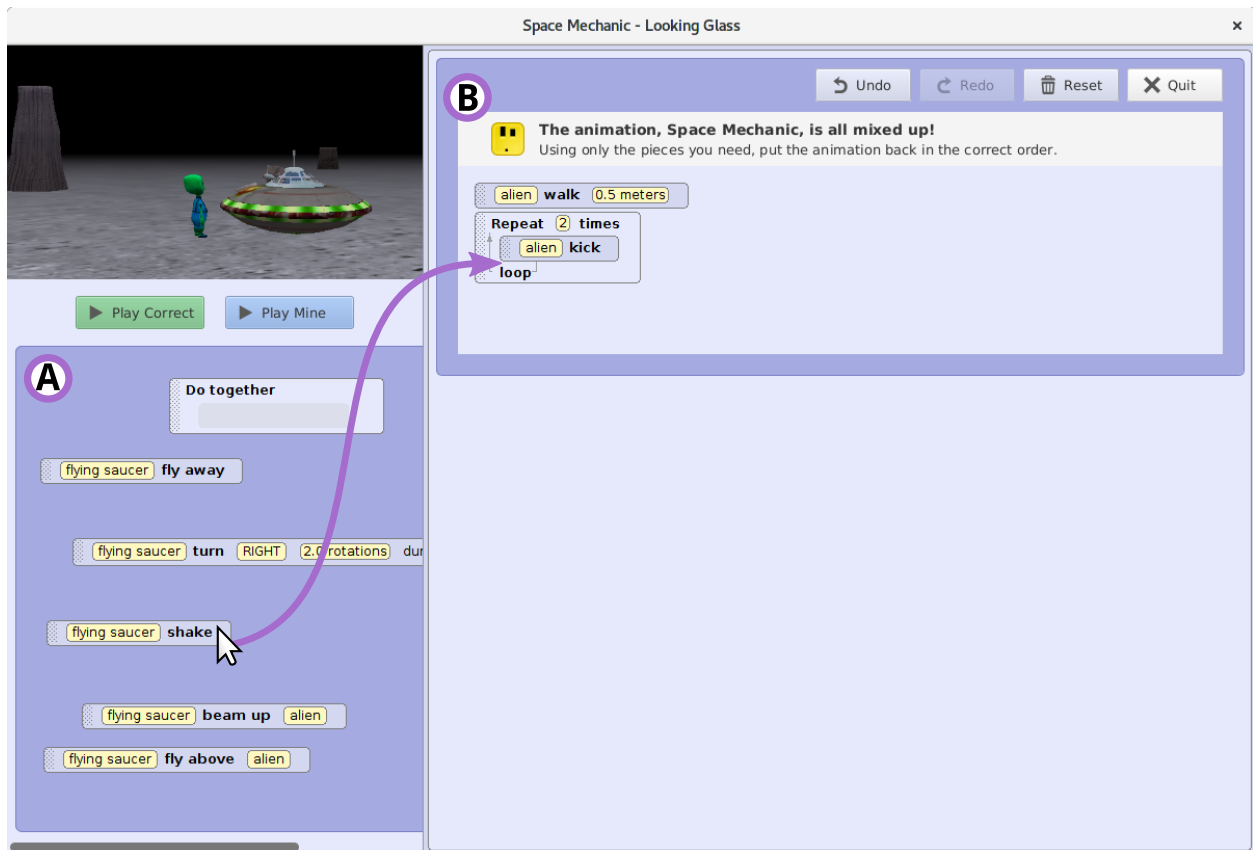


Figure 1.5: Code puzzle completion problems in Looking Glass. Code puzzle completion problems combine the motivating aspects of code puzzles with the effective learning qualities of completion problems. In code puzzle completion problems, learners reassemble an animation from its initial parts (A) by dragging statements into the correct order (B).

of that behavior in order to locate the programming statement responsible for the animation change. Through this process, learners will hopefully acquire programming knowledge that they can then apply towards similar situations in their own projects. In the next section, I discuss my hypotheses for evaluating code puzzle completion problems.

1.3.2 Hypotheses

The main aim of this work is to provide a motivating approach that helps children acquire programming knowledge independently. I hypothesize that middle school children will be motivated to use code puzzle completion problems and that they will be able to successfully apply novel programming constructs in near transfer situations after completing code puzzle completion problems independently. Specifically:

Hypothesis I Middle school children will demonstrate greater learning efficiency of programming constructs by finishing training materials in less time and successfully completing more near transfer tasks, when training independently using code puzzle completion problems compared to current best practice, tutorials.

Hypothesis II Errors in code puzzles, commonly called distractors, will decrease learning efficiency by increasing middle school children's time to complete code puzzle completion problems while also decreasing their ability to successfully utilize programming constructs in near transfer situations.

Hypothesis III When given the freedom to choose their own informal learning resources, middle school children will perceive value in using code puzzle completion problems by expressing more interest, enjoyment, and preference towards using them compared to tutorials.

1.4 Contributions

In this work, I expand upon the body of work related to helping novices learn programming in independent contexts. Specifically my contributions are:

1. From formative testing, I share my lessons learned for developing code puzzle completion problems and an introductory curriculum [70].
2. I demonstrate that code puzzle completion problems are both efficient and effective for supporting novices in using programming constructs in near transfer situations [70].
3. I present evidence contrary to popular opinion that distractors in code puzzles may not be effective for supporting learners [73].
4. Through a qualitative study, I present evidence that independent learners opt to choose to use code puzzle completion problems because they find them enjoyable, challenging, and beneficial to improving their programming skills [69].
5. Based on the results of these evaluations, I demonstrate that code puzzle completion problems are a promising approach that can likely support goal-oriented users to learn programming in informal settings.
6. This work serves as a case study for leveraging cognitive load theory's effects in the domain of computer science in support of independent learning.
7. I have released the code puzzle completion problems to the public in the free download of Looking Glass (<https://lookingglass.wustl.edu/>). The source code is also available at <https://github.com/lookingglass-coding/lookingglass-ide> or <https://github.com/harmsk/puzzle-lookingglass>.

1.5 Summary

Over the remaining chapters, I fully explain my contributions towards using code puzzle completion for learning programming in informal settings. I first report on the initial development of the code puzzle completion problems. I then report on supporting my approach by conducting three summative evaluations to validate my hypotheses. Lastly, I followup with a discussion of future work with new directions for using this approach.

Developing Code Puzzles as Completion Problems In the next chapter (Chapter 2), I explore how to create code puzzles that also possess the qualities of completion problems. Specifically, I report the lessons learned from two formative evaluations used to develop the code puzzle completion problems.

Evaluating the Learning Effectiveness of Code Puzzle Completion Problems After developing the initial puzzle interface, mechanics, and curriculum, I needed to test this approach to see if in fact it can be effective for independent learners. I conducted a summative evaluation to validate Hypothesis I: whether code puzzle completion problems improve learning efficiency. I share the details of this evaluation in Chapter 3.

The Impact of Errors in Code Puzzle Completion Problems Current code puzzle systems frequently employ errors as part of their puzzle approach [107, 148]. Commonly these errors are distractor statements which are not part of the solution to the code puzzle. However, there currently exists little evidence to support that errors in code puzzles are beneficial to learning. In this chapter, we conducted a study to evaluate Hypothesis II: whether errors in code puzzles are detrimental to learning. See Chapter 4 for the results of errors in code puzzles and their effects on learning.

Learners’ Perceived Value of Code Puzzle Completion Problems After determining that code puzzle completion problems (without distractors) help novices to correctly use programming constructs in similar contexts, we conducted a study to explore whether independent learners’ perceive value in this approach. By looking at learners’ perceived value we hoped to discover whether independent users would be motivated to choose to use code puzzle completion problems on their own. We conducted a quantitative study to validate Hypothesis III: code puzzle completion problems are perceived as valuable, motivating, enjoyable and learners will opt to use them independently. In Chapter 5, I report on the results of this evaluation and other factors that participants perceived as valuable in tutorials and code puzzles.

Summary & Future Work In the last chapter (Chapter 6), I follow up with a discussion of the possible directions for future work. I discuss how the code puzzle completion problems can be improved by working to integrate other cognitive load theory effects, like worked examples. I also suggest approaches for extending the code puzzle completion problems by expanding the curriculum and using alternative completion strategies. Lastly, I offer suggestions for further supporting goal-oriented learners.

Chapter 2

Developing Code Puzzles as Completion Problems

Portions of this chapter were originally published as “Enabling Independent Learning of Programming Concepts through Programming Completion Puzzles” in the *Proceedings of the 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* [70].

In the previous chapter, I outlined how middle school children often lack formal opportunities to learn programming on their own. One alternative to helping children learn programming outside traditional avenues, may be to help them learn on their own, independently while following their own goals. Yet, in order for any independent approach to encourage middle school children to learn programming, it must also be compelling enough that children are willing spend their free time using it. Using the storytelling context of an existing novice programming environment, combined with a compelling activity like code puzzles that are also completion problems, may provide that alternative approach. Yet, no such system or tool currently exists to evaluate whether code puzzle completion problems are a motivating

activity that also enables novice programmers to apply constructs learned in the puzzles in similar situations.

The first step in evaluating this approach is to develop code puzzles that also function as completion problems. This means understanding what attributes make a completion problem for a computer program. It also means applying these principles to code puzzles in a way that provides a motivating experience that middle school children enjoy. In this chapter, I explore what properties of writing computer programs can be used to author programming completion problems. The results of this investigation are the code puzzle completion problems that I will then use to evaluate Hypotheses I, II, and III in Chapters 3, 4, and 5.

2.1 Completion Problems & Programming

Authoring completion problems for predictable and well-structured subjects, like algebra, is straightforward for educators: authors simply need to remove key steps in the solution. Unfortunately, complex subjects like programming, have less predictable steps, especially considering there are potentially many different ways to program the solution to the same problem. For example, if a function needs to be called multiple times, one solution is to use duplicate function calls, another is to place the function call inside a loop. In this case, it is not clear what are the *key* steps because depending on the solution strategy, the key steps are different.

The original completion problem research conducted by Van Merriënboer used a “partial solution in the form of a well-structured program” for their programming completion problems [192]. However, it is not clear what attributes specifically made these problems partial programs [192, 193, 194]. Based on the description provided in the text, it appears that students wrote code in the partial programs, but we do not know if they wrote missing lines

of code, functions, control flow constructs, or parameters, etc. [192, 193, 194]. Fortunately, Chang et al. and Garner later explicitly developed approaches specifically inspired by Van Merriënboer’s earlier work that clearly state their partial program strategies: fill-in-the-blank for partial lines of code [21] or placing lines of code in the correct order [52, 53]. In the Chang et al. work, students are given a program where select lines have been partially replaced with a blank [21]. Student complete the program by writing code to fill-in-the-blank for all the partial lines of code. In Garner’s tool, CORT, students are given a set of lines that they must place into the correct location in the partial program [51, 52, 53, 54]; students do not write code, they simply insert whole lines into the program.

Additionally, both Van Merriënboer, Chang et al., and Garner’s work were specifically designed to be used as practice alongside traditional classroom instruction. All authors used setups where the completion problems appear more like homework exercises than something a middle school child might opt to use on their own. These homework like problems appear to be more like generating the Fibonacci sequence, which probably does not interest users who are following their own goals. Using a motivating context, like animated code puzzles, combined with completion problems may provide an alternative to motivate and educate these informal users.

In this chapter, I report on two formative evaluations we conducted to explore the potential for creating code puzzles that are also completion problems. From these two formative evaluations we developed *code puzzle completion problems*. In the next section, I describe the process and the lessons we learned from developing the code puzzle interface and mechanics in our first formative evaluation. I then report on our second formative evaluation for authoring an introductory curriculum for introducing basic programming constructs to novices. Throughout both formative evaluations, I discuss our changes through the lens of cognitive load theory and how we believe our changes help to manage the cognitive load of our users. I also

describe a second round of revisions we made to the interface and curriculum to release our code puzzle completion problems to the public. Lastly, I follow up with a discussion of the different completion problem approaches for programming and how our code puzzle completion problems align with these existing strategies.

2.2 Formative Evaluation I:

Puzzle Interface and Mechanics

In our first formative evaluation we designed the puzzle interface and mechanics through 10 iterations of testing. We recruited 23 users between the ages of 10 and 15 years at the St. Louis Science Center to participate in this evaluation. During each study session, each participant worked independently for 30 minutes while we observed them. Following the study, we compensated participants with a \$5 gift certificate.

When we set out to create code puzzle inspired completion problems, we initially came up with the idea of a *code scramble*. We used the code editor of the novice programming environment, Looking Glass [117], and *scrambled* the order of the statements in the program's main. We asked users to move the statements back in the correct order which then completed an animation. See Figure 2.1 for the first prototype of these code scramble puzzles.

Starting with our initial code scrambles, we used a variety of methods to iteratively design our final puzzle mechanics and interface. We used mock-ups, paper prototyping, Wizard of Oz testing, and working prototypes which enabled us to effectively address each problem we encountered throughout the evaluation. In this section, I report on each of the problems we faced by sharing the lessons we learned as well as the methodology that led us to that solution.

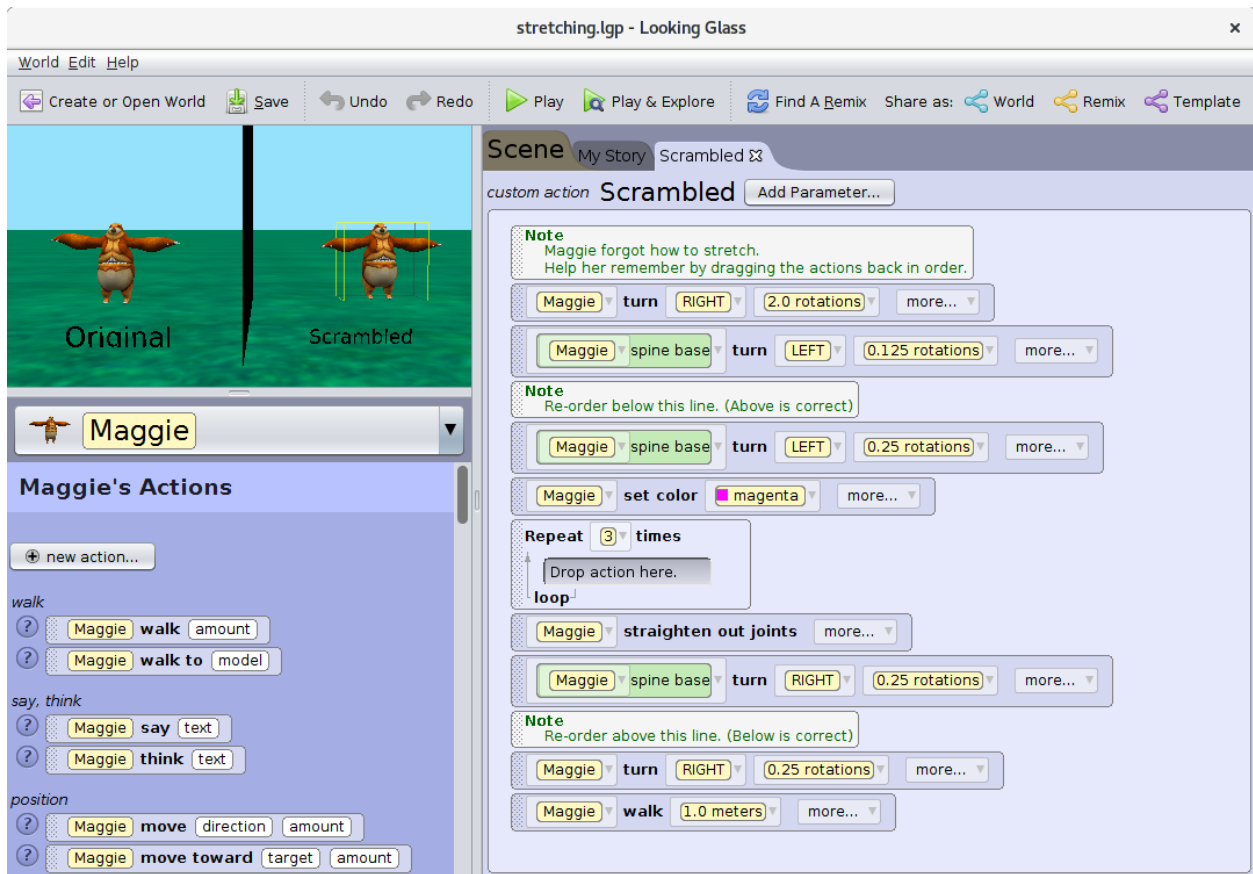


Figure 2.1: In our first iteration of formative testing, we created *code scrambles*. In a code scramble, users reorder the code statements in a conventional code editor so that the scrambled animation matches the correct animation.

2.2.1 Methods & Lessons Learned

From our formative evaluation we learned several important lessons for using code puzzles as completion problems. In this section, I share several of the lessons we learned throughout our iterative design process and the methodology we used for each lesson.

Limit the editable dimensions of the puzzle.

Our initial code scramble iteration simply used the code editor of an existing programming environment (Figure 2.1). In a typical code editor, a programmer can change the code in several dimensions: she may add new statements, and edit, move, or delete existing statements. Even though our directions stated to only reorder the statements, participants changed the code beyond reordering. Users would insert new statements, delete existing statements, and change the values of arguments. We discovered that using a code editor gives users too many dimensions to manipulate. The many editable dimensions also likely impose a high extraneous cognitive load to completing the code scramble puzzle.

To focus the learner on only the reordering dimension, we tried using reorder-only statements. We created a hybrid paper and Wizard of Oz prototype as shown in Figure 2.2. In this prototype, participants could only change the order of the statements by using the up and down buttons on each statement. Behind the scenes a researcher worked on a laptop authoring the code in Looking Glass that matched the paper prototype. When the user would “click” *play* in the paper prototype, the behind the scenes researcher would play the animation on an attached monitor. While we found that this effectively limited the editable dimensions to reordering, it was a rather unsatisfying and cumbersome way to edit the code puzzles.

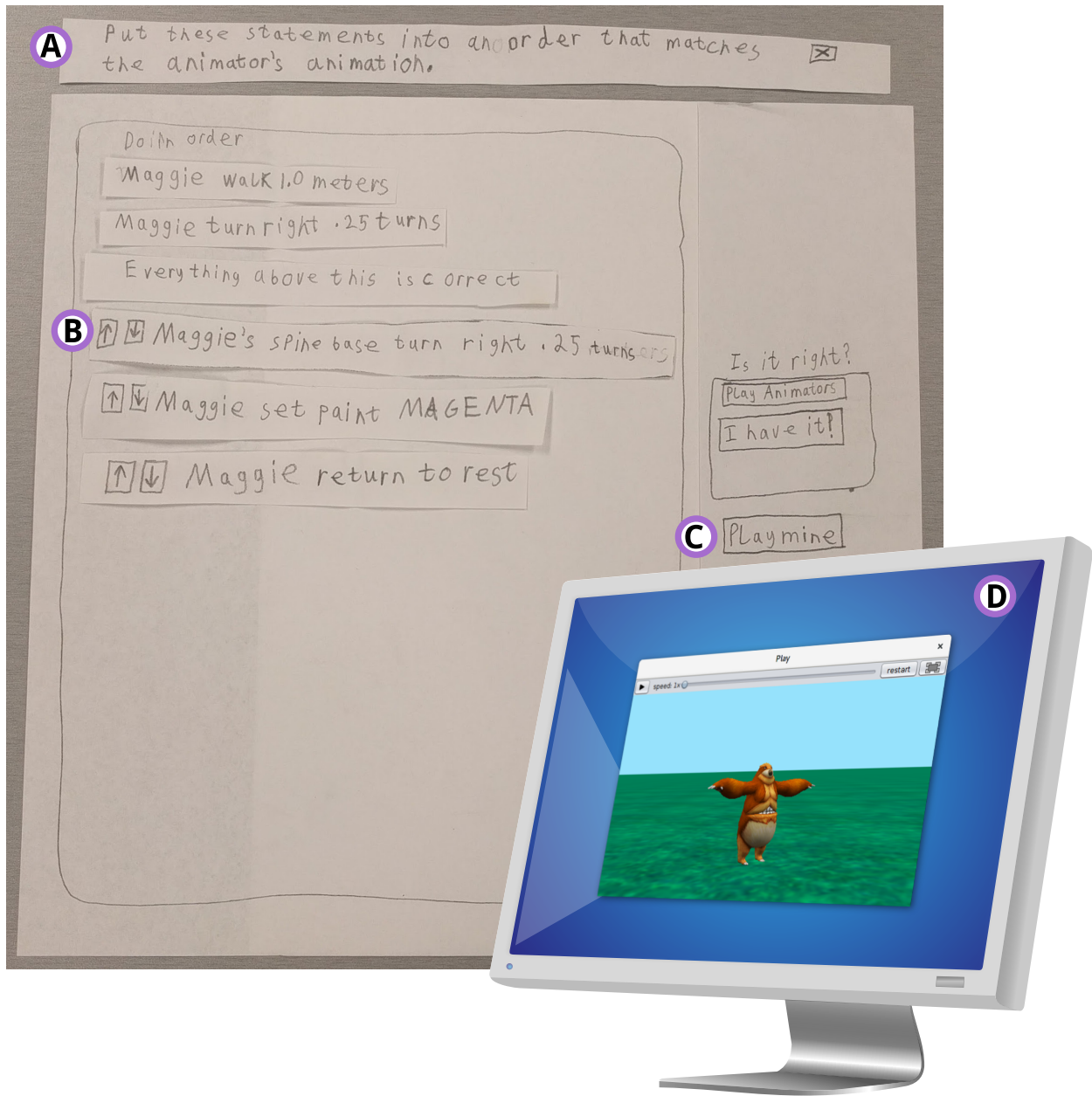


Figure 2.2: A hybrid paper and Wizard of Oz prototype of editing puzzles by using reorder-only statements. Users place the statements in the correct order in the paper prototype (A) by using the up and down buttons (B) on each programming statement. Behind the scene, a researcher authors the same program in Looking Glass, so that when the user plays the animation (C) the researcher can execute the animation on a separate monitor (D).

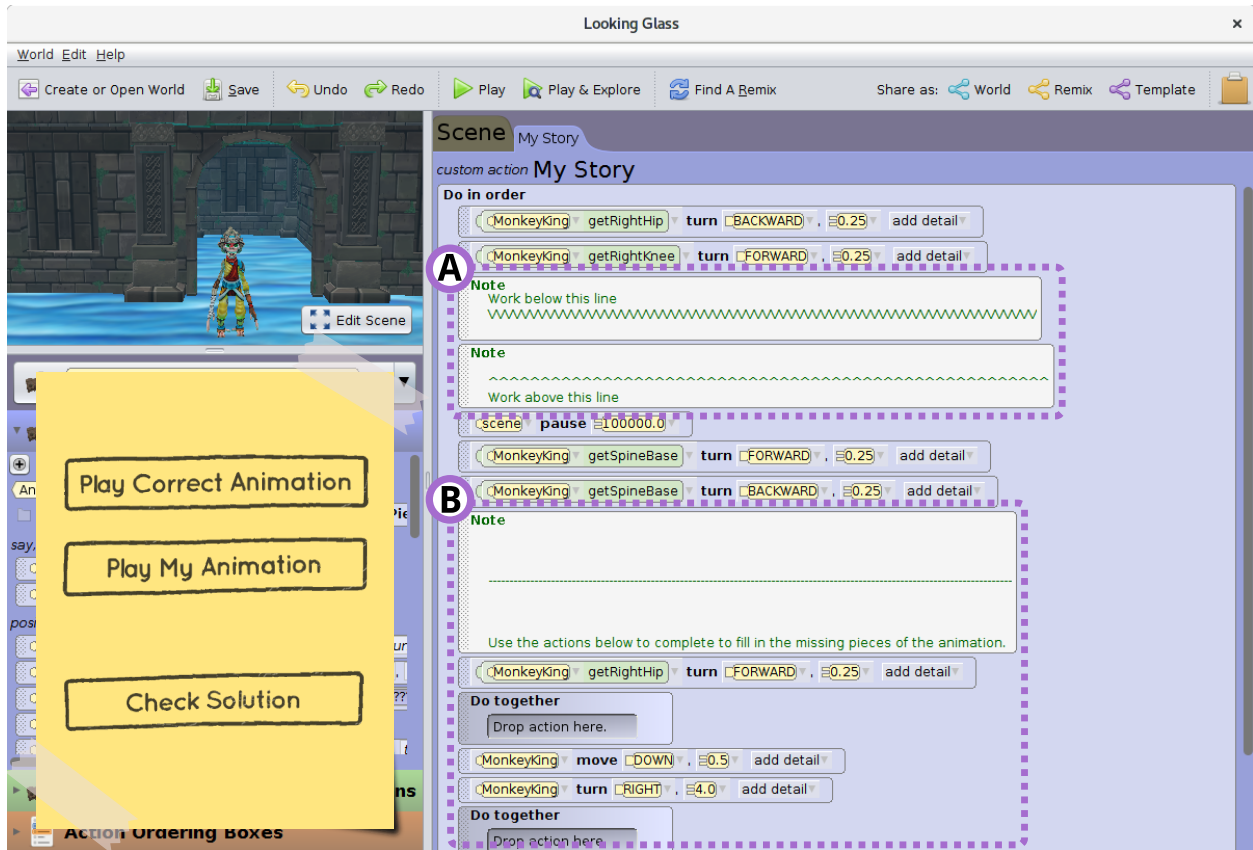


Figure 2.3: An early prototype of using reassembly style puzzles instead of the reorder style code scrambles. We divided a function with code comments into the puzzle workspace (A) and the statement bin (B). We then asked participants to place the statements at the end of the main (B) into the first part of the main (A) to complete the puzzle.

Next, we returned to using a code editor, but instead of asking participants to reorder statements in place, we asked participants to reassemble the statements by moving them from a dedicated bin. We tried creating a bin of statements by using comments in the code editor, as shown in Figure 2.3. The comments divided the editor into the puzzle work area and the available statements bin. In this iteration we also hid some of the other editable dimensions of the code editor by taping a small paper prototype to the screen where the users could play the animations and check their solution. We found that the reassembly strategy worked well, however even with the distinct bins made with comments, participants still changed the code by changing parameters and deleting statement instead of reordering.

In our final version, we created a working prototype where users were limited to editing the puzzles by only reassembling statements into the correct order. This interface still used the statement bin, but instead of being below the puzzle code, the bin is now outside of the code editor as seen in Figure 2.10. We believe that by breaking the interface into two distinct areas: the statement bin and puzzle workspace, along with only allowing reassembly, this helped to reduce the extraneous cognitive load that our earlier prototypes imposed on learners. With the extra detrimental cognitive load lessened, we observed participants having a much easier time completing these code puzzles.

Only show the user's work in the program's output.

Our early code scramble prototypes used reordering, instead of reassembly. Through testing the code scrambles, we realized that when participants viewed the program's output, the output showed both the parts of the program that the user had reordered and the parts that they had not yet reordered. Mixing both elements in the output made it difficult for users to tell where their changes ended and where the scrambled code began. We imagine that the mixing of outputs likely increased the extraneous cognitive load of the task. To

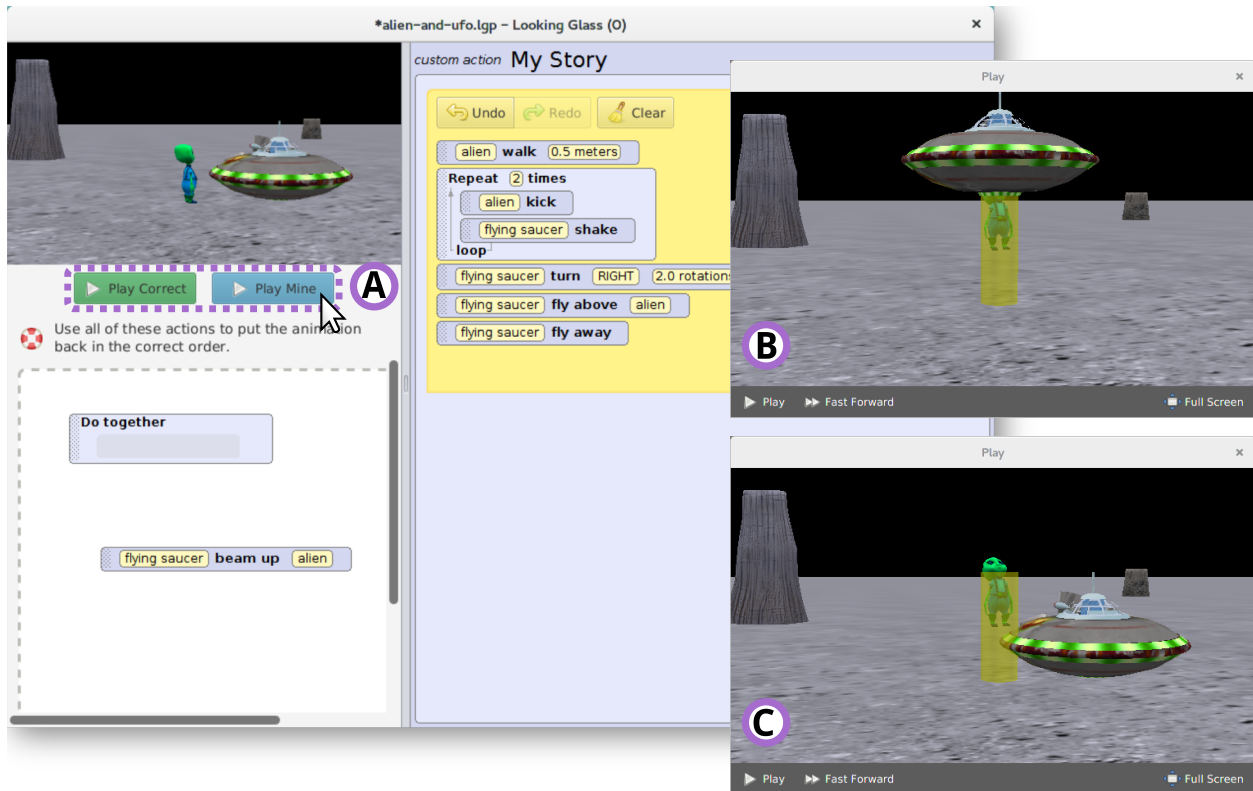


Figure 2.4: In earlier iterations, participants could play (A) the correct animation (B) or their animation (C) in separate windows.

address this we switched from the reorder strategy to the reassembly strategy. Users now place the statements in the correct order by dragging the unused statements from a bin and placing them into the correct order in the puzzle workspace as shown in Figure 2.10. This ensures that only the work that the user has completed is shown in the program’s output, thus leaving any additional cognitive resources to process the novel information.

When executing a program, limit distractions and focus the user’s attention on the program’s output.

To complete a programming puzzle, a user needs to know what the correct output looks like and how it compares to the puzzle’s output. We initially combined the correct output and the puzzle output side-by-side, as shown in Figure 2.1. Unfortunately, users spent most of their

time looking back and forth between both outputs, missing key moments. Any distractions, including looking at the actual code, caused users to miss important details in the program output that would aid self-explanation and help them complete the puzzle.

In order to reduce the extraneous cognitive load imposed by watching and comparing two animations, we separated playing the correct and the user's animation. We gave users two buttons, one button for playing the correct animation and one button for playing their animation. Both buttons would then bring up a window for that animation as shown in Figure 2.4. However, users quickly realized they could quickly click both buttons at the same time and watch both animations concurrently. The two button approach, enabled users to default back to the original behavior of comparing animations. This probably imposed extraneous cognitive load, thereby making the task more difficult on these users.

We tried to address this by only allowing users to toggle between the correct or their animation in the output window; users could not play both animations at the same time. Yet, with only one window, users would move the window so that they could concurrently look at their code and the program output simultaneously. This once again, split the user's attention, likely increasing their extraneous cognitive load.

In our final version, we play the output in an overlay pane (Figure 2.5) that obscures the code and only plays one animation at a time. Without distractions, the user can carefully study the nuances of the program's behavior and later tie the behavior to the appropriate puzzle statements, hopefully encouraging self-explanation.



Figure 2.5: The puzzle feedback overlay pane obscures the code and only allows users to play the correct or their animation at a time.

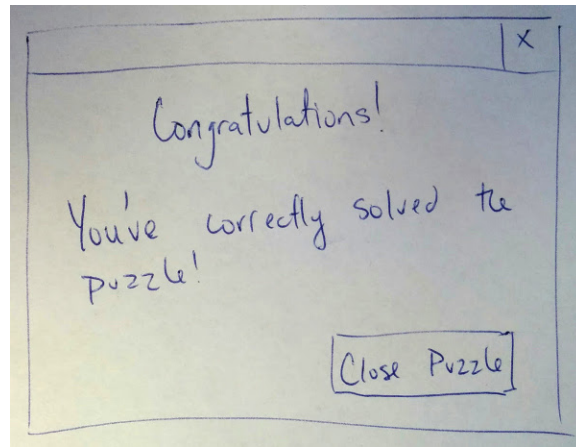


Figure 2.6: This feedback dialog was used early versions of our prototype. When users correctly completed a puzzle, a researcher placed this dialog on the user's screen.

Provide the user with ambiguous and incremental feedback towards the puzzle's solution.

In many of our first iterations, the user only received a notification that the puzzle was complete once every statement was in the correct order. A researcher would place the paper prototype dialog shown in Figure 2.6 on the user's computer screen. Prior to receiving this notification, many participants believed they had correctly completed the puzzle, leading many to terminate the puzzle prematurely.

We designed an incremental feedback indicator to provide feedback throughout solving a puzzle. Our first incremental feedback indicator used a one-to-one or direct mapping to the puzzle solution as shown in Figure 2.7. Unfortunately, direct mapping was easy to game; participants noticed they could count the dots and rearrange the statements until the progress indicator increased without actually having to try to complete the puzzle. This likely reduced participants' germane cognitive load because participants were using the indicator to get the solution, instead thinking through the solution for themselves.

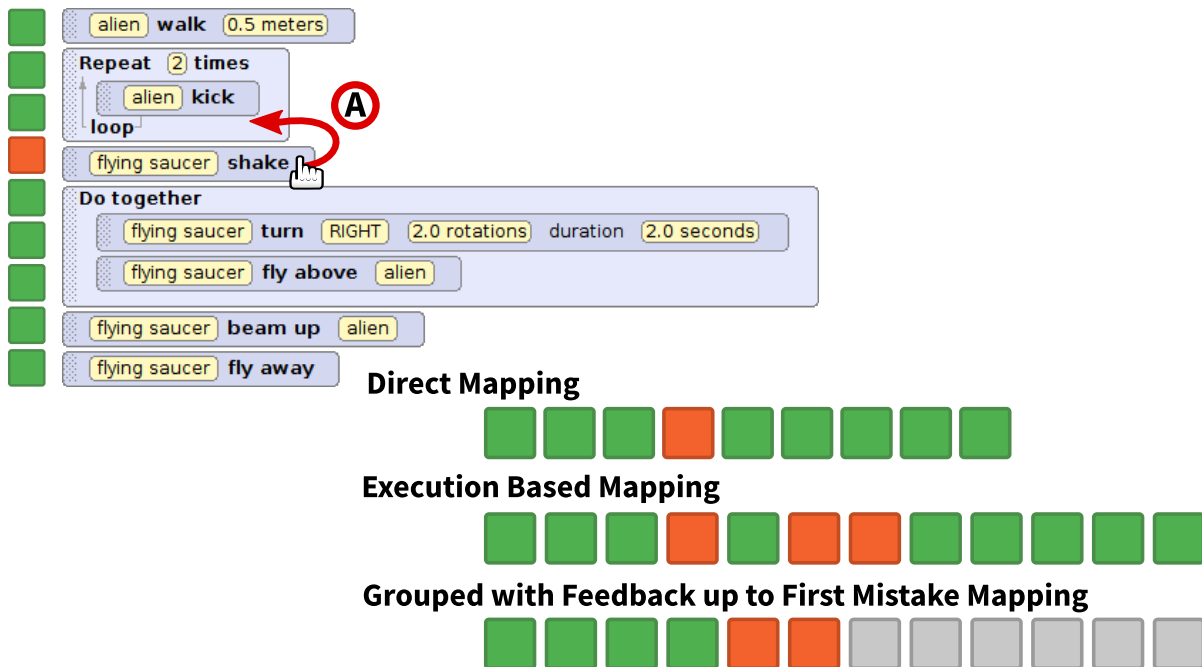


Figure 2.7: We tried several different feedback indicators while trying to provide incremental and ambiguous feedback. The `shake` statement (A) should be inside the *repeat* block. All feedback indicators show the current state of this code puzzle.

To limit this type of behavior and to encourage engagement, we increased the ambiguity within our feedback indicator by drawing a dot for every statement *executed*. In the execution based indicator, every time an incorrect statement executes, we draw an orange dot, instead of a green dot. Because many of our puzzles execute statements in parallel or inside of loops, the execution based feedback was less easy to game. It was also more difficult to understand by users because they had difficulty determining where they made their first mistake.

Figure 2.10-F shows our final incremental and ambiguous progress indicator. Throughout our formative testing, all users used a top-down approach to solve the puzzles. Our progress indicator is designed to provide feedback for the first error in the puzzle, which supports this top-down approach. This indicator shows a new dot each time a statement is executed. A green dot means that the currently executing statement is correct. However, when the first incorrect statement in the puzzle (the `shake` method in Figure 2.7) is executed, a new

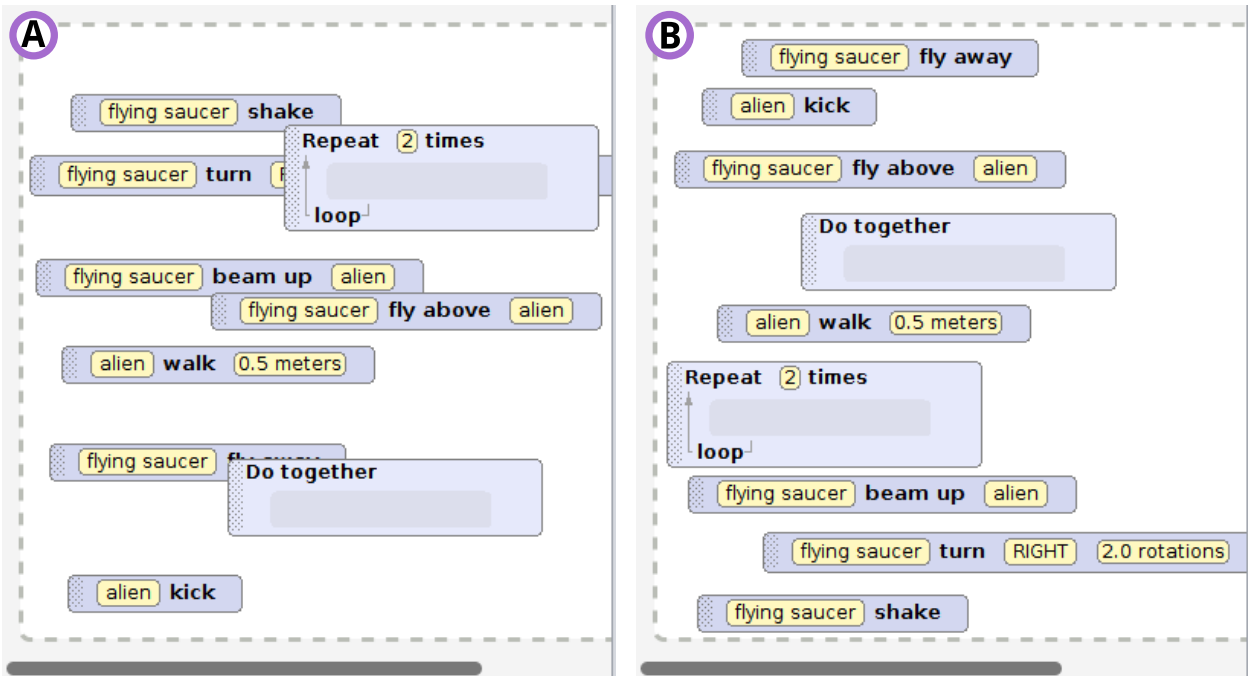


Figure 2.8: The left statement bin (A) has overlapping statements which obscure some of the puzzle statements. The right (B) is laid out using an algorithm which prevents overlapping statements.

orange dot is added. Any executing statements that follow the incorrect statement are added as gray dots. For additional ambiguity, we group all similarly colored dots together; all green dots are grouped together, followed by all orange dots, and then the gray dots (see Figure 2.7). Through this testing, we discovered that a feedback indicator that is execution based with grouped feedback up to the first mistake seems to strike the right balance between incremental and ambiguous feedback.

Provide learners with a comprehensive look of all of the information they need to solve the puzzle.

When solving a puzzle, all of the pieces need to come together in the end for the solution. Piecing together this solution can be difficult if you do not know what pieces you have available. In our puzzle statement bin (Figure 2.10-A), we initially laid out the statements

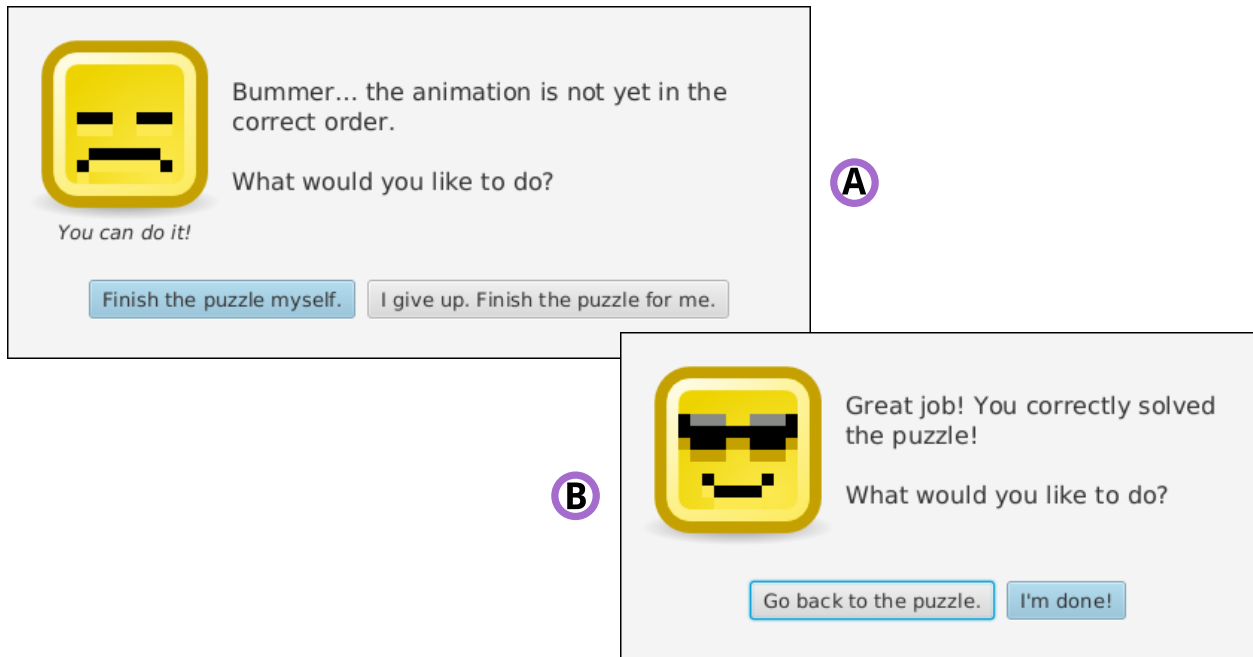


Figure 2.9: When a user chooses to quit a puzzle prematurely, they are shown a dialog to encourage them to continue working (A). Once the user successfully solves a puzzle they are rewarded with a job well done (B).

randomly without consideration for overlapping statements. During formative testing, we observed that the most successful participants often re-laid out the statements so that all statements were visible. From a cognitive load perspective, overlapping statements probably imposed an extraneous cognitive load due to some learners trying to incorrectly align output with a statement that was hidden behind another one. To address this problem, we changed our statement layout algorithm to prevent overlap on the initial placement of statements as shown in Figure 2.8.

Give encouragement and gently nudge learners when necessary.

In independent contexts there is no teacher or person of authority to push a learner to continue working. Throughout our designs we tried to build in subtle encouragement and rewards for users. In our first iterations participants were quick to give up on some puzzles.

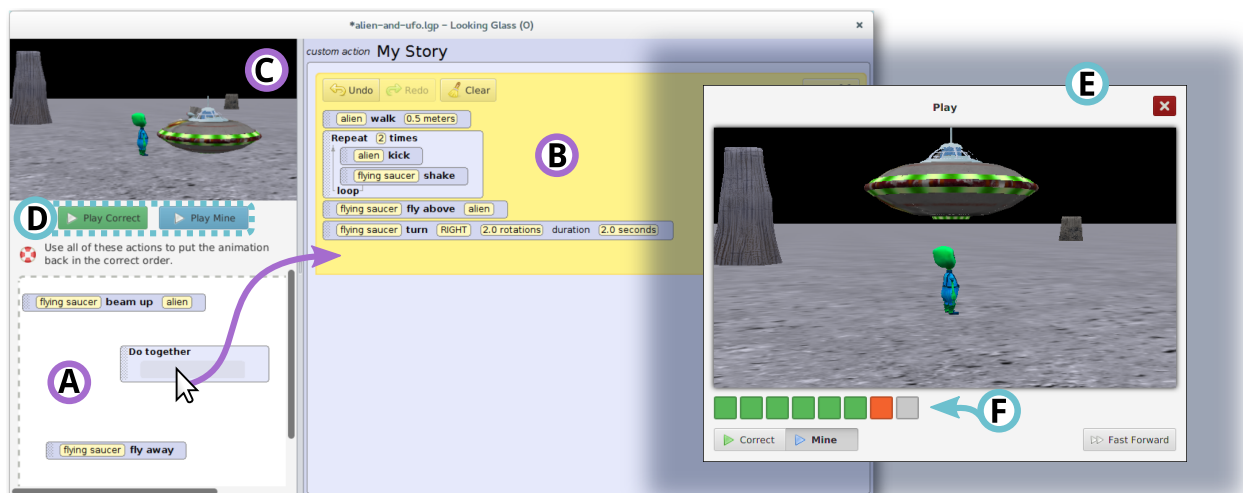


Figure 2.10: Our initial code puzzle completion problems interface. Users drag statements from the unused puzzle statements bin (A) into the puzzle workspace (B). (C) shows the initial starting state of the program. Users view the correct program output or their puzzle output by clicking the appropriate play button (D). Executing programs are shown in the program output overlay (E). When viewing the puzzle’s output, users receive feedback on their progress from the progress indicator (F).

The interface also made quitting very easy. To encourage these users to keep working on the puzzle, we changed the language of the quit dialog to ask participants if they wanted to give up (see Figure 2.9). In later testing we observed that users were less quick to quit because they often hesitated to click the *give up* button. In fact, many of the participants who hesitated often returned to the puzzle to finish it. We also used faces strategically placed throughout our interface (see Figure 2.9) because humans intrinsically find them rewarding. During testing, many participants commented how they really like the smiley face reward after completing a puzzle, as one participant stated, “I got the guy!”

2.2.2 Interface & Mechanics Overview

At the end of our formative evaluation we ended up with the interface and mechanics as shown in Figure 2.10. Users complete the puzzles by dragging statements from the unused

statement bin (Figure 2.10-A) into the puzzle workspace (Figure 2.10-B). Users must use all of the statements in the bin to complete the puzzle. If users need to view the correct program output or view the puzzle’s output, they click the appropriate play button (Figure 2.10-D). This brings up the program output overlay pane (Figure 2.10-E). When the user is viewing the puzzle output, the puzzle feedback indicator (Figure 2.10-F) will show the user’s current progress. The puzzle is correct when all of the indicator’s dots are green. See Appendix A for screenshots of the entire puzzle interface.

2.3 Formative Evaluation II: Introductory Curriculum

After developing the puzzle interface and mechanics, we then moved to creating the actual content for the code puzzles. We based our puzzle curriculum on concepts found in the middle school level (2) of the Computer Science Teacher’s Association’s curricular standards [175]. We selected four constructs at the introductory level that are useful for authoring 3D animations: repeated execution (*repeat*), parallel execution (*do together*), parallel nested within repeated execution (*repeat { do together }*), and repeated nested within parallel execution (*do together { repeat }*).

We recruited 21 participants between the ages of 10 and 15 years from the Academy of Science of St. Louis mailing list to participate in this formative evaluation. During each individual session we asked participants to complete the puzzles in our curriculum. Each session lasted 90 minutes. In total, we tested 8 iterations of our puzzle curriculum on our participants. Afterwards, we compensated participants with a \$10 gift certificate.

In this section, I discuss several lessons we learned through iteratively testing our code puzzles over 8 iterations. I also I describe the resulting curriculum from this second formative evaluation.

2.3.1 Lessons Learned

Through this evaluation we discovered how difficult and time consuming it is to author effective puzzles. Most of our puzzles went through a minimum of 5 iterations of changes and testing. Through this testing we realized that often minor details appear to introduce high extraneous cognitive load and trip up learners in unexpected ways. When authoring puzzles, I strongly encourage rigorous testing of each individual puzzle to eliminate surprise sources of extraneous cognitive load. In the following section, I discuss some of the lessons we learned trying to effectively manage learners' cognitive load while creating our puzzle curriculum.

Author puzzle programs with motivating scenarios.

As I discussed in Chapter 1, completion problems are partial worked examples. In creating our first puzzles that double as completion problems, we initially approached their creation with the mindset of authoring examples. Thus, our early puzzles tended to center around demonstrating construct behavior. Figure 2.11 shows one of these early puzzles which focus on programming constructs. These example-based puzzles were not compelling to users; users were unmotivated to make an ostrich move its neck up and down three times. This was especially problematic given that we intended for our users to complete these puzzles on their own independently. From informal interviews, we realized that we needed to provide users with puzzles that excited them.

One approach we found that motivated users was to author puzzles with a problem-solution scenario. These puzzles contain a compelling initial scene with a scenario that contains a problem and resolution. For example, the *Interstellar Travel Troubleshooting* puzzle in Figure 2.15 shows a initial scene of an alien on a planet. After viewing the correct output, users realize that they have to help the alien leave the planet by repairing his broken spaceship.

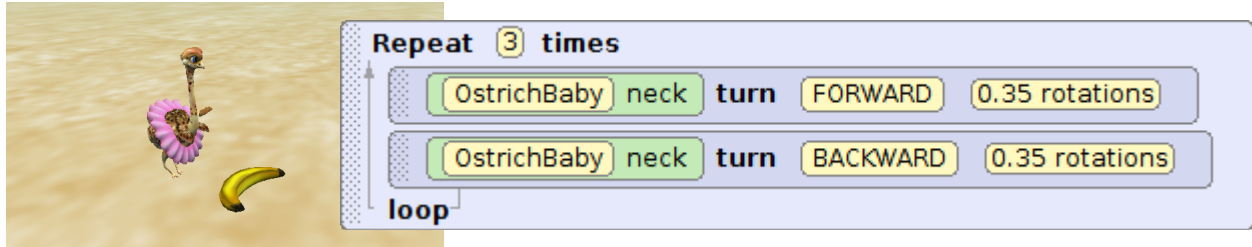


Figure 2.11: In our earlier puzzles we were more concerned about demonstrating a programming construct, rather than using constructs in compelling scenarios. This puzzle demonstrates repeated execution, but fails to motivate users to care why the ostrich is eating a banana.

Users are motivated to finish the puzzle in order to help the alien escape the planet. During testing, we observed that participants tended to work harder to complete these compelling animations. Additionally, the initial scenes shown in Figure 2.15, often excited participants because they wanted try out each puzzle in order to see what would happen in the scenario.

Author puzzle programs with memorable segments.

As part of authoring compelling animations, we originally tried authoring puzzles with unique characters that do interesting things. Users really liked these animations because they were entertaining to watch, however they tended to be a little difficult to actually solve. Figure 2.12 shows an early puzzle that was entertaining, but really difficult to remember due to the random actions in the animation. We found during testing that if an animation or part of an animation was not easily memorable, then users had to continuously flip between the correct output and puzzle output to know what to work on next. From a cognitive load perspective, it is not surprising these random statement puzzles are more difficult to remember because each random statement needs to be processed in working memory. This likely imposed a relatively high extraneous cognitive load on users trying to complete these tasks.

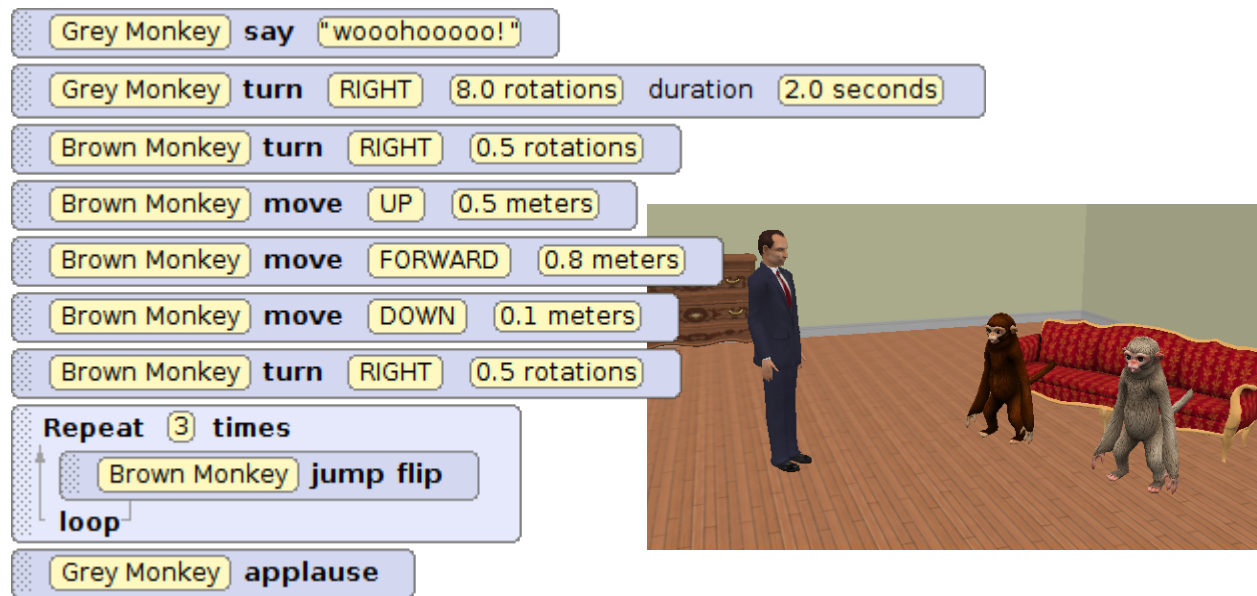


Figure 2.12: An early puzzle that was entertaining to watch, but difficult to solve due to the seemingly random nature of many of the programming statements.

To lessen this extraneous cognitive load, we realized that a user should be able to remember the part of the puzzle they are working on without the need to continuously review the correct output. A simple approach is to author puzzle programs so that there are memorable segments. Figure 2.13 shows an example puzzle with distinct memorable segments about an alien repairing his flying saucer in order to leave a planet. In first segment the alien fixes the flying saucer, in the next segment the flying saucer starts up, and in final segment the alien leaves the planet in the flying saucer. Hopefully each segment is small enough to fit in working memory, but large enough that a user can engage in self-explanation about what statements could cause the output for that segment. We also strongly encourage puzzle authors to use exaggerated or over-the-top animations because they seem to be easier to remember.

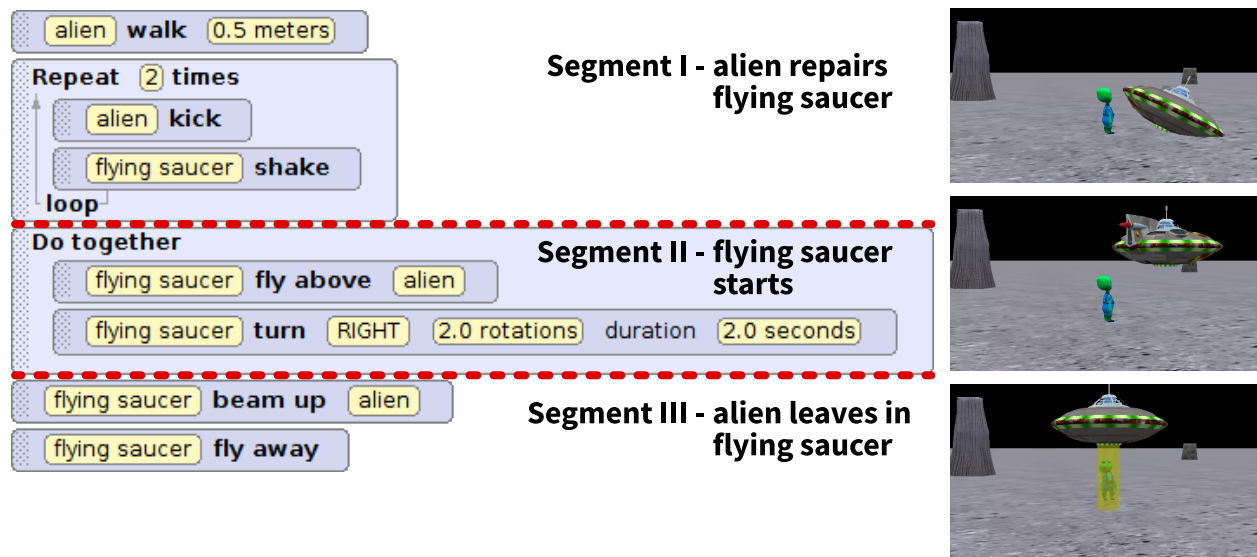


Figure 2.13: In our final puzzles, we authored animations with easily memorable segments. With memorable segments, users can work on each chunk of the puzzle without needing to continuously review the correct output for the next piece to work on.

Provide a challenge without being tricky.

Over the course of our formative sessions, we found that participants sometimes had very different reactions to difficult puzzles. In some cases, completing a difficult puzzle was a very rewarding experience. In others, participants remained frustrated, even after finding the correct solution. Participants' descriptions of these puzzles reflect these two kinds of reactions. Participants often described rewarding puzzles as *challenging* and frustrating puzzles as *tricky*. Through examining a collection of tricky and challenging puzzles, we noticed that what differentiates these puzzles is the source of their difficulty.

Tricky puzzles pull users' mental resources away from the puzzle's programming constructs and instead expend their resources on extraneous details. Frequently, these extraneous details require participants to observe subtle details in the animation or notice subtle differences between statements, as shown in Figure 2.14. For example, an early puzzle included small rotations of a character's body part that participants struggled to perceive when viewing the

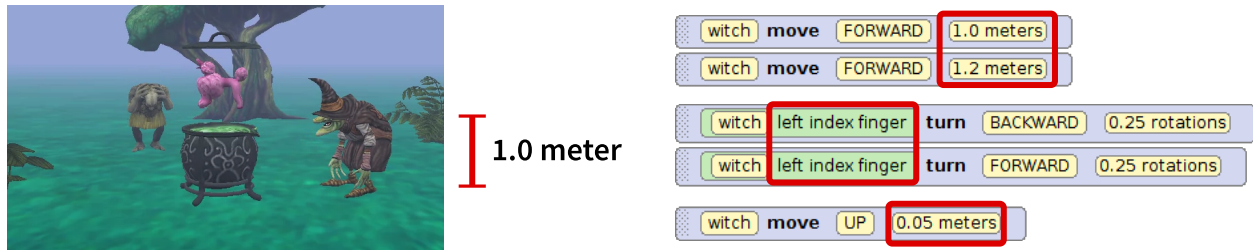


Figure 2.14: Tricky puzzles often uses small changes that are difficult to notice when the animation is playing.

puzzle output. In another case, a puzzle included nearly identical statements, one moved 1.0 meters and the other 1.2 meters. Participants frequently swapped these two statements in the puzzle and perceived the resulting output as matching the correct one. While participants typically solve tricky puzzles, they often finish with a sense that they have wasted a lot of time on unimportant details. Further, these subtle differences likely expended working memory resources away from learning the programming constructs at hand.

In contrast, we found that users really enjoyed challenging puzzles. Frequently, users commented that they liked the challenging puzzles best, “I thought this one was a little challenging, but I liked it!” The source of difficulty in challenging puzzles was the puzzle’s new programming construct. These puzzles were still difficult for users, but this type of difficulty feels authentic because users finish with a new skill; spending time on it feels justified.

However, we note that too many challenging puzzles can be overwhelming for users. During one formative testing session, we replaced all of our easy puzzles with challenging ones. In this session, users no longer enjoyed the puzzles. Providing difficulty variation between different puzzles seemed to result in the best user experience as reflected in our final curriculum shown in Table 2.1.

Table 2.1: Our initial code puzzle curriculum contains six puzzles with an additional introductory puzzle (0).

ID	Title	Constructs	Difficulty	Statements
0	Satellite Inspection	puzzle mechanics	trivial	4
1	Dizzy Walrus	sequential execution	easy	7
2	Monkey Business	repeated execution	challenging	10
3	The Spooky Trick	parallel execution	challenging	9
4	Interstellar Travel Trouble...	repeated & parallel	easy	9
5	Shark Snack	parallel { repeated }	challenging	8
6	Whack-a-Yeti	repeated { parallel }	challenging	10

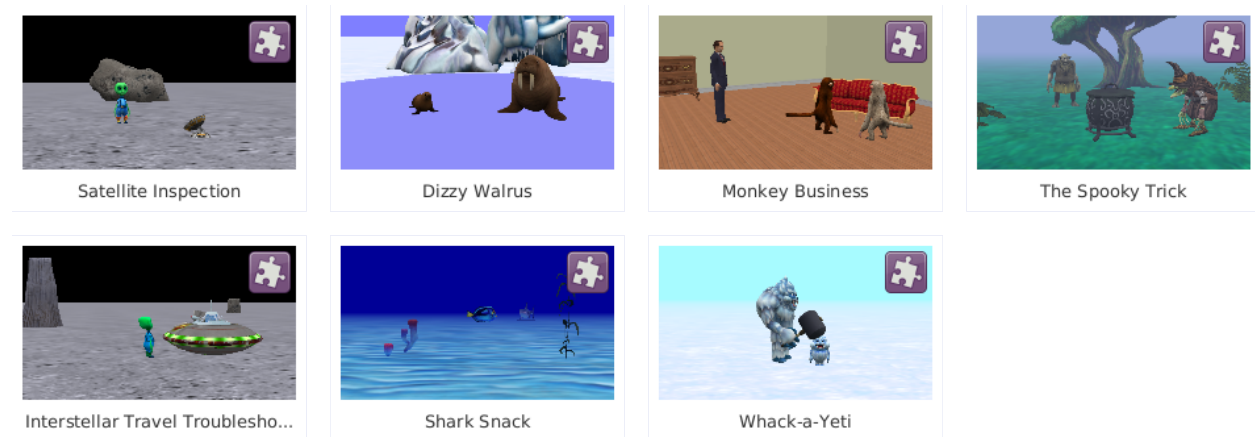


Figure 2.15: The seven animations of the code puzzles in our initial curriculum.

2.3.2 Curriculum Overview

The resulting curriculum for this formative evaluation contains six puzzles with an additional introductory mechanics puzzle. See Table 2.1 for the curriculum and Figure 2.15 for an overview of the animations in the curriculum. See Appendix B for the complete programs for each of the code puzzles.

2.4 Revised Code Puzzle Completion Problems

After developing the initial code puzzle completion problems described in sections 2.2 and 2.3 we conducted an evaluation of Hypothesis I (see Chapter 3). The results of this study demonstrated that code puzzle completion problems are an effective method for learning programming independently. Based on this positive result, we decided to release our code puzzle completion problems to the public. However, before releasing the puzzles to the public we decided to make several improvements to the interface and curriculum. We folded the development of our revisions into the formative evaluation presented in Chapter 4 and the pilot study presented in Chapter 5. We then used the revised interface and curriculum for the studies presented in Chapters 4 and 5. In this section, I share the revisions we made to polish the interface and add additional puzzles to the introductory curriculum.

2.4.1 Revised Puzzle Interface

We updated the look of the puzzle interface to be more polished and consistent with the existing design of Looking Glass. Figure 2.16 shows the updated look for the code puzzle interface in Looking Glass. The changes we made are almost all cosmetic, aside from changing

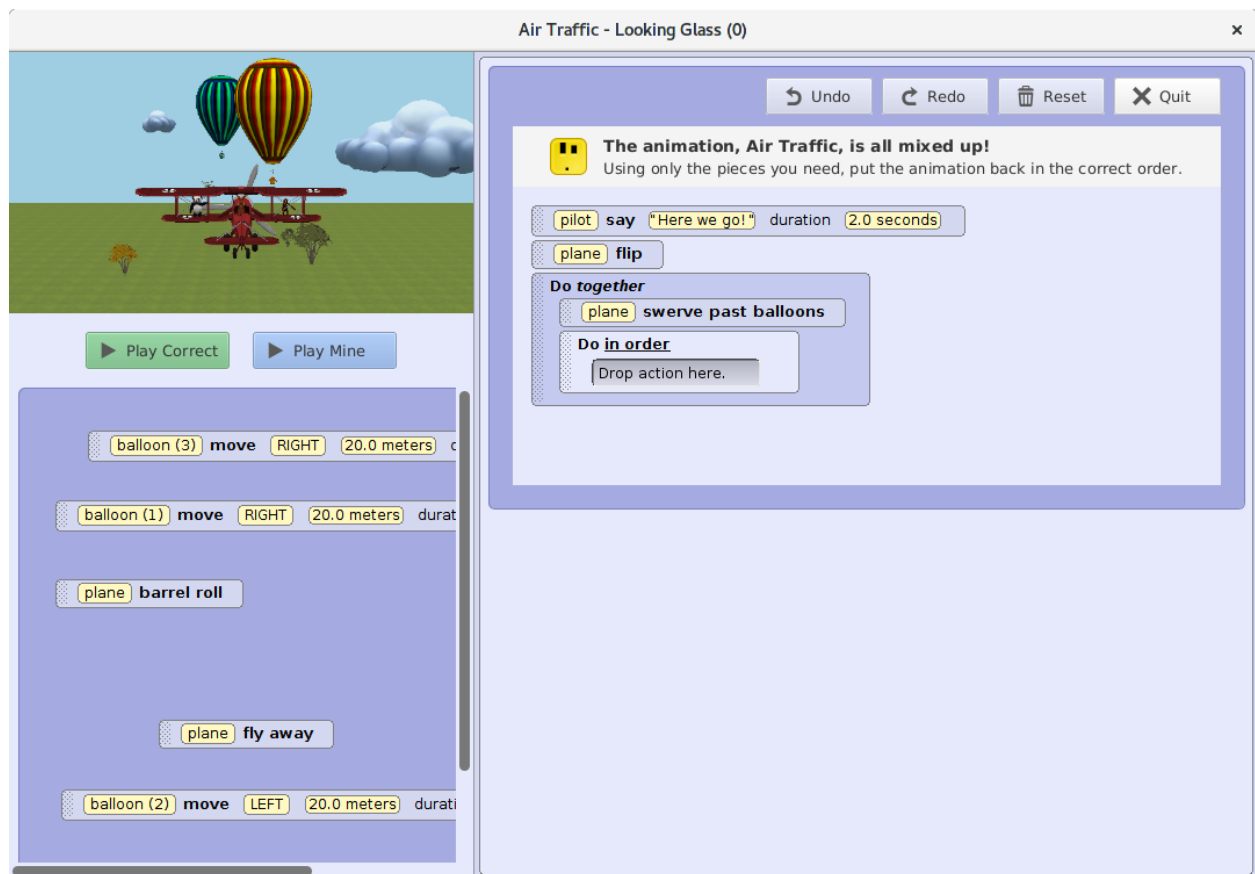


Figure 2.16: We updated the design of the puzzle interface to appear more polished and consistent with the overall look of Looking Glass.

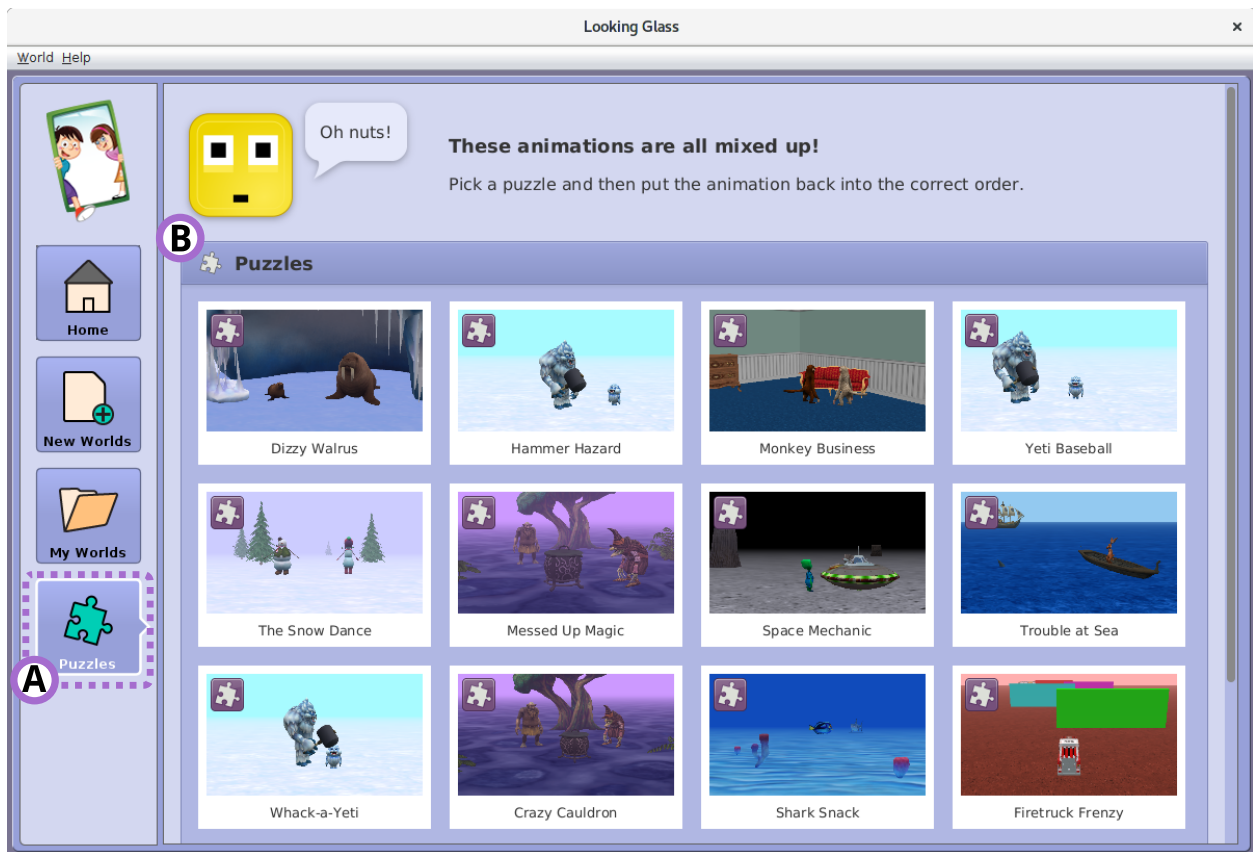


Figure 2.17: We added a puzzles as an activity (A) to the welcome screen in Looking Glass. Once the puzzle activity is selected (A), users can pick which puzzle they would like to work on (B).

the location and content of some of the on screen text; the mechanics are exactly the same as the version produced in the first formative evaluation (Section 2.2).

We also added the puzzles as an activity to the welcome screen in Looking Glass, as shown in Figure 2.17. Now when users launch Looking Glass they are given the option to work on their own programs (known as *worlds*) or work on some puzzles. The puzzle selection activity is an important part of our strategy to provide a learning opportunity within an existing environment that complements users' goals rather than distract from them. See Appendix C for screenshots of the entire revised puzzle interface.

Table 2.2: We revised our curriculum to now include 14 puzzles, including several showcasing a new programming construct: sequential execution nested within parallel execution.

ID	Title	Construct(s)	Difficulty	Statements
1	Dizzy Walrus	sequential execution	easy	7
2	Hammer Hazard	repeated execution	medium	9
3	Monkey Business	repeated execution	challenging	10
4	Yeti Baseball	parallel execution	medium	10
5	The Snow Dance	parallel execution	challenging	11
6	Messed Up Magic	parallel execution	challenging	11
7	Interstellar Travel Trouble... ^a	repeated & parallel	easy	9
8	Trouble at Sea	repeated { repeated }	challenging	9
9	Shark Snack	parallel { repeated }	challenging	11
10	Crazy Cauldron	repeated { parallel }	challenging	10
11	Whack-a-Yeti	repeated { parallel }	challenging	10
12	Firetruck Frenzy	parallel { sequential }	medium	11
13	Air Traffic	parallel { sequential }	challenging	10
14	Polar Surprise	parallel { sequential }	challenging	11

^a *Interstellar Travel Troubleshooting* was later renamed *Space Mechanic* in the public release.

2.4.2 Revised Curriculum

We revised the introductory curriculum by creating additional puzzles as well as adding a new programming construct to the curriculum, sequential execution nested inside of parallel execution (*do together { do in order }*). Table 2.2 shows the updated curriculum which now contains 14 puzzles. See Figure 2.18 for a preview of the new animations from the curriculum. During testing we found that users had no difficulty learning the puzzle mechanics, so we eliminated the introductory mechanics puzzle. See Appendix D for the complete puzzles in the revised curriculum.

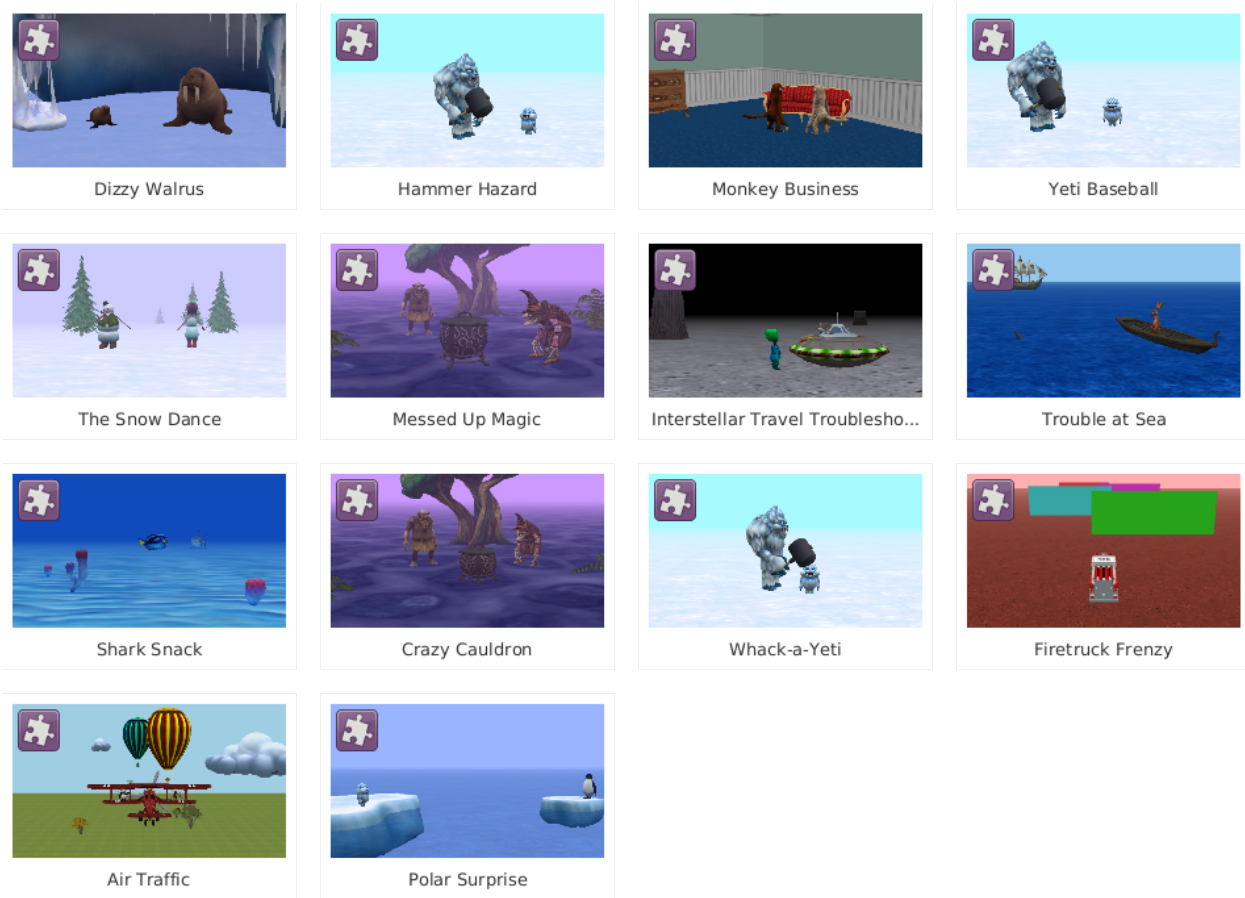


Figure 2.18: The 14 animations of our revised puzzle curriculum.

2.5 Discussion

Overall, the results of our formative evaluations appear to have successfully created code puzzles that also function as completion problems. To help us understand why these puzzles double as completion problems, in this section I explore what it means to be a completion problem in programming.

As a reminder, “a completion problem is a partial worked example where the learner has to complete some key solution steps” [182]. In the context of computer programming, the key steps are less clear due to the complexity of programming. In the original study which validated the completion problem effect, Van Merriënboer does not clearly indicate how their completion problems were partial programs [192]. Inspired by the earlier work, Chang et al. used fill-in-the-blanks for partial lines of code [21], while Garner used line ordering and location as the key steps in completing a program [52]. A similar approach to Garner was employed several years later: *Parson’s programming puzzles*, later re-branded as *Parsons problems* [148]. In Parsons problems, users are only given the lines to a program and they must place the lines in order to produce the entire program, not just place them in a partial program. In this case, the key steps are once again correctly ordering the lines of a program. Yet, in Parsons problems the authors appeared to have developed this approach independently without leveraging Van Merriënboer, Chang et al., or Garner’s earlier work. Regardless, the key steps for each of these systems all involve lines of code; completing the line or placing lines back into the correct location.

Additionally, since completion problems are also used for independent learning, the problems should appropriately take into consideration the learner’s cognitive load. If the completion problems impose extraneous cognitive load without fostering germane cognitive load, then their effectiveness as a tool for informal learning is probably diminished. Unfortunately,

Van Merriënboer, Chang et al., Garner, and Parsons and Haden tested and developed their systems to supplement traditional classroom instruction. Failing to carefully manage cognitive load in a classroom environment, while not ideal, can probably be overcome with the support of a teacher. In the case of independent learning, failure to manage a learner’s cognitive load appropriately likely reduces their opportunity to acquire new programming skills. To manage learners’ cognitive load using our code puzzle completion problems, we incorporated scaffolding throughout the design to support these independent learners. While this additional support is a deviation from the previous code puzzles, it is consistent with existing completion problem approaches and it is consistent with educational materials that support independent learning.

In the remainder of this dissertation, I share three evaluations using the code puzzle completion problems that we developed in these formative studies. In the next chapter, I investigate whether our code puzzles are effective in supporting middle school children to learn programming constructs on their own.

Chapter 3

Evaluating the Learning Effectiveness of Code Puzzle Completion Problems

Portions of this chapter were originally published as “Enabling Independent Learning of Programming Concepts through Programming Completion Puzzles” in the *Proceedings of the 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* [70].

From our formative evaluations in Chapter 2, we developed code puzzle completion problems and an introductory curriculum that we believed would support novices in learning new programming constructs independently. Prior to code puzzle completion problems, if goal-oriented novice programmers wanted to increase their programming knowledge they typically had one readily available option: tutorials. Because tutorials are the most commonly available method for learning programming independently, we decided to investigate how well tutorials and code puzzle completion problems facilitate independent learning. In this chapter, I evaluate Hypothesis I; whether code puzzle completion problems improve novices’ ability to transfer programming constructs compared to using tutorials.

Conveniently, many novice programming environments readily include support for independent learners through tutorials or puzzles. Tutorials are probably the most common method that programming environments use to support novices [7, 71, 72]. However, these tutorials are often focused on teaching the system’s mechanics [55, 77, 174] or improving successful completion by guiding users through a series of steps while minimizing errors [95, 151, 198]. While completing tutorials may help with user interface mechanics, it is not clear how well novice programmers learn programming constructs via tutorials. More recently, some programming environments have begun to use a more puzzle-like approach to support users [8, 30, 45, 107]. Often these environments present some type of challenge with code pieces and encourage users to solve the challenge using the code pieces. However, I am unaware of any work that empirically evaluates the effectiveness of code puzzles to support independent learning [21, 52, 148].

In this chapter, I present a study evaluating the learning efficiency and effectiveness of code puzzle completion problems to validate Hypothesis I. In our evaluation, we compared participants’ ability to transfer programming constructs from training with either tutorials or puzzles. From our evaluation we found that participants who learned using code puzzle completion problems spent 21% less time in the learning phase and performed 33% better on transfer tasks than participants who learned using tutorials.

3.1 Related Work

Currently, there are several approaches available to facilitate learning programming in informal settings. In this section, we discuss how tutors, tutorials, and puzzle-like approaches attempt to support learning programming.

3.1.1 Tutors

Programming tutors typically focus on providing students with programming practice alongside classroom instruction. During practice, many programming tutors focus on helping identify student programming errors [2, 156, 170, 177]. Further, cognitive tutors [37] and intelligent tutoring systems [156] try to provide sufficient practice by modeling students' knowledge and suggesting additional practice for un-mastered concepts. These types of tutors have been found to be effective with classroom instruction [86]. It is not clear whether or not systems designed to complement the classroom are effective when used in independent contexts. Outside of the classroom, online tutors try to provide a classroom-like experience [27, 97]. However, many students often start but fail to finish online instruction [132]. Further, the course-like linear structure of the tutor material is unlikely to support goal-oriented learners who likely seek information that directly supports their goals.

3.1.2 Tutorials

Goal-oriented users may find that following a specific tutorial may better align with their current information needs. However, prior tutorial research has mostly investigated improving tutorial presentation and mechanics, rather than supporting learning independently. Tutorial presentation can take the form of static text [145], pictures [58], or video [23, 151] or can even be interactive [7, 50, 71, 95, 198]. Generally, these systems have focused on improving tutorial completion by providing additional scaffolding or support by using videos [23], onscreen annotations [95, 198], synchronizing user progress with the tutorial [50, 95, 151], or viewing other users' tutorial experiences [102]. Some researchers have used gamification to help motivate users to complete tutorials [110, 111]. Researchers also found that reconstructing code via a tutorial enabled users to perform better on transfer tasks than directly inserting

the code [71]. Beyond this, it is unclear how well tutorials will perform when teaching beyond mechanics.

3.1.3 Puzzle-Like Systems

Puzzle-like systems frequently present some form of problem and require users to formulate a solution to solve that problem [44]. In the context of programming, many puzzle systems give users an objective that requires them to program an object’s path through obstacles on a grid [26, 28, 30, 107, 112]. Frequently, puzzle-like programming environments use tutorials as part of their initial user experience [26, 30, 107, 112]. Introductory tutorials typically take the form of tutorial levels that introduce the puzzle’s mechanics. To encourage learning, some systems use debugging [107] or competition [74, 116]. Outside of informal use, some puzzle-like systems are specifically designed to be use alongside traditional classroom instruction [52, 148]. However, we are not aware of any work that explores the learning effectiveness of programming puzzles for independent learners.

3.2 Summative Evaluation

Using the initial code puzzle completion problems developed in the formative evaluations described in Chapter 2, we conducted a between-subjects evaluation to assess the effectiveness of learning new programming constructs with puzzles compared to tutorials.

We primarily based our tutorial condition on Scratch’s step-by-step video and text tutorials as shown in Figure 3.1 [174]. We developed and tested our tutorial implementation at the tail end of the curriculum formative evaluation (II) presented in the previous chapter (Section 2.3). See Figure 3.2 for our Scratch-style tutorials implemented in Looking Glass. In the tutorials, every step instructs the user to insert a statement into their program. For each step there

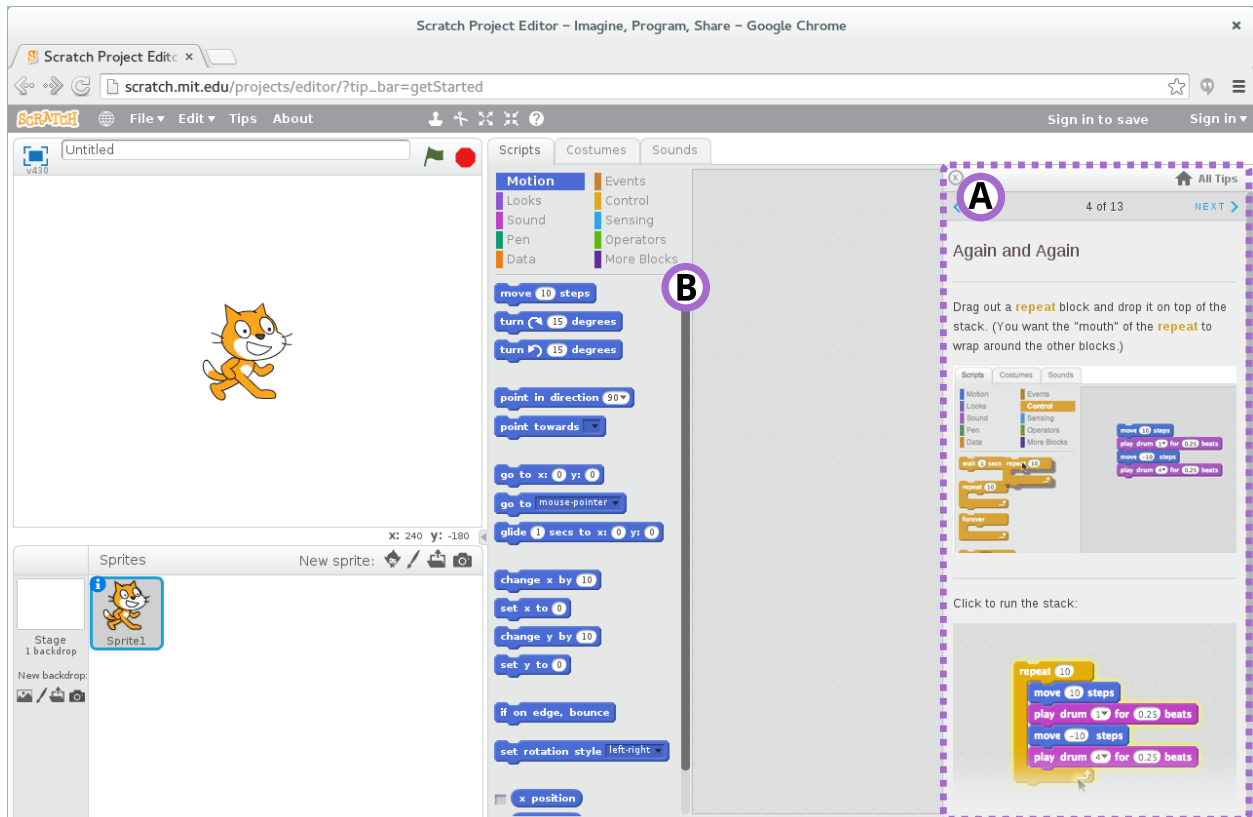


Figure 3.1: Scratch’s step-by-step tutorials [174]. The tutorial window (A) provides step-by-step instructions for authoring code in Scratch (B).

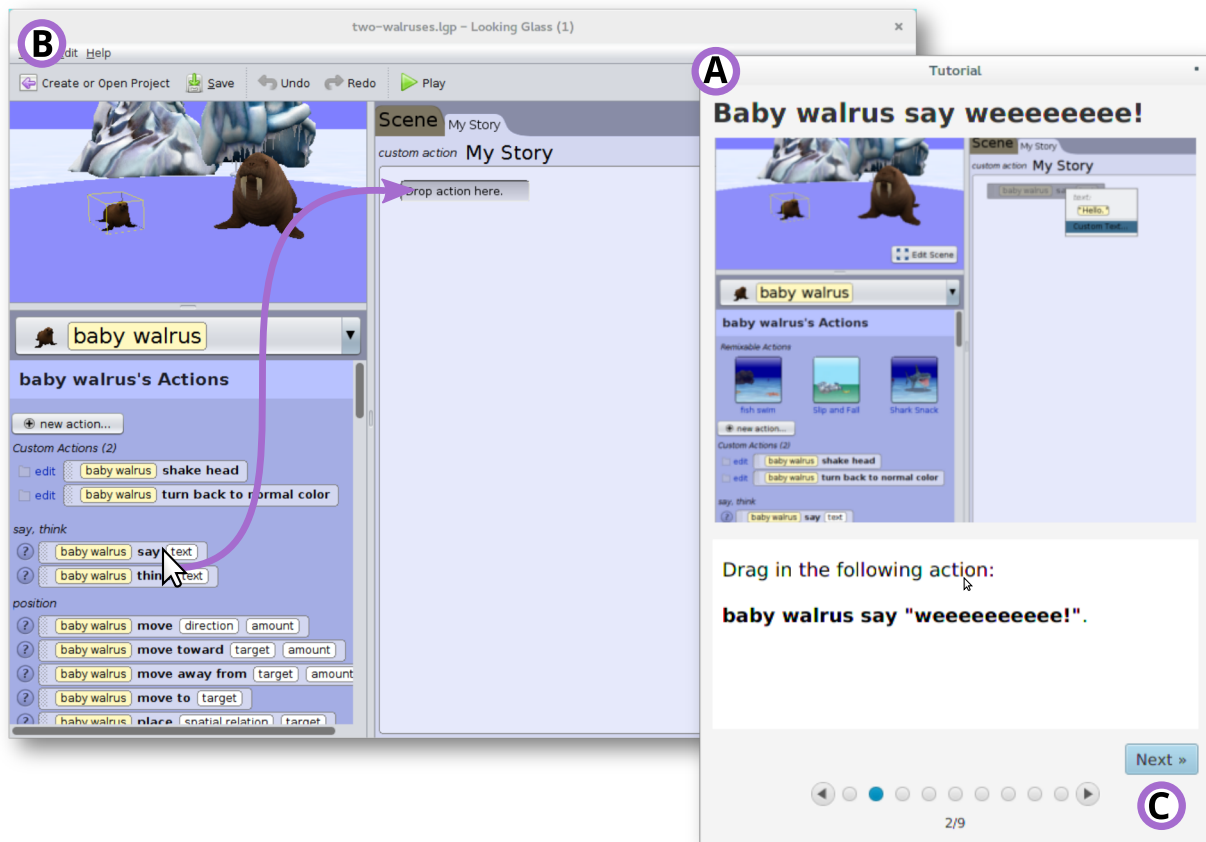


Figure 3.2: The tutorial condition. The tutorial window (A) shows a video and text description of each step. Users complete the step in the programming editor (B) and then manually advance the tutorial (C).

is a looping screencast video showing how to complete the step as well as text directions instructing the user what statement they need to insert.

From this evaluation, we intended to answer the following questions: 1) Do puzzles require a different time and mental investment than tutorials? 2) Are puzzles more motivating than tutorials? and 3) Do puzzle users show more evidence of learning than tutorial users?

3.2.1 Participants

We recruited 39 participants between the ages of 10 and 15 (14 female, 25 male; age: $M = 11.62$, $SD = 1.50$) for two separate experiments that lasted two hours. This chapter presents only the first experiment; the second experiment was for the development of a block-based programming assessment. We recruited participants through the Academy of Science of St. Louis mailing list. The Academy of Science is a not-for-profit organization dedicated to science outreach within the St. Louis metropolitan area. This mailing list is well known by community members and was further forwarded around local schools and other groups. We compensated participants with a \$10 gift certificate.

In the published version of this work, we did not analyze the data for participants with prior minimal programming experience (less than 3 hours total) [70]. Later, I realized that prior programming experience could be controlled through the use of covariates. I have redone the analysis of this study to include all participants regardless of prior programming experience.

3.2.2 Materials

We developed programs and survey materials that participants used during training and transfer study phases. We developed our materials in the Looking Glass programming environment. See Appendix E for all of the materials used in this study.

Table 3.1: The six training tasks and their programming constructs.

Training Task	Programming Constructs	No. Statements
1	Do in Order	7
2	Repeat	10
3	Do Together	9
4	Repeat & Do Together	9
5	Do Together { Repeat }	8
6	Repeat { Do Together }	10

Familiarization Tasks

For each study phase, we developed a phase familiarization task. The familiarization task introduced participants to the format of the tasks for the phase and also introduced participants to the interface mechanics of the programming environment. Each task’s format was identical to the training or transfer tasks. The familiarization training task is puzzle 0 from the initial curriculum shown in Table 2.1.

Training Tasks

We used the puzzles developed during our initial formative evaluation for the content of our six training tasks (See Table 3.1). For our puzzle condition, we used the puzzles in their original form. However, for the tutorial condition we authored tutorials that produced the same programs as the puzzles, but with directions using the code editor of Looking Glass.

Transfer Tasks

To measure participants ability to apply the training information in near transfer situations, we asked participants to complete several transfer tasks. For our transfer tasks we wanted

participants to demonstrate their ability to correctly identify and use a programming construct in a similar situation to the training tasks. We developed four transfer tasks, one for each of the programming constructs introduced in our initial curriculum (See Table 3.1). We asked participants to complete the transfer tasks in the code editor of Looking Glass.

The transfer tasks are complete programs that contain correctly ordered method invocations along with a description and video of the correct program output. Each program is missing the required control flow constructs necessary to create the desired output. To complete a transfer task a user must 1) determine the appropriate constructs, 2) locate the constructs within the programming environment, and 3) drag the construct blocks into the existing code and move the appropriate statements into the construct blocks. No additional method invocations are necessary to complete these tasks. This setup is intended to allow participants to spend their time demonstrating their mastery of the programming constructs rather than recreating animations or writing entire programs from scratch.

Surveys

We used two surveys for our evaluation: a self-developed task survey and the Intrinsic Motivation Inventory’s Task Evaluation Questionnaire (TEQ). The TEQ is a 22 Likert scale item questionnaire with four subscales: interest/enjoyment, perceived competence, perceived choice, and pressure/tension [25]. Participants rate each item on a scale from 1 - *not true at all*, to 7 - *very true*.

Our self-developed task survey was comprised of a multiple-choice question and five Likert scale items: a cognitive load scale and four perceived enjoyment scales. The difficulty survey is a well validated and reliable unidimensional scale from 1 - *extremely easy (very, very low mental effort)*, to 9 - *extremely difficult (very, very high mental effort)* for measuring cognitive

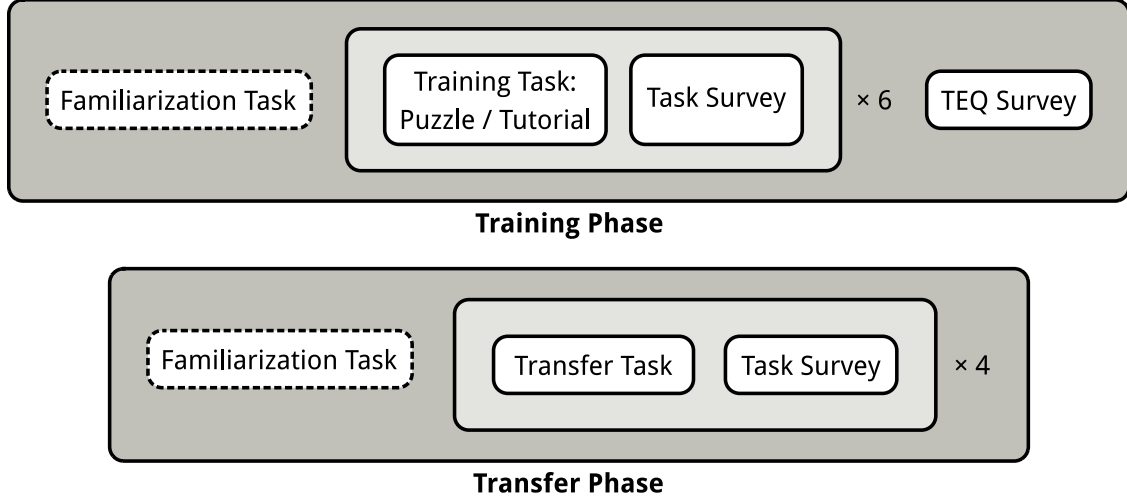


Figure 3.3: Overview of the basic summative evaluation procedure.

load [140, 143]. The perceived enjoyment scale contains four, seven item Likert scale items: *enjoyable – unenjoyable*, *exciting – dull*, *pleasant – unpleasant*, *interesting – boring* [82, 191]. Lastly, we included the *again again table*, which asks participants whether they would like to do this activity again with the following choices: yes, maybe, no [157].

3.2.3 Methods

We conducted our evaluation with four multi-user sessions. Each participant attended one of the two-hour sessions. Even though multiple participants attended each session, each participant worked independently. The study took place in a typical computer lab environment. We seated participants to minimize the potential for viewing other participants' screens.

We randomly assigned the participants to the puzzle or tutorial condition (puzzle: 21, tutorial: 18). Upon arriving to the study location, we asked participants to complete a computing history survey. We then interviewed each participant about their reported prior programming experience from the survey. Participants first completed the training phase

followed by the transfer phase. Following the transfer phase, the participants completed a separate experiment not presented in this dissertation. After completion of the separate experiment, participants were free to author their own programs. See Figure 3.3 for an overview of the basic study procedure for this experiment.

Training Phase

The training phase consisted of the following activities: the familiarization task, six training tasks with task surveys, followed by the TEQ survey. We gave participants a maximum of 7 minutes to finish the familiarization task and each training task. Following each task, we asked participants to complete a task survey. During our pilot studies, almost all participants completed each task within 7 minutes (this is roughly the mean plus a standard deviation). We allowed participants to ask any questions during the familiarization task. We also helped participants complete the familiarization task if they needed assistance with the directions or interface mechanics. For the actual training tasks, participants were not allowed to ask questions and worked independently.

Once participants completed the familiarization task, they were given the 6 training tasks in the curriculum order. Based on the participant's condition they completed all tasks using either the puzzles or tutorials. Following each training task, participants completed the task survey. After completing all training tasks, we asked participants to complete the TEQ.

Transfer Phase

The transfer phase consisted of the familiarization task and four transfer tasks with task surveys. We gave participants a maximum of 6 minutes to complete the familiarization and all transfer tasks. After each task, we asked participants to complete the task survey. We used a Latin squares design to administer the transfer tasks to control for learning effects. Like

the training phase, we assisted participants if necessary during the familiarization task; we provided no assistance for the 4 transfer tasks. After each transfer task, we asked participants to complete the task survey.

3.3 Analysis

For the training and transfer tasks, we analyzed task time and performance, as well as the task surveys.

3.3.1 Transfer Tasks

We analyzed the time each participant took to complete each transfer task and their performance for each transfer task. For the transfer tasks, we measured performance by whether the participant’s solution matches the correct solution exactly; we did not score with partial credit. In the published version of this study, we used a rubric to give partial credit for each of the transfer programs [70]. Upon reflection, I now believe this was a mistake because the rubrics were not validated. Without validated rubrics, I am not confident that the grading will accurately reflect each participants’ understanding of the programming concepts.

3.3.2 Task Surveys

We analyzed the results for the cognitive load/difficulty scale, the four perceived enjoyment scales, and the *again again table*. We analyzed each survey for reliability (Cronbach’s $\alpha > .70$). The cognitive load/difficulty scale was reliable ($\alpha_{training} = .87$; $\alpha_{transfer} = .82$). The four perceived enjoyment scales were reliable: enjoyable ($\alpha_{training} = .85$; $\alpha_{transfer} = .84$), exciting ($\alpha_{training} = .84$; $\alpha_{transfer} = .86$), pleasant ($\alpha_{training} = .82$; $\alpha_{transfer} = .88$), and interesting

($\alpha_{training} = .86$; $\alpha_{transfer} = .87$). Lastly the *again again* scale was also reliable ($\alpha_{training} = .93$; $\alpha_{transfer} = .90$).

3.3.3 Task Evaluation Questionnaire

The interest/enjoyment ($\alpha = .91$) and perceived competence ($\alpha = .75$) subscales of the TEQ are reliable (Cronbach's $\alpha > .70$). The subscales for perceived choice ($\alpha = .57$) and pressure/tension ($\alpha = .60$) are not reliable. We have only included the interest/enjoyment and perceived competence subscales in our analysis.

3.3.4 Controlling External Effects

Our programming experience interview revealed that 31% of participants had limited prior programming experience. Because prior programming experience may influence the outcome of our results, we controlled for prior programming experience, as well as age, and gender, using covariates in our analysis. For all statistical results, where appropriate, we used MANCOVA with Pillai's trace (V) to compare the dependent variables from all six training tasks or all four transfer tasks against the puzzle and tutorial conditions. Our analysis revealed that none of the covariates are significant for any of the results.

When reporting our results, we also include effect size alongside p values. P values are heavily influenced by sample size; the larger the sample size, the more likely a statistical test will return significance ($p < .05$) [179]. Effect size removes the influence of sample size by measuring the magnitude a variable differs between conditions [179]. When reporting effect size for MANCOVA we report eta squared (η^2). We report the magnitude (small, medium, and large) based on the following values: .01, .06, and .14 [99].

3.4 Results

In this section, we address the results of our evaluation in terms of answering our research questions.

3.4.1 Do puzzles require a different time and mental investment than tutorials?

We looked at time and mental investment by examining three factors from the training phase data: time and cognitive load. We wanted to know if there was any difference in completion time between the conditions. We also wanted to know if puzzles required more cognitive load than tutorials. We found that puzzle participants overall took 21% less time and required 66% more cognitive load compared to tutorial participants.

Puzzle and Tutorial Task Time

Overall puzzle participants spent less time completing the training tasks ($M = 3.66$, $SD = 1.19$ minutes) compared to tutorial participants ($M = 4.62$, $SD = 0.62$ minutes), as shown in Figure 3.4. There was a significant and large effect of condition (puzzle or tutorial) on the training task time; $V = .75$, $F(6, 29) = 14.12$, $p < .001$, $\eta^2 = .745$.

Puzzle and Tutorial Cognitive Load

Puzzle participants reported higher cognitive load ($M = 3.95$, $SD = 0.99$, \approx *slightly easy*) compared to tutorial participants ($M = 2.38$, $SD = 0.30$, \approx *very easy*). See Figure 3.5 for the average cognitive load scores for all the training tasks. The effect of condition (puzzle or tutorial) is significant and large; $V = .51$, $F(6, 26) = 4.45$, $p = .003$, $\eta^2 = .507$. Unfortunately,

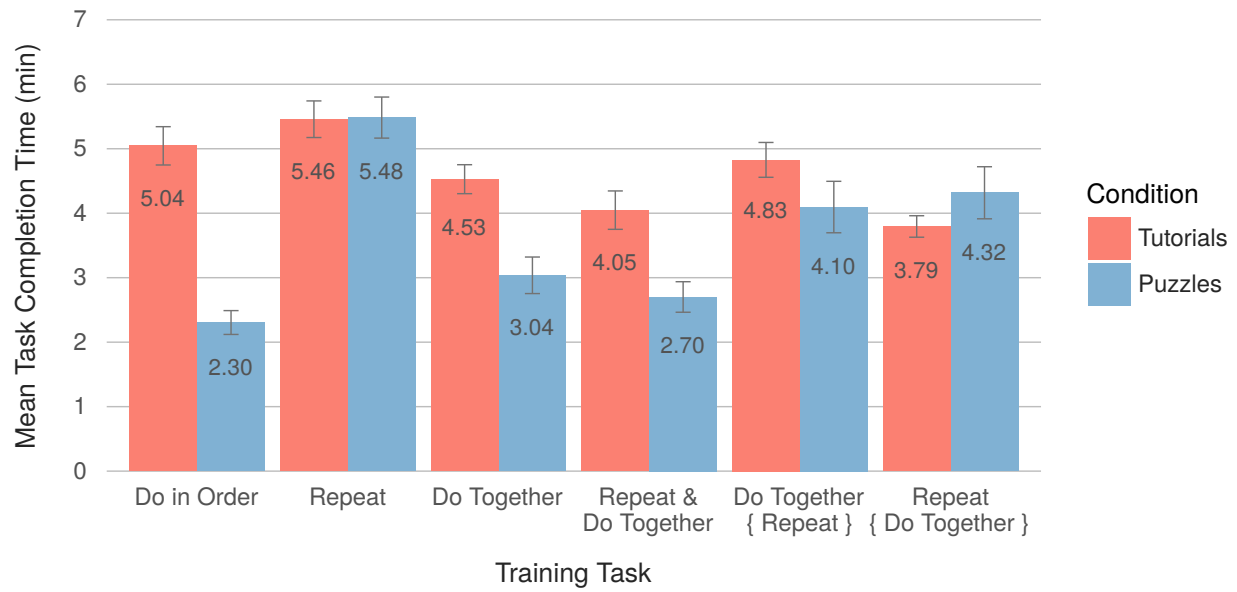


Figure 3.4: Mean training task completion time for puzzle and tutorial participants. ($V = .75$, $F(6, 29) = 14.12$, $p < .001$, $\eta^2 = .745$)

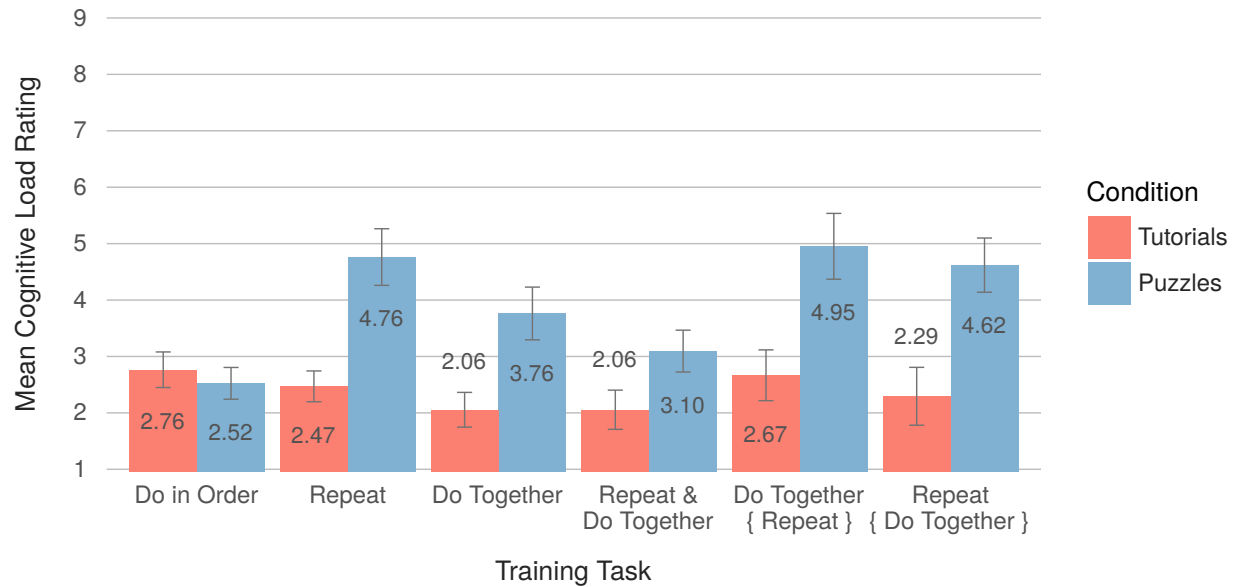


Figure 3.5: Mean training task cognitive load for puzzle and tutorial participants. 1 - extremely easy (very, very low mental effort); 9 - extremely difficult (very, very high mental effort). ($V = .51$, $F(6, 26) = 4.45$, $p = .003$, $\eta^2 = .507$)

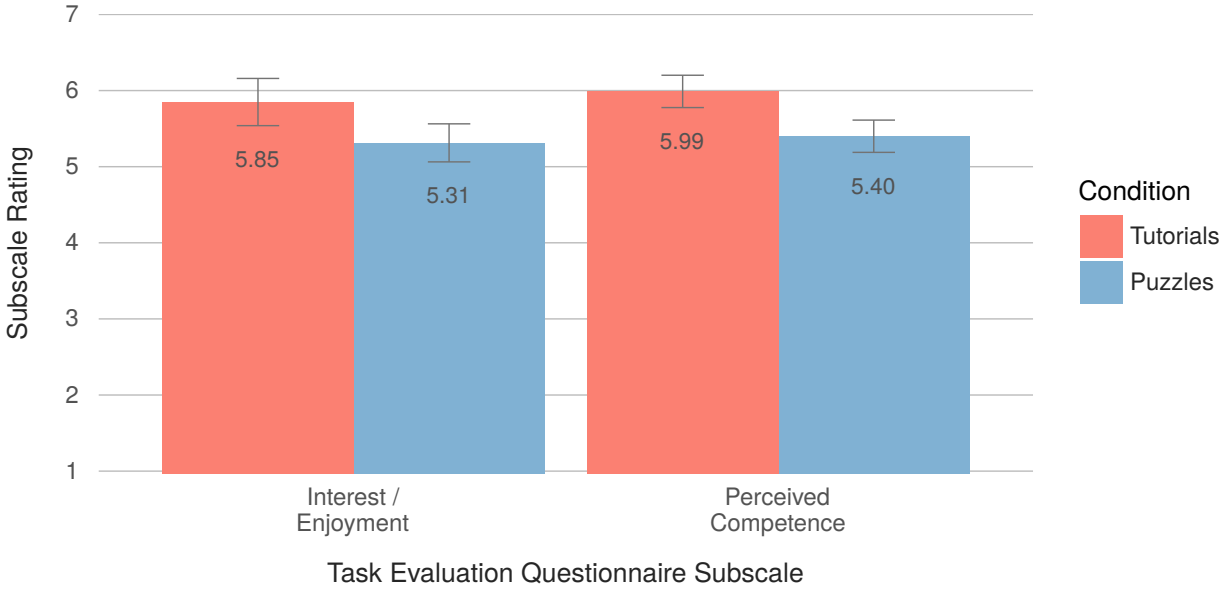


Figure 3.6: Mean subscales for the Task Evaluation Questionnaire (TEQ). 1 - not true; 7 - very true. ($V = .14$, $F(2, 33) = 2.58$, $p = .091$, $\eta^2 = .135$)

due to the study design we are unable to determine whether the increase in cognitive load is extraneous or germane. We discuss the potential types of cognitive load further in the discussion below (Section 3.5).

3.4.2 Are puzzles more motivating than tutorials?

We used the training tasks surveys and the TEQ to help us determine whether the puzzles are more motivating than tutorials. We found no significant effects for these measures between the puzzle and tutorial conditions suggesting no difference in motivation between the conditions.

The TEQ showed no significant changes between the conditions; $V = .14$, $F(2, 33) = 2.58$, $p = .091$, $\eta^2 = .135$. For the TEQ interest/enjoyment subscale (1 - *not true*, 7 - *very true*), the puzzle ($M = 5.31$, $SD = 1.15$) and tutorial ($M = 5.85$, $SD = 1.32$) participants rated that they enjoyed their experience. For the perceived competence subscale, the puzzle

($M = 5.99$, $SD = 0.90$) and tutorial ($M = 5.40$, $SD = 0.97$) participants also rated that they felt competent.

The training task surveys showed no significant differences between conditions. Participants in both conditions rated each task as enjoyable, exciting, pleasant, and interesting. They also rated that they would like to do the task again. For the perceived enjoyment survey the effect of condition is not significant for all scales; *enjoyable* – *unenjoyable*: $V = .20$, $F(6, 26) = 1.06$, $p = .410$, $\eta^2 = .197$, *exciting* – *dull*: $V = .30$, $F(6, 26) = 1.85$, $p = .128$, $\eta^2 = .299$, *pleasant* – *unpleasant*: $V = .18$, $F(6, 26) = 0.93$, $p = .489$, $\eta^2 = .177$, and *interesting* – *boring*: $V = .17$, $F(6, 26) = 0.87$, $p = .528$, $\eta^2 = .168$. The *again again table* also revealed similar non-significant results, $V = .13$, $F(6, 26) = 0.63$, $p = .706$, $\eta^2 = .127$. These results suggest that puzzles are no more or less motivating than tutorials. However, we note that these results do not align with our formative evaluation observations; we discuss this further in the discussion section and in Chapter 5.

3.4.3 Do puzzle users show more evidence of learning than tutorial users?

Lastly, we wanted to know whether users learn programming constructs more effectively using puzzles or tutorials. Overall, we found that puzzle participants performed 33% better on the transfer tasks while requiring equivalent cognitive load and time compared to tutorial participants. This suggests that code puzzle completion problems are an effective means for learning programming constructs independently. Next, we share the results of the transfer task data for performance, time, and cognitive load.

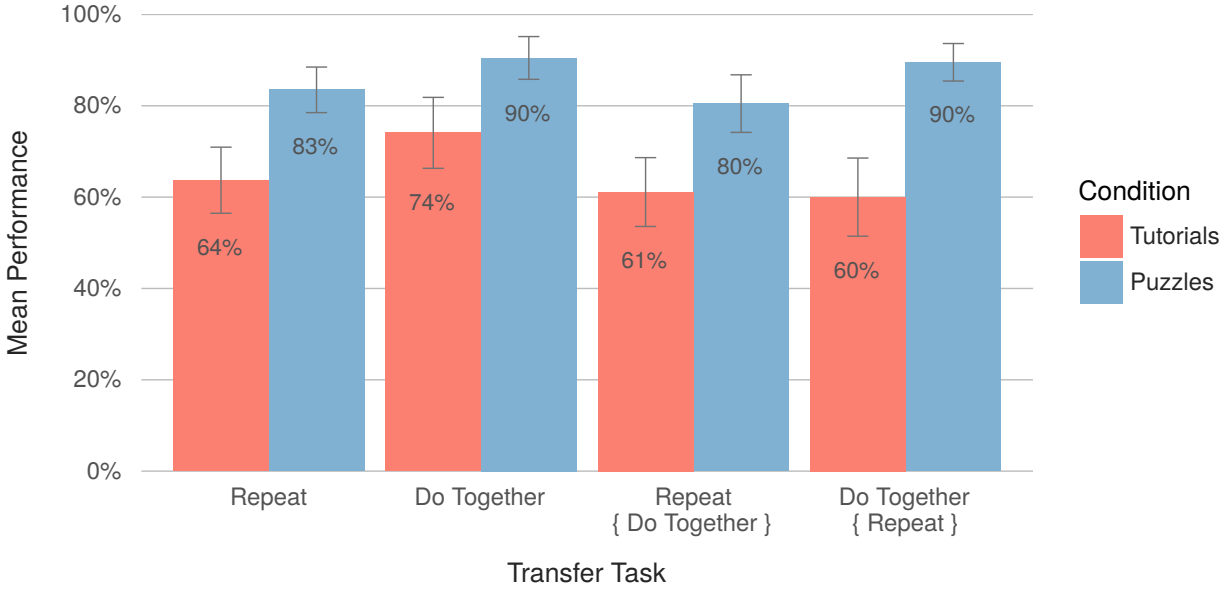


Figure 3.7: Mean transfer task performance across conditions. ($V = .27$, $F(4, 31) = 2.80$, $p = .043$, $\eta^2 = .266$)

Transfer Task Performance

Puzzle participants ($M = 86\%$, $SD = 5\%$) outperformed tutorial participants ($M = 65\%$, $SD = 6\%$) in correctly completing the transfer tasks. Figure 3.7 shows the average performance across all transfer tasks. There was a significant and large effect of condition on transfer task performance; $V = .27$, $F(4, 31) = 2.80$, $p = .043$, $\eta^2 = .266$.

Transfer Task Time

Puzzle participants spent 14% less time completing the transfer tasks ($M = 3.47$, $SD = 0.71$ minutes) compared to tutorial participants ($M = 4.02$, $SD = 0.92$ minutes), as shown in Figure 3.8. However, this difference is not significant; $V = .11$, $F(4, 31) = 0.95$, $p = .450$, $\eta^2 = .109$.

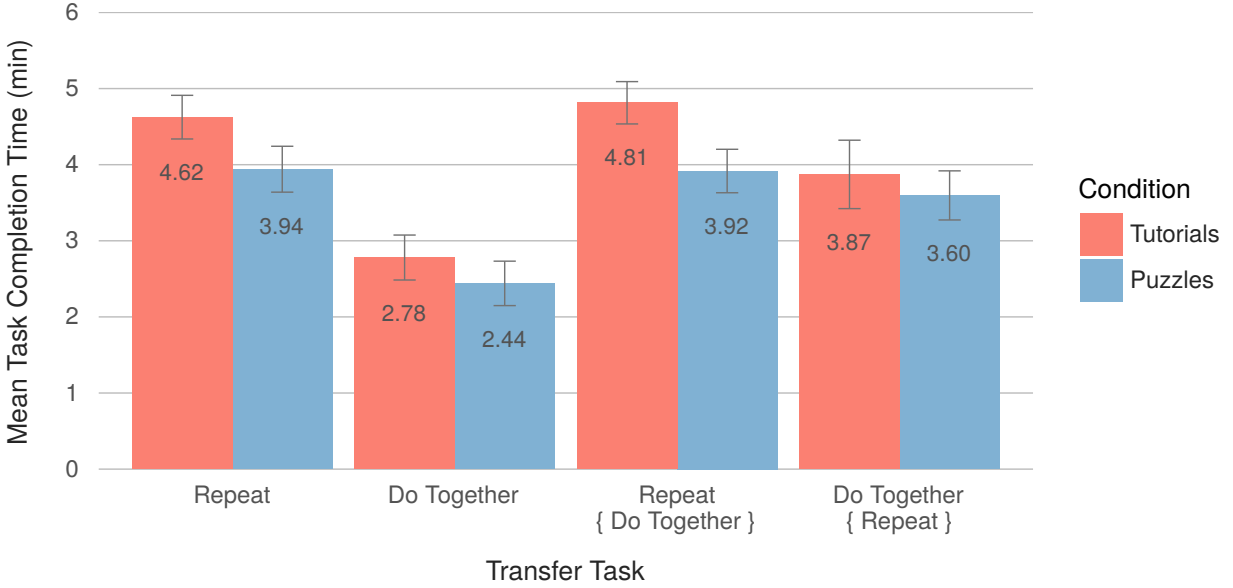


Figure 3.8: Mean transfer task completion time between conditions. ($V = .11$, $F(4, 31) = 0.95$, $p = .450$, $\eta^2 = .109$)

Transfer Task Cognitive Load

Puzzle participants ($M = 3.96$, $SD = 0.57$, \approx *slightly easy*) and tutorial participants ($M = 3.69$, $SD = 0.58$, \approx *slightly easy*) reported approximately equivalent cognitive load, as shown in Figure 3.9. The effect of condition on transfer task cognitive load is not significant; $V = .03$, $F(4, 31) = 0.20$, $p = .937$, $\eta^2 = .025$.

These results suggest that code puzzles improve programming task performance without significant differences in task completion time or cognitive load as compared to tutorials.

3.4.4 Instructional Efficiency

Commonly, approaches that leverage principles and techniques from cognitive load theory report a measure known as *instructional efficiency* [56, 139, 144]. By using the instructional efficiency measure, we can consider the potential cognitive costs for learning. Instructional

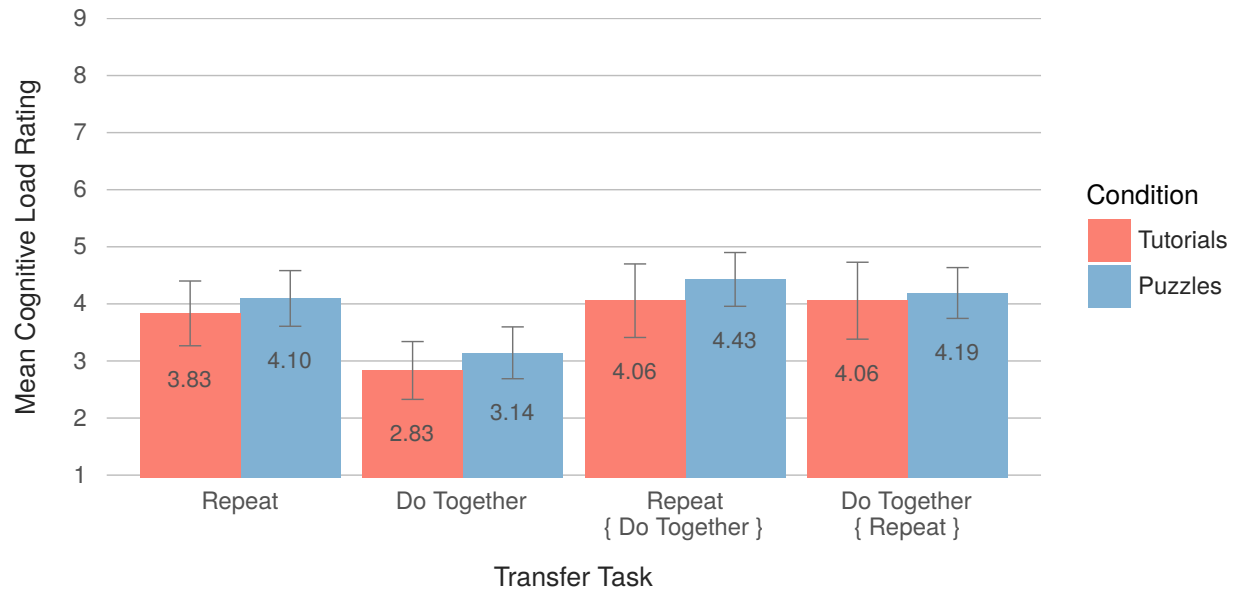


Figure 3.9: Mean reported cognitive load for transfer tasks. 1 - extremely easy (very, very low mental effort); 9 - extremely difficult (very, very high mental effort). ($V = .03$, $F(4, 31) = 0.20$, $p = .937$, $\eta^2 = .025$)

efficiency is calculated by combining cognitive load with task performance [139, 144]. For our evaluation, we computed the instructional efficiency between the tutorial and puzzle conditions by using the task performance data and cognitive load ratings from the transfer phase. Figure 3.10 shows the instructional efficiency for our evaluation. From Figure 3.10, we see that code puzzles are more efficient compared to tutorials.

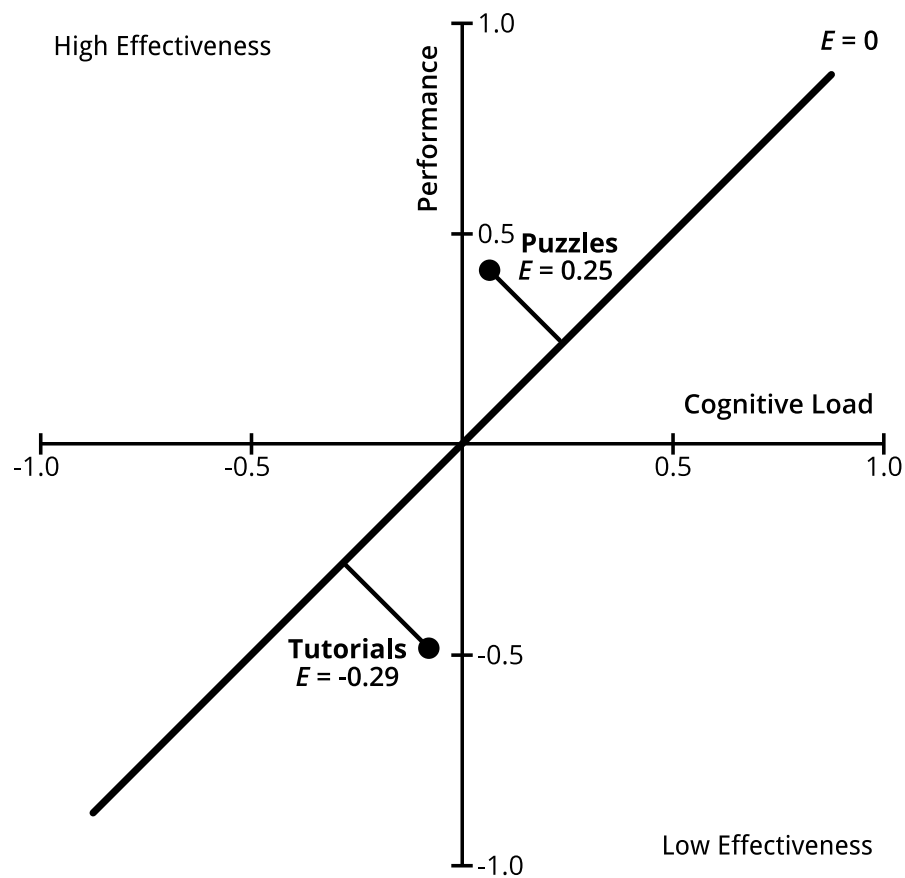


Figure 3.10: Instructional efficiency (E) between the tutorial and puzzle conditions. Points that lie about the $E = 0$ line reside in the high effectiveness area of the graph (top-left corner); points below the $E = 0$ line reside in the low effectiveness area (bottom-right corner).

3.5 Discussion

Overall, our results suggest that code puzzle completion problems are a promising way to support the learning of programming constructs independently.

Cognitive Load Participants who used the puzzles required less time to learn and performed 33% better on the transfer tasks. Puzzle participants also rated the cognitive load required to complete the training tasks 66% higher than tutorial participants. While this increase is significant, puzzle participants' overall cognitive load is still relatively low. Yet, the increase in performance on transfer tasks combined with the increase in cognitive load for puzzle participants, suggests that the increase in cognitive load is likely beneficial to learning. However, without redesigning and rerunning this study we cannot say for certain if the increase is only germane or also additional extraneous cognitive load.

Additionally, the code puzzles may also encourage participants to engage in self-explanation. Tutorial participants could follow the instructions without engaging in self-explanation about each programming construct. The puzzles required participants to think about the behavior of each statement within a program and how to combine them to achieve a given outcome, thus encouraging self-explanation. The puzzle format also intentionally removes unnecessary code authoring barriers to focus learners on practicing directly with the programming constructs. This is in direct contrast to the tutorials which attempt to teach both the interface mechanics and the programming constructs simultaneously by demonstration. While the puzzles omit the interface mechanics, we believe that this approach better prepares learners for using programming constructs in a novel setting, as seen in the transfer tasks.

Motivation Finally, we note that we were surprised by the similarity in attitudinal results between tutorial and puzzle participants. At the tail end of our curriculum formative

evaluation (Section 2.3) participants interacted with both the puzzles and tutorials. During this formative testing we observed a marked preference for the puzzles. As one user noted, “I like the puzzles more [than the tutorials]... so it was fun.” We investigated these motivational differences in a follow-up study presented in Chapter 5. In our follow-up, we conducted a within-subjects study comparing the attitudes towards the tutorials and puzzles. However, the lack of attitudinal difference between puzzles and tutorials may suggest that a motivating context, such as animated storytelling, plays a larger role in supporting motivation in early programming experiences compared to the presentation of learning materials.

3.6 Conclusion

In this study, we demonstrated that code puzzle completion problems can be an effective learning alternative to tutorials thus validating Hypothesis I. Not only do puzzle participants outperform tutorials participants on similar tasks, but they were able to learn the new content in less time. However, for these puzzles to be effective in independent contexts, motivation and enjoyment are extremely important. Due to the mixed motivational results in this study, I follow-up by further investigating users enjoyment of code puzzles in Chapter 5. In the next chapter, I study whether a common technique of puzzles, distractors, further improves the learning effectiveness of code puzzle completion problems.

Chapter 4

The Impact of Errors in Code Puzzle Completion Problems

This chapter was originally published as “Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers” in the *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER)* [73].

In the previous chapter (3) we discovered that code puzzle completion problems can be an effective approach for supporting the near transfer application of programming constructs. In the evaluation, the code puzzles provided learners with only the correct statements needed to solve the puzzle. However, code puzzles frequently include errors, like extra incorrect statements, as a puzzle mechanic [77, 107]. In this chapter, we investigate Hypothesis II; how errors impact the transfer of programming constructs from code puzzle completion problems.

Errors in code puzzles can be an incorrect program that users must debug in order to solve the puzzle [77, 107]. Or more commonly, completion style code puzzles contain extra statements that are not part of the solution, known as distractors [43, 48, 84, 148]. However, little is

known about how distractors and the errors they generate affect learning. Distractors in code puzzles are often asserted as beneficial to learning [48, 148]. Yet, errors in worked examples have been shown to harm novice learners [63]. Given that code puzzles frequently leverage many of the qualities of completion problems, which are partial worked examples, it is unclear how distractors in code puzzles affect learning.

Further, beyond being *extra statements*, little is known about producing distractors that may benefit learning as previous researchers have asserted [148]. Broadly, distractors in code puzzles are often described as unnecessary code [48], extra fragments [76], or erroneous code [83]. Fortunately, some researchers provide a more precise explanation: distractors should be used to “illustrate a particular point” or to “highlight programming principles the student may ignore” [148]. Distractors based on these principles often attempt to illustrate common programming misconceptions and syntax errors [90, 148]. While this seems like a reasonable approach for using distractors, we are unaware of any empirical evidence on the effectiveness of distractors in code puzzles.

In this chapter, we present a study investigating middle school children’s ability to learn programming constructs independently from code puzzle completion problems with distractors. We conducted a formative evaluation exploring the potential use of distractors to encourage and foster a beneficial learning experience. We then report the results of a summative evaluation comparing the effectiveness of using *partial suboptimal path* distractors when learning programming constructs on transfer task performance. Our results show that distractors increased cognitive load, reduced learners’ ability to successfully complete code puzzles, and significantly increased time on task. Distractor participants also showed no difference in transfer task performance compared to participants who trained without distractors.

4.1 Related Work

At its core, a distractor is an *error*. Some suggest that the errors generated from distractors in code puzzles may aid learning [90, 148]. Additionally, the reasons cited for using distractors in code puzzles are similar to the reasons for using distractors in multiple choice tests [98, 113, 148]. In this section, I review research on the impacts of 1) generating errors on learning and 2) the use and impact of distractors in testing.

4.1.1 Learning with Errors

Learners may encounter errors when learning from traditional educational materials or learning based games. We first discuss how errors affect learners in traditional education situations, followed by game based learning.

Humans naturally make errors as part of our learning process. Generating errors during learning can have both positive and negative consequences that educators must be careful to either leverage or mitigate. One approach to reducing errors made during learning is to teach common misconceptions to students when introducing new material [136]. This is especially important for low performing students, who benefited from learning about common misconceptions [136]. Errors also affect high and low performing learners differently. Only high performing students benefited when learning via worked examples that contained errors [63]. Fortunately, providing students with corrective feedback on the errors mitigated the harmful effects for low performing students [199]. Another common approach to learning with errors is using *trial and error*. Researchers demonstrated that generating errors when learning via trial and error helped younger learners, but impaired older adult learners [39]. Specifically in the context of programming, errors did not benefit learners. Students who

learned programming using a method that deliberately encouraged them to make errors had their rote learning efficiency reduced and they tended to make the same errors later [47]. The inconsistency among the literature suggests that educators must take into consideration learners' abilities and age, feedback and even the educational domain when considering the effects of errors during learning.

There is also a growing body of evidence on the effectiveness of game based learning for several disciplines [189]. Some of these games leverage correcting errors as a central game mechanic [77, 104], while others use distractors to lure players into generating errors [176]. Correcting errors as part of a debugging game has been shown to be an effective strategy for helping children learn programming [104]. While completion style code puzzles are not a game, they have been shown, without distractors, to be effective for learning [70]. We are unaware of any empirical evidence of the benefit or harm that generating errors from distractors in completion style code puzzles may cause.

4.1.2 Distractors and Errors in Testing

Multiple choice questions typically have several incorrect answers designed to distract lower level learners away from the correct answer [98]. Test takers also acquire new information from the act of taking a test [165]. Depending on the circumstances, the errors generated from distractors during testing may benefit or harm learners [16]. We discuss how testing affects learning and how the errors generated by distractors impact learners.

While testing may be viewed as assessing learned material, it also plays an important role in learning [163, 165]. When testing, individuals must retrieve information from their memory. Information is not stored statically in memory, but rather it is reconstructed upon retrieval; retrieval can change the information itself [91]. The act of retrieving information from memory

can be an effective way to promote learning and even retention of information [163]. This is known as the *testing effect* [165]. The key to the testing effect is the retrieval process itself. The act of trying to retrieve information during testing, even if unsuccessful, was found to enhance future learning [101, 161]. Further, if an individual fails to retrieve the information, guessing the answer did not hurt future performance [87]. The testing effect persists across testing formats, including multiple choice tests with distractors [115, 164].

However, if a test taker constructs an incorrect answer during retrieval, there is the possibility that the error is committed to memory which can impair future performance [11]. Test authors should be especially careful when authoring multiple choice tests with distractors because individuals may acquire false information from any errors [16, 121, 122, 123, 164]. Additionally, low performing students are more likely to acquire false information from multiple choice testing [16, 121], while higher performing students actually benefit from the errors [16]. Learning false information in testing appears to be due to faulty reasoning when selecting answers [122]. If faulty reasoning is at play, this can be mitigated by providing corrective feedback after testing [15, 17]. Feedback was found to improve learning when test takers generated errors during multiple choice testing [15, 17, 78, 152]. Given the potential for learning false information from multiple choice testing, test administrators should take measures, like feedback, to limit the potential for false learning.

Guidelines based on empirical evidence may also help test authors write better multiple choice tests with distractors [66, 67]. Even then, writing good distractors is often difficult for test authors [188]. Part of the difficulty in writing quality distractors is providing responses that separate low and high performers [162], but that also provide insights about students' misunderstandings [98, 113]. One approach to authoring multiple choice distractors is to utilize common misconceptions [66, 131]. In our context, misconceptions can be common

programming mistakes, for example, failing to properly nest statements in a control flow construct [13].

Overall, generating errors during learning may help learners under the right conditions. However, there is the potential that generating errors may also decrease the effectiveness of learning. Little empirical evidence exists to suggest how generating errors in completion style code puzzles will affect learners. Beneficial use of distractors in code puzzles may only be effective under certain circumstances, similar to the learning with errors research. In this chapter, we explore the potential for using distractors in code puzzle completion problems to help novices learn programming independently.

4.2 Formative Evaluation

We conducted a formative evaluation to explore how to author distractors in code puzzles to exploit the benefits of learning with errors while reducing the potential harm they may cause. See Figure 4.1 for examples of the distractors we developed in this evaluation. Even though the most cited use of a distractor in code puzzles is the programming syntax distractor [43, 48, 76, 84, 90, 148], we did not explore this distractor in our study. Blocks-based programming languages forgo syntax in an effort to reduce the barriers for learning programming. From our evaluation, we share three guidelines for distractors in code puzzles: 1) distractors that create *extra noise* are easy to ignore, 2) distractors should encourage learners to follow a *familiar, but suboptimal path*, and 3) allow only *one possible solution* to a puzzle.

For our formative study, we recruited 16 participants between the ages of 10 and 15 (10 female, 6 male; age: $M = 11.94$, $SD = 1.57$) from the Academy of Science of St. Louis mailing list. The Academy of Science is a not-for-profit organization dedicated to science outreach in the St. Louis metropolitan area. We compensated participants with a \$10 gift certificate.

4.2.1 Extra Noise is Easy to Ignore

Prior work described code puzzle distractors as extra code or unnecessary code [48, 76]. We initially came up with three unnecessary code distractors: 1) create additional *unrelated* random noise, 2) create additional *tangentially related* noise, and 3) insert *unrelated control flow constructs*.

For the *unrelated noise* distractors we added random method invocations to the puzzles, for example: `ufo.resize(2.0)`. Users very quickly realized that the random statements are easy to eliminate from their possible solution space, as one participant stated, “the extra actions (statements) contradict, so you can ignore them.” We also tested *tangentially related* distractors by inserting additional method invocations that could plausibly be part of an animation. For example, in an animation about a dolphin rescuing a sinking boat, we inserted a method call to a background ship object, `ship.sail_towards(boat)`. However, participants gave similar explanations that they knew they could just ignore it because the ship never sailed towards the boat.

Given that participants were quick to dismiss the *extra noise*, we tried another distractor designed to encourage participants to make decisions about which control flow blocks they might need for the solution. We inserted several types of *unrelated control flow blocks* as distractors. We knew that this approach better engaged some participants when they asked, “Do I have to use all of them?” Further, the extra control blocks appeared to encourage thinking for some participants, as one participant stated: “It’s sorta tricky, but it also makes you think harder, like what does the computer want me to do?” However, we note that adding unrelated control blocks were also easy to dismiss as unnecessary.

4.2.2 Use a Familiar, but Suboptimal Path

Instead of adding extra noise to the puzzles, we tried to engage participants by creating distractors that encouraged them to follow a *familiar, but suboptimal path*, when solving the puzzle. The idea is similar to the *misconception* distractors mentioned in Parsons problems [148]. The distractors allow a user to create a suboptimal solution for a puzzle that follows a familiar strategy that they have likely used before. For example, using identical duplicate statements instead of using a loop as shown in Figure 4.1-D. We used three strategies for suboptimal path distractors: 1) insert distractor statements into the solution for the puzzle’s *initial state* (Figure 4.1-A), 2) *initially nest* constructs incorrectly (Figure 4.1-B), and 3) add *alternative statements* that can lead to a suboptimal solution (e.g. duplicate statements instead of a loop) (Figure 4.1-D).

Compared to the previous *noisy* distractors, participants engaged with the suboptimal path distractors frequently. For the *initial state* distractor one participant stated, “This one you have to think that you can move the pieces on the board already off.” In *initial nesting* distractors, participants would recognize that they needed the combined effects of both constructs for the solution. Yet, participants had difficulty realizing that they needed to switch the nesting order, “I’m a little bit irritated. I’ve checked over and over the difference between my video and the correct one.”

When participants followed the suboptimal path, it often led to them failing to recover and failing to discover the optimal solution. This was especially true for the *alternative statement* distractors. Participants had difficulty understanding why their solution’s output looked correct even though the feedback indicated it was incorrect, as one participant stated, “It makes me feel a little frustrated. Well, I looked at the video many times. I figured out what I was missing. I corrected my animation. But looking at it... and it tells me that it’s wrong [and

that] makes me feel frustrated.” Even if many participants failed to recover, the suboptimal path distractors were a promising approach given that participants were frequently actively engaged with these distractors.

4.2.3 Allow Only One Solution

The *familiar, but suboptimal path* distractors succeeded in leading participants down a suboptimal path, but they frequently failed to bring participants towards the optimal solution. The biggest problem with the suboptimal path distractors is that they create several suboptimal solutions in addition to the optimal solution. For example, the *alternative statement* distractors for a loop produce a complete suboptimal solution with the duplicate statements instead of the optimal loop solution. To gently nudge participants towards the optimal solution, we modified the distractors by removing the possibility of suboptimal solutions while still keeping the ability to make some progress down a partial suboptimal path.

One frequent source of suboptimal solutions is empty control flow blocks inserted by participants into the solution. Extra control flow statements can lead to no operation behavior (NOP) [84]. For example, a *repeat* block with no nested statements does nothing when executed. The NOP makes the program output match the correct output, even though the solution is incorrect because of the unused block. This can be particularly confusing for novices when their output looks correct but the puzzle feedback indicates that their solution is incorrect. The *initial nesting* distractors have a high tendency to produce NOP behavior. For example, a participant may unnest an incorrectly nested control block and then forget to remove the block from the solution. Yet, novices see nothing wrong with this behavior, as one participant stated, “I get that it’s extra, but I don’t agree...”

To help participants realize the existence of the optimal solution, we eliminated the existence of suboptimal solutions. For the *alternative statement* distractors we provided enough statements to lead a user down the suboptimal path, but not enough statements to actually finish the suboptimal solution. For example, in Figure 4.1-D the last duplicate statement is missing from the puzzle, so a user can start the suboptimal path, but must seek out the optimal solution to solve the puzzle. We also removed the NOP suboptimal solutions typically produced by the *initial nesting* distractors by locking nested control flow blocks together as shown in Figure 4.1-B/C. Because the blocks are locked together, the user can explore the suboptimal path (Figure 4.1-B), but because they cannot unnest the blocks, they cannot produce a puzzle with NOP behavior. Instead a user must swap out the incorrectly nested blocks with the correctly nested one (Figure 4.1-C). We also carefully authored and tested our puzzles to ensure that only one solution existed for each puzzle.

Not allowing participants to complete the suboptimal path had the desired effect of encouraging participants to seek out the optimal solution: “Are there any other pieces? Looks like I’m missing one.” The *partial suboptimal path* distractors also appeared to lead to greater success. Several participants commented on how they enjoyed the distractors, as one participant stated, “I actually like that. I think it’s pretty cool that you throw in some random ones because that’s kinda how puzzles work.” For the remainder of our study, we used the *partial suboptimal path* distractors.

4.3 Summative Evaluation

Using what we learned in the formative evaluation, we conducted a between-subjects study to assess the impacts that distractors have on learning new programming knowledge. Our

study contained two phases: training and transfer. In the training phase participants learned new programming skills using code puzzles that we then evaluated in the transfer phase.

We developed a series of six puzzles using the *partial suboptimal path* distractors (see Table 4.1). Our puzzles introduced participants to three nested programming constructs: *repeat* nested in a *repeat*, *do together* nested in a *repeat*, and *repeat* nested in a *do together*. From this evaluation, we intended to answer the following questions: 1) What effect do distractors have on task completion time, task success, and cognitive load? and 2) Do distractor participants show more evidence of learning than those who learned programming constructs without them?

4.3.1 Participants

We recruited 102 participants between the ages of 10 and 15 from the Academy of Science of St. Louis mailing list. We screened participants for participation in any of our previous studies. We removed 9 participants from our data set for prior participation. We also removed one participant from our data set for not completing most of the study materials. In total we analyzed the data for 92 participants (32 female, 60 male; age: $M = 12.01$, $SD = 1.67$). We compensated participants with a \$10 gift certificate.

4.3.2 Materials

We developed, tested, and refined our study materials through a pilot study. For the pilot study we recruited 14 participants (9 female, 5 male; age: $M = 12.93$, $SD = 2.23$). See Appendix F for all of the study materials that we used in this evaluation.

Table 4.1: Training task constructs and distractors.

Task	Programming Constructs	Partial Suboptimal Path Distractors
1	Do Together	Alternative Statements
2	Repeat	Alternative Statements \times 2
3	Repeat { Repeat }	Initial State; Alternative Statements
4	Repeat { Repeat }	Initial State; Alternative Statements
5	Repeat { Do Together }	Initial State; Initial Nesting
6	Do Together { Repeat }	Initial Nesting; Alternative Statements

Familiarization Tasks

For each phase of the study, we developed familiarization tasks to help introduce participants to the format of the tasks and the mechanics of the programming environment.

Training Tasks

We developed six training tasks, which were all puzzles. The six tasks were designed to be completed sequentially. See Table 4.1 for the programming constructs and distractor types for each training task.

For the training tasks, we used five puzzles from the revised curriculum shown in Table 2.2 and Figure 2.18. During pilot testing we felt that introducing the nested construct, *repeat { repeat }*, warranted an additional training task. However, the revised curriculum only contained one puzzle (no. 8) that emphasized *repeat { repeat }*, so we authored a new puzzle specific to this evaluation, (training task 3).

We developed two sets of training tasks: one set without distractors, the other with distractors. For the distractors condition, we used the distractor variants of the puzzles we developed in

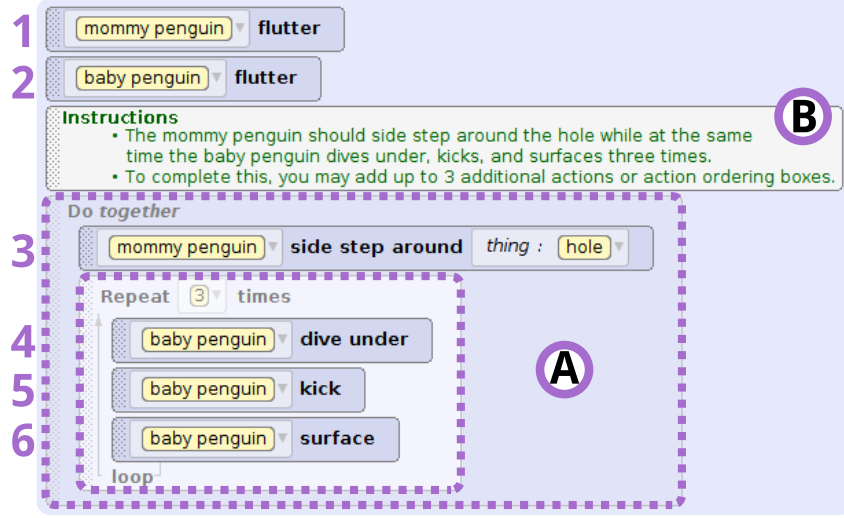


Figure 4.2: Example transfer task. Statements 1–6 are already inserted and correctly ordered. Participants need to insert additional control blocks (A) to make the animation match the instructions (B).

our formative evaluation. The training familiarization task is identical in format to the actual training tasks. However, the training familiarization task did not contain any distractors.

Transfer Tasks

In the transfer phase, we wanted to acquire evidence that the participants learned the nested programming constructs from the training phase. Specifically, we wanted to know if participants could identify the correct programming constructs and the proper nested structure when given a novel problem. We developed three transfer tasks to evaluate participants' mastery of nesting constructs from the training phase: *repeat { repeat }*, *repeat { do together }*, and *do together { repeat }*. The *repeat { do together }* and *do together { repeat }* transfer tasks are identical to the transfer tasks we used in the previous summative evaluation in Chapter 3.

Each transfer task is a complete program with existing and correctly ordered statements as shown in Figure 4.2. The existing statements are only method invocations; there are no

control flow constructs. To complete each task, participants need only identify and insert the correct control flow constructs from the programming environment as shown in Figure 4.2-A. Once a construct is inserted into the program, participants drag the appropriate existing statements, maintaining their order, into the newly inserted control block. Each transfer task also included a video of the correct animation, written instructions for completing the task, and inline comments describing the correct program output (Figure 4.2-B). The instructions noted that the existing statements are in the correct order and that participants are limited to only inserting three additional blocks into the program. We designed the transfer tasks to focus participants' time on demonstrating their programming knowledge, not authoring programs from scratch.

The transfer phase had two separate familiarization tasks. The first familiarization task introduced participants to the mechanics of authoring a sequential program in the programming environment. The second task is identical in format to the actual transfer tasks.

Surveys

We included three surveys in our evaluation: a self-developed programming experience survey, cognitive load task survey, and the Computer Science Cognitive Load Component Survey (CS CLCS) [133]. The programming experience survey gathered information about participants' age, gender, schooling, prior study participation, and prior programming education and experience.

The cognitive load task survey included two validated and reliable scales for measuring cognitive load [140]: mental effort [138] and difficulty [89]. Both scales are Likert item surveys with nine items from 1 - *very, very low mental effort / extremely easy* to 9 - *very, very high mental effort / extremely difficult*. Historically when measuring cognitive load, researchers

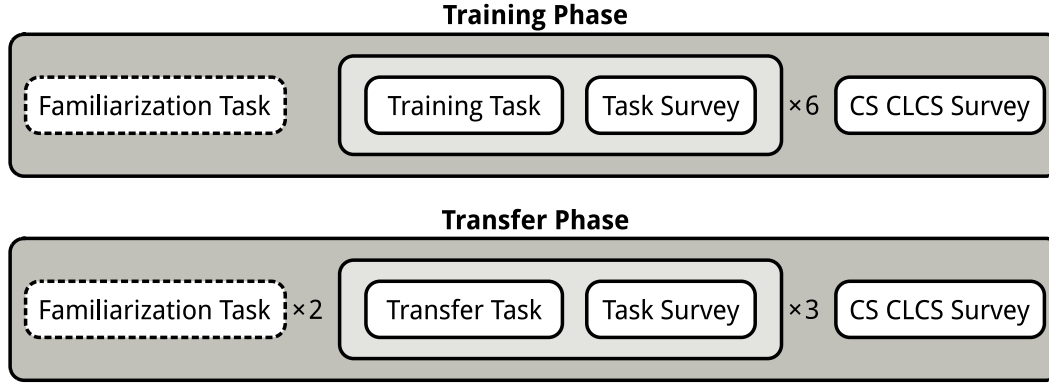


Figure 4.3: Summative evaluation procedure.

used either mental effort [138] or difficulty [89] scales to measure cognitive load. However, research now suggests that while mental effort and difficulty are correlated, they are not the same thing [56]. Because we used the difficulty scale in the prior study presented in Chapter 3, we included both scales in this study for comparison. Please note that when we discuss cognitive load in this study, we only refer to the mental effort scale.

We used a validated survey to measure the different types of cognitive load. The CS CLCS is an adaptation of an existing validated survey, the Cognitive Load Component Survey [108, 109], for the domain of computer science. The CS CLCS is a ten item Likert survey with three separate cognitive load scales: intrinsic, extraneous, and germane [133]. Each question is rated on an eleven-point scale from 0 - *not at all the case* to 10 - *completely the case*.

4.3.3 Methods

We conducted our evaluation over several different multi-user sessions. Each participant attended a single two hour session. We seated participants to minimize viewing other participants' screens and to minimize interaction between participants.

We conducted a between-subjects study with two conditions: control (no distractors) and distractors. We randomly assigned participants to either the control or the distractors condition (control: 47, distractors: 45). The study contained two parts: the training phase and the transfer phase as shown in Figure 4.3. In the training phase, participants completed puzzles with or without distractors as assigned in their condition. In the transfer phase, participants completed three transfer tasks. After completion of the transfer phase, we allowed participants to create their own animation or work on additional puzzles from the revised curriculum.

Before the study, we asked participants to complete the programming experience survey. Once completed, a member of the research team followed up on the survey responses by interviewing each participant about their responses. This follow up interview allowed us to correct any misreported data about participants' prior programming experience.

Training Phase

In the training phase, we asked participants to complete a familiarization task, six training tasks, and the CS CLCS. The familiarization task used the same format as the training tasks.

At the start of the training phase, we gave participants an instruction sheet with directions on how to complete a training task. We gave participants 12 minutes to complete each training task; there was no time limit for the familiarization task. We permitted participants to ask for help during the familiarization task. However, after completion of the familiarization task, we required participants to complete the remaining six training tasks without assistance. Each participant completed the training tasks in the same order. After completing each task, we asked participants to complete the cognitive load task survey where they rated their mental effort and difficulty for that task.

Upon completion of all training tasks, we gave participants a reminder sheet with a picture and the title for each of the training task animations (i.e. not the familiarization task). We then asked participants to complete the same cognitive load task survey ranking their mental effort and difficulty across all six tasks. We then asked them to complete the CS CLCS. We encouraged participants to reference the reminder sheet when completing these surveys to help them recall their experience.

Transfer Phase

After completing the training phase, participants began the transfer phase of the study. Similar to the training phase, participants first completed two familiarization tasks, three transfer tasks, and then the CS CLCS.

With each familiarization task, we provided an instruction sheet. The first familiarization task's instruction sheet demonstrated the basics of the Looking Glass programming environment. In the second familiarization task, the instruction sheet provided directions on how to complete the transfer tasks in the study. We allowed participants to keep both instruction sheets for the remainder of the transfer phase as a reference. During the familiarization tasks, participants could ask for help. However, we provided no assistance during the actual transfer tasks. We assigned the three transfer tasks using a balanced 'Williams' Latin squares design to control for learning effects [201]. We gave participants 6 minutes to complete each transfer task. Following the second familiarization task and each transfer task, we asked participants to complete the cognitive load task survey. Finally, upon completion of all transfer tasks, we gave participants a reminder sheet with each of the three transfer tasks' titles and pictures of each animation. We then asked them to complete a cognitive load task survey ranking their overall mental effort and difficulty for all transfer tasks and the CS CLCS using the reminder sheet.

4.4 Analysis

We analyzed the data gathered in the training and transfer phases in the study. We also discuss how we control for external factors, like prior programming experience, in our analysis.

4.4.1 Training & Transfer Tasks

For the training and transfer tasks, we analyzed task time and performance, as well as the cognitive load scales. We collected cognitive load using several scales: mental effort across all tasks, difficulty across all tasks, and overall intrinsic, extraneous, and germane cognitive loads. For each scale, we computed Cronbach's alpha for reliability (Cronbach's $\alpha > .70$).

Recall that we asked participants to rate their cognitive load for each task as well as rate their overall cognitive load after completing all tasks. Prior research has found that ratings completed at the end of a phase, as opposed to after each task, are often slightly higher than the mean of the ratings from all tasks [195]. The cognitive load task survey required minimal time (pilot testing revealed that this is typically less than 30 seconds) for participants to complete after each task. However, due to timing constraints and concerns over participant fatigue, we decided to only ask participants to complete the CS CLCS once, at the end of each phase. Because the CS CLCS was completed at the end of the phase the results may be slightly higher compared to the per task cognitive load ratings. To account for this potential difference in the CS CLCS results, we use both the immediate cognitive load task survey ratings and the overall ratings in our analysis.

Training Tasks

For the training tasks, we analyzed time on task, task completion, the cognitive load surveys, and distractor usage. Because task completion may have been affected by other factors, we analyzed whether participants ran out of time or gave up on the task. We also analyzed distractor usage within the distractors condition to evaluate the effectiveness of the distractors. We analyzed the percent of participants who used the distractors in each task and the percent of time participants used distractors in each task.

Transfer Tasks

Similar to the training tasks, we analyzed the time each participant took to complete each transfer task and their performance for each transfer task. For the transfer tasks, we measured performance by whether the participant's solution matches the correct solution exactly; we did not score with partial credit.

Cognitive Load

The mental effort scales ($\alpha_{training} = .90$; $\alpha_{transfer} = .89$) and difficulty scales ($\alpha_{training} = .89$; $\alpha_{transfer} = .79$) are both reliable for both phases. The intrinsic ($\alpha_{training} = .89$; $\alpha_{transfer} = .92$), extraneous ($\alpha_{training} = .78$; $\alpha_{transfer} = .71$), and germane ($\alpha_{training} = .87$; $\alpha_{transfer} = .92$) cognitive load scales from the CS CLCS were also reliable. There is a strong correlation between the mean mental effort for all tasks and the overall mental effort $r_{training} = .82$, $p_{training} < .001$; $r_{transfer} = .80$, $p_{transfer} < .001$. Likewise, there is a strong correlation between the mean difficulty for all tasks and the overall difficulty $r_{training} = .85$, $p_{training} < .001$; $r_{transfer} = .80$, $p_{transfer} < .001$. Because these correlations are extremely strong, we only report the results of mental effort using ratings from each task (i.e. we ignore the overall

ratings). Due to this strong correlation, we have not adjusted the results of the CS CLCS. We also see a strong correlation between mean mental effort and difficulty $r_{training} = .92$, $p_{training} < .001$; $r_{transfer} = .88$, $p_{transfer} < .001$. This suggests that mental effort and difficulty are very closely related. Because of this strong correlation, we only report the mental effort scale when discussing cognitive load.

4.4.2 Controlling External Effects

Our programming experience survey with follow-on interview revealed that 71% of participants had limited prior programming experience, notably many participants had used Scratch [174] or participated in an Hour of Code activity [77]. Because prior programming experience may influence the outcome of our results, we controlled for prior programming experience, as well as age, formal programming education, and gender, using covariates in our analysis. For all statistical results, where appropriate, we used ANCOVA or MANCOVA with Pillai's trace (V) to compare the dependent variables from all six training tasks or all three transfer tasks against the control and distractor conditions.

Our analysis revealed that the age and prior experience covariates are significant for most results. In general, older participants and participants with prior experience performed better, while needing less time. The formal education and gender covariates are not significant for any results. This suggests that at the middle school level formal education may not differ much from informal learning. Researchers have also advocated for gender inclusiveness in software, including programming environments, in part because problem solving strategies tend to cluster by gender [14]. Reassuringly, we found no evidence of any gender differences.

When reporting our results, we also include effect size alongside p values. P values are heavily influenced by sample size; the larger the sample size, the more likely a statistical test

will return significance ($p < .05$) [179]. Effect size removes the influence of sample size by measuring the magnitude a variable differs between conditions [179]. When reporting effect size for ANCOVA results we report omega squared (ω^2) and eta squared (η^2) for MANCOVA. We report the magnitude (small, medium, and large) based on the following values: .01, .06, and .14 [99].

4.5 Results

In this section, I share the results of our summative evaluation in terms of how they address our research questions. However, before I report on our research questions we first verify that the distractors in our study performed as expected.

4.5.1 Distractor Usage

We expected participants to interact with the distractors or use the distractors when solving the training tasks. We used two measures to identify whether participants made use of the distractors: whether participants interacted with the distractors and the percent of time spent using those distractors in each training task.

All distractor participants used distractors in at least two of the training tasks ($Mdn = 5$). In fact, the majority of distractor participants (84%) used distractors in four or more of the six training tasks. When authoring multiple choice tests, researchers recommend removing infrequently chosen distractors; distractors should have a response frequency greater than 5% [65, 67]. Across each training task, a majority of participants used distractors in each task ($M = 76\%$, $SD = 19\%$). See Figure 4.4 for the percentage of participants who used distractors for each task. We also note that participants spent on average 36% of their time

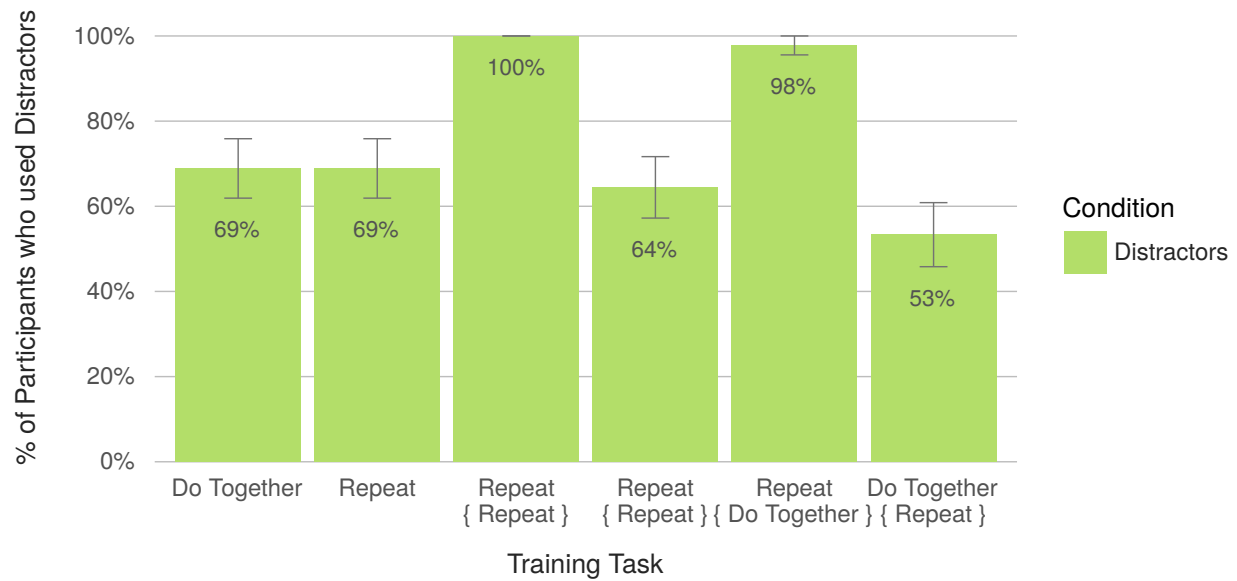


Figure 4.4: Mean percentage of distractor participants who used the distractor statements in each training task.

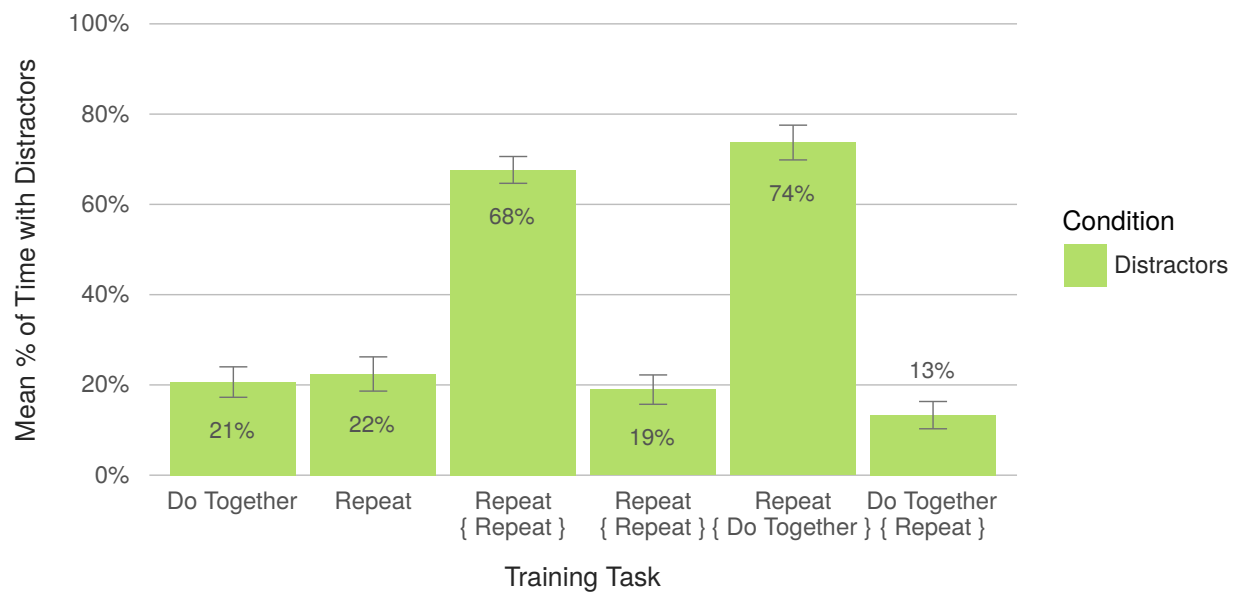


Figure 4.5: Mean percentage of time spent with distractors in training tasks across all distractor participants.

interacting with the distractors ($SD = 27\%$). Figure 4.5 also shows the average percent of time that participants used the distractors for each training task.

In general, we believe that this evidence suggests that participants used the distractors an appropriate amount. All participants in the distractors condition used distractors and spent an average of 36% of their time during the training phase using the distractors. While the specific numbers vary across tasks and participants, we believe that spending roughly 30% of task time generating errors, is a reasonable amount of time. We are confident that the distractors worked as intended and so we spend the remainder of the results section investigating our research questions.

4.5.2 How do distractors affect task completion time, task success, and cognitive load?

In this section, we report how the distractors affected the training task experience for participants. Overall, we found that distractors caused participants to spend 14% more time on the training tasks, while completing 26% less tasks and increasing participants' cognitive load by 11%.

Training Task Time

Distractor participants spent more time completing the training tasks ($M = 6.15$, $SD = 1.00$ minutes) compared to the control condition ($M = 5.42$, $SD = 1.11$ minutes). There was a significant and large effect of distractors on training task completion time $V = .17$, $F(6, 81) = 2.86$, $p = .014$, $\eta^2 = .175$. See Figure 4.6 for the average training task time for each condition.

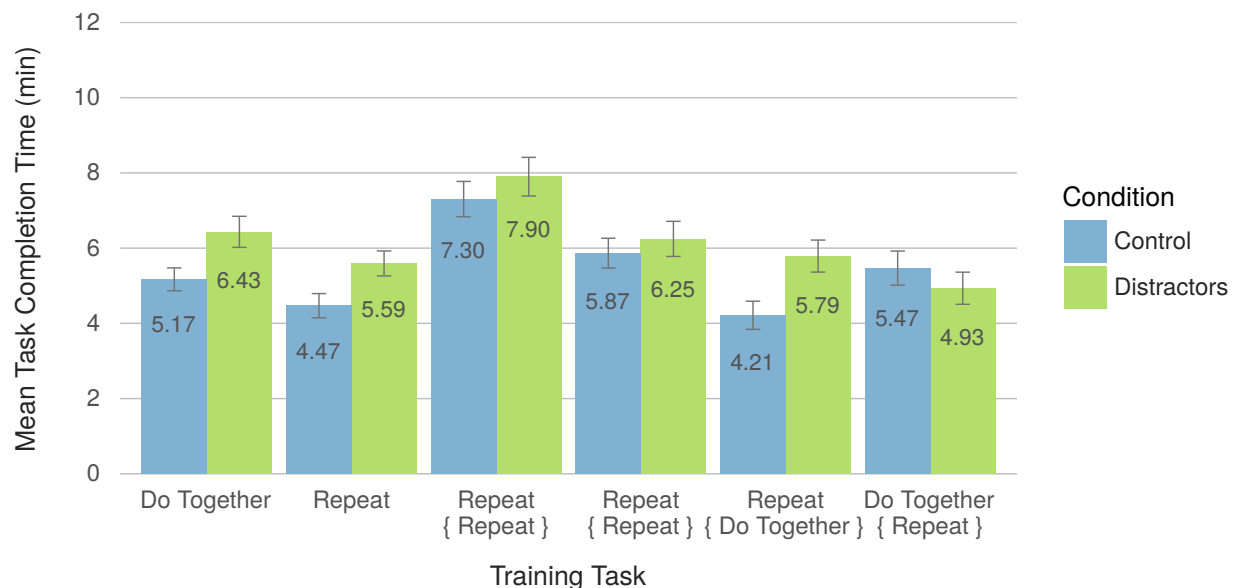


Figure 4.6: The mean training task completion time for both the control and distractors conditions. ($V = .17$, $F(6, 81) = 2.86$, $p = .014$, $\eta^2 = .175$)

Training Task Performance

Distractor participants ($M = 65\%$, $SD = 23\%$) correctly completed fewer training tasks than the control participants ($M = 88\%$, $SD = 11\%$). This effect was significant and large, $V = .30$, $F(6, 81) = 5.65$, $p < .001$, $\eta^2 = .295$. Roughly, this translates to control participants correctly completing about five tasks ($Mdn = 6$) while the distractor participants only completed close to four tasks ($Mdn = 4$). For an overview of the number of participants that completed each training task, see Figure 4.7.

Recall that participants were given twelve minutes to complete each training task along with the option to quit working on a task at any point. Quitting a task early is analogous to giving up on the task. Distractor participants were significantly more likely to give up on a task compared to control participants, $V = .20$, $F(6, 81) = 3.33$, $p = .006$, $\eta^2 = .198$. In fact, distractor participants quit an average of 1.8 tasks ($SD = 2.16$, $Mdn = 1$) compared to an

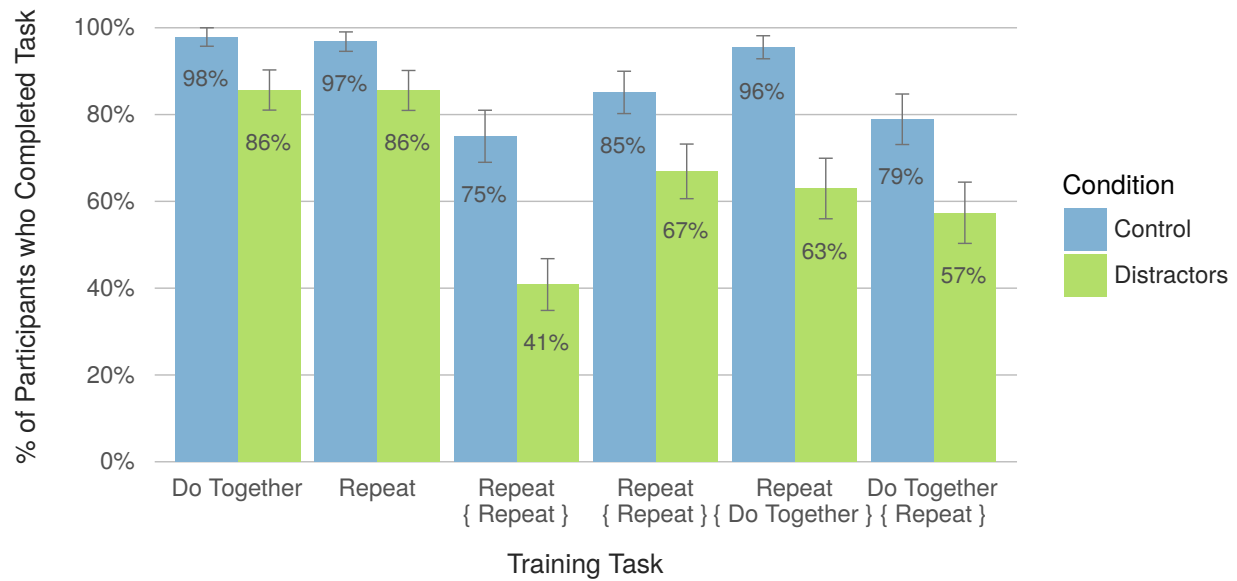


Figure 4.7: The percentage of participants in each condition that completed each training task. ($V = .30$, $F(6, 81) = 5.65$, $p < .001$, $\eta^2 = .295$)

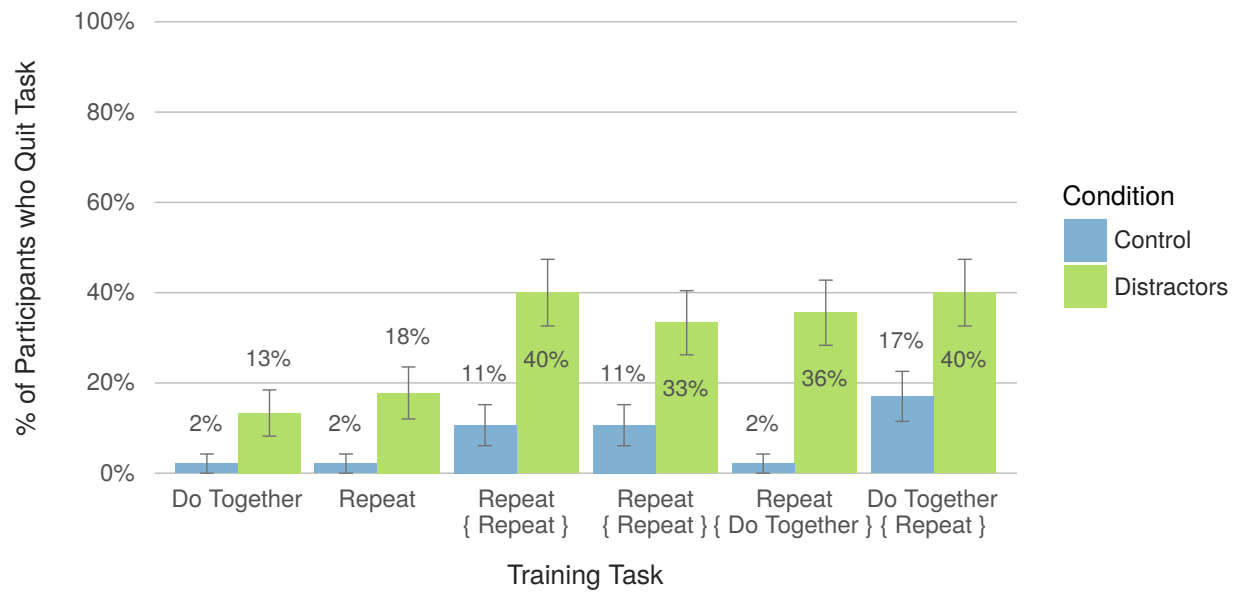


Figure 4.8: The percent of participants in each condition that gave up on completing the training tasks. ($V = .20$, $F(6, 81) = 3.33$, $p = .006$, $\eta^2 = .198$)

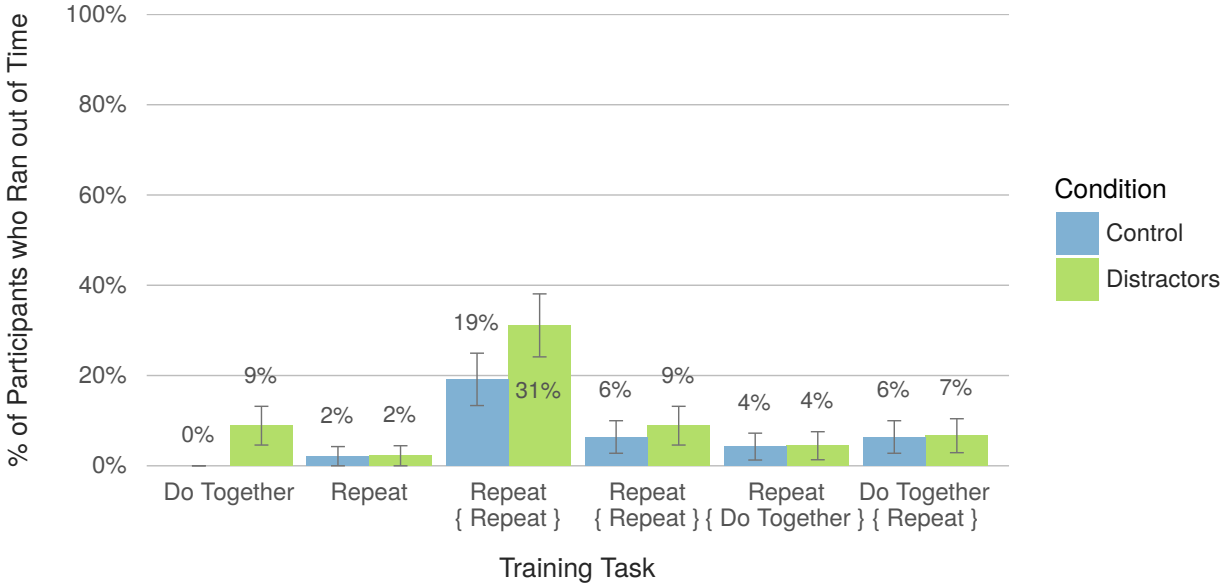


Figure 4.9: The percent of participants who did not have enough time to complete the training tasks. ($V = .06$, $F(6, 81) = 0.92$, $p = .486$, $\eta^2 = .064$)

average of 0.45 tasks ($SD = 1.00$, $Mdn = 0$) for control participants. See Figure 4.8 for the number of participants who quit each training task.

There is not a significant effect of distractors on participants running out of time while working on a task, $V = .06$, $F(6, 81) = 0.92$, $p = .486$, $\eta^2 = .064$. This suggests that participants in both conditions had sufficient time to complete the training tasks (See Figure 4.9). Overall, we see that distractors appear to have a negative impact on participants' ability to complete the code puzzles.

Training Task Cognitive Load

Across all training tasks, distractor participants reported higher cognitive load than control participants. On average control participants reported their mental effort as $5.18 \approx \text{neither low nor high}$ ($SD = 1.67$), whereas distractor participants reported their mental effort as $5.73 \approx \text{rather high}$ ($SD = 1.57$). See Figure 4.10 for the cognitive load for each training task.

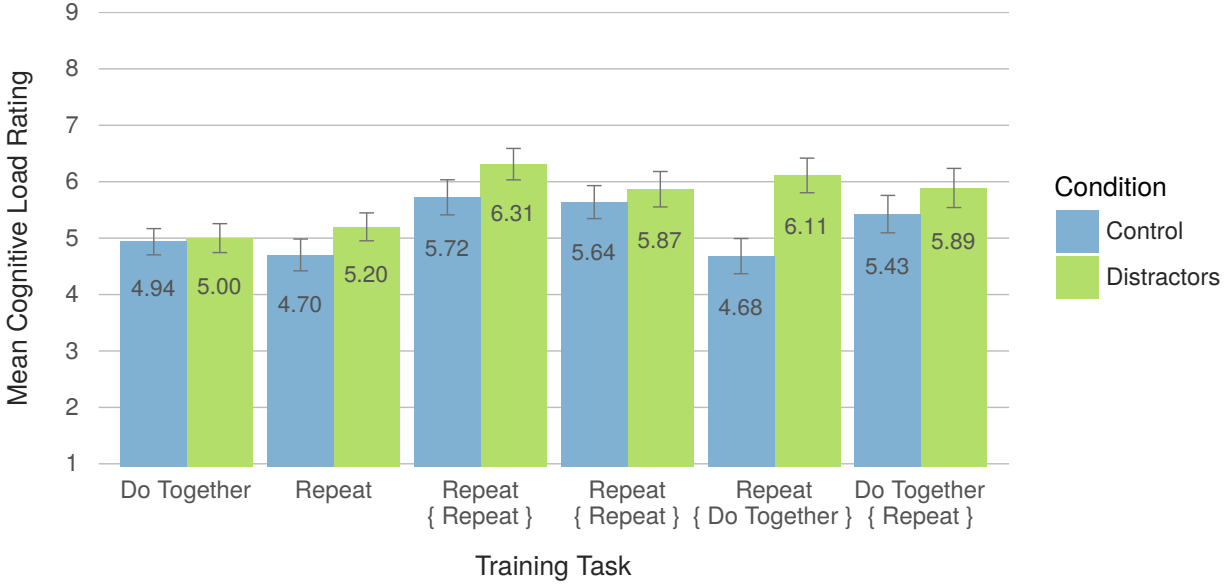


Figure 4.10: Mean cognitive load scores across all training task between control and distractor conditions. 1 - very, very low mental effort; 9 - very, very high mental effort. ($V = .21$, $F(6, 81) = 3.55$, $p = .004$, $\eta^2 = .208$)

The effect of distractors on cognitive load is significant and large, $V = .21$, $F(6, 81) = 3.55$, $p = .004$, $\eta^2 = .208$. These results suggest that distractors increase cognitive load for learners.

The CS CLCS reveals that the distractors significantly increased extraneous cognitive load, but not intrinsic or germane load. See Figure 4.11 for the average intrinsic, extraneous, and germane cognitive load reported for each condition. Control participants ($M = 3.56$, $SD = 2.44$) reported roughly the same intrinsic cognitive load as distractor participants ($M = 3.87$, $SD = 2.81$); $F(1, 86) = 0.01$, $p = .943$, $\omega^2 = .008$. Control participants also ($M = 5.10$, $SD = 2.55$) reported roughly the same germane cognitive load as distractor participants ($M = 4.79$, $SD = 2.40$); $F(1, 86) = 0.35$, $p = .557$, $\omega^2 = .007$. Control participants ($M = 1.43$, $SD = 1.62$) reported less extraneous cognitive load compared to distractor participants ($M = 2.56$, $SD = 2.46$). The effect of distractors on extraneous cognitive load is significant, but small, $F(1, 86) = 6.03$, $p = .016$, $\omega^2 = .049$. The increase

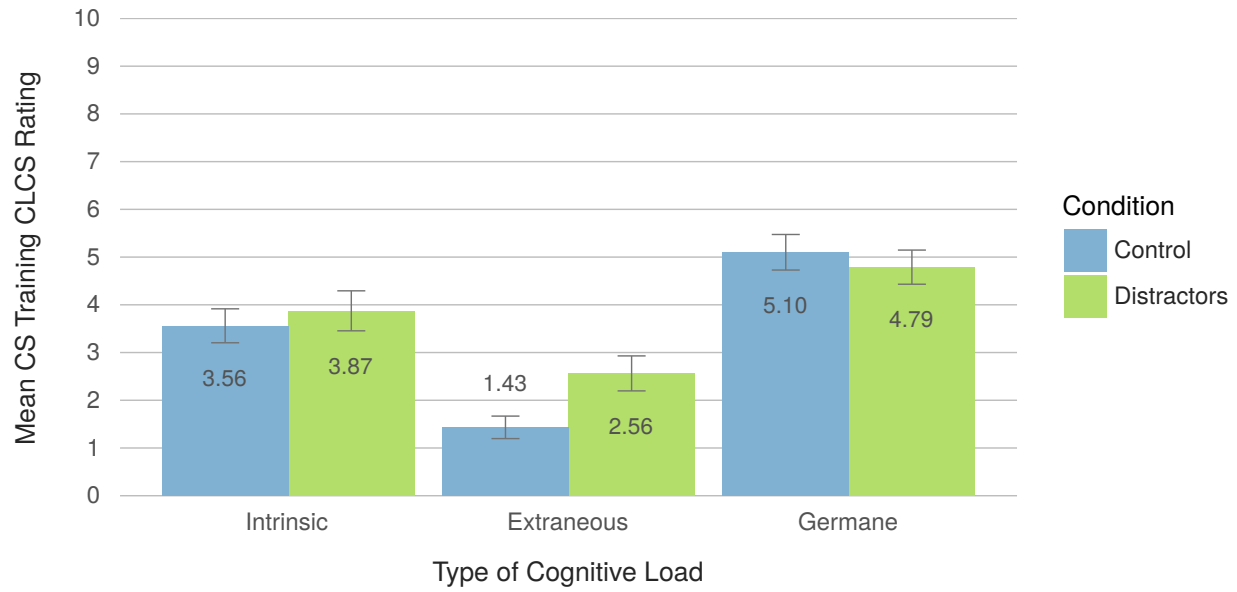


Figure 4.11: Mean cognitive load ratings in the training phase for each of type of cognitive load from the CS CLCS. 0 - not at all the case; 10 - completely the case. (intrinsic: $F(1, 86) = 0.01$, $p = .943$, $\omega^2 = .008$; extraneous: $F(1, 86) = 6.03$, $p = .016$, $\omega^2 = .049$; germane: $F(1, 86) = 0.35$, $p = .557$, $\omega^2 = .007$.)

in extraneous cognitive load suggests that distractors are detrimental for novices learning programming.

4.5.3 Do distractor participants show more evidence of learning?

Lastly, we wanted to know how distractors affected participants' ability to learn programming knowledge. Overall, we found no differences in control and distractor participants' transfer task time, transfer task performance, and cognitive load.

Transfer Task Time

Control participants ($M = 3.88$, $SD = .60$ minutes) spent roughly the same amount of time working on the transfer tasks as distractor participants ($M = 3.37$, $SD = .51$ minutes).

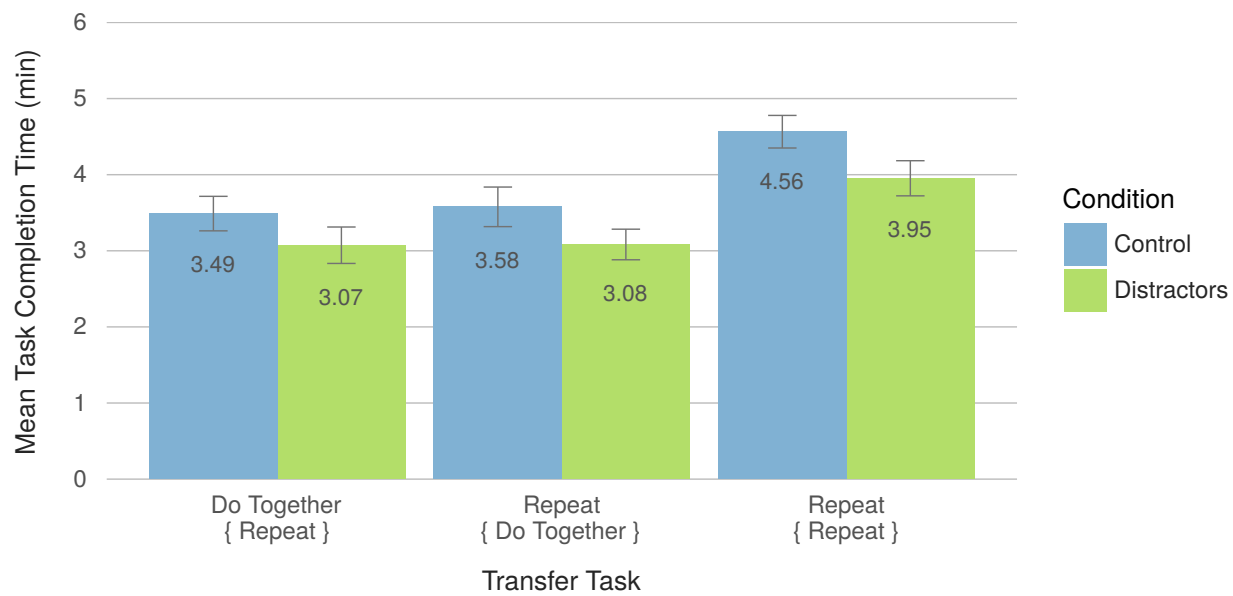


Figure 4.12: The mean transfer task completion time for all transfer tasks for both distractor and control participants. ($V = .06$, $F(3, 84) = 1.73$, $p = .168$, $\eta^2 = .058$)

There is not a significant effect of distractors on transfer task completion time, $V = .06$, $F(3, 84) = 1.73$, $p = .168$, $\eta^2 = .058$. See Figure 4.12 for the mean transfer task completion time for all transfer tasks.

Transfer Task Performance

Control participants ($M = 55\%$, $SD = 12\%$) completed roughly the same number of transfer tasks correctly as distractor participants ($M = 43\%$, $SD = 6\%$). There is not a significant effect of distractors on correctly completing transfer tasks, $V = .06$, $F(3, 84) = 1.78$, $p = .158$, $\eta^2 = .060$. See Figure 4.13 for the percent of participants who correctly completed each transfer task.

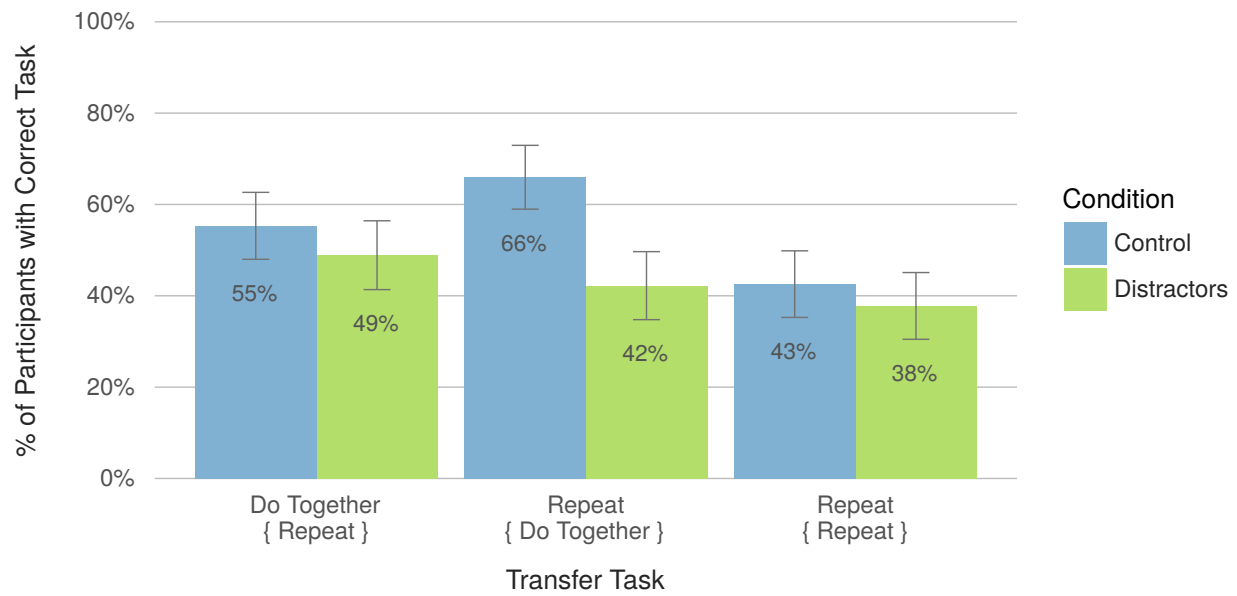


Figure 4.13: Percent of participants in the distractors or control conditions who *correctly* completed each transfer task. ($V = .06$, $F(3, 84) = 1.78$, $p = .158$, $\eta^2 = .060$)

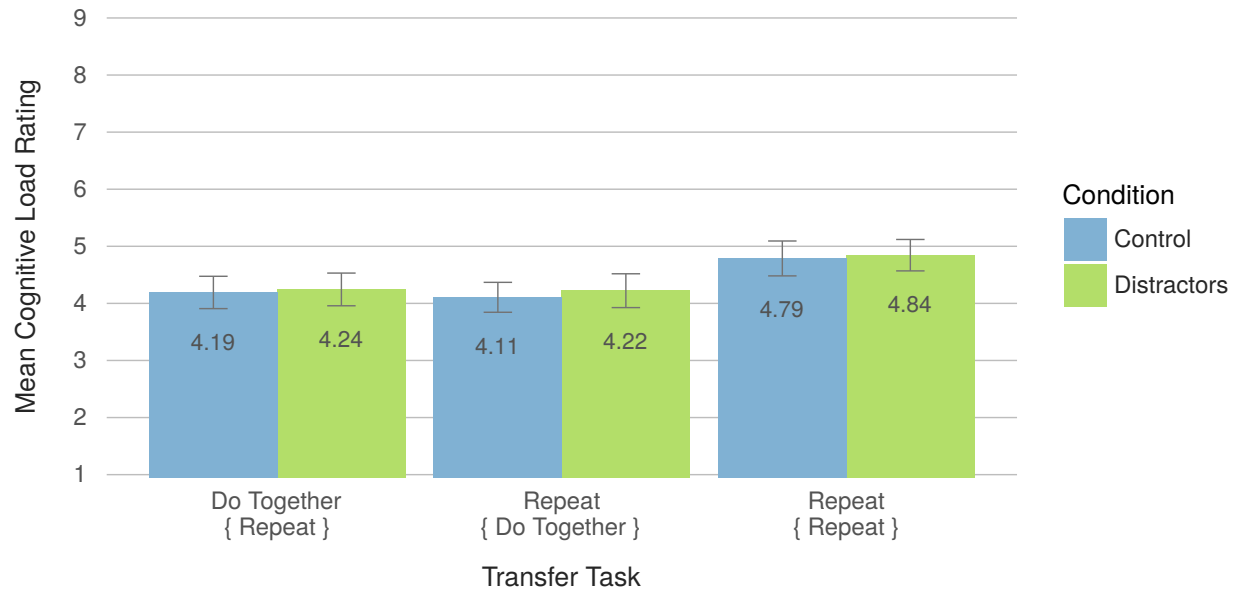


Figure 4.14: Mean reported cognitive load for each transfer task across both conditions. 1 - very, very low mental effort; 9 - very, very high mental effort. ($V = .00$, $F(3, 84) = 0.03$, $p = .992$, $\eta^2 = .001$)

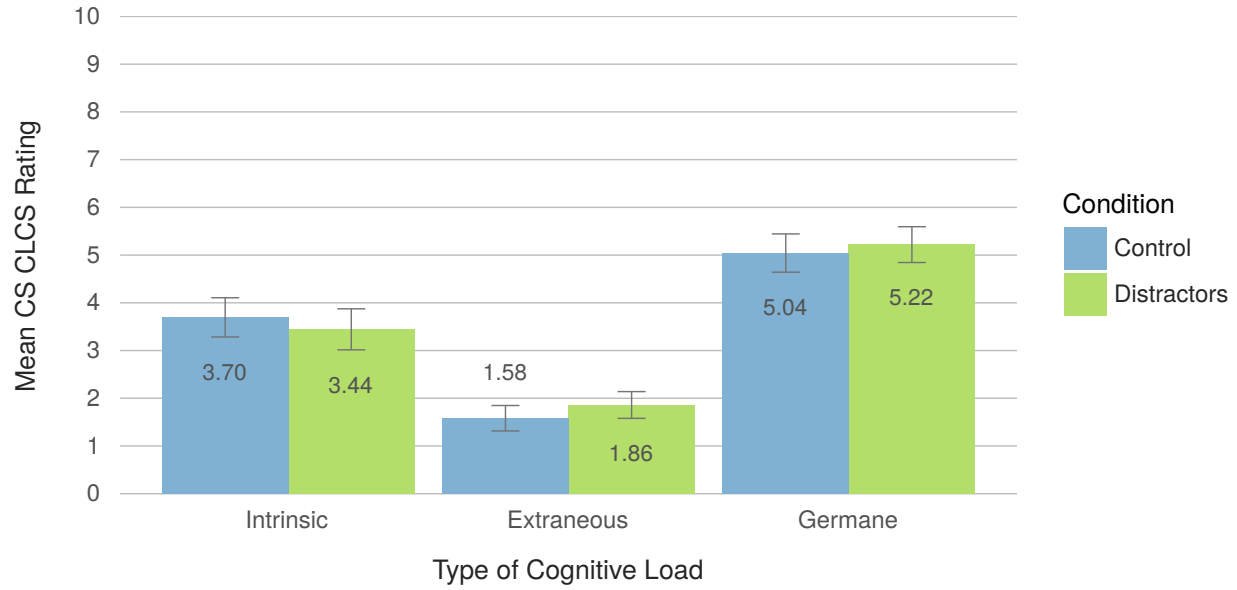


Figure 4.15: Mean cognitive load ratings in the transfer phase for each of type of cognitive load from the CS CLCS. 0 - not at all the case; 10 - completely the case. (intrinsic: $F(1, 86) = 0.52$, $p = .474$, $\omega^2 = .005$; extraneous: $F(1, 86) = 0.63$, $p = .431$, $\omega^2 = .004$; germane: $F(1, 86) = 0.02$, $p = .889$, $\omega^2 = .010$.)

Transfer Task Cognitive Load

Across all transfer tasks control and distractor participants reported roughly the same cognitive load. On average control participants ($M = 4.36$, $SD = 1.77$) and distractor participants ($M = 4.44$, $SD = 1.73$) reported *rather low* mental effort. The effect of distractors on cognitive load is not significant, $V = .00$, $F(3, 84) = 0.03$, $p = .992$, $\eta^2 = .001$. See Figure 4.14 for the mean reported cognitive load for all transfer tasks.

Further, there is no significant difference between conditions for intrinsic, $F(1, 86) = 0.52$, $p = .474$, $\omega^2 = .005$, extraneous, $F(1, 86) = 0.63$, $p = .431$, $\omega^2 = .004$, and germane, $F(1, 86) = 0.02$, $p = .889$, $\omega^2 = .010$, cognitive load. See Figure 4.15 for the average intrinsic, extraneous, and germane cognitive load reported for each condition.

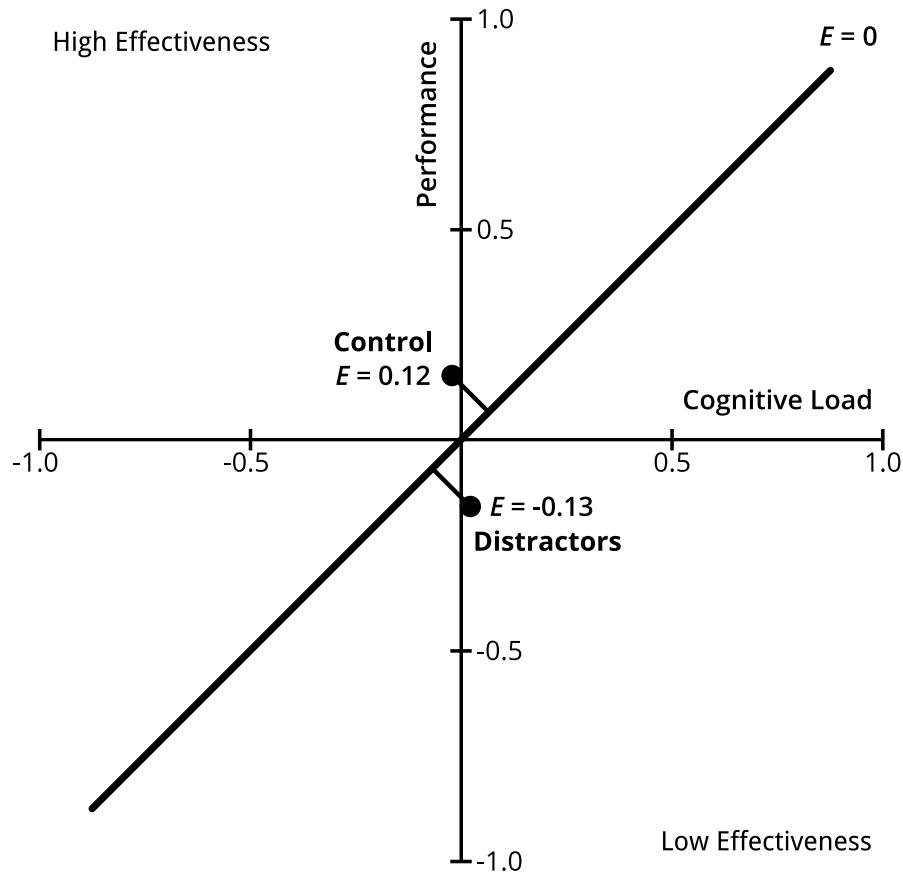


Figure 4.16: Instructional efficiency (E) between the control and distractors conditions. Points that lie above the $E = 0$ line reside in the high effectiveness area of the graph (top-left corner); points below the $E = 0$ line reside in the low effectiveness area (bottom-right corner).

4.5.4 Instructional Efficiency

To get an idea of how participants' cognitive load related to their performance in the transfer tasks, we computed *instructional efficiency*. Instructional efficiency is a measure that combines the transfer phase's cognitive load and performance in order to compare the effectiveness of instructional conditions [139, 144]. Figure 4.16 shows the instructional efficiency between the control and distractor conditions. From Figure 4.16, we see that the control condition is slightly more effective than the distractors condition.

4.6 Threats to Validity

This study looked at one type of code puzzle distractor, the *partial suboptimal path* distractor, in a very specific context: middle school children learning programming independently. There are other types of distractors and contexts that this study did not explore. For example, Parsons problems frequently contain *syntax error* distractors [90, 148]. Considering that syntax is often a significant hurdle for new programmers, future research is necessary to understand how other types of distractors affect learners.

We also note that our recruiting mailing list often includes parents who consistently seek additional learning opportunities for their children, possibly enabling selection bias towards higher performing students. Thus our results may not fully represent the wide range of abilities of middle school children.

4.7 Discussion

The results of this study suggest that errors in code puzzle completion problems provide no benefit for novices learning programming independently. In fact, the reduced learning efficiency that distractor participants experienced may impact other factors like motivation. In this section, we highlight several issues related to the use of distractors in code puzzles.

Transfer Performance Distractor participants experienced reduced learning efficiency during the training phase, while experiencing no difference in performance in the transfer phase. Based on our related work, we expected distractors to increase or decrease transfer task performance: the Parsons problem literature often asserts that distractors are beneficial [90, 148], suggesting increased transfer task performance; alternatively, based on the learning with

errors and testing literature we would have expected distractors to decrease transfer task performance [63]. However, the lack of difference in transfer task performance is inconsistent with either expectation. One possible explanation for the lack of negative performance for distractor participants is that the harmful consequences of the testing effect can be mitigated by providing test takers with corrective feedback after testing [15, 17]. The puzzle feedback indicator may have filled a similar role; the feedback indicator may have provided sufficient corrective feedback that any programming construct misconceptions from the training phase were corrected before completing the transfer tasks.

Further, the impact of distractors on transfer performance may be influenced by favorable prior knowledge. Große and Renkl found that if students had sound prior knowledge then their far transfer skills were enhanced when learning with correct and incorrect worked examples [63]. This suggests that distractors in code puzzles may be beneficial for experienced novice programmers, but only when learning from puzzles with and without errors. Because our participants lacked sufficient prior knowledge, additional research is necessary to determine if this is a factor for distractors in code puzzles.

Motivation The higher failure rate for completing the puzzles in the training phase for the distractor condition is worrisome (mean completion: control 88%, distractors 65%). Lower success may leave users feeling deflated and less motivated to continue learning programming. This is especially concerning for independent learners who may feel discouraged from learning programming informally and may opt out altogether.

Programming Experience Most participants had little prior exposure to the programming constructs tested in the transfer phase and on average performed poorly on all transfer tasks (control: 55%, distractors: 43%). The overall low performance suggests that participants were struggling to simply understand the basic concepts.

This is somewhat surprising considering the control condition and the puzzle condition from the evaluation in Chapter 3 completed two of the same transfer tasks: *do together { repeat }* and *repeat { do together }*. Yet, the control participants in this study performed much worse on both the *do together { repeat }* (55% versus 90%) and *repeat { do together }* (66% versus 80%) transfer tasks. This suggests that the training tasks in this study were much less effective than the training tasks in Chapter 3's study. The largest difference between the training tasks is Chapter 3's tasks gradually introduce nested constructs whereas this study very aggressively introduces them without additional puzzles to practice the constructs without nesting first.

4.8 Conclusion

In this study, we demonstrated that errors in the form of distractors in code puzzle completion problems provide no clear benefit for independent learners while also reducing their learning efficiency. Due to this result I partially accept Hypothesis II because even though distractors reduced learning efficiency they had no significant impact on transfer performance. Without clear evidence of benefit, educators should use caution if they choose to use distractors in code puzzles intended for novices because the result may not match their expectations. Based on the evidence I have presented so far, code puzzle completion problems are an effective approach to support independent learning, so long as distractors are not used. In the next chapter, I explore whether code puzzle completion problems are considered motivating to learners and whether learners view code puzzles as supportive of learning in independent contexts.

Chapter 5

Learners’ Perceived Value of Code Puzzle Completion Problems

This chapter was originally published as “Learning Programming from Tutorials and Code Puzzles: Children’s Perceptions of Value” in the *Proceedings of the 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* [69].

Up to this point, I have demonstrated that code puzzle completion problems can support learners in acquiring new programming knowledge. However, if learners are not motivated to use the code puzzles because they do not perceive them as valuable, then this approach is unlikely to succeed in independent contexts. In independent contexts, learners are unlikely to choose to use instructional materials, like puzzles, unless they directly align and support the learner’s specific goals. Additionally, in Chapter 3 we found a surprising mismatch in motivation between the formative and summative phases of the study [70]. These findings suggest that children may find both tutorials and puzzles useful. In this chapter, I investigate the ways in which code puzzle completion problems and tutorials are perceived as valuable to independent learners.

To investigate Hypothesis III, we conducted a qualitative study to explore users' perceptions of both instructional formats. Over the course of the study, we asked participants to select and complete six programs using their choice of tutorials and puzzles. By giving participants a choice, we gained valuable insight into how well learners choose to spend their time. This is especially important, because while learners are often motivated to learn when given a choice [34, 172, 204], they often have difficulty selecting skill appropriate tasks [36].

In this chapter, we report the results of our study with an explicit analysis of the motives and decisions middle school children make when choosing a particular instructional format. Throughout the study, we interviewed participants about their experiences and intentions for choosing between the instructional formats. From the interviews, we hoped to learn about the 1) circumstances under which novices choose to use tutorials or puzzles, and 2) novices' perceptions of value, including motivation, for each of the instructional formats. The results suggest that users' own personal interests play a large role in their instructional choice. Overall participants generally preferred the puzzles. However, we also found that the range of scaffolding provided by both formats enables learners to better follow and manage their own goals.

5.1 Related Work

We consider two areas of related work: 1) adult students and their perceptions of learning, and 2) the effectiveness of learning programming independently, especially for children.

5.1.1 Learners' Perceptions

Learners' perceptions and choices influence the way they navigate learning materials. For example, independent learners often choose not to read directions and tend to avoid repetitive

practice exercises when following instructional materials [20, 118]. Instead, learners prefer to begin working on their own tasks immediately and use instructional materials as a reference only when needed [19, 20]. Traditional-style instructional manuals can be especially frustrating for these task-oriented learners [19]. Aligning instructional materials more appropriately to learners' preferences improves their learning efficacy [19].

An alternative to aligning materials to learners' preferences is giving learners a choice in their learning and enabling them to make their own learning decisions. In the context of programming, students liked the ability to choose the instructional format (e.g. self-guided lab or tutorial) of supplemental classroom material [68, 154]. Students also felt that choosing their learning materials, like programming languages [155], projects [178], and online discussion format [93] helped them learn better. Similarly, students reported that compared to traditional classroom instruction, informal resources (e.g. online courses) enabled them to choose materials that were appropriate for their level, which they perceived as improving their ability to learn [9]. However, beyond students' perceptions, these researchers present little evidence that demonstrates that choice actually helps students learn better. Furthermore, we know little about how these results relate to independent learners, especially children, when trying to learn programming outside of the classroom experience.

5.1.2 Independent Learning Support

Novice programmers, and in particular children, who seek to learn programming independently can choose from several different instructional formats. Broadly, these formats fall into four categories: learning from online courses, examples, tutorials, and puzzle-like systems.

Massive open online courses (MOOCs) like Codecademy [27], Khan Academy [97], and edX [46], are an increasingly popular way for novices to learn programming. However, online

courses typically have high attrition rates, which can be attributed to an inconsistency between learners’ goals and the style of instruction [6, 169].

Instead, task-oriented learners can leverage example code and tutorials when seeking instruction [166]. Tutorons annotate online example code with explanation to help learners understand the code [75]. Online tutorials that incorporate example code with live program replay, like Online Python Tutor [64] and Codepourri [57] may also help learners understand code. Further, tutorials have been shown to help children learn new programming constructs [71]. Yet, it is unclear what value novices perceive in these tools.

An increasingly popular alternative is puzzle-like support for learning programming independently. Probably the most well-known puzzle-like system is the Hour of Code [77]. The Hour of Code is specifically targeted at enabling children to learn programming independently. Another popular type of code puzzle, the Parsons problem [148], have also been used in independent contexts [48], but they are more commonly used in classrooms. Puzzle-like systems generally set an explicit goal that users accomplish through programming. Frequently, this involves navigating an object through a grid [29, 104, 112]. Puzzle-like systems have also been shown to be effective tools to enable children to learn programming independently [70, 104]. However, little is known about children’s perceptions of instructional formats and how those perceptions factor into their decisions on when to use a particular format.

5.2 Evaluation

The goal of our exploratory study was to identify the factors that influenced participants’ decisions about learning content and instructional format. We asked participants to complete six instructional tasks. For each task, participants selected both the program (i.e. animation) they wanted to learn to create and the instructional format (i.e. tutorial or puzzle). We

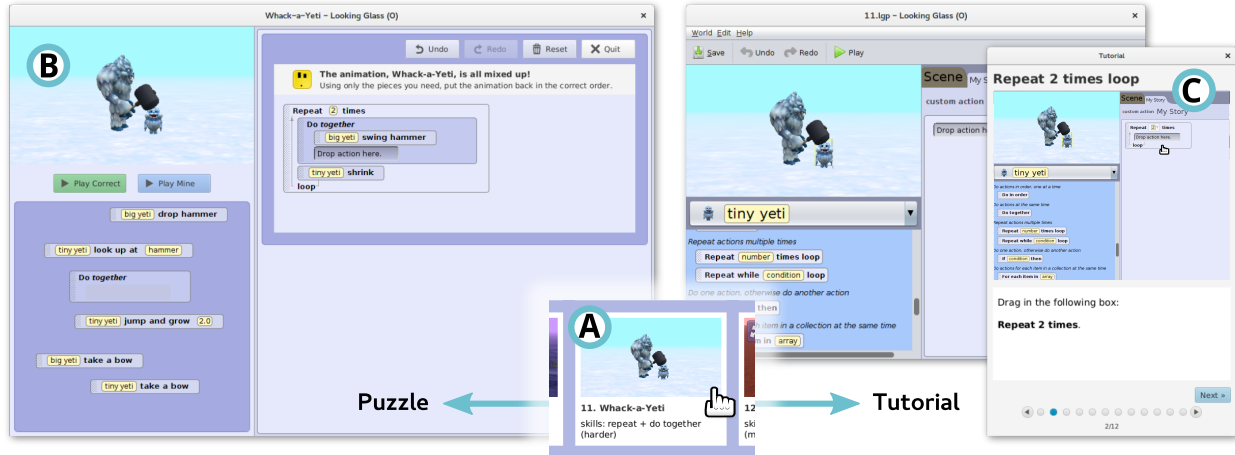


Figure 5.1: Participants are given a list of 14 instructional tasks (A). Participants choose whether to complete an instructional task (A) as a puzzle (B) or a tutorial (C).

interviewed participants throughout the study to gather information about the choices they made when choosing an instructional format.

5.2.1 Instructional Formats

We decided to give participants a choice between using tutorials and puzzles to complete their choice 6 of the 14 programs. We used the same tutorial interface from the summative evaluation in Chapter 3 and the revised code puzzle completion problems from section 2.4. See Figure 5.1 for both of the instructional formats used in this evaluation.

5.2.2 Participants

We recruited 30 participants between the ages of 10 and 15 years (14 female, 16 male; age: $M = 11.2$, $SD = 1.3$) from the Academy of Science of St. Louis mailing list. The Academy of Science is a not-for-profit organization in the St. Louis metropolitan area dedicated to science outreach. Participants self-reported their prior programming experience. Eighteen participants had less than three hours of prior programming experience, whereas twelve

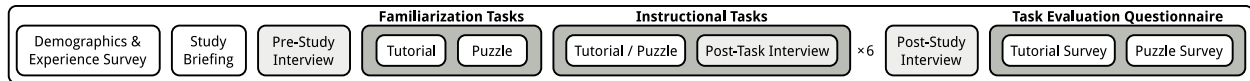


Figure 5.2: From left to right, the order of the study’s activities.

participants had at least three hours of prior programming experience. We compensated participants with a \$10 gift card.

5.2.3 Methods

We conducted our study through 30 individual, two-hour sessions. See Figure 5.2 for an overview of the study procedure. In the first part of the study, we familiarized participants with the study procedure. When participants arrived, we asked them to complete a demographic and programming experience survey. We then briefed participants on the study format. During the briefing we also primed participants by requesting that they focus on the goal of increasing their programming skills while working their way through the study. During our pilot testing we discovered that many participants made decisions based on whether they found an animation interesting. We have chosen to include this priming in our study in order to gather perceptions of value beyond compelling animations.

After the briefing, we conducted the pre-study interview and then asked participants to complete two familiarization tasks. The familiarization tasks were designed to introduce participants to the experience of using a tutorial and puzzle. We randomly assigned participants to complete a tutorial or puzzle first. During the familiarization tasks, we informed each participant that we could provide assistance and answer questions. We also informed participants that once they began the instructional task part of the study, we would be unable to provide any assistance.

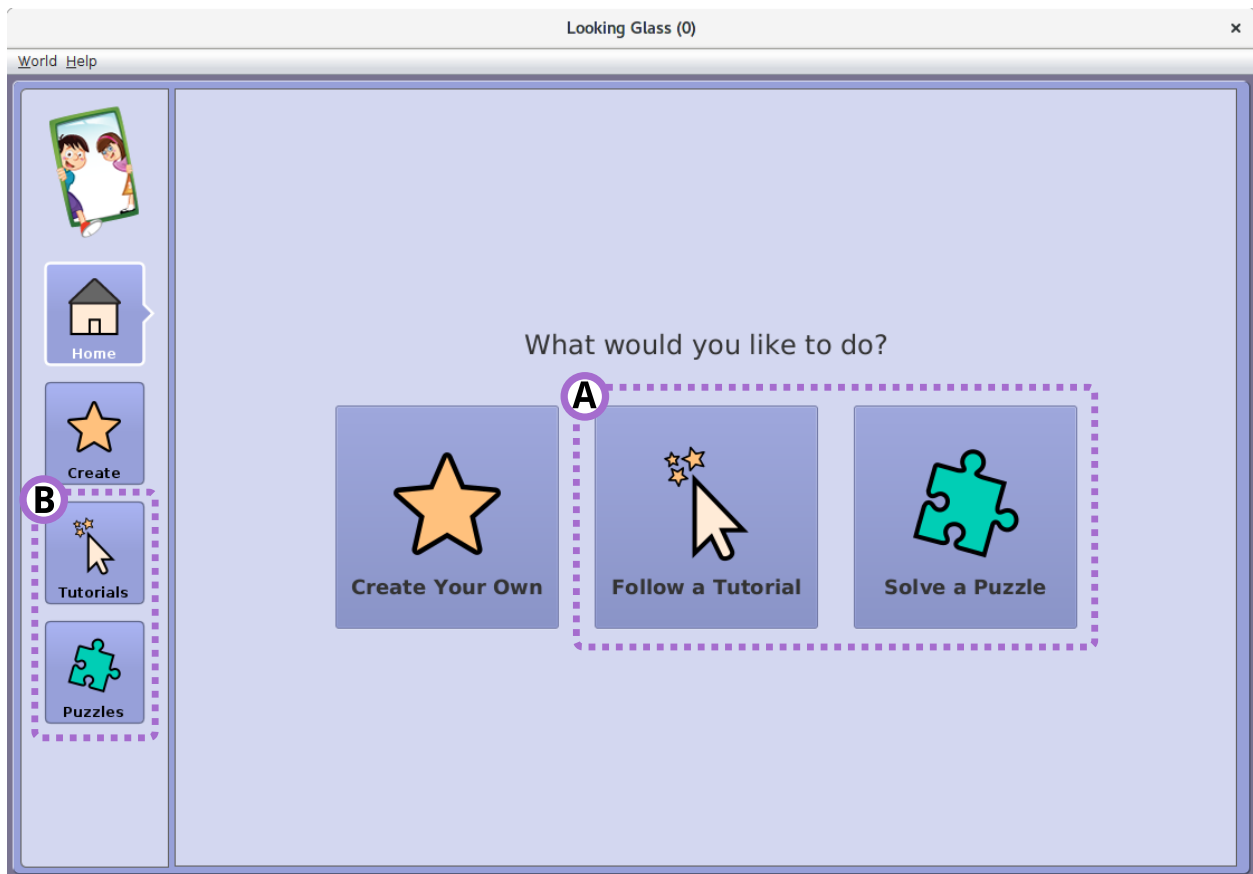


Figure 5.3: Instructional format selection screen. Participants first select the type of instructional format they want to use for an instructional task: tutorial or puzzle (A). Participants may also select the instructional format in the side pane (B).

For the remainder of the study, we asked participants to complete six instructional tasks. Participants would first select whether they would use a tutorial or a puzzle to learn how to create an animation, as shown in Figure 5.3. Participants then selected from among fourteen animations. See Figure 5.4 for the instructional task selection screen for tutorials. There was no time limit for the instructional tasks. Upon completion of each task, a researcher interviewed the participant about that task. We allowed participants to stop working on a task at any time. If a participant choose to stop before completing a task, we interviewed them about their decision to quit that task.

After completing the instructional tasks, we interviewed participants about their entire study experience in the post-study interview. We then asked them to complete two motivation inventories, one for tutorials and one for puzzles. Participants decided which motivation inventory they completed first.

5.2.4 Materials

Our study included familiarization tasks, instructional tasks, semi-structured interviews, and surveys. We iteratively developed and refined these materials through a pilot study with 29 participants (9 female, 20 male; age: $M = 11.2$, $SD = 1.8$). See Appendix G for all the materials, including interviews, used in this evaluation.

Familiarization Tasks

During piloting, we discovered that participants always chose to complete a tutorial and puzzle to get a feel for each format. Because this behavior was consistent and common among all participants, we decided to create two familiarization tasks to remove this variability from our final evaluation. We designed the familiarization tasks to give participants a taste

Table 5.1: The fourteen instructional tasks.

Task	Title	Programming Constructs
1	Dizzy Walrus	Do in Order
2	Hammer Hazard	Repeat
3	Monkey Business	Repeat
4	Yeti Baseball	Do Together
5	The Snow Dance	Do Together
6	Messed Up Magic	Do Together
7	Interstellar Travel Troubleshooting	Repeat & Do Together
8	Trouble at Sea	Repeat { Repeat }
9	Shark Snack	Do Together { Repeat }
10	Crazy Cauldron	Repeat { Do Together }
11	Whack-a-Yeti	Repeat { Do Together }
12	Firetruck Frenzy	Do Together { Do in Order }
13	Air Traffic	Do Together { Do in Order }
14	Polar Surprise	Do Together { Do in Order }

of each instructional format without exposing them to new programming constructs. The familiarization tasks each consisted of a simple, sequential animation with four statements. Each animation could be completed as a tutorial or as a puzzle.

Instructional Tasks

We developed fourteen instructional tasks as shown in Table 5.1. Each instructional task is a program (i.e. animation) that participants can choose to complete as a tutorial or as a puzzle. The animations spanned a range of difficulties and programming constructs. Programming constructs included sequential execution (*do in order*), *repeat*, *do together*,

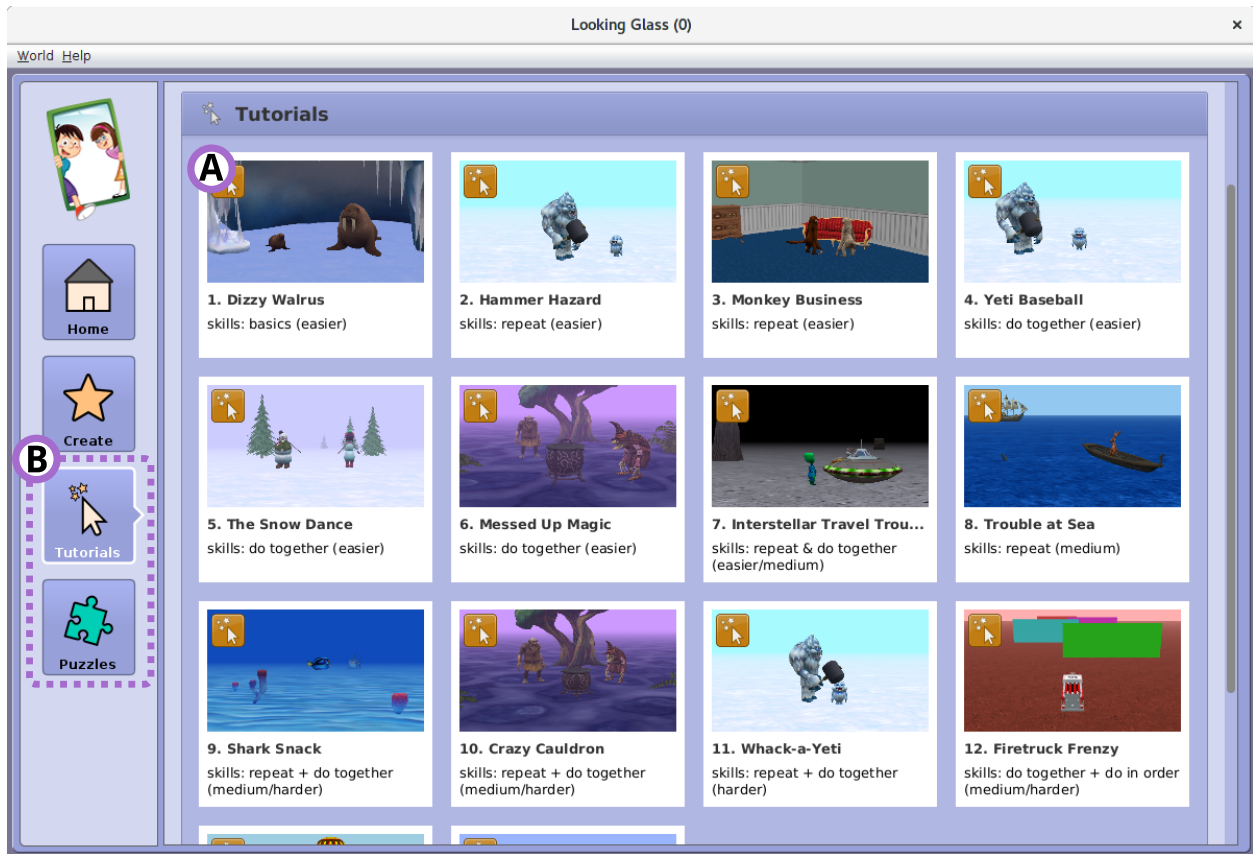


Figure 5.4: Instructional task selection screen. After selecting the type of instructional format (Figure 5.3), tutorial or puzzle, participants select an instructional task from among 14 animation programs (A). Participants may change the instructional format at anytime by clicking the desired format button (B).

repeat & do together, *repeat { repeat }*, *repeat { do together }*, *do together { repeat }*, and *do together { do in order }*.

Additionally, we developed an interface that allows participants to select an instructional format and task (Figures 5.3 and 5.4 respectively). For the instructional task selection screen (Figure 5.4), we adapted the puzzle selection screen from our revised puzzle interface (Figure 2.17). Our adaptations included adding numbering and a difficulty rating to each task. We established the difficulty ordering based on pilot testing. Altogether, the interface contains a thumbnail of each animation, the animation's title, the ranked difficulty (1-14), a

difficulty label (easier, medium, and harder), and the programming constructs demonstrated in the program. Participants could also click the thumbnail to view the full animation for each task.

Semi-Structured Interviews

We developed questions for four semi-structured interviews: a *pre-study interview*, a *post-task interview*, an *early termination interview*, and a *post-study interview*. Each interview contained questions that sought extended responses and one-word responses. We captured participants' responses in audio recordings. The interview questions were informed and refined by the themes that emerged through our pilot study.

The *pre-study interview* contained five questions about participants' prior programming experience. Additionally, we asked participants to rank their programming skill using a Likert scale from 1 - *novice* to 5 - *expert* [89]. For all Likert scales in the study, we presented both numerical and ordinal values.

In the *post-task interview*, we asked participants a series of 17 questions about their decision, the quality and value of their experience with the task, and what they planned to do next. Three questions asked participants to rank their expected difficulty for the task, the actual difficulty for the task, and how difficult the task would have been if they had used the other instructional format using a Likert scale ranging from 1 - *extremely easy* to 9 - *extremely difficult* [89].

The *early termination interview* consisted of a subset of the questions from the post-task interview. These questions focused on perceived difficulty and their plans for what to work on next. We included an additional question asking participants about their reasons for terminating the task early.

In the *post-study interview*, we asked participants 20 questions exploring their experiences during the study, their preferences regarding instructional task format, difficulties with tasks, and comparative questions about the value between the instructional formats. We also asked participants to re-rate their programming experience from 1 - *novice* to 5 - *expert*.

Surveys

Our study included two surveys. The first survey, completed before starting the study, asked participants to self-report their age, gender, schooling and prior programming experience. For the second survey, completed at the end of the study, we used the Intrinsic Motivation Inventory's Task Evaluation Questionnaire (TEQ) [85]. The TEQ is a 22-item Likert survey with four subscales: *interest/enjoyment*, *perceived competence*, *perceived choice*, and *pressure/tension*. Participants rated their agreement with individual statements from 1 - *not true at all* to 7 - *very true*. We asked participants to complete two TEQ surveys, one for tutorials and one for puzzles.

5.3 Analysis

We analyzed the reliability of the TEQ motivation subscales and performed a qualitative coding on the interview responses.

5.3.1 Task Evaluation Questionnaires

We determined the reliability of the TEQ motivation subscales by combining the results from the tutorial and puzzle TEQ surveys and computing Cronbach's alpha for each subscale (Cronbach's $\alpha > .70$). Three of the subscales were reliable: *interest/enjoyment* ($\alpha = .92$),

perceived choice ($\alpha = .77$), and pressure/tension ($\alpha = .73$). We have not reported the results for the perceived competence subscale ($\alpha = .68$).

5.3.2 Interview Responses

The primary data analysis we performed was the qualitative coding of the interview data. Over the course of the study, we collected an average of 40.2 ($SD = 8.8$) minutes of recorded interview responses for each participant. We transcribed the audio recordings and separated the responses by question. This produced a total of 3,915 question responses. Because we used semi-structured interviews, the responses to some questions included follow-on questions and responses. We grouped any follow-on questions with the original interview question.

Using a grounded theory approach, we developed a set of high-level categories for the responses. We chose to categorize based on participants' responses rather than the questions for two reasons: 1) the interviews included related questions and 2) participants sometimes responded with information that did not perfectly align to each question. We developed the initial categories by manually sorting roughly 10% of the responses into similar groups. We identified five categories: *decision rationales*, *expected task difficulty*, *sources of ease and difficulty*, *experience outcomes*, and *other*. Reassuringly, the categories that emerged from this process align closely with the themes we tried to incorporate into the interviews. Once the categories emerged, two researchers labeled a new random sample of 20% of the responses. We reached very high agreement (Cohen's $\kappa > .8$) for the categories ($\kappa = .95$, $p < .001$). One researcher then categorized the remaining responses.

We subsequently repeated this process to develop sublabels for each of the high-level categories, with the exception of the *other* category. We omitted the *other* category, because many of the responses were unrelated. Because the sublabels were based on the original high-level

Table 5.2: Sublabel interrater agreement.

High-Level Category	% of Responses	Cohen’s κ	adj. p
Decision Rationales	16%	0.89	$p < .001$
Expected Task Difficulty	13%	0.85	$p < .001$
Sources of Ease & Difficulty	15%	0.88	$p < .001$
Experience Outcomes	38%	0.86	$p < .001$
Other	18%	n/a	n/a

categories, we report Bonferroni adjusted p values for Cohen’s κ . We reached very high agreement for all sublabels as shown in Table 5.2. We report the sublabels for each category in the results section.

5.4 Results

We share our findings in this study by reporting each category separately. Each of the categories provides insight into the decisions that participants made, the factors that influenced those decisions, and participants’ perceptions of value that they received from the task and instructional format. For each category, we share relevant survey results and summarize the general themes that emerged from participants’ responses.

When summarizing the interview responses, we report each category’s sublabels in two ways: 1) the overall percentage a sublabel was cited across all tasks in the study and 2) the percentage of participants who cited the sublabel at least once during the study. The percentage across all tasks gives insight into what types of decisions participants made, whereas the percentage across all participants helps demonstrate what factors participants identify as important. We report a summary of each high-level category’s sublabels in Tables 5.3–5.6. In these tables we summarize the interview responses in the same two ways: 1) *% of Tasks*, and 2) *% of*

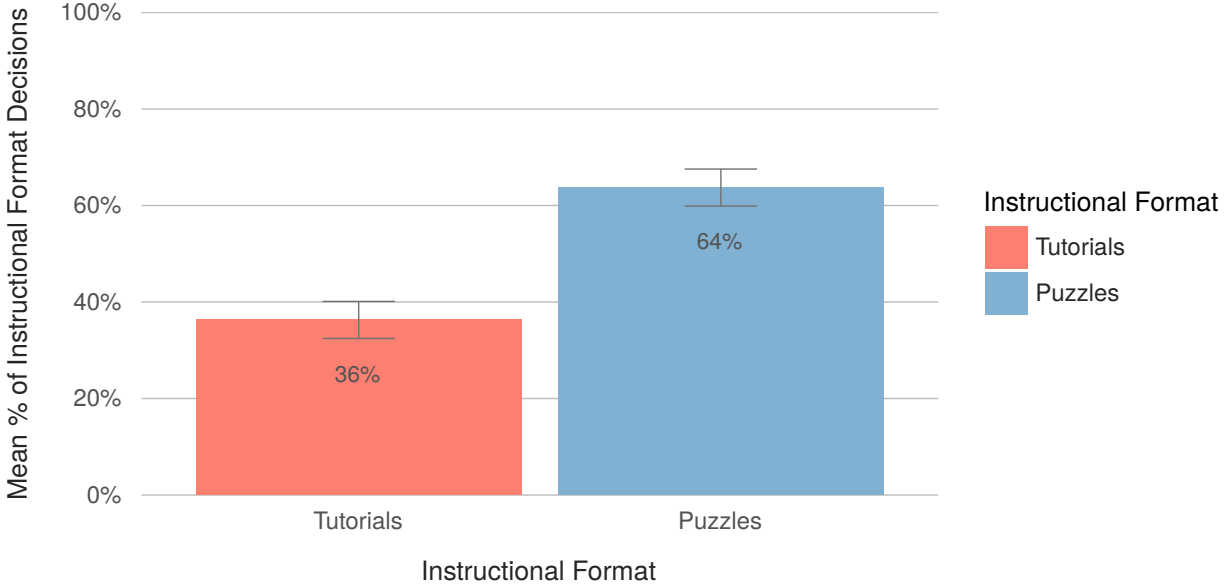


Figure 5.5: Mean percentage of tutorial and puzzles participants decided to work on for their instructional tasks. ($t(29) = 3.40$, $p = .002$, $d = 1.13$)

Participants, respectively. For each high-level category we had two additional sublabels that we do not discuss in our results: *no reason / neutral* and *other*. The *no reason / neutral* sublabel captured responses that provided no reason and were often neutral in sentiment, for example “it was good.” The *other* sublabel captures responses that did not answer the question or that are off topic.

We begin by first discussing the decision rationales category to understand participants’ decisions. We then follow-up with the remaining high-level categories shown in Table 5.2.

5.4.1 Decision Rationales

The decision rationales category analyzes the explicit reasons participants gave for choosing an instructional task and format. In total, participants worked on 167 instructional tasks (62 tutorials, 105 puzzles). 21 participants (70%) completed all six instructional tasks; the remaining 9 participants (30%) completed between three and five tasks ($Mdn = 4$, $M = 4$,

$SD = .83$). Most participants chose to work on both tutorials ($Mdn = 2$) and puzzles ($Mdn = 3$). As shown in Figure 5.5, participants were more likely to opt to use more puzzles than tutorials; this effect is significant and large: $t(29) = 3.40$, $p = .002$, $d = 1.13$. Overall, 10% of participants decided to work on more tutorials than puzzles, 60% of participants worked on more puzzles, and 30% of participants decided to work on an equal number of both.

Participants indicated that they valued both the tutorial and puzzle formats. In the post-study interview, we asked participants which format they were more likely to use on their own after this study. 53% of participants responded that they would use both, while 30% said puzzles, and the remaining 17% said tutorials. This is further supported by the variety of decisions participants made during the study. Overall, we see that the majority of participants (73%) used both tutorials and puzzles. While a small minority show a strong preference (number of tasks minus one) for tutorials (3%) or puzzles (23%).

From the sublabels, we see that participants made decisions based on personal preference, improving their programming skills, and challenge (see Table 5.3). We found that participants were more likely to choose a task based on their personal preference rather than choosing a task based on improving their programming skills. However, when they did make a decision to improve their programming skills, many participants chose the tutorials, the format that we demonstrated in Chapter 3 as less effective [70]. Throughout the study participants also actively managed the difficulty of their tasks by choosing an instructional format that they believed would increase or decrease each task's challenge.

Table 5.3: Decision rationale response themes.

	% of Tasks (Tutorials, Puzzles) [†]	% of Participants (Females, Males)
Personal Preference	56% (52%, 58%)	87% (71%, 100%)
Enjoyed Animation	48% (47%, 49%)	83% (71%, 94%)
Liked Format	16% (8%, 20%)	53% (43%, 63%)
Improve Programming Skills	32% (52%, 21%)	73% (79%, 69%)
Improve Skill	12% (19%, 8%)	37% (36%, 38%)
Discover Level	21% (34%, 13%)	53% (64%, 44%)
Challenge	66% (65%, 68%)	100% (100%, 100%)
Avoid Challenge	22% (39%, 12%)	60% (43%, 75%)
Seek Challenge	48% (32%, 57%)	87% (93%, 81%)
Other	17% (13%, 19%)	53% (36%, 69%)
No Reason / Neutral	12% (13%, 11%)	40% (29%, 50%)
Other	5% (0%, 8%)	27% (21%, 31%)

[†] Percent of sublabels for all tutorials/puzzles; not percent across all tasks.

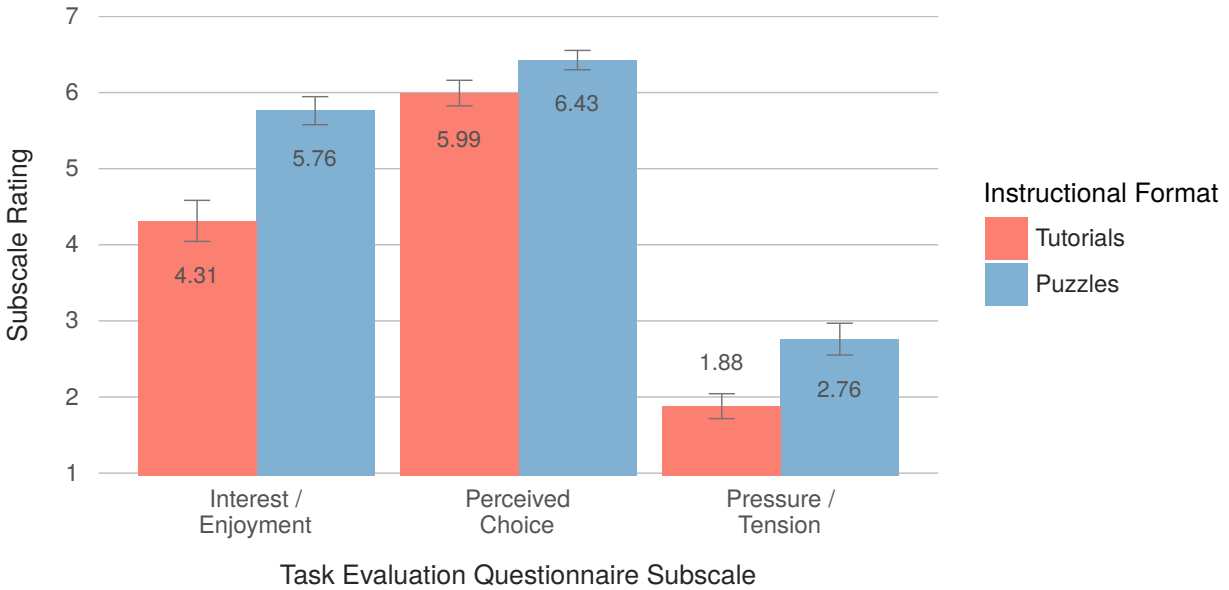


Figure 5.6: Mean subscales for the Task Evaluation Questionnaire (TEQ). 1 - not true; 7 - very true. (Interest/Enjoyment: $\chi^2(1) = 17.52$, $p < .001$; Perceived Choice: $\chi^2(1) = 6.48$, $p = .011$; Pressure/Tension: $\chi^2(1) = 10.46$, $p = .001$)

Personal Preference

Personal preference was a common reason participants cited for their decisions. Overall, our results suggest that participants enjoyed completing puzzles more than tutorials.

Participants cited enjoying the instructional format as an important factor in their decisions. In total, 53% of participants made a decision during the study based on whether they liked the instructional format. 43% of participants based at least one decision on enjoying the puzzles, “I like the puzzles; they’ve all been fun.” In contrast, only 17% of participants cited making a decision based on liking the tutorials, “I like the tutorials better.” The greater enjoyment of puzzles is also reflected in the percentage of tasks: participants cited enjoyment of puzzles in 20% of tasks, while only citing enjoyment in 8% of tutorials.

In the post-study interview, we asked participants which instructional format they enjoy more. Again, we see a preference towards puzzles: 80% of participants stated that they enjoy the puzzles more, compared to 13% who enjoy the tutorials more, and 7% who enjoy both equally. This preference is also reflected in the TEQ motivation survey’s interest/enjoyment scale, as shown in Figure 5.6. Using a multilevel model, we found that participants rated the puzzles as significantly more enjoyable ($M = 5.8$, $SD = 1.0$) than the tutorials ($M = 4.3$, $SD = 1.5$), $\chi^2(1) = 17.52$, $p < .001$.

In 48% of tasks, participants cited enjoying an animation as the reason for making a decision. In total, 83% of participants made a decision that was based on whether or not they liked the animation. Participants cited the overall appeal of an animation: “Probably just because it looked like a fun animation to do.” Some participants also described specific attributes: “I liked how the alien kicked the spaceship to start it up and when it flew away.” The specific content presented via learning resources is an important attribute for children learning programming.

Improve Programming Skills

Participants made fewer decisions that prioritized improving their programming skills compared to their personal interests. When making decisions to improve their programming skills, they did so either directly; by seeking to acquire new programming skills or indirectly; by trying to gain a better understanding of their skill level and the instructional formats.

A relatively small number of participants (37%) cited a goal of improving their programming skills across only 12% of tasks. This is a surprisingly low result given that we primed participants to make decisions to improve their skills. However, when participants did cite improving programming skills as a goal, they were more than twice as likely to have chosen a

tutorial (19% of tasks) than a puzzle (7% of tasks). This difference seems to reflect a belief from users that they learn more using tutorials. When asked for the reason behind a tutorial decision, one participant explained, “Because it was a new skill that I hadn’t learned before, so if I did it as a puzzle I’d probably not quite understand exactly what it was trying to teach me.” This tendency to select tutorials when prioritizing learning is interesting because it does not reflect the results of our evaluation in Chapter 3 which found that puzzles were both more efficient and effective for learning new constructs [70].

Choosing appropriate learning goals also requires both accurate knowledge of the available learning resources and knowledge of their own abilities. Participants described a variety of goals that contribute towards these types of self-knowledge. In 21% of tasks, participants described their desire to discover their skill level. Some attempted discovery by equally using both formats. One participant justified a decision as “Just so I can get a feel for both tutorials and puzzles.” Other participants talked about comparing the difficulty between the two formats: “I wanted it to be a little harder than the tutorial because I wanted to see if it’d make a big difference or not.”

Participants’ responses indicate a strong preference towards enjoyment rather than improving skills. However, when attempting to improve their programming skills, their responses suggest a nuanced approach in which they explore what roles resources could play, assess their own skill level, and select resources that they believe could contribute towards that goal.

Challenge

Participants frequently cited *challenge*, pushing their abilities, as a factor in their decisions. Overall, 87% of participants sought challenge while 60% also avoided challenge. Plotting the expected difficulty of each instructional task for each participant, we observed fluctuation

between easy and hard tasks for roughly 80% of participants. Of the remaining 20%, 7% of participants consistently avoided challenge and 13% consistently sought challenge. This suggests that participants do not always want to work on the hardest task, but instead they prefer a mix between challenging and non-challenging tasks.

When seeking challenge, participants tended to select puzzles. Participants described seeking a challenge for 57% of the puzzles and 32% of the tutorials. When avoiding challenge, participants favored tutorials. In 39% of tutorial decisions and 12% of puzzle decisions, participants expressed a desire to decrease the challenge level. This disparity suggests that participants perceived tutorials as being easier. The post-interview provided further support for the perception that tutorials are easier. In each post-task interview we asked participants to rank the difficulty of their chosen instructional format from 1 to 9. Participants ranked puzzles as significantly more difficult ($M = 4.8 \approx \text{neither easy nor difficult}$, $SD = 1.9$) than tutorials ($M = 3.8 \approx \text{slightly easy}$, $SD = 1.6$), $\chi^2(27) = 86.75$, $p < .001$. Further, participants also never ranked a tutorial above a 6 $\approx \text{slightly difficult}$, suggesting a difficulty ceiling for tutorials. Participants also reported significantly less pressure/tension in the TEQ motivation survey when using the tutorials ($M = 1.9$, $SD = .9$) compared to puzzles ($M = 2.8$, $SD = 1.1$), $\chi^2(1) = 10.46$, $p = .001$ (See Figure 5.6).

Participants' desires to seek challenge were mostly straightforward: "I like having to challenge my mind more." We saw two classes of reasons for avoiding challenge. The first group of participants expressed a desire to complete an animation they expected to be too challenging for them. These participants perceived tutorials as a more gentle introduction to the content required for a challenging animation. "I picked it as a tutorial because it looked like it had lots more complexity than the other ones, and I didn't want to just jump right in without knowing what I was doing." The second group of participants avoided challenge after

extending themselves outside of their comfort range. “I think I’m going to do mostly tutorials from now on because this was a bit hard.”

While there is some variety in what participants perceived as too difficult, participants who gave up on a task before successfully completing it arguably felt that they had taken on too much. We note that this was relatively rare; it happened in 8 out of 167 total tasks. However, all eight of these tasks were puzzles. For these eight, participants uniformly went on to easier tasks afterwards. Six chose to complete a tutorial for their next task. The other two chose an easier puzzle. While the failed tasks are a more extreme situation, participants’ decisions to reduce the difficulty were consistent with the overall pattern of pushing a little bit and then scaling back. The presence of both puzzles and tutorials empowered participants to adjust the experience to meet their own learning and confidence needs.

5.4.2 Expected Task Difficulty

In the decision rationales section (5.4.1) we saw that challenge was a prominent reason given for making a decision. When trying to manage challenge, participants considered a variety of information that informed their expectations about task difficulty (see Table 5.4). Our results suggest that participants used appropriate factors for predicting task difficulty. Participants typically weighed several practical factors: their prior programming experience, their perception of the instructional format’s intrinsic difficulty, and the animation’s labeled difficulty. However, there is one factor that may hamper their ability to do this accurately: animation complexity.

Table 5.4: Expected task difficulty response themes.

	% of Tasks (Tutorials, Puzzles) [†]	% of Participants (Females, Males)
Programming Experience	23% (21%, 24%)	77% (71%, 81%)
Programming Experience	20% (19%, 21%)	70% (71%, 69%)
Code Structure	2% (2%, 3%)	10% (0%, 19%)
Instructional Format	14% (31%, 4%)	47% (50%, 44%)
Labeled Difficulty	31% (21%, 36%)	67% (64%, 69%)
Animation Complexity	23% (23%, 23%)	63% (71%, 56%)
Other	16% (13%, 18%)	43% (50%, 38%)
No Reason / Neutral	14% (10%, 17%)	40% (50%, 31%)
Other	2% (3%, 1%)	10% (7%, 13%)

[†] Percent of sublabels for all tutorials/puzzles; not percent across all tasks.

Programming Experience

70% of participants considered their own programming experience when reasoning about difficulty. One participant expected a task to be easy “Because all of the skills in this one I had already learned.” An additional 10% of participants reasoned about code structure. Another participant expected a selected task to be difficult: “Because it was a new skill that I hadn’t learned and I’d have to figure out that you would have to put the *do in order* inside the *do together* with something outside of the *do in order*.”

Instructional Format

47% of participants based their expectations of difficulty on the instructional format. For example, one participant expected that, “The tutorial experience would be pretty easy.” Participants often stated that tutorials were easier “Because all you do is look at the tutorial, it tells you what to do, you do that, and then you turn the page and you do that again with the next instruction.”

During the post-study interview, we asked participants to summarize their advice to potential new users about when they should use tutorials and puzzles. Their advice reiterates the theme of using the different instructional formats to manage challenge. 17% of participants stated that you should use a tutorial when you expect a task to be difficult. 13% of participants stated that you should use a puzzle when you want to challenge yourself. Lastly, 41% of participants thought that if something is new for you, you should do a tutorial first, and then practice with puzzles later.

Labeled Difficulty

Our task selection interface labeled all tasks with one of three difficulty labels: easier, medium, and harder. In 31% of tasks, participants based their expectations on these difficulty labels.

Animation Complexity

63% of participants stated that they expected a task to be easy or difficult based on watching the animation. One participant expected a task to be challenging “just because there was, I mean, it looked like there was lots of stuff happening.” In total, participants used animation complexity to inform their difficulty assessment in 23% of tasks. While this may be a natural thing to do, it creates an interesting tension. Participants often considered the appeal of an animation in selecting a task. Participants’ desire to complete a compelling animation creates an incentive to create more complex and visually interesting animations. However, the same push towards compelling animations may also lead users to perceive them as out of reach.

5.4.3 Sources of Ease & Difficulty

When making decisions, participants commonly used their generalizations about difficulty as discussed in the previous sections. However, in the post-task interviews participants identified what they believed to be the source of difficulty for that task. Participants’ expectations often aligned with their actual experience of difficulty. In this section, we discuss the elements that participants felt contributed to the ease or difficulty of completing tasks (see Table 5.5). Overall, participants identified fairly common sources of difficulty. Participants felt that the instructional format, their prior programming experience, the interface’s mechanics, and their own personal degree of struggle, were sources of task difficulty.

Table 5.5: Sources of ease & difficulty response themes.

	% of Tasks (Tutorials, Puzzles) [†]	% of Participants (Females, Males)
Instructional Format	31% (58%, 15%)	87% (79%, 94%)
Prior Experience & Barriers	55% (37%, 66%)	100% (100%,100%)
Prior Experience	23% (24%, 22%)	60% (50%, 69%)
Conceptual Barrier	37% (15%, 51%)	93% (86%,100%)
Translation Barrier	2% (0%, 4%)	13% (7%, 19%)
Mechanics	21% (29%, 16%)	63% (57%, 69%)
Instructional Format	11% (10%, 11%)	30% (36%, 25%)
Programming Environment	13% (24%, 6%)	57% (50%, 63%)
Degree of Struggle	49% (32%, 59%)	90% (86%, 94%)
My Experience	37% (21%, 47%)	83% (79%, 88%)
Length	16% (13%, 18%)	57% (79%, 38%)
Other	25% (26%, 25%)	73% (71%, 75%)
No Reason / Neutral	25% (26%, 24%)	70% (71%, 69%)
Other	1% (2%, 1%)	7% (0%, 13%)

[†] Percent of sublabels for all tutorials/puzzles; not percent across all tasks.

Instructional Format

Unsurprisingly, 87% of participants cited the instructional format as a source of ease or difficulty. The responses here were consistent with the overall perception that tutorials are easier as we have noted elsewhere. 77% of participants identified tutorials as easy or difficult because of the format while only 43% said the same for puzzles. We note however, that 74% of these responses for tutorials are cited for being easy, whereas the responses for puzzles are spread across easy and difficult.

Prior Experience & Barriers

For many users, content familiarity made tasks easier, and a lack of it created the potential for barriers. If a participant had prior experience with a programming construct, they tended to perceive it as easier, “It was much easier than I thought it was, just using the stuff that I had already learned.” In total, 60% of participants cited that their familiarity with a programming construct made that task easier for them. Participants’ experiences of difficulty based on their background were consistent with their background-based expectations of difficulty. In contrast, 93% of participants cited a conceptual barrier, or not understanding a programming construct, as a source of difficulty in the tasks. One participant said that “Not knowing that it was actually possible to put *do in order* inside with *do together*” made a task difficult. Conceptual barriers are also strongly associated with puzzles. 87% of participants stated that a conceptual barrier caused them difficulty in puzzles, whereas only 23% of participants said the same for tutorials.

We also saw a translation barrier: connecting a programming statement with its output, as a source of difficulty for participants working on puzzles. 13% of participants stated that the translation barrier was a source of difficulty for them.

Mechanics

30% of participants cited the instructional format’s mechanics as a cause of difficulty. In the tutorial, participants found watching and following the video difficult. Whereas in the puzzle, participants found watching the correct output difficult. Participants also found the programming environment’s mechanics difficult. In total, 57% of participants cited the mechanics of the programming environment as a source of difficulty.

Degree of Struggle

Lastly, participants cited their own perceptions as a source of ease and difficulty. These perceptions were often very self-referential in nature: in essence, participants seemed to express “It was hard because I found it hard.” Some participants talked about “having to do it over and over and over and over again.” Others referenced the amount of time necessary to complete the task: “It was hard and long. It took me a really long time to do it.” In 18% of puzzles and 13% of tutorials, participants cited length as a source of ease or difficulty. Using a multilevel model, we noted that participants spent significantly longer working on puzzles ($M = 8.2$, $SD = 6.6$ minutes) than tutorials ($M = 7.9$, $SD = 4.6$ minutes), $\chi^2(27) = 67.39$, $p < .001$. Overall, 83% of participants explained their perception of difficulty through their own experience at least once during the study.

5.4.4 Experience Outcomes

Our final category, experience outcomes, explores the values that participants perceived while completing instructional tasks (see Table 5.6). Participants received enjoyment from completing the tasks and also improved their programming skills. However, in many tasks

Table 5.6: Experience outcomes response themes.

	% of Tasks (Tutorials, Puzzles) [†]	% of Participants (Females, Males)
Enjoyment	80% (79%, 80%)	97% (100%, 94%)
Liked Animation	27% (27%, 27%)	63% (64%, 63%)
Working on Task	26% (32%, 22%)	73% (71%, 75%)
Finishing Task	50% (45%, 53%)	90% (86%, 94%)
Improve Programming Skills	62% (61%, 63%)	97% (93%,100%)
Learned Skill	44% (50%, 40%)	87% (71%,100%)
Improved Competency	22% (23%, 21%)	63% (64%, 63%)
Practice	34% (32%, 35%)	87% (86%, 88%)
No Benefit	66% (71%, 64%)	97% (93%,100%)
Other	92% (100%, 87%)	100% (100%,100%)
No Reason / Neutral	87% (97%, 81%)	100% (100%,100%)
Other	35% (44%, 31%)	87% (86%, 88%)

[†] Percent of sublabels for all tutorials/puzzles; not percent across all tasks.

participants were unable to state what outcome or benefit they received from completing the task.

Enjoyment

In the decision rationales section (5.4.1), we noted that participants often made decisions based on their preferences. When we asked participants in the post-task interviews what they got out of a task, one of the most common responses was that they simply enjoyed it. 63% of participants enjoyed the animation content, 73% enjoyed working on the task, and 90% enjoyed completing the task. 83% and 53% of participants stated that they enjoyed puzzles and tutorials, respectively. The most interesting responses described struggling to complete the task and taking strong satisfaction from that success. As one participant said, “The more you struggle at something the more exciting it is when you finish it.” Echoing the other results, satisfaction from completing a task is a theme we saw more strongly with puzzles than tutorials. One possible explanation for the enjoyment disparity between puzzles and tutorials is that participants saw value in the freedom to solve the problem: “The best part was having to figure everything out.” As shown in Figure 5.6, our TEQ results suggest that participants did perceived more choice in the puzzles ($M = 6.4$, $SD = .7$) compared to the tutorials ($M = 6.0$, $SD = .9$), $\chi^2(1) = 6.48$, $p = .011$.

Improve Programming Skills

Participants stated improving their programming skills as an outcome. We note that this may be due in part to our priming to prioritize learning. Participants often distinguished between three concepts: learning, general competency, and practice.

When participants talked about learning, they focused on their introduction to a new concept or programming element. Overall, 87% of participants stated they learned a specific

programming skill from completing a task. 70% of participants reported they learned new programming skills from the puzzles, compared to 60% of participants who reported the same for tutorials. This contrasts with participants' stated decisions around learning, where we found that they tended to select tutorials when citing a learning goal (see the improve programming skills section in 5.4.1). 63% of participants reported improved computing competency. 87% of participants also reported finding value in practicing using the instructional tasks. As one participant stated, "I enjoyed it. It was fun to practice on my own with the skills that I had just learned." 70% of participants reported that the puzzles helped them reinforce their skills, compared to just 50% of participants who said the same thing about tutorials.

We compared the reported programming experience from the pre-study and post-study interviews: 80% of participants felt more competent after completing the study, 17% reported no change, and a single participant reported feeling less competent.

No Benefit

In 66% of tasks participants responded to some interview questions that they did not benefit from the task. In most cases, this was in response to a question about what specific programming knowledge did they learn from the task. Recall that in only 44% of tasks was learning a specific skill cited. Frequently, participants did not know what they had just learned. Many recognized that the task was useful (i.e. practice) but could not name a specific programming construct that they had learned.

5.5 Threats to Validity

In this study we used a semi-structured interview that we specifically geared towards learning programming. We also primed participants to improve their programming skills. Our specific focus on learning may skew our results away from other factors that also affect decisions and perceptions of value. However, even when we explicitly encouraged participants to prioritize learning, participants were more likely to make their decisions based on their preferences and desire for challenge.

This study sought to identify the perceptions of value that novice programmers see in two specific instructional formats. While the specifics of these perceptions may not hold to other formats, our general conclusions about the factors that participants feel are important, like challenge, will likely hold for other instructional formats. However, outside of middle school children, these results would likely differ for other demographics, like expert programmers. We also note that participants with a stronger interest in programming may have self-selected to participate in this study.

5.6 Discussion

Below we discuss how participants' responses and behavior lend further support for code puzzle completion problems as a motivating and independent learning resource.

Choice & Learning Prior work has found that giving learners full control over their instructional task choices can lead to both positive and negative outcomes [92, 202]. Overall, we believe the participants in this study made reasonable choices that helped improve their programming knowledge. This suggests that for independent contexts, our approach of letting

participants opt to complete code puzzle completion problems alongside working towards their own goals, will likely benefit learners.

While participants probably benefited from their choices, their selection of instructional tasks and formats was suboptimal for maximum learning gains. Participants often made choices that were motivated by enjoyment rather than learning. When participants did make decisions about improving their programming skills, they often chose tutorials, as they perceived tutorials as more supportive of learning. Yet after completing a code puzzle, participants frequently cited them as more useful for improving their programming skills. The mixed responses from participants suggest that novices lack the expertise needed to efficiently self-regulate their learning choices [204].

Fortunately, self-regulation is itself an important skill for independent learners to acquire [42]. Through the process of contrasting and leveraging tutorials and puzzles throughout the study, participants may have indirectly improved their self-regulating ability for future decisions. During the evaluation, a majority of participants made choices based on trying to discover which tasks and instructional formats were appropriate for their skill level. This suggests that participants were attempting to self-regulate for tasks that align with their current expertise. Over time, learners' ability to self-regulate between tutorials and puzzles will likely improve, thus further supporting that novices have the capacity to make effective use of code puzzle completion problems in independent contexts.

Motivation & Challenge In our first study comparing tutorials and code puzzles (Chapter 3) we found no differences in motivation between the two instructional formats. From this evaluation, we found that participants are frequently motivated by things that they enjoy. Not only did participants enjoy the animations of the instructional tasks, but they had fun completing the puzzles. Further, participants opted to complete more code puzzles than

tutorials throughout this evaluation which lends further evidence to support that code puzzle completion problems are motivating to independent learners.

Beyond simple enjoyment, participants often stated that it was *challenge* that motivated them. In fact, almost every participant made a decision during the study where they were seeking challenge. Reassuringly, participants found the code puzzles to be more challenging than the tutorials.

Lastly, the extra motivation provided by challenge in the code puzzles may explain the difference in task completion time between tutorials and puzzles in the study. In our first study, we found that puzzles took significantly less time to complete than tutorials. We suspect that the opposite result in this study is due to participants purposefully choosing puzzles which would challenge them. Challenging puzzles likely took longer to complete, thus possibility reducing learning efficiency. However, it is not clear how self-imposed challenge affects learners' cognitive load. It is possible that self-imposed challenge in code puzzle completion problems may increase germane cognitive load thereby increasing programming knowledge while simultaneously motivating learners.

5.7 Conclusion

In this chapter, we presented a qualitative study into validating Hypothesis III by investigating which instructional formats users preferred and why they chose to use them. From this evaluation we learned that independent learners' care about more than just an enjoyable activity. We discovered that enjoyment, challenge, and perceived value all play important roles in a learner's decision to choose between tutorials and code puzzles. The results of this study suggest that code puzzle completion problems are viewed as a useful resources by independent learners that also complement and support their goals. But, a key feature of

this support is that it is balanced against other informal learning resources, like tutorials. In the next chapter, I discuss what role code puzzle completion problems may have in the future and how they might further support independent learners.

Chapter 6

Summary & Future Work

I started this dissertation with the goal of developing a motivating way for middle school children to learn programming while working towards their own goals. To achieve this goal, we created code puzzle completion problems. Through two formative evaluations we worked to bring together the motivating aspects of code puzzles and storytelling with an effective approach for learning independently: completion problems (see Chapter 2). We also developed an introductory puzzle curriculum to introduce basic programming constructs like iteration and simple parallelism. Through three additional summative evaluations we investigated my hypotheses (see Section 1.3.2) that code puzzle completion problems can support novices in using programming constructs in similar situations on their own, and that they will likely be motivated to do so.

In our first summative evaluation (Chapter 3), I reported on a study to investigate Hypothesis I: whether novices using code puzzle completion problems demonstrate evidence of learning new programming constructs independently. We found that when novices trained with code puzzle completion problems they performed 33% better on transfer tasks compared

to training with tutorials while also taking 21% less time to complete the training tasks. This evaluation demonstrated that code puzzle completion problems increase the learning efficiency and effectiveness for novices learning to apply basic programming constructs on their own. This result supports Hypothesis I, thus validating our approach of using code puzzles and completion problems in support of learning programming independently.

In further support of improving the learning potential of code puzzle completion problems, we also investigated whether errors may support or hinder learning in these puzzles. (see Chapter 4). Current research into leveraging errors in learning provides conflicting evidence between whether errors benefit or harm learners. In Hypothesis II, we expected distractors to increase novices' time completing puzzles while also decreasing their ability to apply new programming constructs in transfer tasks. Through our second summative evaluation, we discovered that distractors do in fact increase the time to complete a code puzzle by 14%, but do not decrease performance on near transfer tasks. Instead, distractor participants demonstrated no performance differences on the transfer tasks compared to participants who trained without distractors. Given this result, I partially accept Hypothesis II; distractors do decrease learning efficiency, but do not reduce learning effectiveness in code puzzle completion problems.

Lastly in Chapter 5, I investigated whether middle school children perceived value in code puzzle completion problems when given the freedom to select their own instructional format: tutorials or puzzles. In this evaluation, we investigated Hypothesis III: novices will perceive more interest and enjoyment, but also opt to use code puzzle completion problems more than tutorials. Through this qualitative study we interviewed and analyzed participants about their decisions to use the tutorials or puzzles. In our study participants exhibited more preference for using code puzzles over tutorials. Our results also demonstrate that middle school children perceived code puzzles as more motivating and enjoyable while also useful for

helping them improve their programming skills. Further, we found that while participants did not find tutorials as enjoyable as puzzles, they did find them as also valuable for improving their programming skills. The results validate Hypothesis III that code puzzle completion problems are in fact motivating and that when given a choice, participants may opt to use them on their own in informal contexts.

The results of these evaluations suggest that code puzzle completion problems can be an effective approach for motivating middle school children to learn programming constructs in informal settings. They also suggest that code puzzle completion problems are also an effective way to help novices use programming constructs from the code puzzles in similar circumstances; thereby aiding independent learners as they work towards their own goals. While this dissertation does demonstrate that code puzzle completion problems are a viable approach, there is still a lot of future work to do to fully realize the goals I outlined in Chapter 1. In this section, I discuss some of the potential future opportunities for improving the effectiveness of code puzzle completion problems, extending their use beyond the introductory programming constructs, and further validating their use to support goal-oriented independent learners.

6.1 Improving the Efficiency & Effectiveness of Code Puzzle Completion Problems

Based on prior work in cognitive load theory there are several additional methods we can employ to improve both the efficiency and effectiveness of code puzzle completion problems. In this section, I share my thoughts for future work on improving the ability for learners to acquire new programming knowledge from these code puzzles. Specifically, I discuss the

possibility of using worked examples, promoting self-explanation, and using errors to improve the future utility of code puzzle completion problems.

6.1.1 Using Worked Examples

Studying worked examples is known to be an effective method for fostering schema development and learning [185]. This is especially true for the early phases of learning when intrinsic load is high due to the absence of schemas [141]. The act of studying worked examples fosters germane cognitive load and therefore contributes to schema development [141]. However, as I discussed in Chapter 1, worked examples are often ill suited to complex domains like computer programming. This is unfortunate because pairing completion problems with worked examples is known to be a highly effective method for facilitating schema development [5, 138]. We can likely improve the learning efficiency and effectiveness of code puzzles completion problems by integrating the completion problem approach with worked examples.

Worked Examples + Completion Problems

Using both the worked example and completion problems effects is known to be an effective approach for facilitating learning [22, 124, 185, 190]. The idea behind this approach is that learners will not study worked examples unless they have a problem to solve [22, 185, 190]. By giving learners completion problems with paired worked examples, learners pay more attention to the worked examples which further helps their performance on completing problems [124, 190].

In fact, I originally planned on using both completion problems with paired worked examples when I started this work. Before my formative evaluations in Chapter 2, I conducted a separate formative evaluation with Michelle Ichinco to try and develop worked examples for

programming [80]. Due to the complex nature of programming, it was not clear to us what a worked example should look like for programming. One approach is to add comments to the code examples to describe what the purpose of each code segment [135]. At the time, Morrison, Margulieux, and Guzdial’s approach of adding subgoal labels to code examples was not yet published [135]. So we decided to try out different ways of annotating code examples in order to produce programming worked examples.

In our formative evaluation we developed three annotation styles which we later tested in a summative evaluation [80]. I then tested using these annotated worked examples with the completion problems in my first formative evaluation for the code puzzle completion problems (see Section 2.2). I programmed the code puzzles so that an example would automatically open up in a separate window. I also added an *example* button to the toolbar of the code puzzle workspace. During testing I found that users did not use the example and often closed the example window. Due to the difficulty of getting participants to use the worked example, I abandoned this approach. Moving forward, exploring ways to use worked examples with code puzzle completion problems is an important next step for improving the effectiveness of our approach.

Programming Worked Examples

Additionally, future work is required to author examples for programming that are actually *worked* examples. I now believe that the annotated code examples used in prior studies [80, 135] are not actually worked examples because they fail to demonstrate the key steps used to arrive at the final code snippet. While comments or annotations are useful for understanding the code snippet, they fail to show how a programmer might go about creating that snippet of code. A useful starting point may be to use Nasehi et al.’s observations of step-by-step examples from StackOverflow [137].

I suggest conducting an exploratory study that looks at the iterative process that experts use when coding a problem. By looking at experts' approach to coding, we can gain insights into the key steps they use to author code. For example, when I am coding I often break my bigger problem into many incremental steps that build upon each other. If I needed to search a bunch of files for string, I might first write the code to open a file and then test that I can see all of the file's contents. Once I know that I am correctly reading this file, I would write the code to iterate over all the characters to locate my string. After testing that my search algorithm works, I would then write the code to process many files with the search algorithm. While the exact results of this exploratory study will be different, expert programmers likely do not write code using top-down approach, which is what the 'worked' examples above assume.

I suspect that worked examples for programming need to be broken down into incremental steps. One approach might be to use annotations in the code like: step 1) write and test file reading, 2) write and test searching file contents, and 3) write and test code to read in all files and search. An alternative approach maybe to annotate the worked example with the different incremental states using a code diff visualization showing what the programmer worked on first in one color and the remaining steps in other colors. Whatever the outcome of this exploratory study, knowing what constitutes a worked example for programming is key to using worked examples with the code puzzle completion problems in the future.

Faded Worked Examples

As discussed above, the addition of worked examples to the code puzzle completion problems will likely further facilitate learning and schema construction. Using faded worked examples with code puzzle completion problems can likely improve this situation even more. Faded worked examples gradually over time, leave steps of the worked example incomplete [5, 159,

182]. Because the example is incomplete, learners are forced to generate self-explanations about the missing steps in the worked example to solve the completion problem [5, 160]. Gray et al. provide some initial suggestions on how to fade code examples [59]. However, the authors provide no evidence that their fading suggestions are an effective approach [59]. Future investigations should study how to fade code examples effectively and whether faded worked examples when used in conjunction with code puzzle completion problems further promote self-explanation in independent learners.

Example Variability

Lastly, example variability may also improve the learning efficiency and effectiveness of code puzzle completion problems. Worked examples that vary in their surface and structural features compared to the completion problems are known to promote deep learning [144, 181, 184]. Examples with low variability compared to the completion problems have small surface level feature differences. For example, for a completion problem and worked example that both have a loop, a worked example with only a different loop index is a surface level difference. Low variability examples are useful for promoting learning for near transfer tasks [181]. However, worked examples with high variability differ in terms of structural changes in content and format, and promote learning for far transfer tasks. A structural level change in a worked example compared to a completion problem might be a completion problem that uses a loop, but an example that has a loop nested inside of a conditional statement. A key advantage of using high variability examples is that they are associated with better learning gains [144, 153].

For this dissertation, I only sought to improve near transfer for novice programmers as a starting point for validating the code puzzle completion problem approach. Given that my target audience is goal-oriented learners, learning how to apply programming constructs in

far transfer situations may further encourage informal users to seek out even more ambiguous projects on their own. Future investigation is necessary to author high variability worked examples for code puzzle completion problems to further increase their utility for independent learners, especially in far transfer situations.

6.1.2 Promoting Self-Explanation

As mentioned in the introduction (Chapter 1), self-explanation is essential to the construction of schema [22, 182]. One way to promote self-explanation in learners is to provide them with active, rather than passive instructional tasks [5, 159]. Prior work for completion problems suggests that the active process of completing a problem promotes the process of self-explanation [124, 182].

Unfortunately, we have no hard evidence to suggest that code puzzle completion problems promote self-explanation in novice programmers. However, many of the responses given during the study in Chapter 5 suggest that self-explanation likely occurred. As one participant stated,

“I mean, like I could use a *repeat* block but I didn’t really understand exactly what it was going to do until I tinkered with it a bunch. [...] It’s just a lot easier because you can repeat things in a specific order, so it’ll repeat two times the dolphin bump thing, and then the fishing boat will shake, and then it’ll move down point one meters, and then repeat. And so it just makes more sense that way.”

Or as another participant stated,

“I put the plane swerves past the balloons and the balloon one, two, and three all in the *do together* box and I never used the *do in order* box. I was supposed to put the *do in order* box and put the balloons in there and then put that in the *do together* box and also in the *do together* box I was supposed to put the planes swerves [past the] balloons.”

Both of these responses highlight how the problems that these participants faced while working on the code puzzles helped them better understand how the programming constructs functioned.

Moving forward, one way to investigate whether code puzzle completion problems promote self-explanation is to ask participants to use the think-aloud protocol while working on the puzzles. Additionally, exploring ways to further promote self-explanation in code puzzle completion problems may also improve both their efficiency and effectiveness.

Eliciting Purposeful Self-Explanation

To promote self-explanation in instructional tasks, prior studies asked learners to talk aloud in order to promote self-explanation [158]. Yet, this is unlikely to work in informal settings because many users may opt out of this voluntary procedure. Instead of asking learners to talk aloud, we could ask them to type an explanation of a code puzzle instead. However, this too is unlikely to work because prior work found that typing self-explanations was not particularly effective [173]. Alternatively, researchers successfully promoted self-explanation using a multiple choice question for selecting the main principle of an instructional task [5]. Additional investigation is necessary to see if these existing approaches can effectively promote self-explanation in code puzzles or whether new approaches are necessary.

As a start for this investigation, exploring how to integrate the main principle multiple choice questions into code puzzles is one promising approach. Additionally, because subgoal labels [134] and annotations [80] are effective in code examples, they may also provide an avenue for promoting self-explanation in code puzzles. One approach would be to create puzzle comment blocks that are subgoal labels or annotations. Learners would then place the comment tiles into the correct location of the puzzle’s solution. Through the active process of completing the puzzle first and through the secondary process of placing the code comments in the appropriate locations, learners may engage in further self-explanation. Alternative approaches are also likely possible, but this is an area of future work.

6.1.3 Errors for Experienced Learners

As novices complement their project oriented work with code puzzle completion problems, they are likely to become more experienced programmers with better developed schemas. Unfortunately, the educational strategies that work for less experienced learners can often backfire for more experienced learners. When the additional scaffolding and extra material necessary to reduce novices’ cognitive load becomes redundant to more experienced learners due to their mature schemas, the redundant information will impose extraneous cognitive load [88]. This is known as the *expertise reversal effect* [88]. This suggests that the extra scaffolding provided by code puzzle completion problems may actually harm more experienced learners. Changing the format of the code puzzle completion problems by introducing errors or distractors may help mitigate this effect.

In Chapter 4, we found that errors did not aid in acquiring programming knowledge for novices and in fact they increased extraneous cognitive load. Similar to our result, Große and Renkl demonstrated that errors in worked examples benefit high performing students, while negatively impacting lower performing ones [63]. It is likely that the errors in the

worked examples promoted germane cognitive load for experienced learners, while the errors produced extraneous cognitive load for novices. This suggests that errors in code puzzle completion problems may reduce the expertise reversal effect and may actually be beneficial to more experienced programmers. I leave the investigation of the effects of distractors in code puzzle completion problems on programmer expertise as future work.

6.2 Extending Code Puzzle Completion Problems

The code puzzle completion problems that we developed in this work are limited in their breadth of computing concepts and in their context. The introductory curriculum only exposes novices to elementary repetition, parallelism, and sequential execution in a 3D animated storytelling context. Further, the code puzzle completion problems we developed are only based on blocks (i.e. lines) of code. There are many potential avenues to extending code puzzle completion problems beyond this initial approach. In this section, I share my thoughts on how code puzzle completion problems can be extended into other programming contexts, how we can extend the curriculum to better support independent learners, and how alternative completion strategies may support the expanded curriculum.

6.2.1 Alternative Contexts

Our approach to code puzzle completion problems depends heavily on 3D animated program output. We specifically developed code puzzles that produced animations where learners can recognize any programming construct behavior. We further supported learners, by adding live feedback that complements the executing animation. In other contexts, like traditional text based output, our scaffolding for feedback may not be sufficient. However, it is likely that

many of the lessons we learned by providing incremental, ambiguous, and grouped feedback can be adapted to other programming contexts.

Outside of 3D animations, I believe that code puzzle completion problems will likely work well in many contexts that produce graphical output. Text based approaches like Parsons problems [48, 148], may be difficult to develop into code puzzles, given that the context may not be as compelling. Many of these text based problems currently resemble uninspiring programming homework problems, like sorting numbers or fixing out-of-bound arrays, rather than producing rewarding output. However, other contexts like event-driven user interface programming, statistical modeling and graphing, responsive web programming or even gaming can likely be adapted into code puzzles that goal-oriented learners will find motivating. Future exploration is necessary to determine how code puzzle completion problems can benefit independent learners in other motivating contexts.

6.2.2 Curricular Content

There is much more to programming than the three programming constructs and their nested variants we used in our evaluations. Future work is necessary to extend the code puzzle curriculum to cover the greater depth of programming concepts and skills used by experienced programmers.

Conditional Statements & Expressions While repetition and simple parallelism are important for animation, few interesting programs can be written for other contexts without conditional logic. Conditional statements and expressions should be added to the code puzzle completion problems curriculum. However, conditional statements and expressions are distinctly different skills that may each require unique treatments or approaches in the

code puzzles. Future work must investigate how to create puzzles that both demonstrate conditional execution while also helping develop learners' schema for how expressions function.

Organization & Abstraction Likewise, very few complex programs can be written without methods. Methods are critical for reusing functionality and organizing programs. Code puzzle completion problems should support novices in using and authoring methods. An initial approach may be to have users place statements in both the main and in a separate method.

Additionally, objects and abstraction can further help organize programs and reduce program maintenance. Future work might investigate how to promote these ideas by having users place statements in abstract methods or add properties to objects. For example, in an animation with three dancers, learners would reassemble the abstract dance animation to make two dance, while the third dancer has a overridden method that the learner also reassembles.

Programming Syntax & Skills Blocks based programming is useful for novices because it likely reduces extraneous cognitive load when learning to program. Eventually, goal-oriented learners may desire to learn how to write programs using syntax. Code puzzle completion problems should be able to support this given that all known Parsons problem implementations are syntax based. However, future work is necessary to explore how to transition learners from blocks to syntax and how to adapt the puzzles or mechanics to effectively support program syntax.

Additionally, programmers often use many skills when writing code that would benefit goal-oriented learners, like refactoring. Code puzzle completion problems may be able to be adapted to support the development of refactoring skills. Programmers also often incrementally write and test code. During our evaluations we observed that many users do

in fact use this approach when solving puzzles. A future evaluation should study whether this incremental write-a-bit, test-a-bit approach is transferred to users' own projects. An evaluation may provide insight into the other types of programming skills learners develop beyond programming constructs from using code puzzle completion problems.

The curriculum should also be modified to include more advanced programming concepts like recursion. Especially because recursion is often completely misunderstood by inexperienced programmers. In fact, novice programmers in Looking Glass accidentally created recursive function calls frequently enough that we added a warning to prevent users from making recursive calls. Code puzzle completion problems can likely be adapted to demonstrate recursion, given that many of recursive applications are graphical in nature. However, this is an area for future exploration.

Programming Paradigms In this study we exposed novices to programming constructs in the imperative programming paradigm. Other paradigms, like event-based programming will likely also benefit goal-oriented learners. Goal-oriented learners may desire to create games or apps, which both require an understanding of event-based programming. Functional programming may also be desirable given an appropriate context, like statistical modeling and graphing. Future work might investigate how to use other paradigms in code puzzle completion problems. For example, having novices reconstruct user interface events, like button clicks.

6.2.3 Alternative Completion Approaches

Adding the new curricular content discussed above likely involves authoring new code puzzles, but also adapting the puzzle interface and mechanics. In this work, we used a blocks-based approach as our completion strategy; learners reassemble just lines of code. Future work

should investigate other mechanics for changing the completion strategy of code puzzle completion problems, especially to support new curricular content.

For example, in our code puzzle completion problems we asked learners to reassemble the entire animation. For introducing a new programming concept like conditional execution, it may be desirable to fix or lock all statements in the puzzle solution except for the lines inside of the conditional statement. Learners would then place just the statements for the *if* and *else* blocks into the solution. This may help highlight to novices that there are two executing paths in the program without having to impose additional cognitive load for reassembling the entire program. Likewise, refactoring skills might be encouraged by locking all statements in the solution and having novices drag and drop constants to replace literals in the code. Additional completion approaches may simply ask learners to reassemble just an expression while learning conditional logic.

There may also be a nice opportunity to gradually transition from blocks to syntax using the completion strategy. One approach may fix all the statements in the program, but leave out the brackets. Learners would then just insert the brackets to recreate the familiar “blocks” structure. Other transitional approaches for using syntax for just a part of a code puzzle, like using syntax for method calls, parameters, or loop indices, may also be worthwhile to explore.

6.3 Supporting Goal-Oriented Learners

Lastly, improving code puzzle completion problems to better support goal-oriented learners is another area for future work. Supporting goal-oriented learners is especially important because these users are likely to only engage in activities that further their pursuit of their own goals. In the future, we should continue to explore the use of choice in supporting their

goals, but also learn how to better support self-directing learning and self-assessment for these learners.

6.3.1 Choices for Instructional Formats

In the evaluation in Chapter 5, participants choose to work on more puzzles than tutorials. The reasons they gave for choosing the puzzles were often enjoyment, seeking challenge, or improving their programming skills. While participants did choose to use the puzzles in a controlled study, learners may make different decisions about how to spend their time when they are working towards their own goals during their own time. The reasons behind their decisions may also be different than the ones we observed in this study. Conducting an in-the-wild study, would help shed light into the decisions children make while working on their own projects: do they use code puzzle completion problems? when do they use them? and why do they choose to use them?

We also discovered in the study in Chapter 5 that participants see value in both tutorial and puzzle instructional formats. This suggests that there is not a one-size-fits-all approach to supporting independent learners. Instead, code puzzle completion problems should be just one of several independent learning resources available to goal-oriented learners. Future work should investigate new types of independent resources for learning programming as well as exploring how best to have several instructional formats work together to support learners. Novice programming environments like Looking Glass, might provide multiple formats with a range of scaffolding that enables users to obtain adequate support while pursuing their interests. These formats might support users in learning interface mechanics, like tutorials. Or instead, they might help users develop their near transfer skills like code puzzle completion problems. Other alternative instructional formats might include remixing code [62, 71] or providing hints or suggestions while users work on their own projects [79, 149].

New approaches could also be developed that actively integrate with the learner’s goals, rather than hope that the learner passively opts to complete a tutorial or puzzle. For example, suggestions could be given to novices about how to improve their animation in Looking Glass [79]. However, in order to utilize the suggestion, a code puzzle completion problem is generated from the user’s current project with suggested adaptation in the project. In this way, the learner is benefiting from using code puzzle completion problems, but is still actively working towards their goal at the same time. Future work is needed to understand how the inclusion of different instructional formats, like the one proposed above, might empower users to improve their skills while making decisions based on their own goals.

Lastly, a majority of participants made a decision based on trying to discover which tasks were appropriate for their skill level in the study from Chapter 5. Because this rationale appears to be both important for users and commonplace, programming environment designers should consider ways that enable participants to explore their abilities. We observed that in several MOOCs and puzzle-like systems, users are not allowed to progress until they have finished the current lesson or level. Giving users control to make their own decisions may better help them meet their goals. But this does not come without risks. Users may feel discouraged after failing to complete a task. However, as we observed in this study, many of the participants who failed to complete a puzzle were driven to discover their mistake by using another instructional format or by selecting an easier task.

6.3.2 Self-Directed Learning

Goal-oriented learners are responsible for directing their own learning. The design of code puzzle completion problems assumes that learners will make their own choices about which puzzles they complete and when to do so. Choosing learners’ tasks for them, or sharing task

selection with them, are all possible alternatives to aid participants in better directing their own learning.

In terms of choosing tasks for learners, systems could force learners to complete all tasks in a predetermined order, like MOOCs, or personalize the order of the tasks to meet the needs of that learner. Researchers have found that adapting or personalizing tasks, based on input from the learner, leads to more efficient training and better transfer performance compared to a predetermined sequence [34]. Unfortunately, predetermined sequences are not likely to work well for goal-oriented learners due to the disconnect between their goals and the current content. Personalization, may adapt better to the learners' goals, however learners may find the lack of choice as to how they spend their time less motivating [34].

Giving users control seems like the most promising approach for working with self-directed, goal-oriented learners on code puzzle completion problems. Not only is control motivating [34] for learners, but the effectiveness of task selection is also directly tied to whether learners recognize they have control of their tasks [171]. Unfortunately, evidence suggests that having learners select their own tasks can sometimes be ineffective [92, 202]. One solution is to give self-directed learners advice about tasks selection [187]. The code puzzle completion problem task selection could then be modified to advise users on which puzzles might benefit them the most given their current skill level.

However, other research demonstrates that personalizing tasks to the learner and sharing control for final task selection is both highly motivating and very effective [33, 35, 172]. For code puzzle completion problems, this would mean instead of giving learners access to all 14 puzzles at once, the system decides on a smaller set of puzzles that are appropriate for the current learner, and then lets the user pick the among the appropriate tasks. Ultimately, given that prior research demonstrated that shared control led to higher learning outcomes and

more invested effort into learning [36], code puzzle completion problems should be adapted to utilize this approach. Future work is necessary to best realize how to personalize the code puzzles to the expertise of the current user and then enabling users to select between those puzzles. With control still in the learner’s hands, this approach may still align with our goal-oriented learners. However, any exploration into using personalized and shared controlled of code puzzle completion problems must carefully examine how this may additionally effect goal-oriented learners’ motivations.

6.3.3 Self-Assessment & Scaffolding

Self-directed, goal-oriented learners, need to be able to assess their performance in order to continue working to improve their skills [125]. In self-directed learning, learners must first perform a task, assess their performance on that task, and lastly select their next task [125]. The evidence from the study in Chapter 5 suggests that learners do reflect and self assess their abilities. The act of finishing a code puzzle completion problem or using the incremental feedback indicator seemed to facilitate some self-assessment. Participants would often scale back to easier puzzles, especially if they had difficulty completing harder ones.

Additional feedback will likely better help novices assess their current performance in code puzzle completion problems. Future work should look into ways to support users when they become stuck or fail to complete a puzzle. If the puzzle interface provides additional scaffolding for when they get stuck, novices may be able to use the extra help as an additional self-assessment opportunity. Further, the extra scaffolding may prevent feeling deflating after failing a task.

Possible avenues for future scaffolding may be to provide hints for solving the puzzles. One approach may be to use annotations in the animations to highlight a key principle. Lin

et al. found that highlighting important pieces of an animation positively impacted learning, cognitive load, and intrinsic motivation [114]. The code puzzle completion problem feedback window could be modified to provide these annotations once users fail to make forward progress on their puzzle. Similarly, the correct statements in the puzzle could be frozen in place to reduce the learner from carelessly swapping statements, if they cannot figure out what statement is causing the incorrect behavior in their output. Other avenues may be to simply provide the solution to a puzzle if a user decides they cannot complete it. The solution to the puzzle then becomes a worked example. From studying the worked example, the learner can reflect on their misconceptions, which may help them realize and assess their current skill level. There are other potential opportunities for providing scaffolding to encourage self-assessment, however this is an area I leave for future work.

In this dissertation, I have reported on the development of code puzzle completion problems and the evaluations into their effectiveness as a tool for goal-oriented users to learn programming on their own. This work demonstrates that code puzzle completion problems are an alternative approach in supporting these independent learners. Middle school children successfully completed more transfer tasks when training using the code puzzles compared to training with tutorials. Participants also found the code puzzles to be more enjoyable and motivating, and were more likely to opt to use puzzles over tutorials. However, I have only demonstrated the early potential for this approach. Future work is absolutely necessary to refine code puzzle completion problems to fully support middle school children who desire to learn programming, but lack the resources at their school to do so.

References

- [1] J. R Anderson and B. J Reiser. “The LISP Tutor”. In: *Byte* 10.4 (1985), pp. 159–175.
- [2] J. R Anderson and E. Skwarecki. “The Automated Tutoring of Introductory Computer Programming”. In: *Commun. ACM* 29.9 (Sept. 1986), pp. 842–849. DOI: 10.1145/6592.6593.
- [3] John R. Anderson, Frederick G. Conrad, and Albert T. Corbett. “Skill Acquisition and the LISP Tutor”. In: *Cognitive Science* 13.4 (Oct. 1989), pp. 467–505. DOI: 10.1016/0364-0213(89)90021-9.
- [4] Robert K. Atkinson, Sharon J. Derry, Alexander Renkl, and Donald Wortham. “Learning from Examples: Instructional Principles from the Worked Examples Research”. In: *Review of Educational Research* 70.2 (Jan. 6, 2000), pp. 181–214. DOI: 10.3102/00346543070002181.
- [5] Robert K. Atkinson, Alexander Renkl, and Mary Margaret Merrill. “Transitioning From Studying Examples to Solving Problems: Effects of Self-Explanation Prompts and Fading Worked-Out Steps”. In: *Journal of Educational Psychology* 95.4 (2003), pp. 774–783. DOI: 10.1037/0022-0663.95.4.774.
- [6] Klara Benda, Amy Bruckman, and Mark Guzdial. “When Life and Learning Do Not Fit: Challenges of Workload and Communication in Introductory Computer Science Online”. In: *Trans. Comput. Educ.* 12.4 (Nov. 2012), 15:1–15:38. DOI: 10.1145/2382564.2382567.
- [7] Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. “DocWizards: A System for Authoring Follow-Me Documentation Wizards”. In: *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*. UIST ’05. New York, NY, USA: ACM, 2005, pp. 191–200. DOI: 10.1145/1095034.1095067.
- [8] *Blockly Games*. URL: <https://blockly-games.appspot.com/>.
- [9] Jonas Boustedt, Anna Eckerdal, Robert McCartney, Kate Sanders, Lynda Thomas, and Carol Zander. “Students’ Perceptions of the Differences Between Formal and Informal Learning”. In: *Proceedings of the Seventh International Workshop on Computing Education Research*. ICER ’11. New York, NY, USA: ACM, 2011, pp. 61–68. DOI: 10.1145/2016911.2016926.

- [10] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. “Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '09. New York, NY, USA: ACM, 2009, pp. 1589–1598. DOI: 10.1145/1518701.1518944.
- [11] Emma K. Bridger and Axel Mecklinger. “Errorful and Errorless Learning: The Impact of Cue–target Constraint in Learning from Errors”. In: *Memory & Cognition* 42.6 (Mar. 11, 2014), pp. 898–911. DOI: 10.3758/s13421-014-0408-z.
- [12] P.L. Brusilovsky. “Intelligent Tutor, Environment and Manual for Introductory Programming”. In: *Innovations in Education & Training International* 29.1 (1992), pp. 26–34. DOI: 10.1080/0954730920290104.
- [13] Philip Sheridan Buffum, Eleni V. Lobene, Megan Hardy Frankosky, Kristy Elizabeth Boyer, Eric N. Wiebe, and James C. Lester. “A Practical Guide to Developing and Validating Computer Science Knowledge Assessments with Application to Middle School”. In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. New York, NY, USA: ACM, 2015, pp. 622–627. DOI: 10.1145/2676723.2677295.
- [14] Margaret Burnett, Scott D. Fleming, Shamsi Iqbal, Gina Venolia, Vidya Rajaram, Umer Farooq, Valentina Grigoreanu, and Mary Czerwinski. “Gender Differences and Programming Environments: Across Programming Populations”. In: *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '10. New York, NY, USA: ACM, 2010, 28:1–28:10. DOI: 10.1145/1852786.1852824.
- [15] Andrew C. Butler, Jeffrey D. Karpicke, and Henry L. Roediger III. “The Effect of Type and Timing of Feedback on Learning from Multiple-Choice Tests”. In: *Journal of Experimental Psychology: Applied* 13.4 (2007), pp. 273–281. DOI: 10.1037/1076-898X.13.4.273.
- [16] Andrew C. Butler, Elizabeth J. Marsh, Michael K. Goode, and Henry L. Roediger. “When Additional Multiple-Choice Lures Aid versus Hinder Later Memory”. In: *Applied Cognitive Psychology* 20.7 (Nov. 1, 2006), pp. 941–956. DOI: 10.1002/acp.1239.
- [17] Andrew C. Butler and Henry L. Roediger. “Feedback Enhances the Positive Effects and Reduces the Negative Effects of Multiple-Choice Testing”. In: *Memory & Cognition* 36.3 (Apr. 2008), pp. 604–616. DOI: 10.3758/MC.36.3.604.
- [18] Sarah Carr. “As Distance Education Comes of Age, the Challenge Is Keeping the Students”. In: *The Chronicle of Higher Education* 46.23 (2000), A39–A41.
- [19] John M. Carroll, Penny L. Smith-Kerker, James R. Ford, and Sandra A. Mazur-Rimet. “The Minimal Manual”. In: *Human–Computer Interaction* 3.2 (June 1, 1987), pp. 123–153. DOI: 10.1207/s15327051hci0302_2.
- [20] John Millar Carroll and Sandra A. Mazur. *LisaLearning*. Citeseer, 1985.

- [21] Kuo-En Chang, Bea-Chu Chiao, Sei-Wang Chen, and Rong-Shue Hsiao. “A Programming Learning System for Beginners-a Completion Strategy Approach”. In: *IEEE Transactions on Education* 43.2 (May 2000), pp. 211–220. DOI: 10.1109/13.848075.
- [22] Michelene T.H. Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. “Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems”. In: *Cognitive Science* 13.2 (Apr. 1, 1989), pp. 145–182. DOI: 10.1207/s15516709cog1302_1.
- [23] Pei-Yu Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. “MixT: Automatic Generation of Step-by-Step Mixed Media Tutorials”. In: *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. UIST ’12. New York, NY, USA: ACM, 2012, pp. 93–102. DOI: 10.1145/2380116.2380130.
- [24] Ruth Colvin Clark, Frank Nguyen, John Sweller, and Melissa Baddeley. “Efficiency in Learning: Evidence-Based Guidelines to Manage Cognitive Load”. In: *Performance Improvement* 45.9 (Oct. 1, 2006), pp. 46–47. DOI: 10.1002/pfi.4930450920.
- [25] Andy Cockburn, Philip Quinn, and Carl Gutwin. “Examining the Peak-End Effects of Subjective Experience”. In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI ’15. New York, NY, USA: ACM, 2015, pp. 357–366. DOI: 10.1145/2702123.2702139.
- [26] *Code Avengers*. URL: <http://www.codeavengers.com/>.
- [27] *Codecademy*. URL: <http://www.codecademy.com/>.
- [28] *CodeCombat*. URL: <http://codecombat.com/>.
- [29] *CodeCombat: Learn to Code by Playing a Game*. URL: <http://codecombat.com> (visited on 03/13/2016).
- [30] *Code.Org*. URL: <http://code.org/>.
- [31] *Computing Education and Future Jobs: A Look at National, State, and Congressional District Data*. National Center for Women & Information Technology, Dec. 16, 2011.
- [32] Graham Cooper and John Sweller. “Effects of Schema Acquisition and Rule Automation on Mathematical Problem-Solving Transfer”. In: *Journal of Educational Psychology* 79.4 (1987), pp. 347–362. DOI: 10.1037/0022-0663.79.4.347.
- [33] Gemma Corbalan, Liesbeth Kester, and Jeroen J.G. van Merriënboer. “Dynamic Task Selection: Effects of Feedback and Learner Control on Efficiency and Motivation”. In: *Learning and Instruction* 19.6 (Dec. 2009), pp. 455–465. DOI: 10.1016/j.learninstruc.2008.07.002.
- [34] Gemma Corbalan, Liesbeth Kester, and Jeroen J. G. Van Merriënboer. “Towards a Personalized Task Selection Model with Shared Instructional Control”. In: *Instructional Science* 34.5 (Sept. 1, 2006), pp. 399–422. DOI: 10.1007/s11251-005-5774-2.

- [35] Gemma Corbalan, Liesbeth Kester, and Jeroen J. G. van Merriënboer. “Learner-Controlled Selection of Tasks with Different Surface and Structural Features: Effects on Transfer and Efficiency”. In: *Computers in Human Behavior*. Current Research Topics in Cognitive Load Theory Third International Cognitive Load Theory Conference 27.1 (Jan. 2011), pp. 76–81. DOI: 10.1016/j.chb.2010.05.026.
- [36] Gemma Corbalan, Liesbeth Kester, and Jeroen J. G. van Merriënboer. “Selecting Learning Tasks: Effects of Adaptation and Shared Control on Learning Efficiency and Task Involvement”. In: *Contemporary Educational Psychology* 33.4 (Oct. 2008), pp. 733–756. DOI: 10.1016/j.cedpsych.2008.02.003.
- [37] Albert T. Corbett and John R. Anderson. “Student Modeling and Mastery Learning in a Computer-Based Programming Tutor”. In: *Intelligent Tutoring Systems*. Ed. by Claude Frasson, Gilles Gauthier, and Gordon I. McCalla. Lecture Notes in Computer Science 608. Springer Berlin Heidelberg, Jan. 1, 1992, pp. 413–420.
- [38] N. Cowan. “The Magical Number 4 in Short-Term Memory: A Reconsideration of Mental Storage Capacity”. In: *The Behavioral and Brain Sciences* 24.1 (Feb. 2001), 87–114, discussion 114–185. PMID: 11515286.
- [39] Andrée-Ann Cyr and Nicole D. Anderson. “Mistakes as Stepping Stones: Effects of Errors on Episodic Memory among Younger and Older Adults”. In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 41.3 (2015), pp. 841–850. DOI: 10.1037/xlm0000073.
- [40] Shaundra B. Daily, Alison E. Leonard, Sophie Jörg, Sabarish Babu, and Kara Gundersen. “Dancing Alice: Exploring Embodied Pedagogical Strategies for Learning Computational Thinking”. In: *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE ’14. New York, NY, USA: ACM, 2014, pp. 91–96. DOI: 10.1145/2538862.2538917.
- [41] *Debugems to Share - Scratchdebugems*. URL: <https://sites.google.com/site/scratchdebugems/debugems-to-share> (visited on 12/31/2016).
- [42] Edward L. Deci, Robert J. Vallerand, Luc G. Pelletier, and Richard M. Ryan. “Motivation and Education: The Self-Determination Perspective”. In: *Educational Psychologist* 26 (3-4 June 1, 1991), pp. 325–346. DOI: 10.1080/00461520.1991.9653137.
- [43] Paul Denny, Andrew Luxton-Reilly, and Beth Simon. “Evaluating a New Exam Question: Parsons Problems”. In: *Proceedings of the Fourth International Workshop on Computing Education Research*. ICER ’08. New York, NY, USA: ACM, 2008, pp. 113–124. DOI: 10.1145/1404520.1404532.
- [44] Tao Dong, Mira Dontcheva, Diana Joseph, Karrie Karahalios, Mark Newman, and Mark Ackerman. “Discovery-Based Games for Learning Software”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’12. New York, NY, USA: ACM, 2012, pp. 2083–2086. DOI: 10.1145/2207676.2208358.

- [45] Michael Eagle and Tiffany Barnes. “Experimental Evaluation of an Educational Game for Improved Learning in Introductory Computing”. In: *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*. SIGCSE '09. New York, NY, USA: ACM, 2009, pp. 321–325. DOI: 10.1145/1508865.1508980.
- [46] *edX*. URL: <https://www.edx.org/>.
- [47] Warwick B. Elley. “The Role of Errors in Learning with Feedback”. In: *British Journal of Educational Psychology* 36.3 (Nov. 1, 1966), pp. 296–300. DOI: 10.1111/j.2044-8279.1966.tb01882.x.
- [48] Barbara J. Ericson, Mark J. Guzdial, and Briana B. Morrison. “Analysis of Interactive Features Designed to Enhance Learning in an Ebook”. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ICER '15. New York, NY, USA: ACM, 2015, pp. 169–178. DOI: 10.1145/2787622.2787731.
- [49] Stella Fayer, Alan Lacey, and Audrey Watson. *STEM Occupations: Past, Present, And Future*. US Bureau of Labor Statistics, Jan. 2017.
- [50] Jennifer Fernquist, Tovi Grossman, and George Fitzmaurice. “Sketch-Sketch Revolution: An Engaging Tutorial System for Guided Sketching and Application Learning”. In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST '11. New York, NY, USA: ACM, 2011, pp. 373–382. DOI: 10.1145/2047196.2047245.
- [51] S. Garner. “The Learning of Plans in Programming: A Program Completion Approach”. In: *International Conference on Computers in Education, 2002. Proceedings*. International Conference on Computers in Education, 2002. Proceedings. 2002, 1053–1057 vol.2. DOI: 10.1109/CIE.2002.1186149.
- [52] Stuart Garner. “A Tool to Support the Use of Part-Complete Solutions in the Learning of Programming”. In: *Proceeding de Conférence*. 2001, pp. 222–228.
- [53] Stuart Garner. “Learning to Program Using Part-Complete Solutions”. In: (2003).
- [54] Stuart Garner. “Reducing the Cognitive Load on Novice Programmers”. In: *World Conference on Educational Multimedia, Hypermedia and Telecommunications 2002* 2002.1 (2002), pp. 578–583.
- [55] *Gidget*. URL: <http://www.helpgidget.org/>.
- [56] Tamara van Gog and Fred Paas. “Instructional Efficiency: Revisiting the Original Construct in Educational Research”. In: *Educational Psychologist* 43.1 (Jan. 28, 2008), pp. 16–26. DOI: 10.1080/00461520701756248.
- [57] M. Gordon and P. J. Guo. “Codepourri: Creating Visual Coding Tutorials Using a Volunteer Crowd of Learners”. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Oct. 2015, pp. 13–21. DOI: 10.1109/VLHCC.2015.7357193.

- [58] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. “Generating Photo Manipulation Tutorials by Demonstration”. In: *ACM SIGGRAPH 2009 Papers*. New Orleans, Louisiana: ACM, 2009, pp. 1–9. DOI: 10.1145/1576246.1531372.
- [59] Simon Gray, Caroline St. Clair, Richard James, and Jerry Mead. “Suggestions for Graduated Exposure to Programming Concepts Using Fading Worked Examples”. In: *Proceedings of the Third International Workshop on Computing Education Research*. ICER ’07. New York, NY, USA: ACM, 2007, pp. 99–110. DOI: 10.1145/1288580.1288594.
- [60] J. Griffin, E. Kaplan, and Q. Burke. “Debug’ems and Other Deconstruction Kits for STEM Learning”. In: *IEEE 2nd Integrated STEM Education Conference*. IEEE 2nd Integrated STEM Education Conference. Mar. 2012, pp. 1–4. DOI: 10.1109/ISECon.2012.6204168.
- [61] Paul Gross and Caitlin Kelleher. “Non-Programmers Identifying Functionality in Unfamiliar Code: Strategies and Barriers”. In: *Journal of Visual Languages & Computing* 21.5 (Dec. 2010), pp. 263–276. DOI: 10.1016/j.jv1c.2010.08.002.
- [62] Paul Gross, Jennifer Yang, and Caitlin Kelleher. “Dinah: An Interface to Assist Non-Programmers with Selecting Program Code Causing Graphical Output”. In: *Proceedings of the 2011 Annual Conference on Human Factors in Computing Systems*. CHI ’11. New York, NY, USA: ACM, 2011, pp. 3397–3400. DOI: 10.1145/1978942.1979448.
- [63] Cornelia S. Große and Alexander Renkl. “Finding and Fixing Errors in Worked Examples: Can This Foster Learning Outcomes?” In: *Learning and Instruction* 17.6 (Dec. 2007), pp. 612–634. DOI: 10.1016/j.learninstruc.2007.09.008.
- [64] Philip J. Guo. “Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education”. In: *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*. SIGCSE ’13. New York, NY, USA: ACM, 2013, pp. 579–584. DOI: 10.1145/2445196.2445368.
- [65] Thomas M. Haladyna and Steven M. Downing. “Functional Distractors: Implications for Test-Item Writing and Test Design.” In: (Apr. 1988).
- [66] Thomas M. Haladyna and Steven M. Downing. “How Many Options Is Enough for a Multiple-Choice Test Item?” In: *Educational and Psychological Measurement* 53.4 (Jan. 12, 1993), pp. 999–1010. DOI: 10.1177/0013164493053004013.
- [67] Thomas M. Haladyna and Steven M. Downing. “Validity of a Taxonomy of Multiple-Choice Item-Writing Rules”. In: *Applied Measurement in Education* 2.1 (Jan. 1, 1989), pp. 51–78. DOI: 10.1207/s15324818ame0201_4.
- [68] Mohamed Hamada. “Web-Based Tools for Active Learning in Information Theory”. In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’07. New York, NY, USA: ACM, 2007, pp. 60–64. DOI: 10.1145/1227310.1227332.

- [69] K. J. Harms, E. Balzuweit, J. Chen, and C. Kelleher. “Learning Programming from Tutorials and Code Puzzles: Children’s Perceptions of Value”. In: *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Sept. 2016, pp. 59–67. DOI: 10.1109/VLHCC.2016.7739665.
- [70] K. J. Harms, N. Rowlett, and C. Kelleher. “Enabling Independent Learning of Programming Concepts through Programming Completion Puzzles”. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Oct. 2015, pp. 271–279. DOI: 10.1109/VLHCC.2015.7357226.
- [71] Kyle J. Harms, Dennis Cosgrove, Shannon Gray, and Caitlin Kelleher. “Automatically Generating Tutorials to Enable Middle School Children to Learn Programming Independently”. In: *Proceedings of the 12th International Conference on Interaction Design and Children*. IDC ’13. New York, NY, USA: ACM, 2013, pp. 11–19. DOI: 10.1145/2485760.2485764.
- [72] Kyle J. Harms, Jordana H. Kerr, and Caitlin L. Kelleher. “Improving Learning Transfer from Stencils-Based Tutorials”. In: *Proceedings of the 10th International Conference on Interaction Design and Children*. IDC ’11. New York, NY, USA: ACM, 2011, pp. 157–160. DOI: 10.1145/1999030.1999050.
- [73] Kyle James Harms, Jason Chen, and Caitlin L. Kelleher. “Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers”. In: *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ICER ’16. New York, NY, USA: ACM, 2016, pp. 241–250. DOI: 10.1145/2960310.2960314.
- [74] Ken Hartness. “Robocode: Using Games to Teach Artificial Intelligence”. In: *J. Comput. Small Coll.* 19.4 (2004), pp. 287–291.
- [75] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann. “Tutorons: Generating Context-Relevant, on-Demand Explanations and Demonstrations of Online Code”. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). Oct. 2015, pp. 3–12. DOI: 10.1109/VLHCC.2015.7356972.
- [76] J. Helminen, P. Ihanola, V. Karavirta, and S. Alaoutinen. “How Do Students Solve Parsons Programming Problems? – Execution-Based vs. Line-Based Feedback”. In: *Learning and Teaching in Computing and Engineering (LaTiCE), 2013*. Learning and Teaching in Computing and Engineering (LaTiCE), 2013. Mar. 2013, pp. 55–61. DOI: 10.1109/LaTiCE.2013.26.
- [77] *Hour of Code*. URL: <http://csedweek.org/>.
- [78] Barbie J. Huelser and Janet Metcalfe. “Making Related Errors Facilitates Learning, but Learners Do Not Know It”. In: *Memory & Cognition* 40.4 (Dec. 9, 2011), pp. 514–527. DOI: 10.3758/s13421-011-0167-z.

- [79] M. Ichinco. “Towards Crowdsourced Large-Scale Feedback for Novice Programmers”. In: *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). July 2014, pp. 189–190. DOI: 10.1109/VLHCC.2014.6883049.
- [80] Michelle Ichinco, Kyle J. Harms, and Caitlin Kelleher. “Towards Understanding Successful Novice Example Use in Blocks-Based Programming”. In: *Journal of Visual Languages and Sentient Systems: Special Issue on Blocks Programming* (2017). Forthcoming.
- [81] *IEEE Job Site*. URL: http://careers.ieee.org/article/European_Job_Outlook_0312.php (visited on 01/22/2013).
- [82] Magid Igbaria, Juhani Iivari, and Hazem Maragahh. “Why Do Individuals Use Computer Technology? A Finnish Case Study”. In: *Information & Management* 29.5 (Nov. 1995), pp. 227–238. DOI: 10.1016/0378-7206(95)00031-0.
- [83] Petri Ihantola, Juha Helminen, and Ville Karavirta. “How to Study Programming on Mobile Touch Devices: Interactive Python Code Exercises”. In: *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. Koli Calling ’13. New York, NY, USA: ACM, 2013, pp. 51–58. DOI: 10.1145/2526968.2526974.
- [84] Petri Ihantola and Ville Karavirta. “Two-Dimensional Parson’s Puzzles: The Concept, Tools, and First Observations”. In: *Journal of Information Technology Education* 10 (2011).
- [85] *Intrinsic Motivation Inventory*. URL: <http://www.selfdeterminationtheory.org/questionnaires/10-questionnaires/50>.
- [86] Wei Jin. “Pre-Programming Analysis Tutors Help Students Learn Basic Programming Concepts”. In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’08. New York, NY, USA: ACM, 2008, pp. 276–280. DOI: 10.1145/1352135.1352231.
- [87] H. K, Harold Pashler, Nicholas J. Cepeda, Doug Rohrer, Shana K. Carpenter, and Michael C. Mozer. “Does Incorrect Guessing Impair Fact Learning?” In: *Journal of Educational Psychology* 103.1 (2011), pp. 48–59. DOI: 10.1037/a0021977.
- [88] Slava Kalyuga, Paul Ayres, Paul Chandler, and John Sweller. “The Expertise Reversal Effect”. In: *Educational Psychologist* 38.1 (2003), pp. 23–31. DOI: 10.1207/S15326985EP3801_4.
- [89] Slava Kalyuga, Paul Chandler, and John Sweller. “Managing Split-Attention and Redundancy in Multimedia Instruction”. In: *Applied cognitive psychology* 13.4 (1999), pp. 351–371.
- [90] Ville Karavirta, Juha Helminen, and Petri Ihantola. “A Mobile Learning Application for Parsons Problems with Automatic Feedback”. In: *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. Koli Calling ’12. New York, NY, USA: ACM, 2012, pp. 11–18. DOI: 10.1145/2401796.2401798.

- [91] Jeffrey D. Karpicke. “Retrieval-Based Learning Active Retrieval Promotes Meaningful Learning”. In: *Current Directions in Psychological Science* 21.3 (June 1, 2012), pp. 157–163. DOI: 10.1177/0963721412443552.
- [92] Idit Katz and Avi Assor. “When Choice Motivates and When It Does Not”. In: *Educational Psychology Review* 19.4 (Dec. 1, 2007), p. 429. DOI: 10.1007/s10648-006-9027-y.
- [93] Fengfeng Ke and Kui Xie. “Online Discussion Design on Adult Students’ Learning Perceptions and Patterns of Online Interactions”. In: *Proceedings of the 9th International Conference on Computer Supported Collaborative Learning - Volume 1*. CSCL’09. Rhodes, Greece: International Society of the Learning Sciences, 2009, pp. 219–226.
- [94] C. Kelleher, D. Cosgrove, D. Culyba, C. Forlines, J. Pratt, and R. Pausch. “Alice2: Programming without Syntax Errors”. In: *User Interface Software and Technology*. 2002.
- [95] Caitlin Kelleher and Randy Pausch. “Stencils-Based Tutorials: Design and Evaluation”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’05. New York, NY, USA: ACM, 2005, pp. 541–550. DOI: 10.1145/1054972.1055047.
- [96] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. “Storytelling Alice Motivates Middle School Girls to Learn Computer Programming”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’07. New York, NY, USA: ACM, 2007, pp. 1455–1464. DOI: 10.1145/1240624.1240844.
- [97] *Khan Academy*. URL: <http://www.khanacademy.org>.
- [98] Kelly V. King, Doug A. Gardner, Sasha Zucker, and Margaret A. Jorgensen. “The Distractor Rationale Taxonomy: Enhancing Multiple-Choice Items in Reading and Mathematics”. In: *Assessment Report*. Pearson (2004).
- [99] Roger E. Kirk. “Practical Significance: A Concept Whose Time Has Come”. In: *Educational and Psychological Measurement* 56.5 (Jan. 10, 1996), pp. 746–759. DOI: 10.1177/0013164496056005002.
- [100] Paul A. Kirschner, John Sweller, and Richard E. Clark. “Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching”. In: *Educational Psychologist* 41.2 (2006), pp. 75–86. DOI: 10.1207/s15326985ep4102_1.
- [101] Nate Kornell, Matthew Jensen Hays, and Robert A. Bjork. “Unsuccessful Retrieval Attempts Enhance Subsequent Learning”. In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 35.4 (2009), pp. 989–998. DOI: 10.1037/a0015729.
- [102] Benjamin Lafreniere, Tovi Grossman, and George Fitzmaurice. “Community Enhanced Tutorials: Improving Tutorials with Multiple Demonstrations”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’13. New York, NY, USA: ACM, 2013, pp. 1779–1788. DOI: 10.1145/2470654.2466235.

- [103] Karen Lang, R. Galanos, J. Goode, D. Seehorn, F. Trees, P. Phillips, and C. Stephenson. “Bugs in the System: Computer Science Teacher Certification in the US”. In: *The Computer Science Teachers Association and The Association for Computing Machinery* (2013).
- [104] Michael J. Lee and Andrew J. Ko. “Comparing the Effectiveness of Online Learning Approaches on CS1 Learning Outcomes”. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ICER ’15. New York, NY, USA: ACM, 2015, pp. 237–246. DOI: 10.1145/2787622.2787709.
- [105] Michael J. Lee and Andrew J. Ko. “Personifying Programming Tool Feedback Improves Novice Programmers’ Learning”. In: *Proceedings of the Seventh International Workshop on Computing Education Research*. ICER ’11. New York, NY, USA: ACM, 2011, pp. 109–116. DOI: 10.1145/2016911.2016934.
- [106] Michael J. Lee, Andrew J. Ko, and Irwin Kwan. “In-Game Assessments Increase Novice Programmers’ Engagement and Level Completion Speed”. In: *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*. ICER ’13. New York, NY, USA: ACM, 2013, pp. 153–160. DOI: 10.1145/2493394.2493410.
- [107] M.J. Lee et al. “Principles of a Debugging-First Puzzle Game for Computing Education”. In: *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). July 2014, pp. 57–64. DOI: 10.1109/VLHCC.2014.6883023.
- [108] Jimmie Leppink, Fred Paas, Tamara van Gog, Cees P. M. van der Vleuten, and Jeroen J. G. van Merriënboer. “Effects of Pairs of Problems and Examples on Task Performance and Different Types of Cognitive Load”. In: *Learning and Instruction* 30 (Apr. 2014), pp. 32–42. DOI: 10.1016/j.learninstruc.2013.12.001.
- [109] Jimmie Leppink, Fred Paas, Cees P. M. Van der Vleuten, Tamara Van Gog, and Jeroen J. G. Van Merriënboer. “Development of an Instrument for Measuring Different Types of Cognitive Load”. In: *Behavior Research Methods* 45.4 (Apr. 10, 2013), pp. 1058–1072. DOI: 10.3758/s13428-013-0334-1.
- [110] Wei Li, Tovi Grossman, and George Fitzmaurice. “CADament: A Gamified Multiplayer Software Tutorial System”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’14. New York, NY, USA: ACM, 2014, pp. 3369–3378. DOI: 10.1145/2556288.2556954.
- [111] Wei Li, Tovi Grossman, and George Fitzmaurice. “GamiCAD: A Gamified Tutorial System for First Time Autocad Users”. In: *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*. UIST ’12. New York, NY, USA: ACM, 2012, pp. 103–112. DOI: 10.1145/2380116.2380131.
- [112] *Lightbot*. URL: <http://lightbot.com/>.

- [113] Jie Lin, Kwang-lee Chu, and Ying Meng. “Distractor Rationale Taxonomy: Diagnostic Assessment of Reading with Ordered Multiple-Choice Items”. In: *American Educational Research Association, Denver, CO* (2010).
- [114] Lijia Lin, Robert K. Atkinson, Wilhelmina C. Savenye, and Brian C. Nelson. “Effects of Visual Cues and Self-Explanation Prompts: Empirical Evidence in a Multimedia Environment”. In: *Interactive Learning Environments* 24.4 (May 18, 2016), pp. 799–813. DOI: 10.1080/10494820.2014.924531.
- [115] Jeri L. Little and Elizabeth Ligon Bjork. “The Persisting Benefits of Using Multiple-Choice Tests as Learning Events”. In: *Proceedings of the 34th Annual Conference of the Cognitive Science Society*. 2012, pp. 683–688.
- [116] Ju Long. “Just for Fun: Using Programming Games in Software Programming Training and Education—A Field Study of IBM Robocode Community”. In: *Journal of Information Technology Education* 6 (2007), pp. 279–290.
- [117] *Looking Glass*. URL: <http://lookingglass.wustl.edu/>.
- [118] Robert L. Mack, Clayton H. Lewis, and John M. Carroll. “Learning to Use Word Processors: Problems and Prospects”. In: *ACM Trans. Inf. Syst.* 1.3 (July 1983), pp. 254–271. DOI: 10.1145/357436.357440.
- [119] Matt MacLaurin. “Kodu: End-User Programming and Design for Games”. In: *Proceedings of the 4th International Conference on Foundations of Digital Games*. Orlando, Florida: ACM, 2009, pp. xviii–xix. DOI: 10.1145/1536513.1536516.
- [120] Matthew B. MacLaurin. “The Design of Kodu: A Tiny Visual Programming Language for Children on the Xbox 360”. In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 241–246. DOI: 10.1145/1925844.1926413.
- [121] Elizabeth J. Marsh, Pooja K. Agarwal, and Henry L. Roediger III. “Memorial Consequences of Answering SAT II Questions”. In: *Journal of Experimental Psychology: Applied* 15.1 (2009), pp. 1–11. DOI: 10.1037/a0014721.
- [122] Elizabeth J. Marsh, Henry L. Roediger, Robert A. Bjork, and Elizabeth L. Bjork. “The Memorial Consequences of Multiple-Choice Testing”. In: *Psychonomic Bulletin & Review* 14.2 (Apr. 2007), pp. 194–199. DOI: 10.3758/BF03194051.
- [123] Elizabeth Marsh and Allison Cantor. “Learning from the Test: Dos and Don’ts for Using Multiple-Choice Tests”. In: *Integrating Cognitive Science with Innovative Teaching in STEM Disciplines* (Sept. 10, 2014). DOI: 10.7936/K7Z60KZK.
- [124] Jeroen J. G. Van Merriënboer and Hein P. M. Krammer. “Instructional Strategies and Tactics for the Design of Introductory Computer Programming Courses in High School”. In: *Instructional Science* 16.3 (Sept. 1, 1987), pp. 251–285. DOI: 10.1007/BF00120253.
- [125] Jeroen J. G. van Merriënboer and Dominique M. A. Shuijsmans. “Toward a Synthesis of Cognitive Load Theory, Four-Component Instructional Design, and Self-Directed Learning”. In: *Educational Psychology Review* 21.1 (Mar. 1, 2009), pp. 55–66. DOI: 10.1007/s10648-008-9092-5.

- [126] Jeroen J. G. van Merriënboer and John Sweller. “Cognitive Load Theory and Complex Learning: Recent Developments and Future Directions”. In: *Educational Psychology Review* 17.2 (June 1, 2005), pp. 147–177. DOI: 10.1007/s10648-005-3951-0.
- [127] George A. Miller. “The Magical Number Seven, plus or Minus Two: Some Limits on Our Capacity for Processing Information”. In: *Psychological Review* 63.2 (1956), pp. 81–97. DOI: 10.1037/h0043158.
- [128] Misook Heo and Anthony Chow. “The Impact of Computer Augmented Online Learning and Assessment Tool”. In: *Journal of Educational Technology & Society* 8.1 (Jan. 2005), pp. 113–123.
- [129] *MIT App Inventor*. URL: <http://appinventor.mit.edu/>.
- [130] A. Mitrovic. “An Intelligent SQL Tutor on the Web”. In: *International Journal of Artificial Intelligence in Education* 13.2 (2003), pp. 173–197.
- [131] Artur Mkrtchyan. “Distractor Quality Analyze In Multiple Choice Questions Based On Information Retrieval Model”. In: *EDULEARN11 Proceedings* (2011), pp. 1624–1631.
- [132] Johnette Moody. “Distance Education: Why Are the Attrition Rates so High?” In: *Quarterly Review of Distance Education* 5.3 (2004), pp. 205–210.
- [133] Briana B. Morrison, Brian Dorn, and Mark Guzdial. “Measuring Cognitive Load in Introductory CS: Adaptation of an Instrument”. In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER ’14. New York, NY, USA: ACM, 2014, pp. 131–138. DOI: 10.1145/2632320.2632348.
- [134] Briana B. Morrison, Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial. “Subgoals Help Students Solve Parsons Problems”. In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. SIGCSE ’16. New York, NY, USA: ACM, 2016, pp. 42–47. DOI: 10.1145/2839509.2844617.
- [135] Briana B. Morrison, Lauren E. Margulieux, and Mark Guzdial. “Subgoals, Context, and Worked Examples in Learning Computing Problem Solving”. In: *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*. ICER ’15. New York, NY, USA: ACM, 2015, pp. 21–29. DOI: 10.1145/2787622.2787733.
- [136] D.a. Muller, J. Bewes, M.d. Sharma, and P. Reimann. “Saying the Wrong Thing: Improving Learning with Multimedia by Including Misconceptions”. In: *Journal of Computer Assisted Learning* 24.2 (Apr. 1, 2008), pp. 144–155. DOI: 10.1111/j.1365-2729.2007.00248.x.
- [137] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. “What Makes a Good Code Example?: A Study of Programming Q Amp;A in StackOverflow”. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 2012 28th IEEE International Conference on Software Maintenance (ICSM). Sept. 2012, pp. 25–34. DOI: 10.1109/ICSM.2012.6405249.

- [138] Fred G. Paas. “Training Strategies for Attaining Transfer of Problem-Solving Skill in Statistics: A Cognitive-Load Approach”. In: *Journal of Educational Psychology* 84.4 (1992), pp. 429–434. DOI: 10.1037/0022-0663.84.4.429.
- [139] Fred G. W. C. Paas and Jeroen J. G. Van Merriënboer. “The Efficiency of Instructional Conditions: An Approach to Combine Mental Effort and Performance Measures”. In: *Human Factors: The Journal of the Human Factors and Ergonomics Society* 35.4 (Jan. 12, 1993), pp. 737–743. DOI: 10.1177/001872089303500412.
- [140] Fred G. W. C. Paas, Jeroen J. G. Van Merrienboer, and Jos J. Adam. “Measurement of Cognitive Load in Instructional Research”. In: *Perceptual and Motor Skills* 79.1 (Aug. 1, 1994), pp. 419–430. DOI: 10.2466/pms.1994.79.1.419.
- [141] Fred Paas, Alexander Renkl, and John Sweller. “Cognitive Load Theory and Instructional Design: Recent Developments”. In: *Educational Psychologist* 38.1 (2003), pp. 1–4. DOI: 10.1207/S15326985EP3801_1.
- [142] Fred Paas, Alexander Renkl, and John Sweller. “Cognitive Load Theory: Instructional Implications of the Interaction between Information Structures and Cognitive Architecture”. In: *Instructional Science* 32 (1-2 Jan. 1, 2004), pp. 1–8. DOI: 10.1023/B:TRUC.0000021806.17516.d0.
- [143] Fred Paas, Juhani E. Tuovinen, Huib Tabbers, and Pascal W. M. Van Gerven. “Cognitive Load Measurement as a Means to Advance Cognitive Load Theory”. In: *Educational Psychologist* 38.1 (2003), pp. 63–71. DOI: 10.1207/S15326985EP3801_8.
- [144] Fred Paas and Jeroen Van Merrienboer. “Variability of Worked Examples and Transfer of Geometrical Problem-Solving Skills: A Cognitive-Load Approach”. In: *Journal of Educational Psychology* 86.1 (1994), pp. 122–133. DOI: 10.1037/0022-0663.86.1.122.
- [145] Susan Palmiter, Jay Elkerton, and P. Baggett. “Animated Demonstrations vs. Written Instructions for Learning Procedural Tasks: A Preliminary Investigation”. In: *Int. J. Man-Mach. Stud.* 34.5 (1991), pp. 687–701.
- [146] Angie Parker. “A Study of Variables That Predict Dropout from Distance Education”. In: *International Journal of Educational Technology* 1.2 (1999), pp. 1–10.
- [147] D. Parmar, J. Isaac, S. V. Babu, N. D’Souza, A. E. Leonard, S. Jörg, K. Gundersen, and S. B. Daily. “Programming Moves: Design and Evaluation of Applying Embodied Interaction in Virtual Environments to Enhance Computational Thinking in Middle School Students”. In: *2016 IEEE Virtual Reality (VR)*. 2016 IEEE Virtual Reality (VR). Mar. 2016, pp. 131–140. DOI: 10.1109/VR.2016.7504696.
- [148] Dale Parsons and Patricia Haden. “Parson’s Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses”. In: *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*. ACE ’06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 157–163.

- [149] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. “Autonomously Generating Hints by Inferring Problem Solving Policies”. In: *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*. L@S ’15. New York, NY, USA: ACM, 2015, pp. 195–204. DOI: 10.1145/2724660.2724668.
- [150] Peter L. Pirolli and John R. Anderson. “The Role of Learning from Examples in the Acquisition of Recursive Programming Skills”. In: *Canadian Journal of Psychology/Revue canadienne de psychologie* 39.2 (1985), pp. 240–272. DOI: 10.1037/h0080061.
- [151] Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F. Cohen. “Pause-and-Play: Automatically Linking Screencast Video Tutorials with Applications”. In: *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*. UIST ’11. New York, NY, USA: ACM, 2011, pp. 135–144. DOI: 10.1145/2047196.2047213.
- [152] Rosalind Potts and David R. Shanks. “The Benefit of Generating Errors during Learning”. In: *Journal of Experimental Psychology: General* 143.2 (2014), pp. 644–667. DOI: 10.1037/a0033194.
- [153] Jill L. Quilici and Richard E. Mayer. “Role of Examples in How Students Learn to Categorize Statistics Word Problems”. In: *Journal of Educational Psychology* 88.1 (1996), pp. 144–161. DOI: 10.1037/0022-0663.88.1.144.
- [154] Atanas Radenski. “Digital Support for Abductive Learning in Introductory Computing Courses”. In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE ’07. New York, NY, USA: ACM, 2007, pp. 14–18. DOI: 10.1145/1227310.1227318.
- [155] Atanas Radenski. “Freedom of Choice As Motivational Factor for Active Learning”. In: *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE ’09. New York, NY, USA: ACM, 2009, pp. 21–25. DOI: 10.1145/1562877.1562891.
- [156] Haider Ramadhan. “An Intelligent Discovery Programming System”. In: *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing: Technological Challenges of the 1990’s*. SAC ’92. New York, NY, USA: ACM, 1992, pp. 149–159. DOI: 10.1145/143559.143630.
- [157] Janet C. Read. “Validating the Fun Toolkit: An Instrument for Measuring Children’s Opinions of Technology”. In: *Cognition, Technology & Work* 10.2 (May 22, 2007), pp. 119–128. DOI: 10.1007/s10111-007-0069-9.
- [158] Alexander Renkl. “Learning from Worked-Out Examples: A Study on Individual Differences”. In: *Cognitive Science* 21.1 (Jan. 1, 1997), pp. 1–29. DOI: 10.1207/s15516709cog2101_1.
- [159] Alexander Renkl, Robert K. Atkinson, and Cornelia S. Große. “How Fading Worked Solution Steps Works – A Cognitive Load Perspective”. In: *Instructional Science* 32 (1-2 Jan. 1, 2004), pp. 59–82. DOI: 10.1023/B:TRUC.0000021815.74806.f6.

- [160] Alexander Renkl, Robin Stark, Hans Gruber, and Heinz Mandl. “Learning from Worked-Out Examples: The Effects of Example Variability and Elicited Self-Explanations”. In: *Contemporary Educational Psychology* 23.1 (Jan. 1998), pp. 90–108. DOI: 10.1006/ceps.1997.0959.
- [161] Lindsey E. Richland, Nate Kornell, and Liche Sean Kao. “The Pretesting Effect: Do Unsuccessful Retrieval Attempts Enhance Learning?” In: *Journal of Experimental Psychology: Applied* 15.3 (2009), pp. 243–257. DOI: 10.1037/a0016496.
- [162] Michael C. Rodriguez, Ryan J. Kettler, and Stephen N. Elliott. “Distractor Functioning in Modified Items for Test Accessibility”. In: *SAGE Open* 4.4 (Oct. 1, 2014), p. 2158244014553586. DOI: 10.1177/2158244014553586.
- [163] Henry L. Roediger III and Andrew C. Butler. “The Critical Role of Retrieval Practice in Long-Term Retention”. In: *Trends in Cognitive Sciences* 15.1 (Jan. 2011), pp. 20–27. DOI: 10.1016/j.tics.2010.09.003.
- [164] Henry L. Roediger III and Elizabeth J. Marsh. “The Positive and Negative Consequences of Multiple-Choice Testing”. In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 31.5 (2005), pp. 1155–1159. DOI: 10.1037/0278-7393.31.5.1155.
- [165] Henry L. Roediger and Jeffrey D. Karpicke. “Test-Enhanced Learning Taking Memory Tests Improves Long-Term Retention”. In: *Psychological Science* 17.3 (Mar. 1, 2006), pp. 249–255. DOI: 10.1111/j.1467-9280.2006.01693.x. pmid: 16507066.
- [166] Mary Beth Rosson, Julie Ballin, and Jochen Rode. “Who, What, and How: A Survey of Informal and Professional Web Developers”. In: *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. VLHCC ’05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 199–206. DOI: 10.1109/VLHCC.2005.73.
- [167] Mary Beth Rosson and John M. Carroll. “The Reuse of Uses in Smalltalk Programming”. In: *ACM Trans. Comput.-Hum. Interact.* 3.3 (Sept. 1996), pp. 219–253. DOI: 10.1145/234526.234530.
- [168] M.B. Rosson, J. Ballin, and H. Nash. “Everyday Programming: Challenges and Opportunities for Informal Web Development”. In: *2004 IEEE Symposium on Visual Languages and Human Centric Computing*. 2004 IEEE Symposium on Visual Languages and Human Centric Computing. 2004, pp. 123–130. DOI: 10.1109/VLHCC.2004.26.
- [169] Alfred P. Rovai. “In Search of Higher Persistence Rates in Distance Education Online Programs”. In: *The Internet and Higher Education* 6.1 (2003/0/1st/), pp. 1–16. DOI: 10.1016/S1096-7516(02)00158-6.
- [170] Warren Sack, Elliot Soloway, and P. Weingrad. “From PROUST to CHIRON: Its Design as Iterative Engineering: Intermediate Results Are Important”. In: *Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches*. Lawrence Erlbaum Associates, Hillsdale, NJ (1992), pp. 239–274.

- [171] Katharina Scheiter and Peter Gerjets. “Learner Control in Hypermedia Environments”. In: *Educational Psychology Review* 19.3 (Sept. 1, 2007), pp. 285–307. DOI: 10.1007/s10648-007-9046-3.
- [172] Heidi L. Schnackenberg and Howard J. Sullivan. “Learner Control over Full and Lean Computer-Based Instruction under Differing Ability Levels”. In: *Educational Technology Research and Development* 48.2 (June 1, 2000), pp. 19–35. DOI: 10.1007/BF02313399.
- [173] Silke Schworm and Alexander Renkl. “Computer-Supported Example-Based Learning: When Instructional Explanations Reduce Self-Explanations”. In: *Computers & Education* 46.4 (May 2006), pp. 426–445. DOI: 10.1016/j.compedu.2004.08.011.
- [174] *Scratch*. URL: <https://scratch.mit.edu/>.
- [175] Deborah Seehorn et al. *CSTA K–12 Computer Science Standards: Revised 2011*. New York, NY, USA: ACM, 2011.
- [176] Adam M. Smith, Erik Andersen, Michael Mateas, and Zoran Popović. “A Case Study of Expressively Constraining Level Design Automation Tools for a Puzzle Game”. In: *Proceedings of the International Conference on the Foundations of Digital Games*. FDG ’12. New York, NY, USA: ACM, 2012, pp. 156–163. DOI: 10.1145/2282338.2282370.
- [177] Elliot Soloway, Eric Rubin, Beverly Woolf, Jeffrey Bonar, and W. Lewis Johnson. *MENO-II: An AI-Based Programming Tutor*. Aug. 1983.
- [178] Jeffrey A. Stone and Elinor M. Madigan. “The Impact of Providing Project Choices in CS1”. In: *SIGCSE Bull.* 40.2 (June 2008), pp. 65–68. DOI: 10.1145/1383602.1383637.
- [179] Gail M. Sullivan and Richard Feinn. “Using Effect Size—or Why the P Value Is Not Enough”. In: *Journal of Graduate Medical Education* 4.3 (Sept. 2012), pp. 279–282. DOI: 10.4300/JGME-D-12-00156.1. pmid: 23997866.
- [180] J. Sweller. “Instructional Design in Technical Areas.” In: *Adelaide, SA: National Centre for Vocational Education Research* (1999).
- [181] John Sweller. “Element Interactivity and Intrinsic, Extraneous, and Germane Cognitive Load”. In: *Educational Psychology Review* 22.2 (June 1, 2010), pp. 123–138. DOI: 10.1007/s10648-010-9128-5.
- [182] John Sweller, Paul Ayres, and Slava Kalyuga. *Cognitive Load Theory*. Springer, Apr. 7, 2011. 280 pp.
- [183] John Sweller and Paul Chandler. “Why Some Material Is Difficult to Learn”. In: *Cognition and Instruction* 12.3 (1994), pp. 185–233. DOI: 10.1207/s1532690xci1203_1.
- [184] John Sweller, Paul Chandler, Paul Tierney, and Martin Cooper. “Cognitive Load as a Factor in the Structuring of Technical Material”. In: *Journal of Experimental Psychology: General* 119.2 (1990), pp. 176–192. DOI: 10.1037/0096-3445.119.2.176.
- [185] John Sweller and Graham A. Cooper. “The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra”. In: *Cognition and Instruction* 2.1 (1985), pp. 59–89. DOI: 10.1207/s1532690xci0201_3.

- [186] John Sweller, Jeroen J. G. van Merriënboer, and Fred G. W. C. Paas. “Cognitive Architecture and Instructional Design”. In: *Educational Psychology Review* 10.3 (Sept. 1, 1998), pp. 251–296. DOI: 10.1023/A:1022193728205.
- [187] E.m.c. Taminiau, L. Kester, G. Corbalan, J.m. Spector, P.a. Kirschner, and J.j.g. Van Merriënboer. “Designing On-Demand Education for Simultaneous Development of Domain-Specific and Self-Directed Learning Skills”. In: *Journal of Computer Assisted Learning* 31.5 (Oct. 1, 2015), pp. 405–421. DOI: 10.1111/jcal.12076.
- [188] Marie Tarrant, James Ware, and Ahmed M. Mohammed. “An Assessment of Functioning and Non-Functioning Distractors in Multiple-Choice Questions: A Descriptive Analysis”. In: *BMC Medical Education* 9 (2009), p. 40. DOI: 10.1186/1472-6920-9-40.
- [189] Sigmund Tobias, J. Dexter Fletcher, and Alexander P. Wind. “Game-Based Learning”. In: *Handbook of Research on Educational Communications and Technology*. Ed. by J. Michael Spector, M. David Merrill, Jan Elen, and M. J. Bishop. Springer New York, 2014, pp. 485–503. DOI: 10.1007/978-1-4614-3185-5_38.
- [190] John Gregory Trafton and Brian J. Reiser. “The Contributions of Studying Examples and Solving Problems to Skill Acquisition”. 1994.
- [191] Hans Van der Heijden. “User Acceptance of Hedonic Information Systems”. In: *MIS quarterly* (2004), pp. 695–704. JSTOR: 25148660.
- [192] Jeroen J. G. Van Merriënboer. “Strategies for Programming Instruction in High School: Program Completion vs. Program Generation”. In: *Journal of Educational Computing Research* 6.3 (Jan. 1, 1990), pp. 265–285. DOI: 10.2190/4NK5-17L7-TWQV-1EHL.
- [193] Jeroen J. G. Van Merriënboer and Marcel B. M. De Croock. “Strategies for Computer-Based Programming Instruction: Program Completion Vs. Program Generation”. In: *Journal of Educational Computing Research* 8.3 (Jan. 1, 1992), pp. 365–394. DOI: 10.2190/MJDX-9PP4-KFMT-09PM.
- [194] Jeroen JG Van Merriënboer and Hein PM Krammer. “The ”Completion Strategy” in Programming Instruction: Theoretical and Empirical Support”. In: *Research on instruction* (1990), pp. 45–61.
- [195] Tamara van Gog, Femke Kirschner, Liesbeth Kester, and Fred Paas. “Timing and Frequency of Mental Effort Measurement: Evidence in Favour of Repeated Measures”. In: *Applied Cognitive Psychology* 26.6 (Nov. 1, 2012), pp. 833–839. DOI: 10.1002/acp.2883.
- [196] J. J. G van Merriënboer, J. G Schuurman, M. B. M de Croock, and F. G. W. C Paas. “Redirecting Learners’ Attention during Training: Effects on Cognitive Load, Transfer Test Performance and Training Efficiency”. In: *Learning and Instruction* 12.1 (Feb. 2002), pp. 11–37. DOI: 10.1016/S0959-4752(01)00020-2.

- [197] Jeroen J. G. van Merriënboer, Liesbeth Kester, and Fred Paas. “Teaching Complex Rather than Simple Tasks: Balancing Intrinsic and Germane Load to Enhance Transfer of Learning”. In: *Applied Cognitive Psychology* 20.3 (Apr. 1, 2006), pp. 343–352. DOI: 10.1002/acp.1250.
- [198] Cheng-Yao Wang, Wei-Chen Chu, Hou-Ren Chen, Chun-Yen Hsu, and Mike Y. Chen. “EverTutor: Automatically Creating Interactive Guided Tutorials on Smartphones by User Demonstration”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’14. New York, NY, USA: ACM, 2014, pp. 4027–4036. DOI: 10.1145/2556288.2557407.
- [199] M. Wang, Z. K. Yang, S. Y. Liu, H. N. H. Cheng, and Z. Liu. “Using Feedback to Improve Learning: Differentiating between Correct and Erroneous Examples”. In: *2015 International Symposium on Educational Technology (ISET)*. 2015 International Symposium on Educational Technology (ISET). July 2015, pp. 99–103. DOI: 10.1109/ISET.2015.28.
- [200] *What’s Wrong with This Picture?* URL: <https://code.org/promote> (visited on 02/21/2017).
- [201] EJ Williams. “Experimental Designs Balanced for the Estimation of Residual Effects of Treatments”. In: *Australian Journal of Chemistry* 2.2 (Jan. 1, 1949), pp. 149–168.
- [202] Michael D. Williams. “Learner-Control and Instructional Technologies”. In: *D. H. Jonassen (Ed.), Handbook of research on educational communications and technology* (1996), pp. 957–982.
- [203] Cameron Wilson, Leigh Ann Sudol, Chris Stephenson, and Mark Stehlik. *Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age*. Association for Computing Machinery, 2010.
- [204] Christopher A. Wolters. “Regulation of Motivation: Evaluating an Underemphasized Aspect of Self-Regulated Learning”. In: *Educational Psychologist* 38.4 (Dec. 1, 2003), pp. 189–205. DOI: 10.1207/S15326985EP3804_1.
- [205] Xinming Zhu and Herbert A. Simon. “Learning Mathematics From Examples and by Doing”. In: *Cognition and Instruction* 4.3 (1987), pp. 137–166. DOI: 10.1207/s1532690xc10403_1.
- [206] Stuart Zweben and Betsy Bizot. *2015 Taulbee Survey: Continued Booming Undergraduate CS Enrollment; Doctoral Degree Production Dips Slightly*. Computing Research Association, pp. 2–60.

Appendix A

Initial Puzzle Interface

Figures A.1–A.8 show the puzzle interface and mechanics developed during the first formative evaluation presented in Chapter 2.

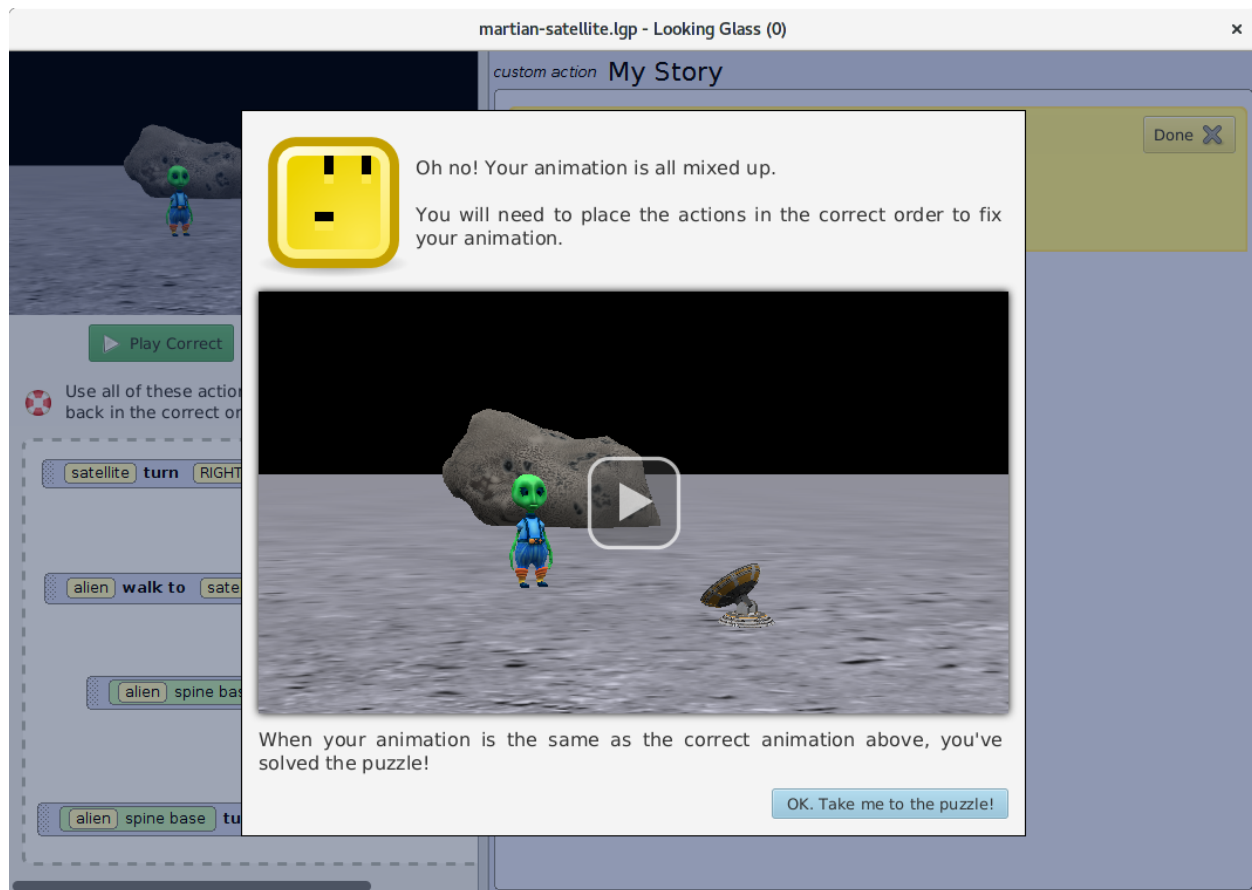


Figure A.1: Upon opening a puzzle, participants are presented with the directions pane.

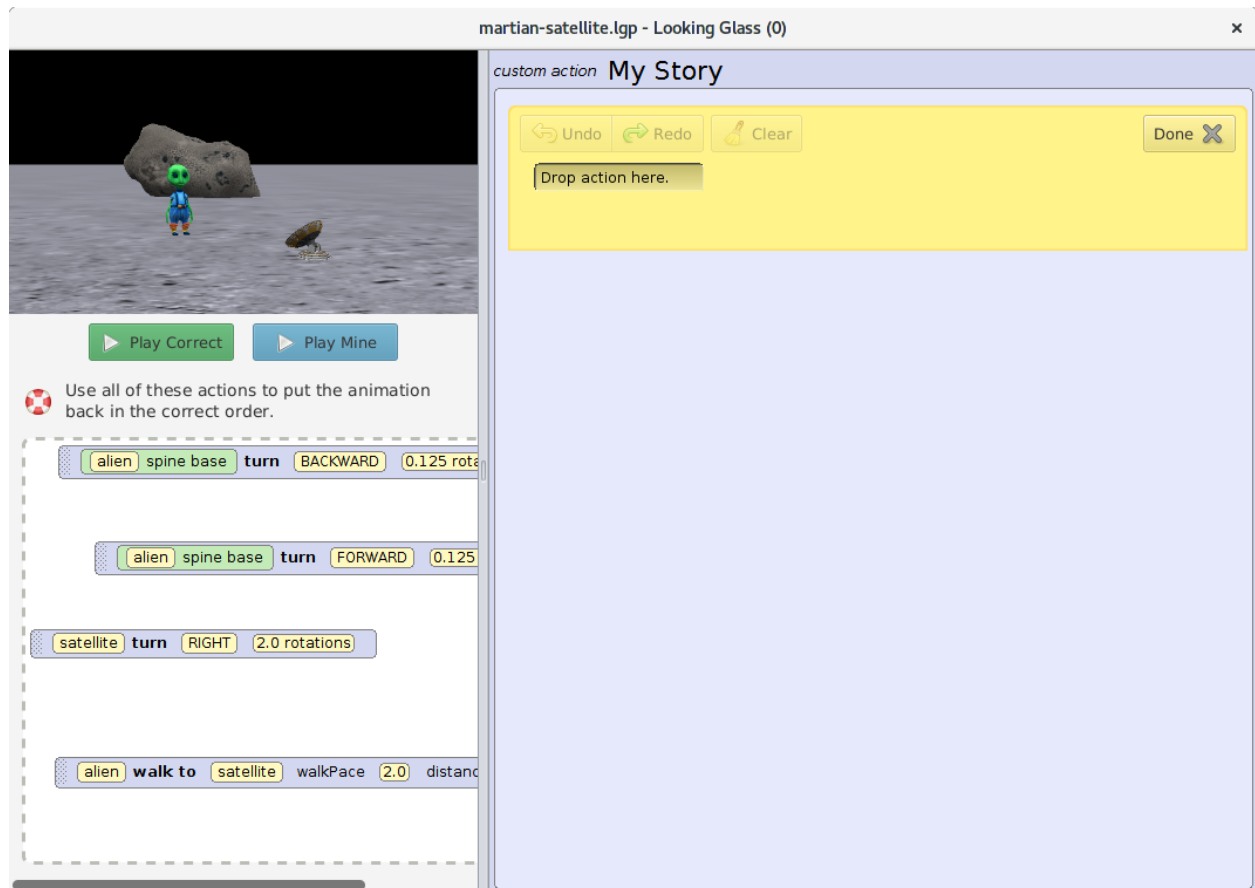


Figure A.2: After closing the directions pane, this is the state of the puzzle interface. Initially, all puzzle statements are located in the bin on the left side of the interface.

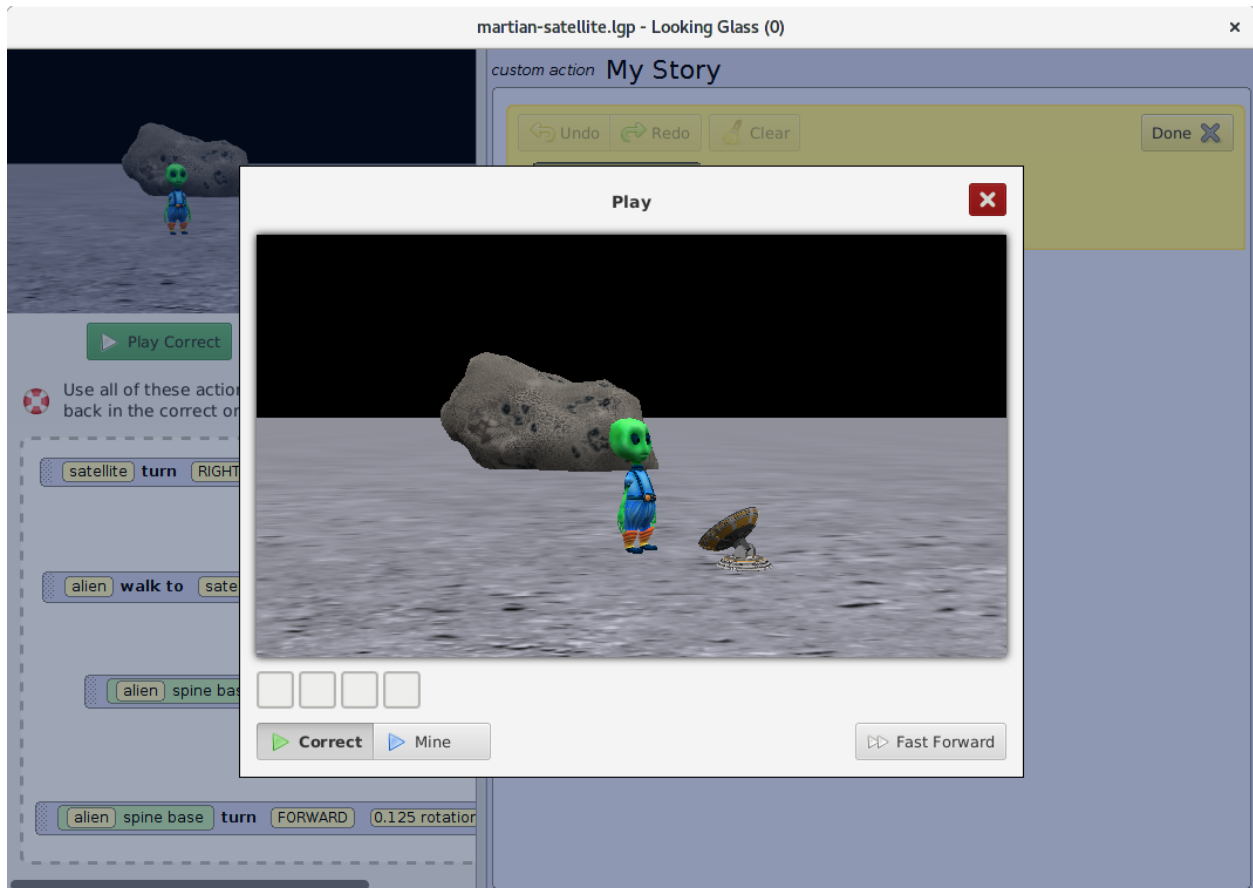


Figure A.3: When users click the *Play Correct* button, this pane will play the correct animation. Notice that the feedback indicator for the correct animation only shows white squares; there is no colored feedback for playing the correct animation.

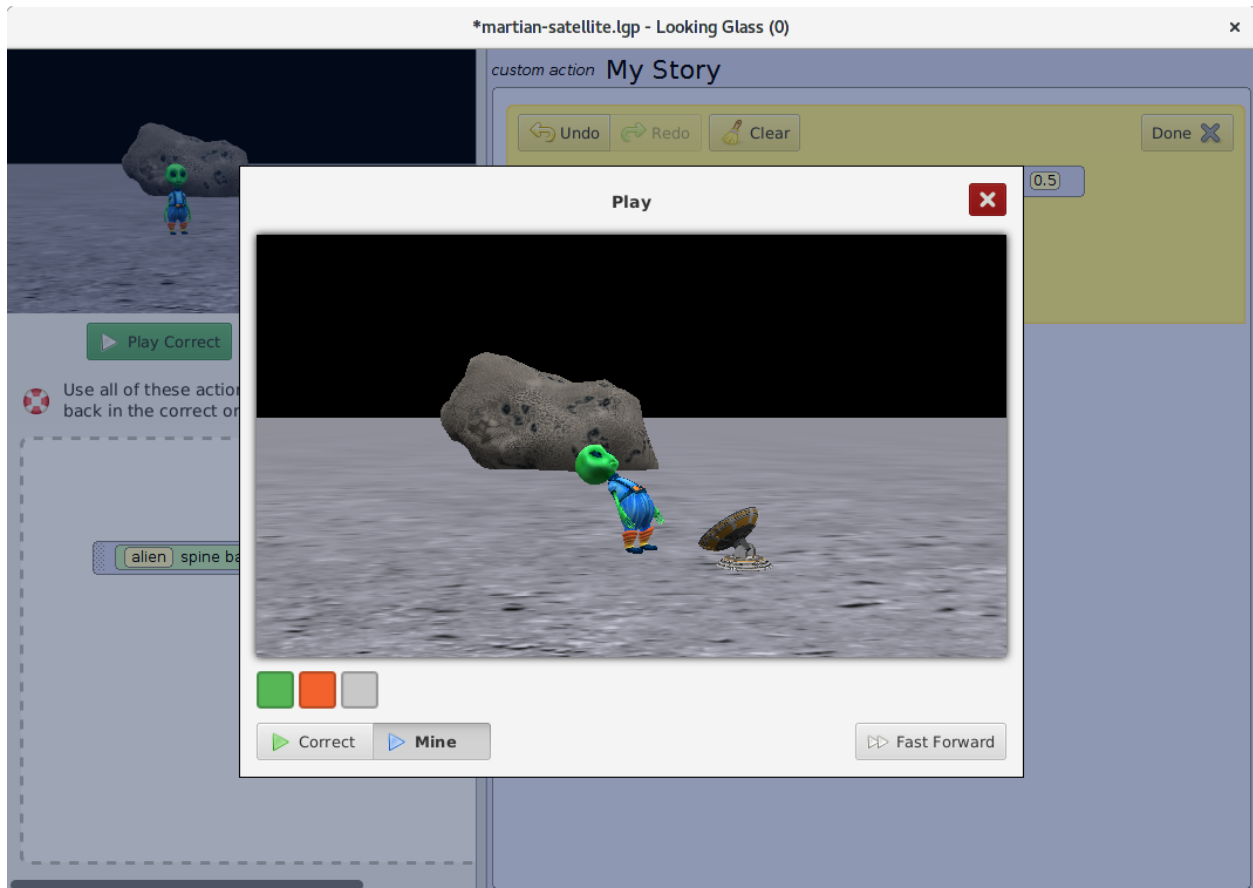


Figure A.4: When a user clicks the *Play Mine* button, this pane will play the user's animation, which is the current state of the puzzle. If the animation is not yet correct, the feedback indicator shows orange and gray squares.

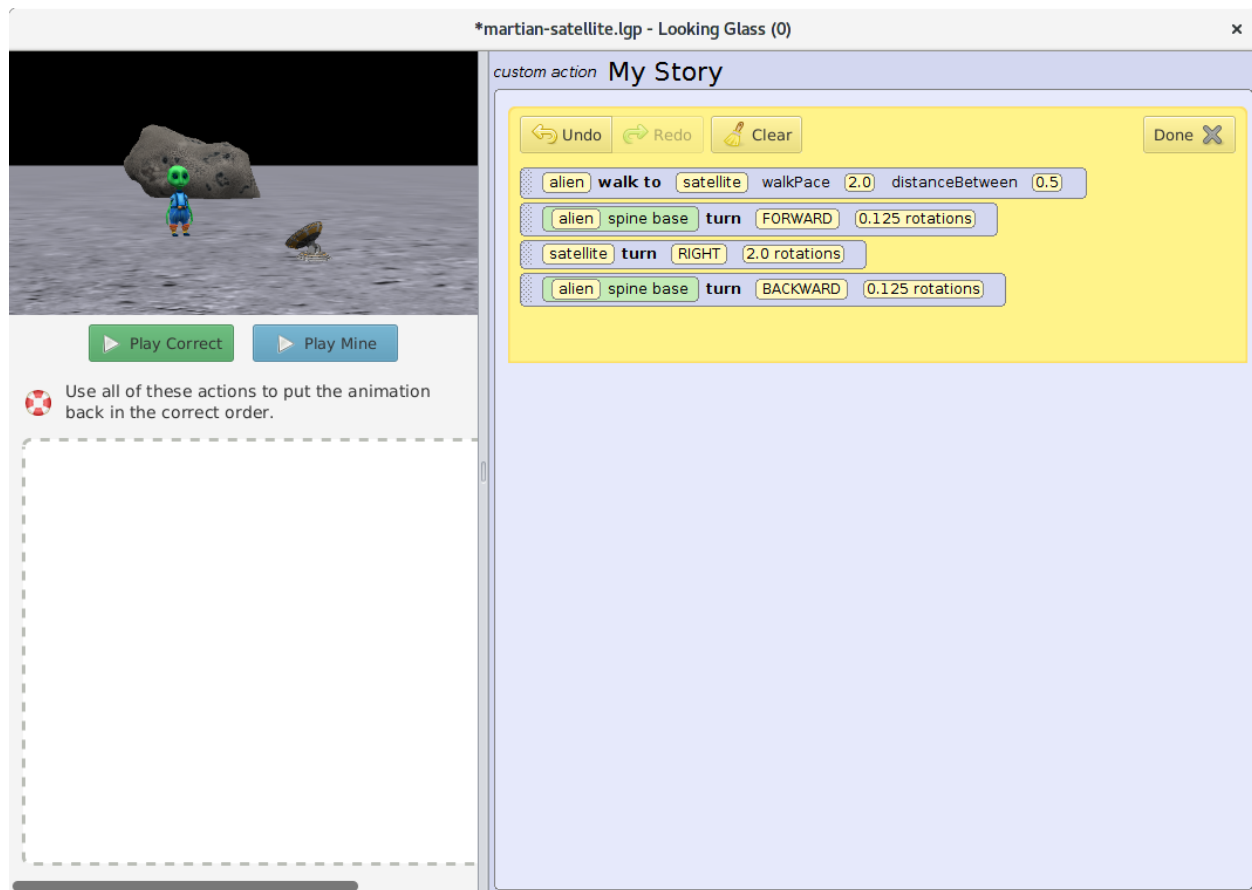


Figure A.5: A correctly completed puzzle.

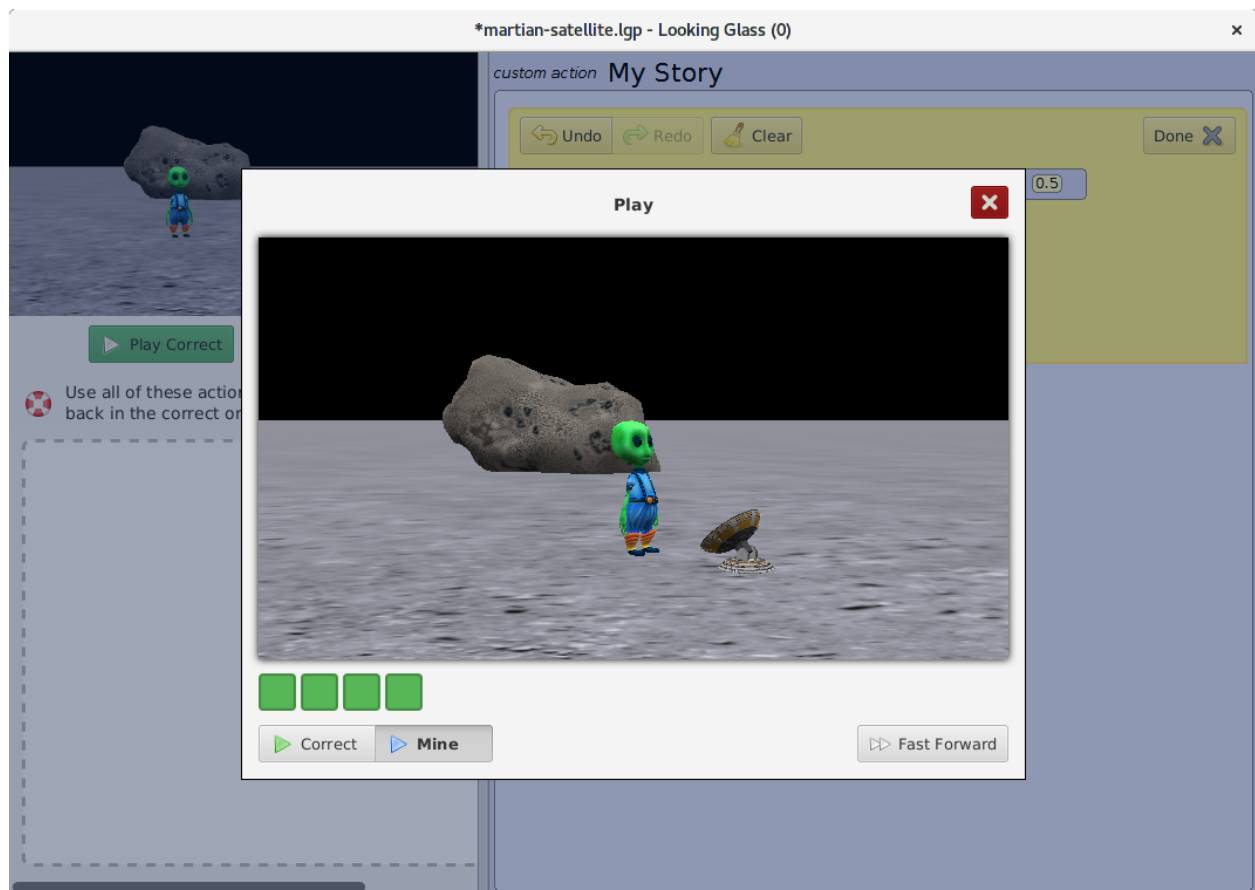


Figure A.6: When the user plays their animation, and it is correct, then the feedback indicator shows all green dots.

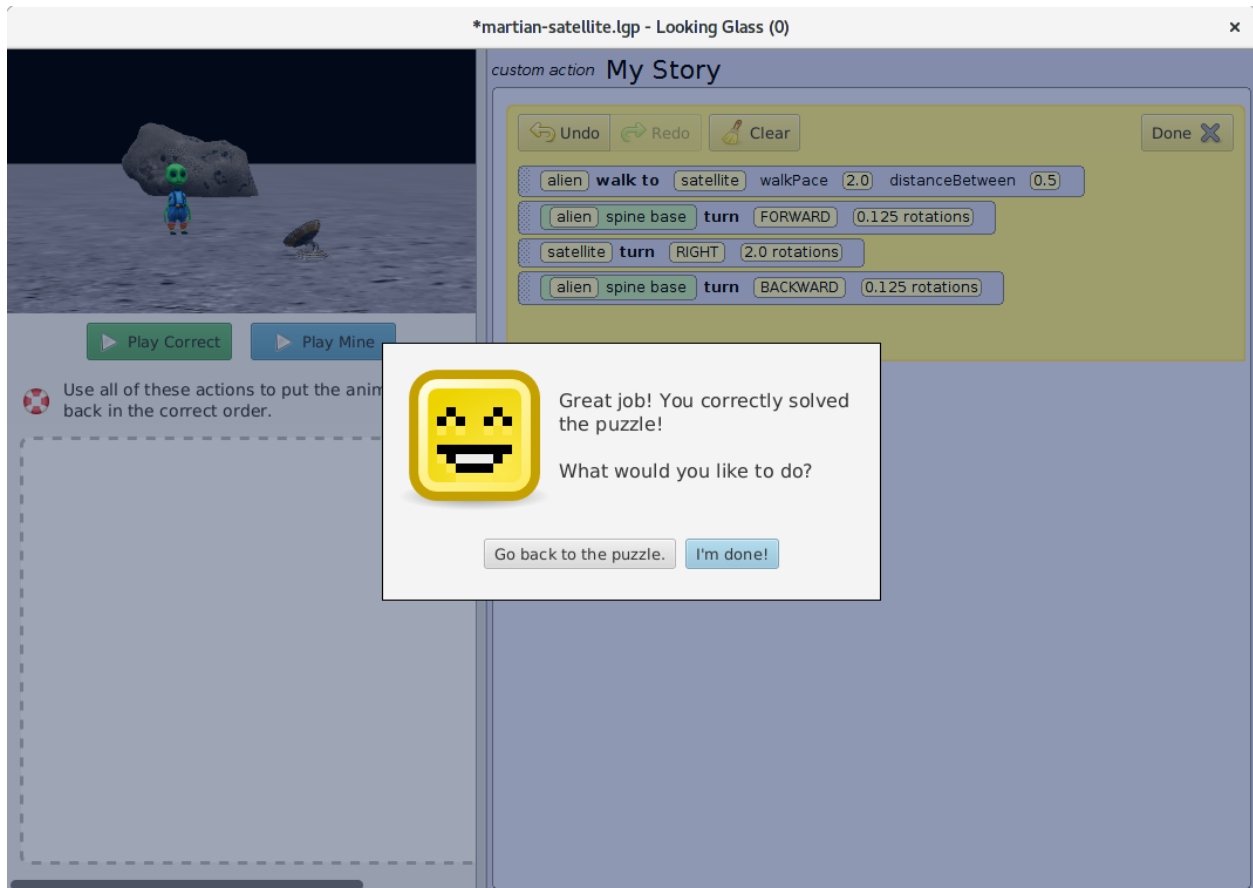


Figure A.7: When the user closes the play animation pane after correctly completing the puzzle, the user is shown this feedback. They also receive this feedback if they click the *Done* button when the puzzle is correct.

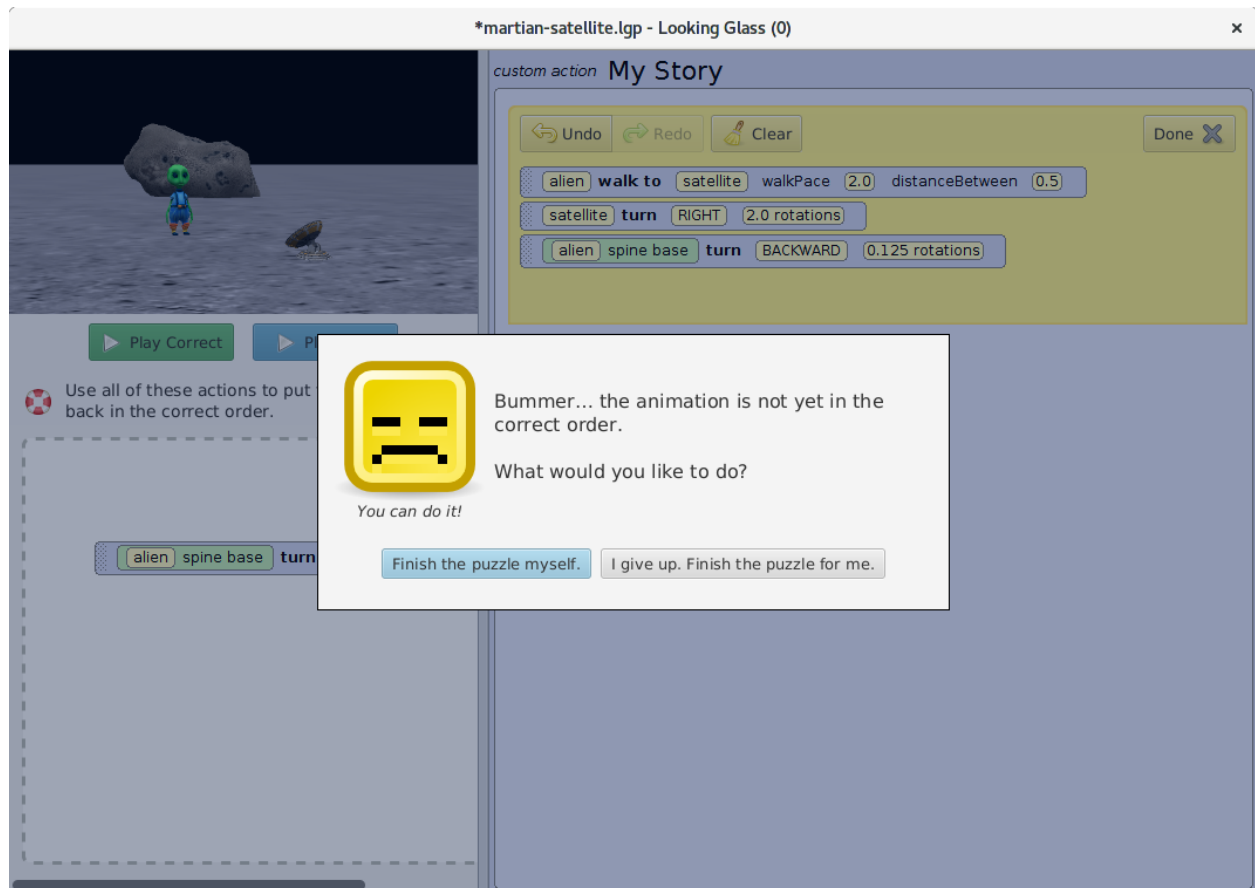


Figure A.8: If the user clicks the *Done* button and the puzzle is not yet correct, they are shown this dialog to encourage them to keep working on solving the puzzle.

Appendix B

Initial Puzzle Curriculum

Table B.1 and Figures B.1–B.7 show the initial puzzle curriculum developed during the second formative evaluation presented in Chapter 2.

Table B.1: The initial code puzzle curriculum contains six puzzles with an additional introductory puzzle which emphasizes puzzle mechanics.

Puzzle	Title	Programming Constructs
0	Satellite Inspection	<i>puzzle mechanics</i>
1	Dizzy Walrus	Do in Order
2	Monkey Business	Repeat
3	The Spooky Trick	Do Together
4	Interstellar Travel Troubleshooting	Repeat & Do Together
5	Shark Snack	Do Together { Repeat }
6	Whack-a-Yeti	Repeat { Do Together }

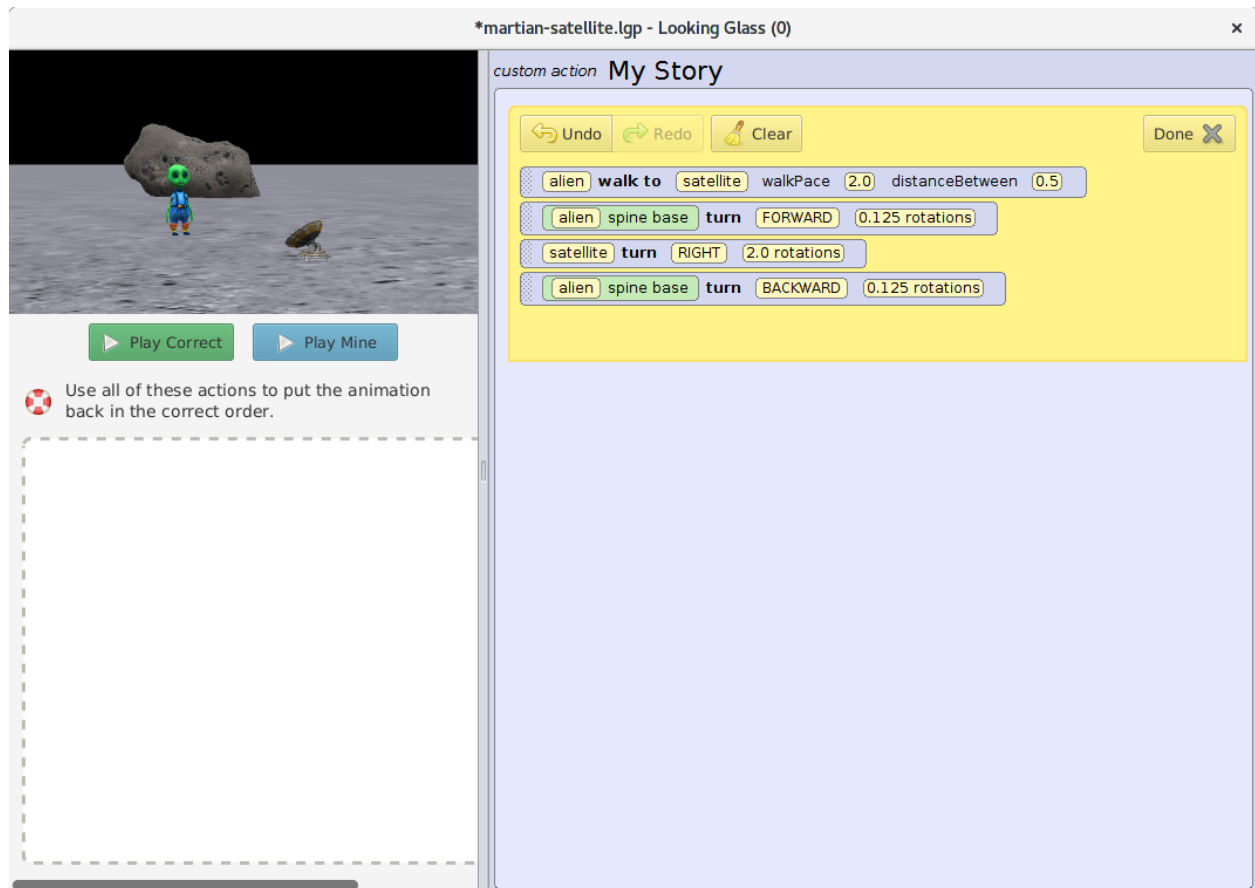


Figure B.1: *Satellite Inspection* (0) is the introductory puzzle that encourages users to become familiar with the puzzle interface and the mechanics of solving a puzzle.



Figure B.2: The *Dizzy Walrus* (1) puzzle introduces sequential execution (*Do in Order*).



Figure B.3: The *Monkey Business* (2) puzzle introduces learners to the *Repeat* programming construct.

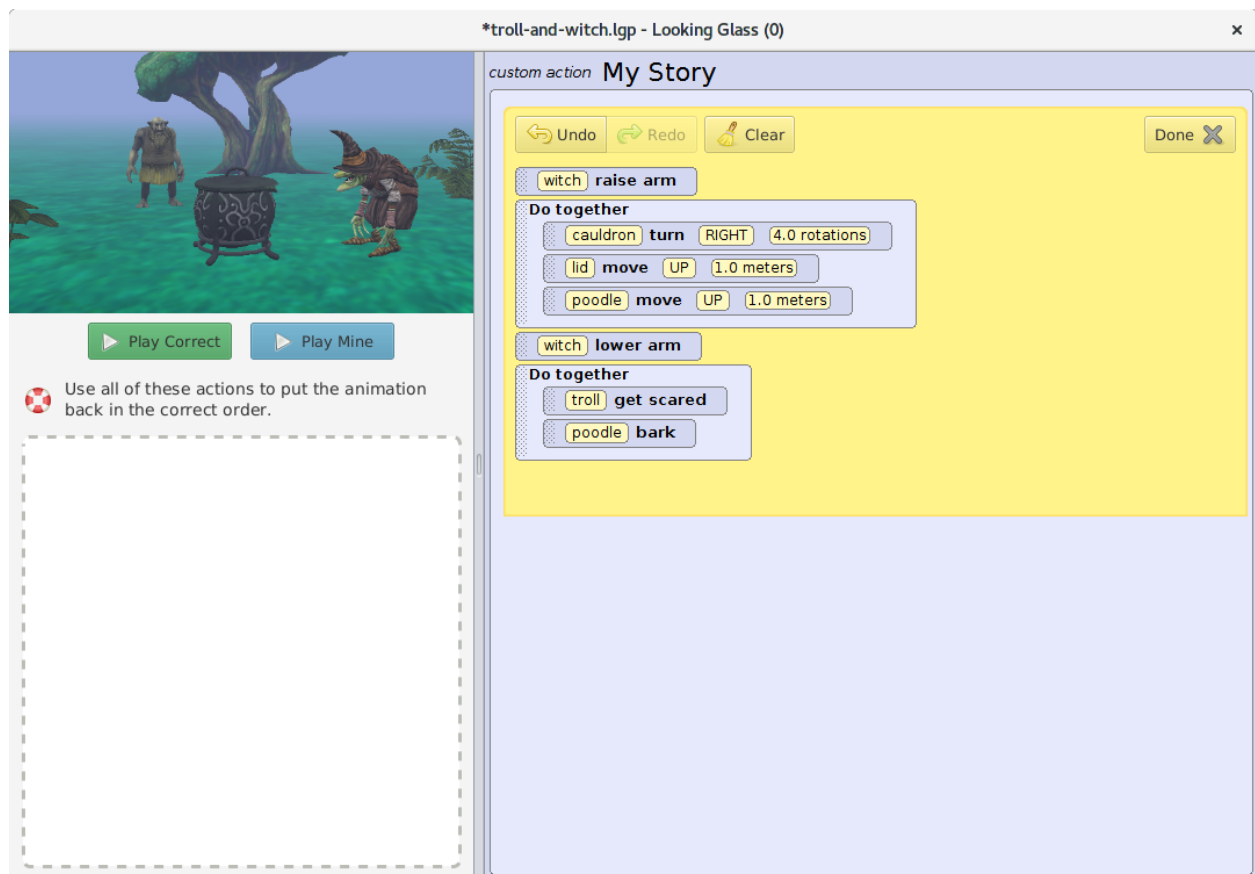


Figure B.4: The *Spooky Trick* (3) puzzle introduces learners to the *Do Together* programming construct.

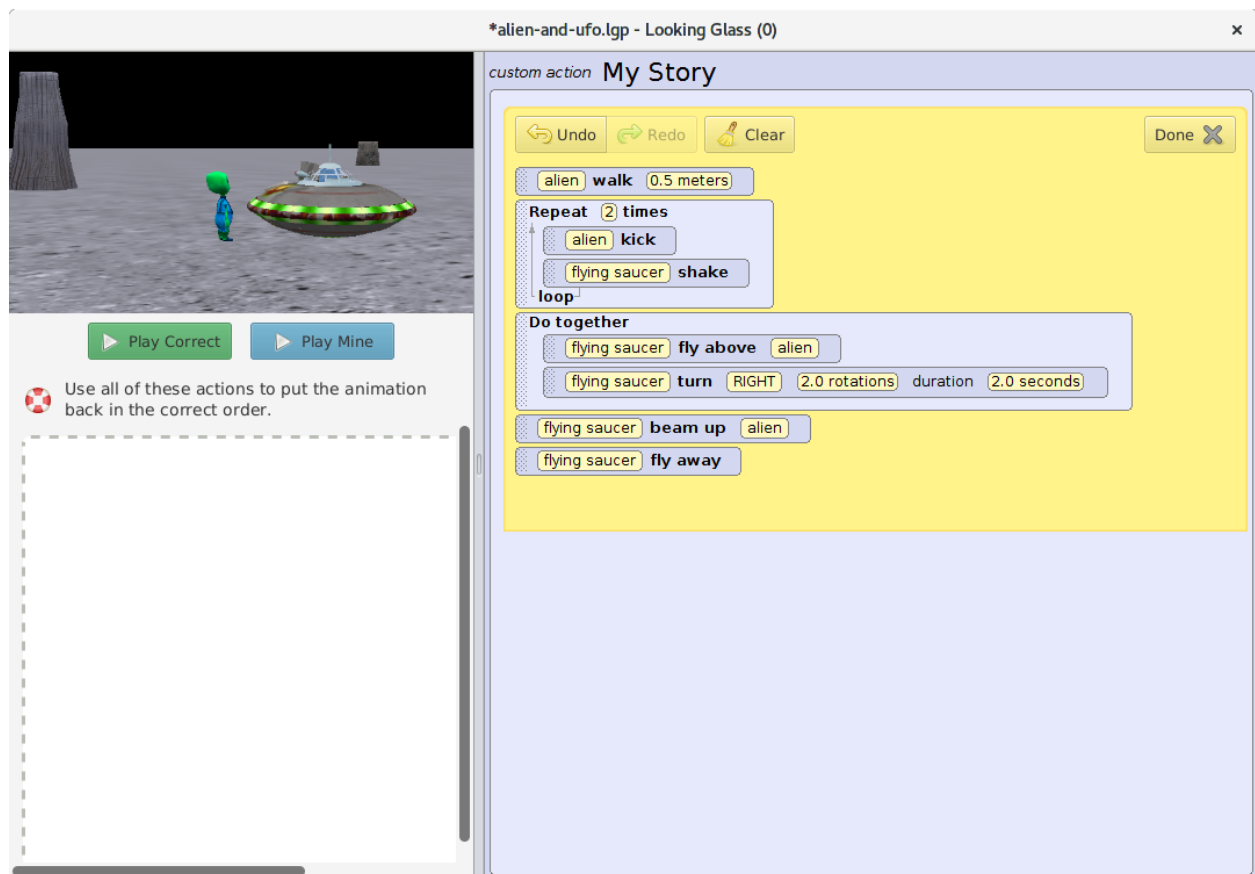


Figure B.5: The *Interstellar Travel Troubleshooting* (4) puzzle uses both the *Repeat* and *Do Together* programming constructs in one puzzle.

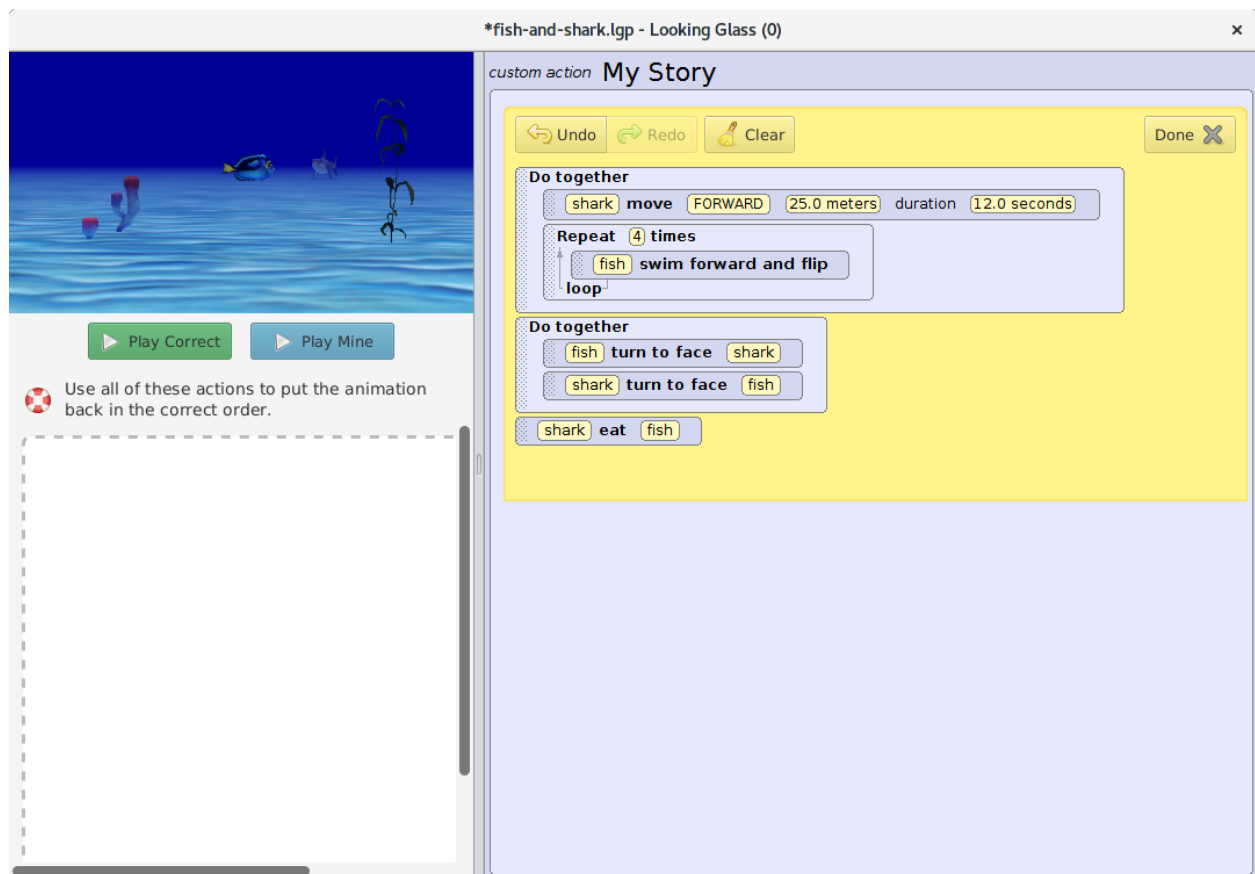


Figure B.6: The *Shark Snack* (5) puzzle exposes learners to nesting a *Repeat* block within a *Do Together* block.

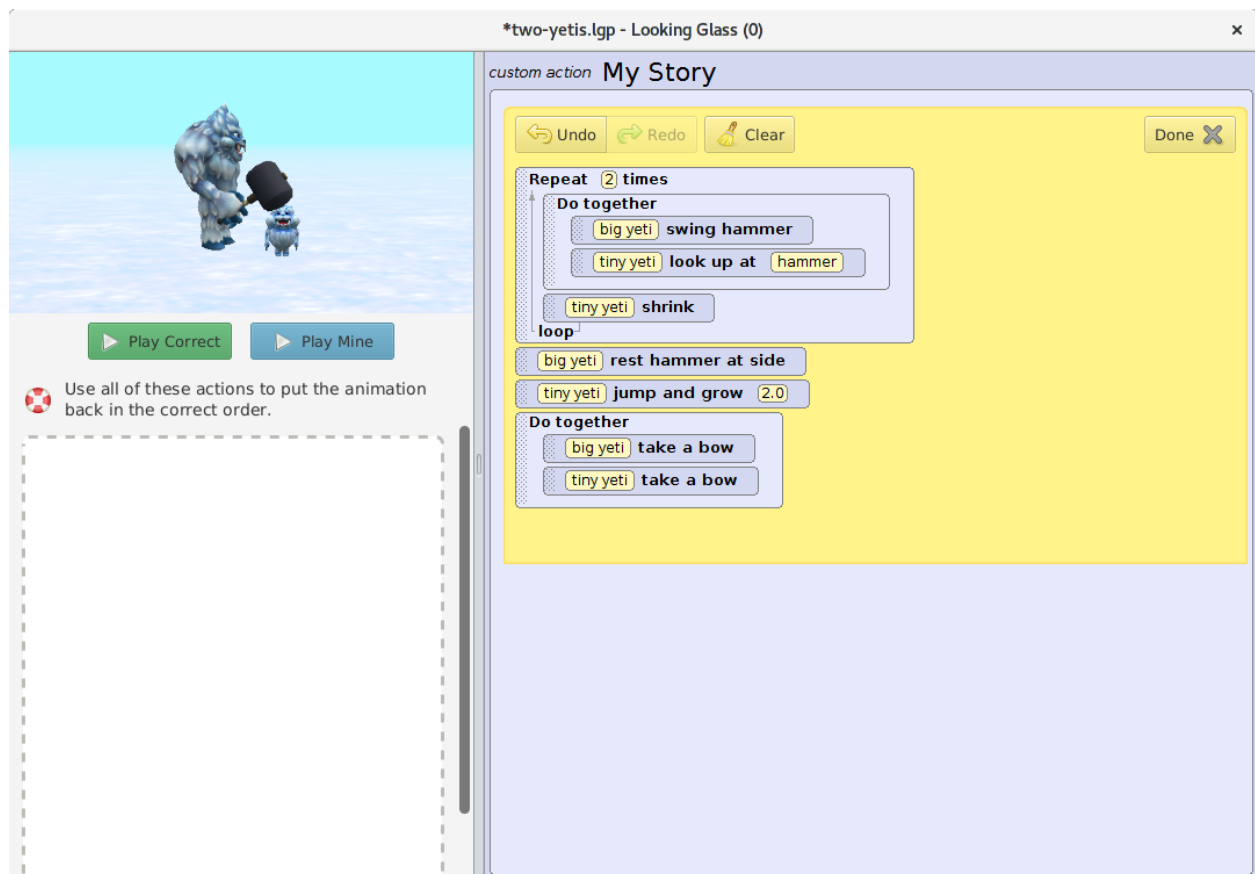


Figure B.7: The *Whack-a-Yeti* (6) puzzle exposes learners to nesting a *Do Together* block within a *Repeat* block.

Appendix C

Revised Puzzle Interface

Figures C.1–C.11 show the revised puzzle interface presented in Chapter 2. The redesign of the interface was done by Danielle Clemons. This version of the puzzle interface is currently available within the Looking Glass programming environment. To download Looking Glass, please visit <https://lookingglass.wustl.edu/>. The source code for this revision is also available at <https://github.com/lookingglass-coding/lookingglass-ide> or <https://github.com/harmsk/puzzle-lookingglass>.

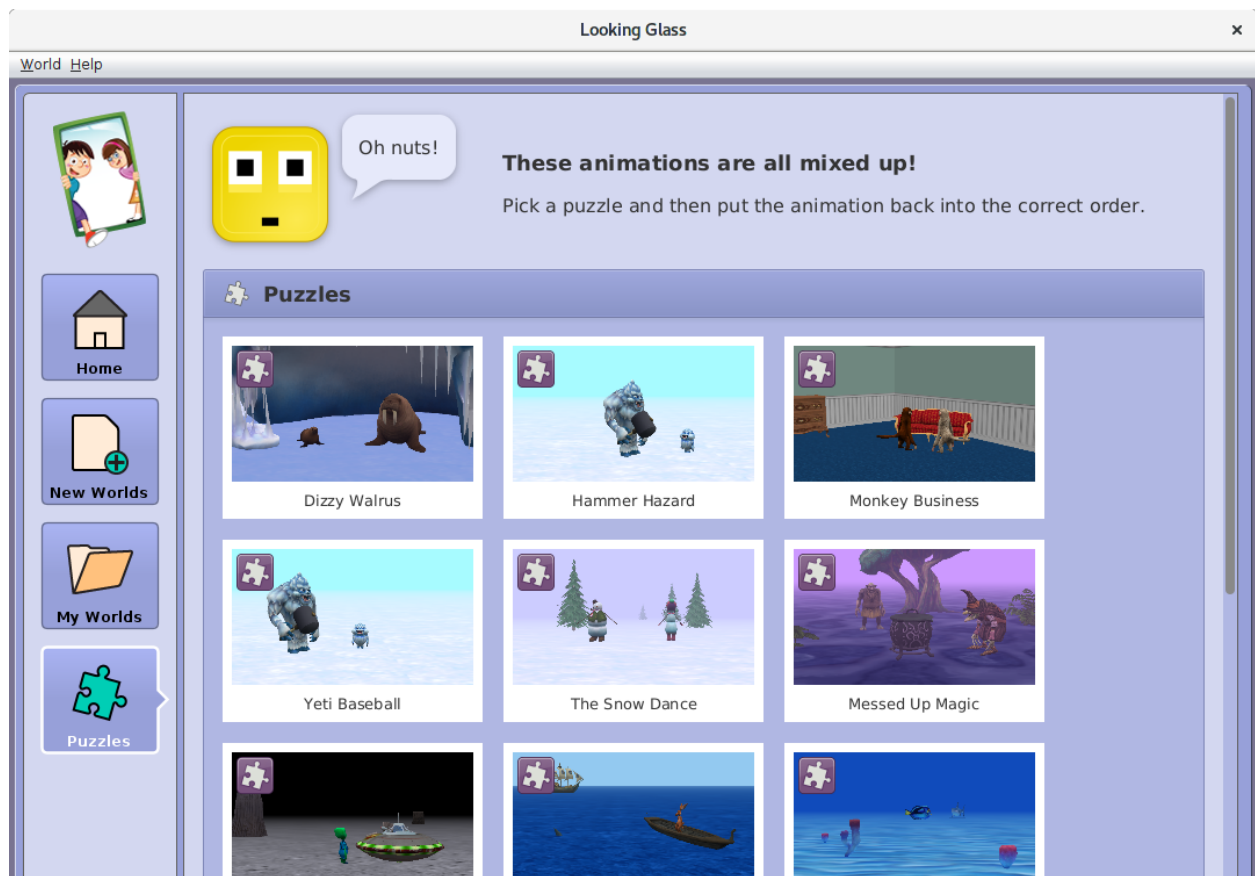


Figure C.1: In the activities panel of Looking Glass, users can now select puzzles as an activity alongside programming their own 3D animations.

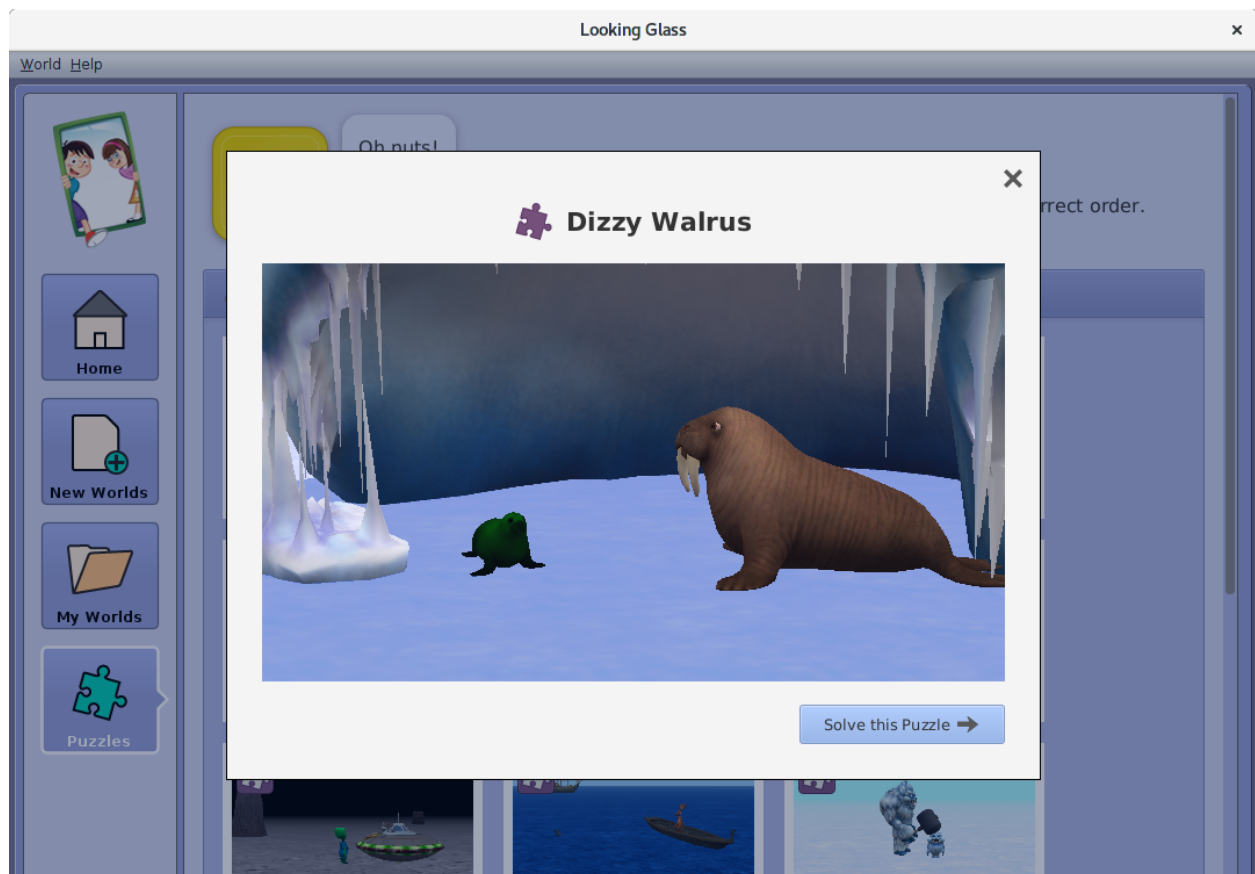


Figure C.2: After selecting a puzzle in the puzzle selection pane (Figure C.1), users are shown a preview of the animation before they decide to work on that puzzle.

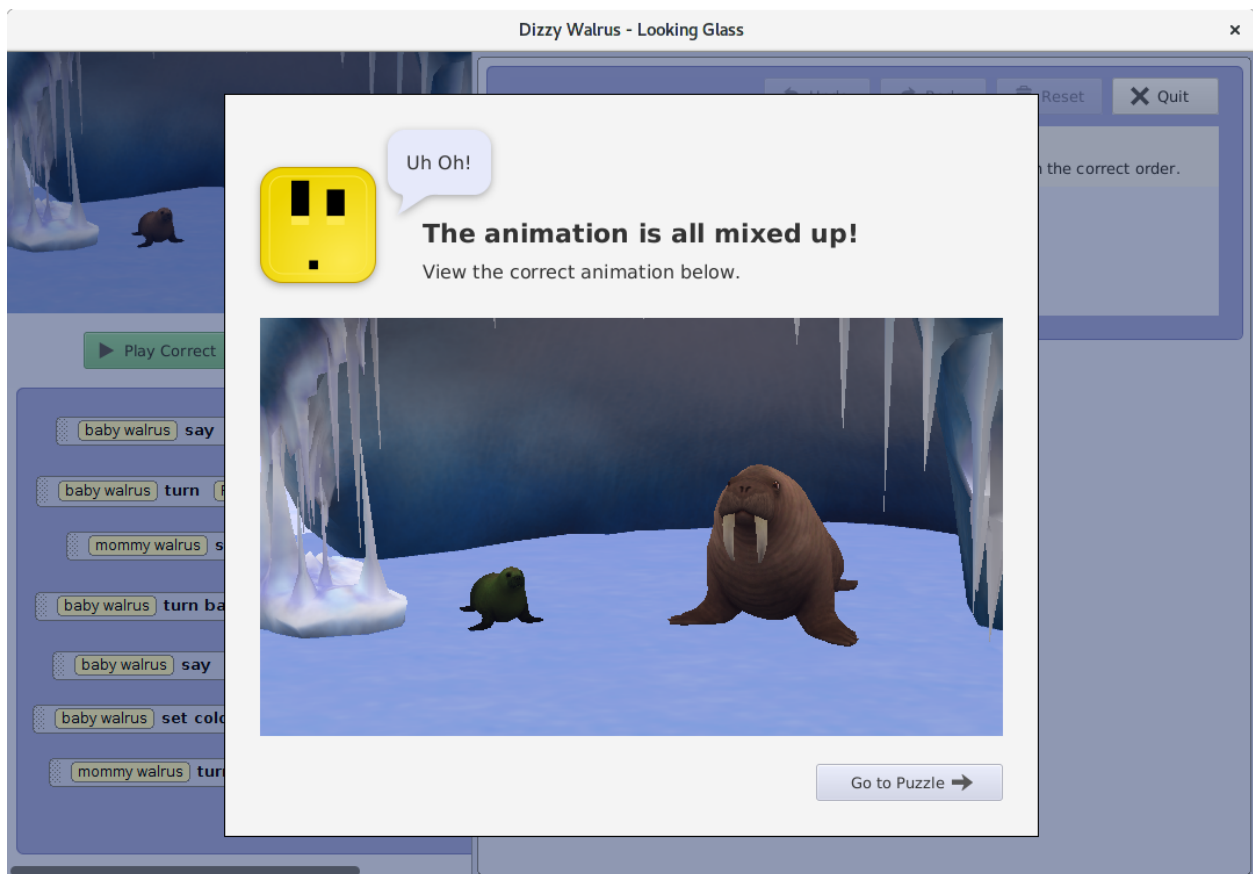


Figure C.3: After choosing to work on a puzzle, users first see the directions pane within the puzzle interface.

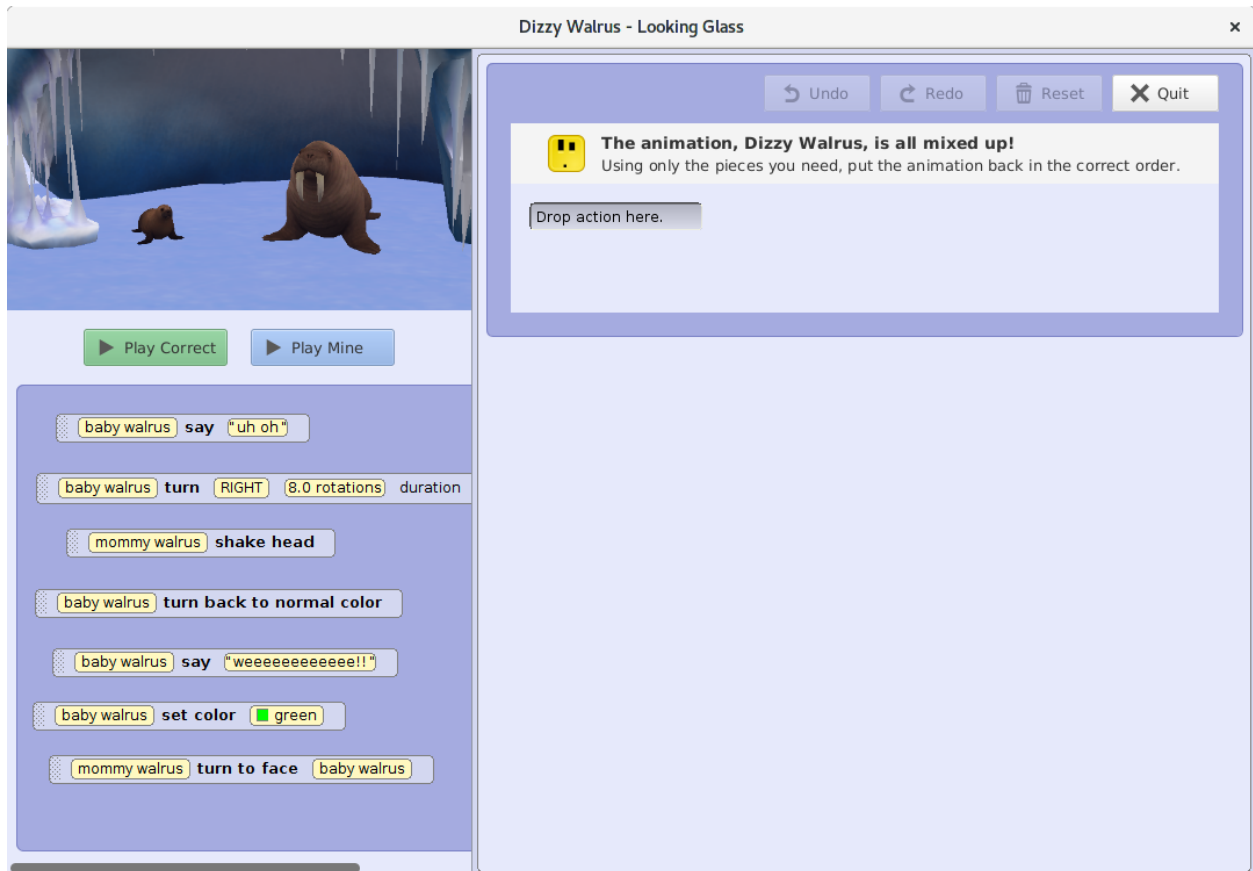


Figure C.4: This is the initial state of the puzzle after closing the directions pane.

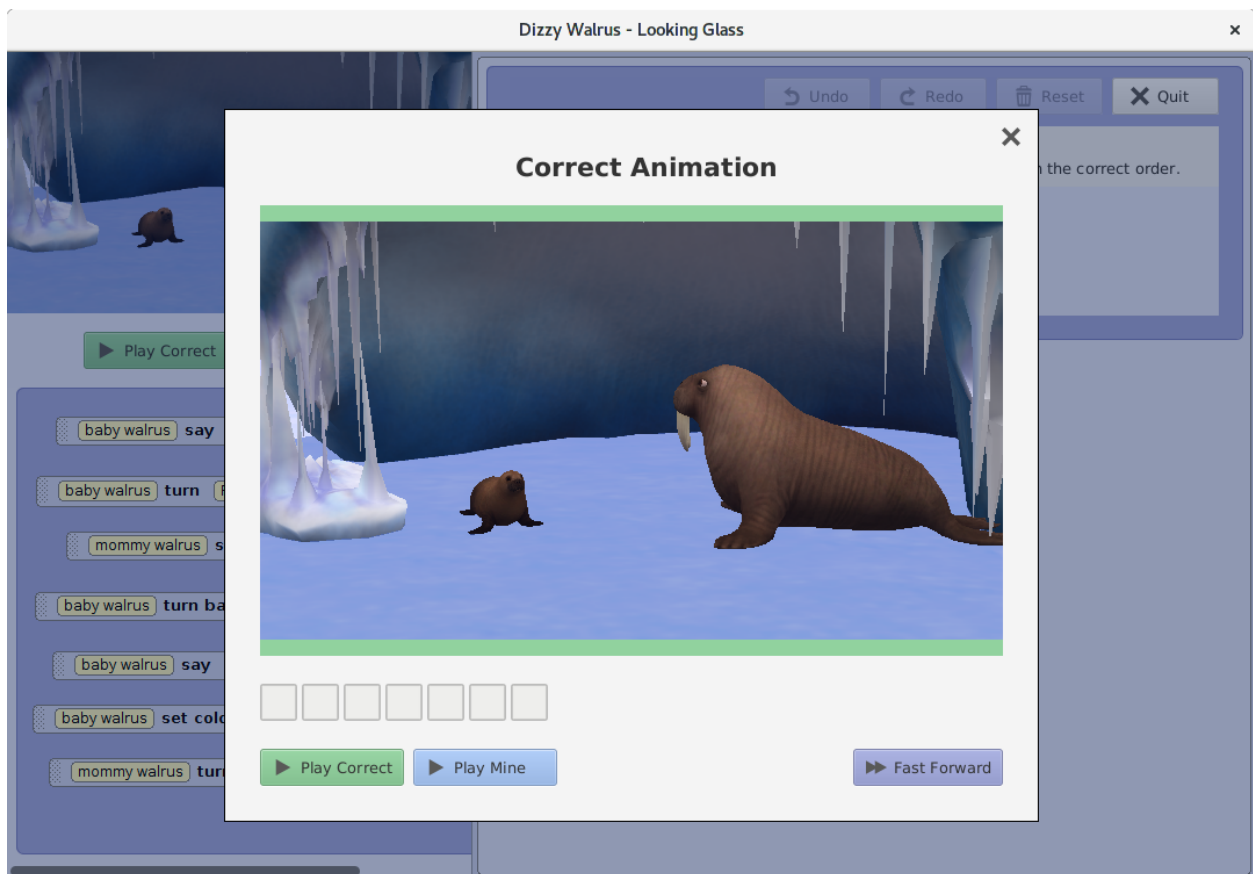


Figure C.5: When users click the *Play Correct* button, the play pane executes the correct animation. Notice that the feedback indicator shows all white squares for each executed statement.

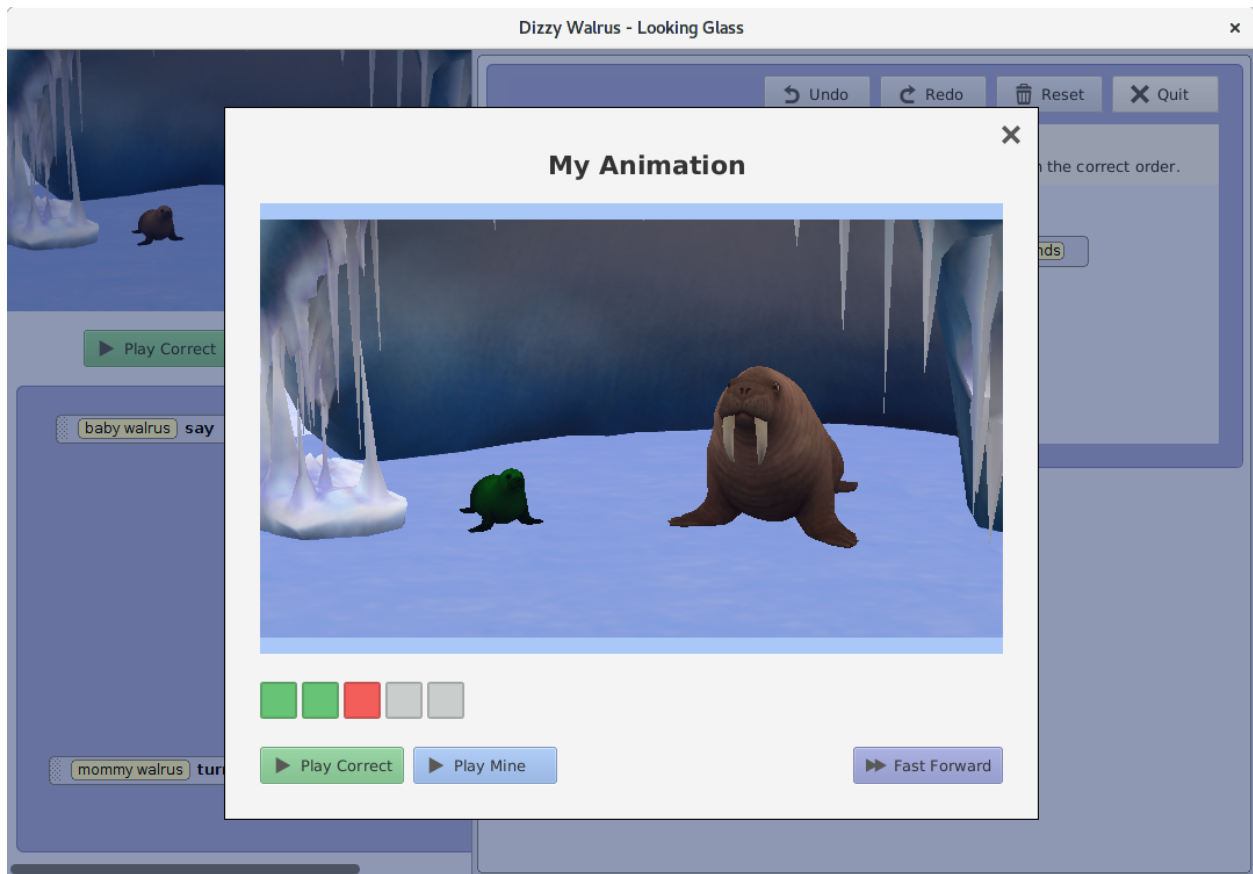


Figure C.6: After clicking the *Play Mine* button, the user can watch the output of the current state of their puzzle. Notice that the feedback indicator shows that something is wrong with the user's current solution due to the presence of the orange dot.

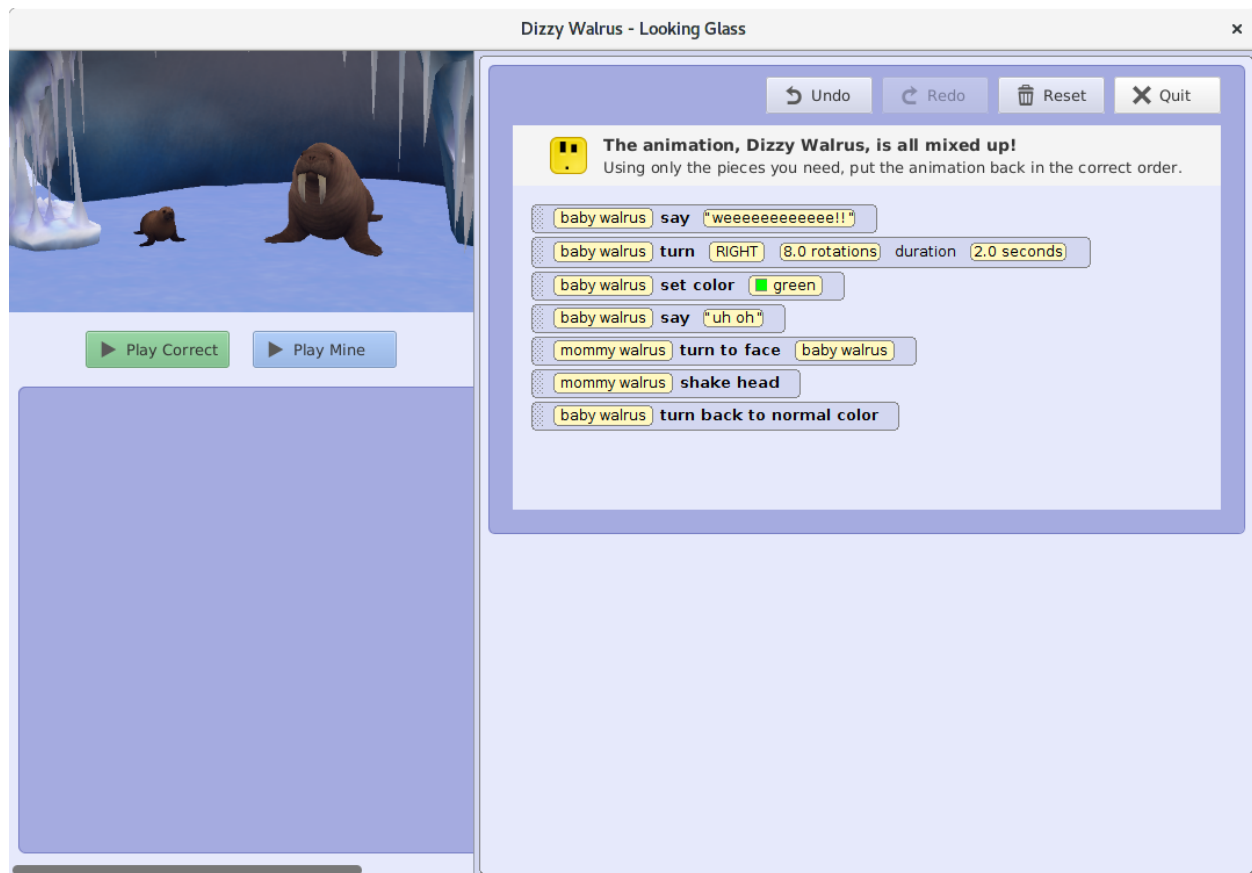


Figure C.7: A successfully completed puzzle.

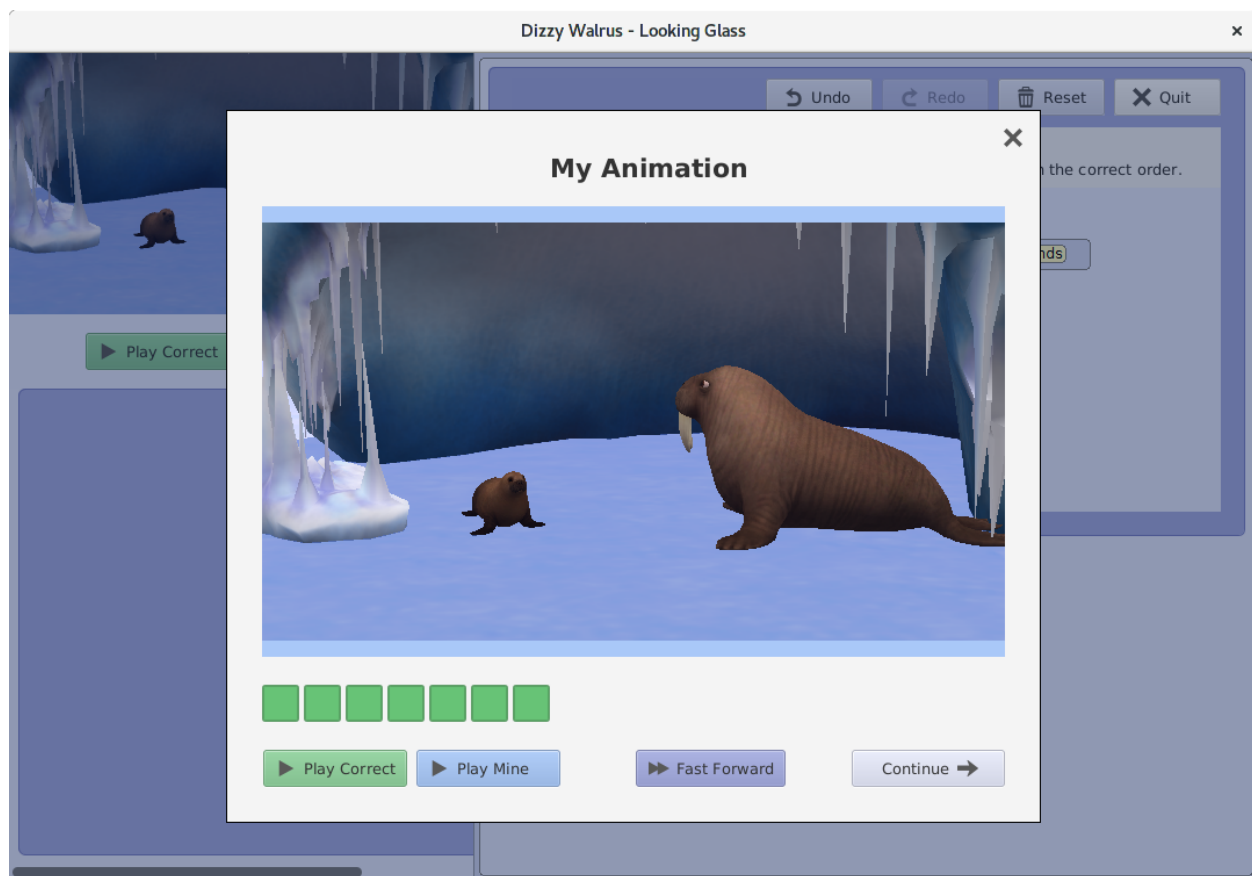


Figure C.8: After successfully completing the puzzle and clicking the *Play Mine* button, users get feedback that shows that every statement in their program is in the correct location. Notice that after all the feedback squares are green, a new *Continue* button is added to the play pane. The continue button will then bring up the correct dialog (Figure C.9).

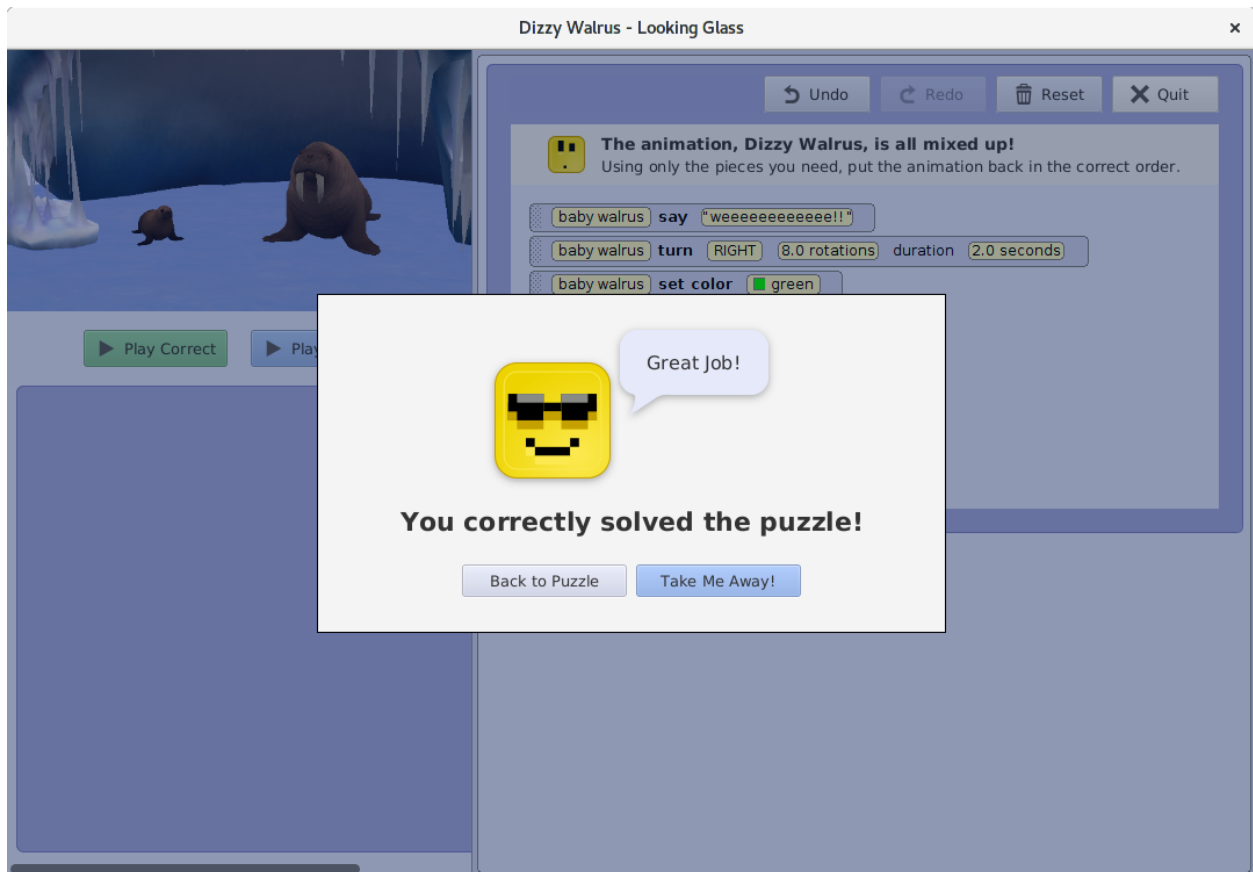


Figure C.9: If the puzzle is complete, users are shown this pane after closing the play pane or clicking on the *Quit* button.

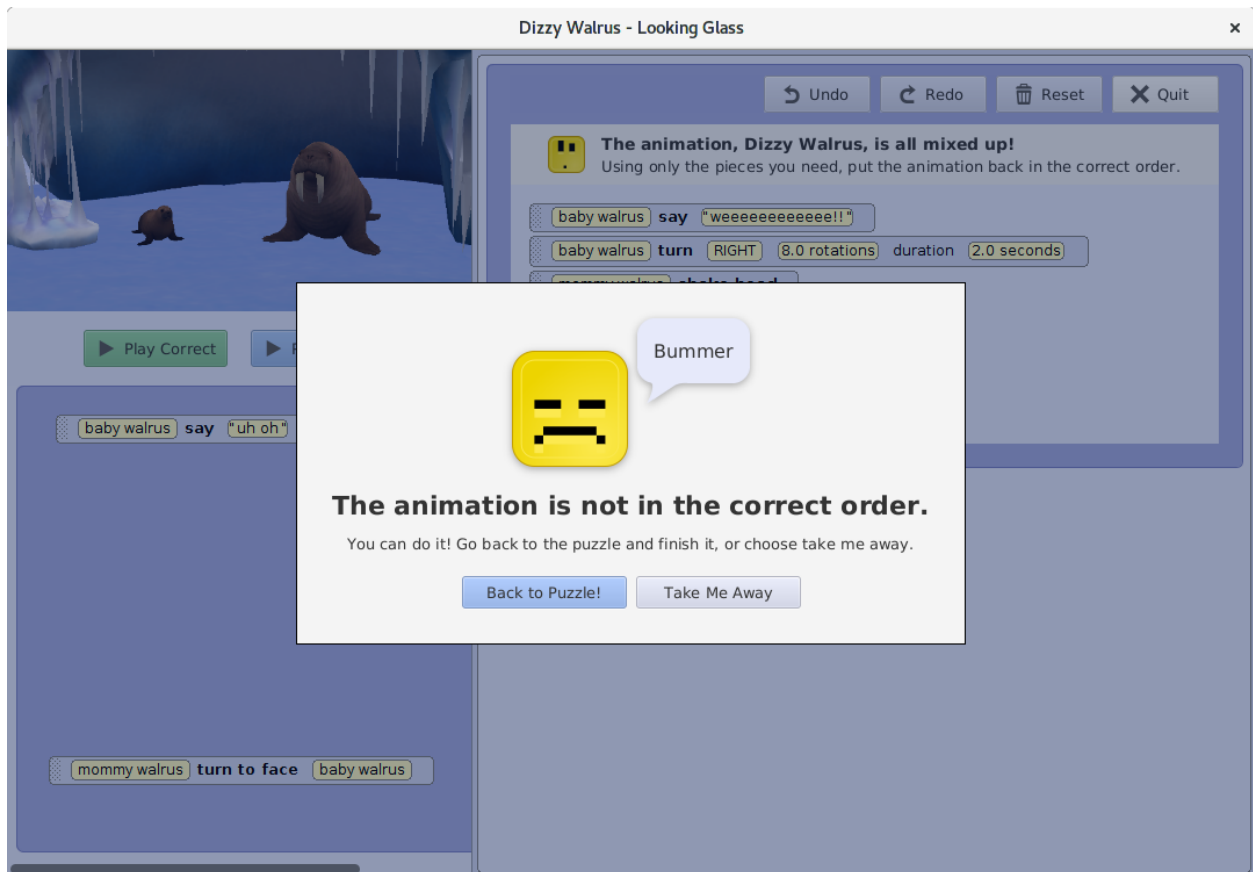


Figure C.10: If a user has not yet completed the puzzle and they click the *Quit* button, they are shown this dialog to encourage them to continue working towards the solution.

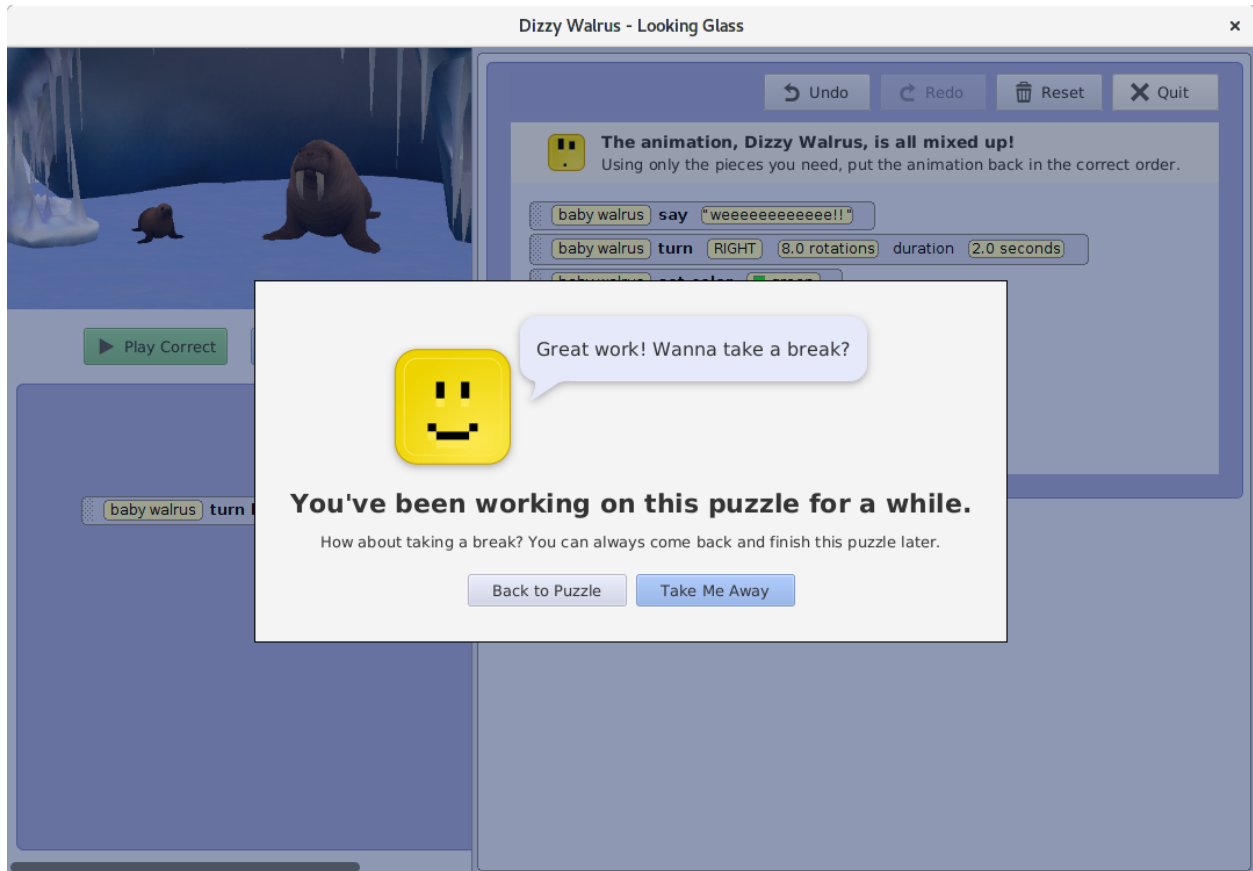


Figure C.11: Through additional testing we observed that many participants were very resistant to give up on a puzzle that they were having trouble solving. After 12 minutes this dialog is shown to users if they have not yet solved the puzzle. The dialog encourages participants to take a break in an effort to reduce any frustration that they might be experiencing.

Appendix D

Revised Puzzle Curriculum

Table D.1 and Figures D.1–D.14 show the revised puzzle curriculum presented in Chapter 2.

Table D.1: The revised code puzzle completion problem curriculum.

Puzzle	Title	Programming Constructs
1	Dizzy Walrus	Do in Order
2	Hammer Hazard	Repeat
3	Monkey Business	Repeat
4	Yeti Baseball	Do Together
5	The Snow Dance	Do Together
6	Messed Up Magic	Do Together
7	Interstellar Travel Troubleshooting ^a	Repeat & Do Together
8	Trouble at Sea	Repeat { Repeat }
9	Shark Snack	Do Together { Repeat }
10	Crazy Cauldron	Repeat { Do Together }
11	Whack-a-Yeti	Repeat { Do Together }
12	Firetruck Frenzy	Do Together { Do in Order }
13	Air Traffic	Do Together { Do in Order }
14	Polar Surprise	Do Together { Do in Order }

^a Later renamed *Space Mechanic* in the public release.

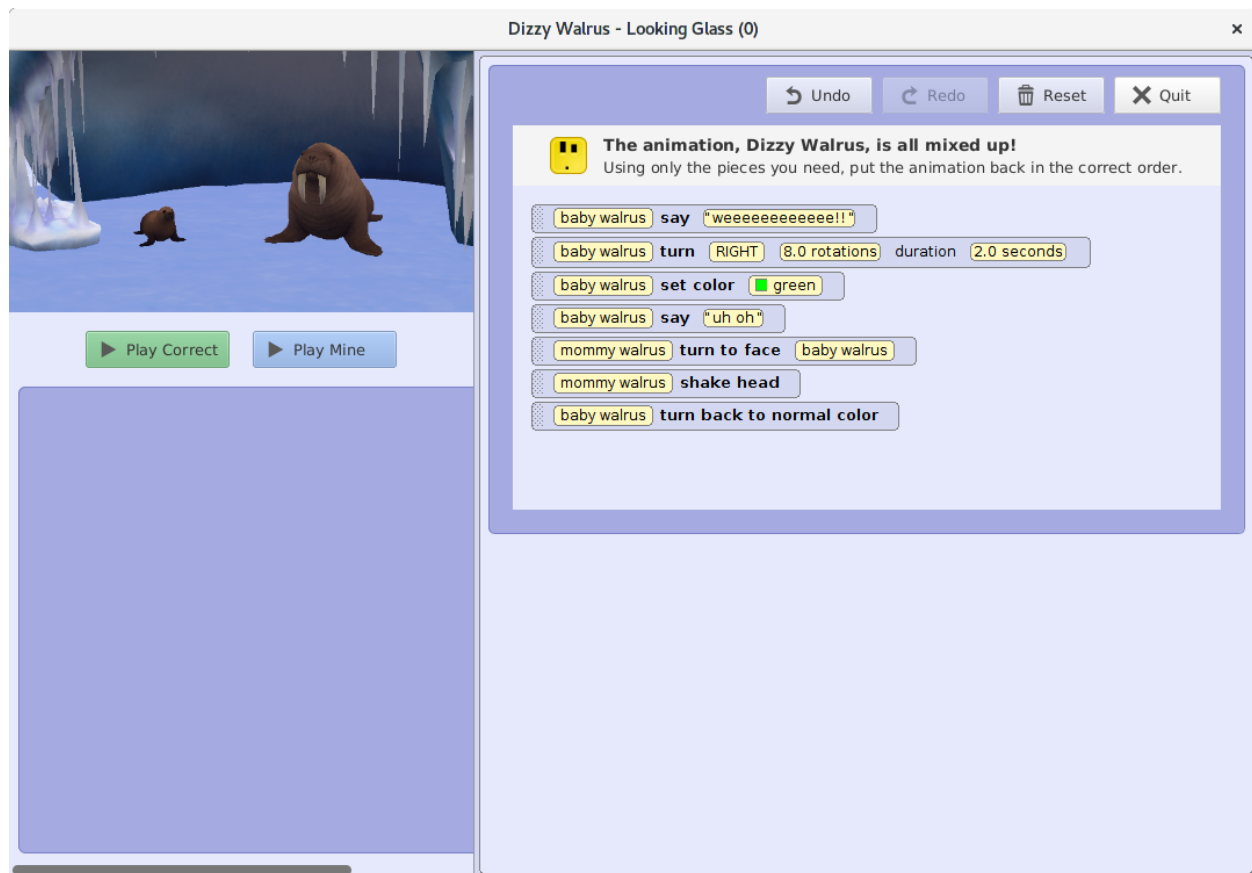


Figure D.1: The *Dizzy Walrus* (1) puzzle introduces learners to sequential execution.

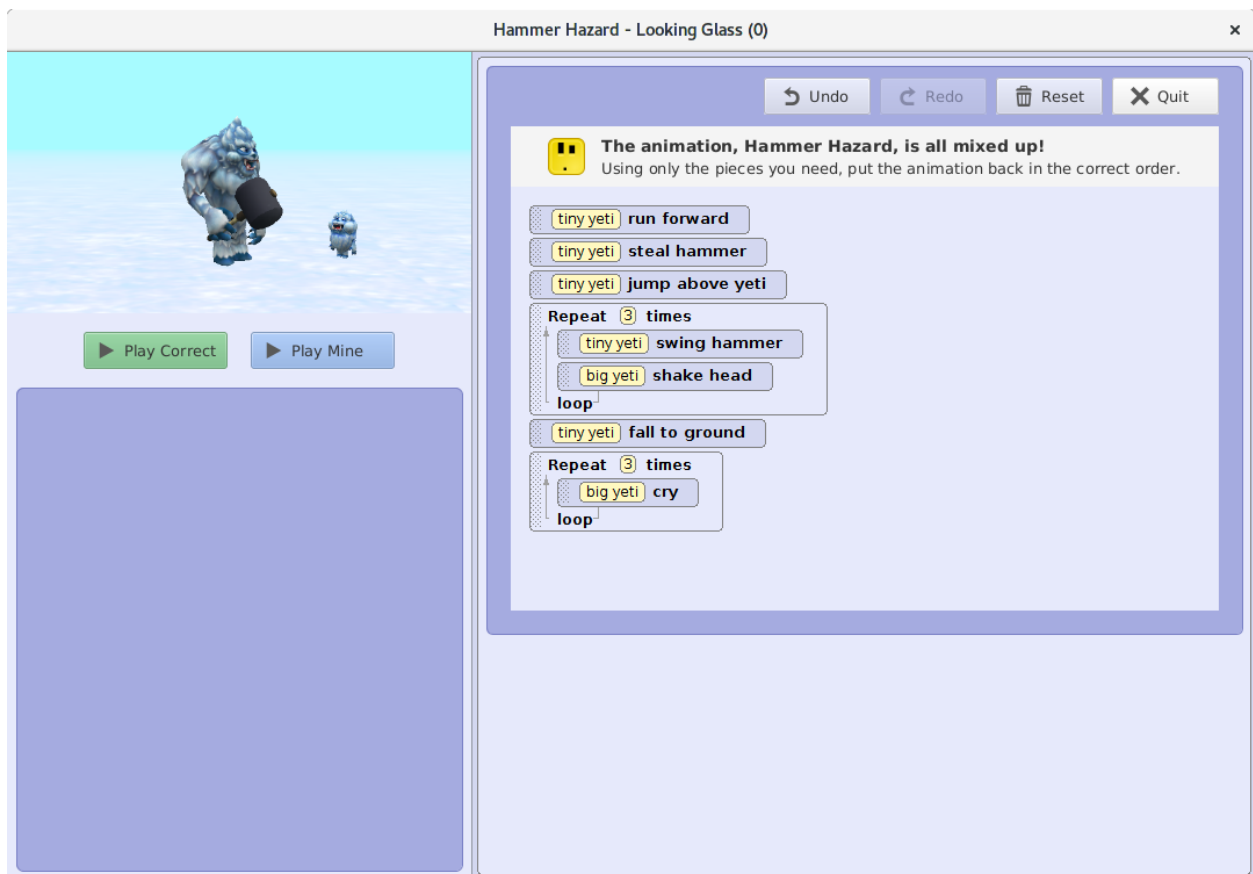


Figure D.2: The *Hammer Hazard* (2) puzzle introduces learners to the *Repeat* programming construct.

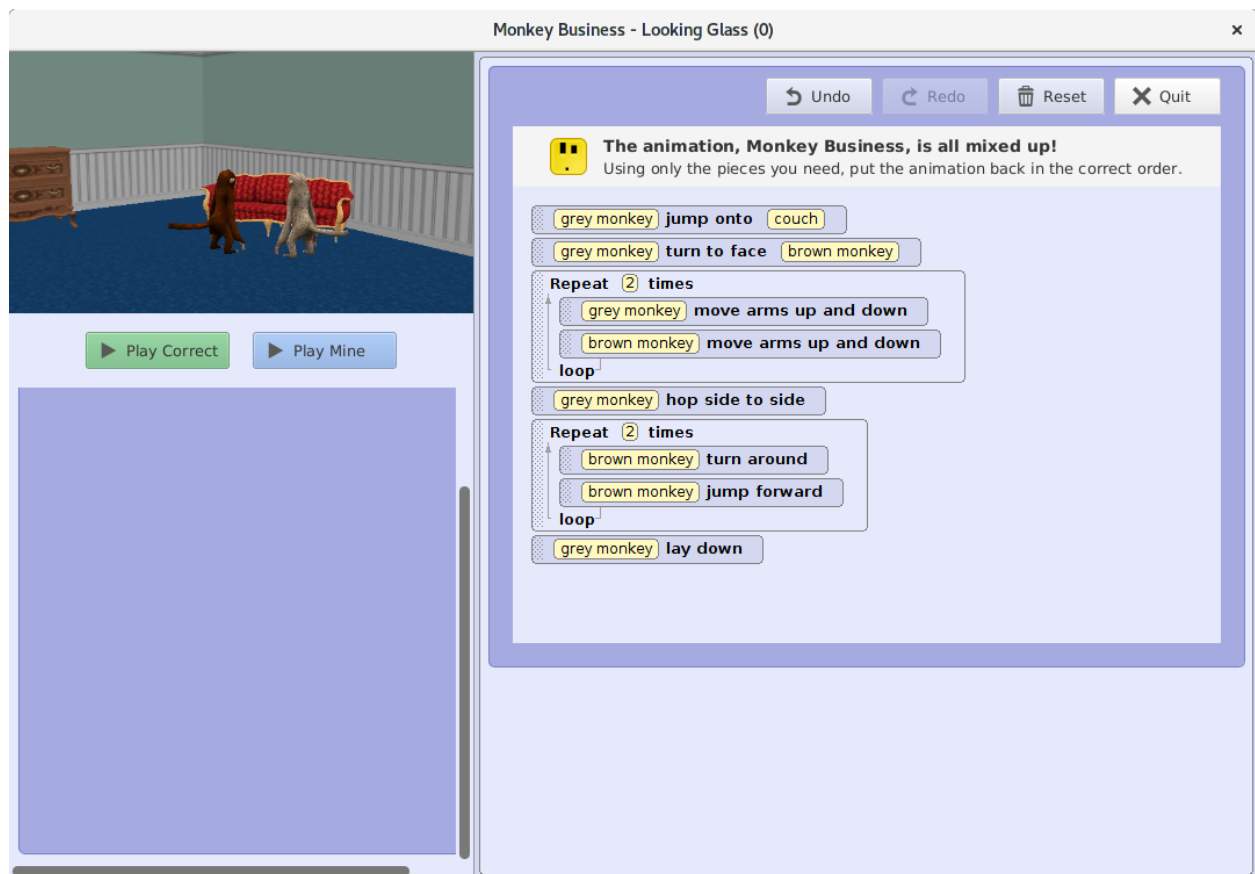


Figure D.3: The *Monkey Business* (3) puzzle enables learners to practice with the *Repeat* programming construct.

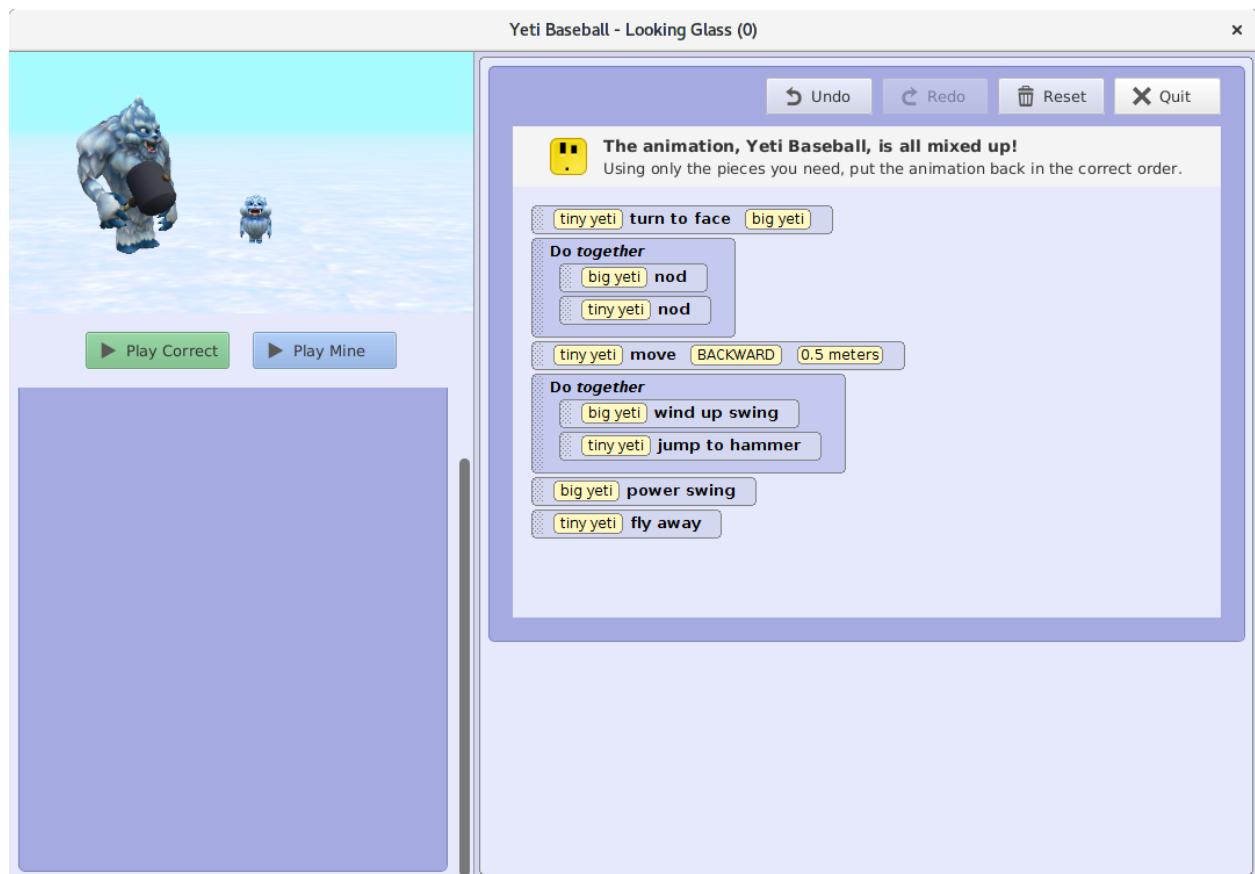


Figure D.4: The *Yeti Baseball* (4) puzzle introduces learners to the *Do Together* programming construct.

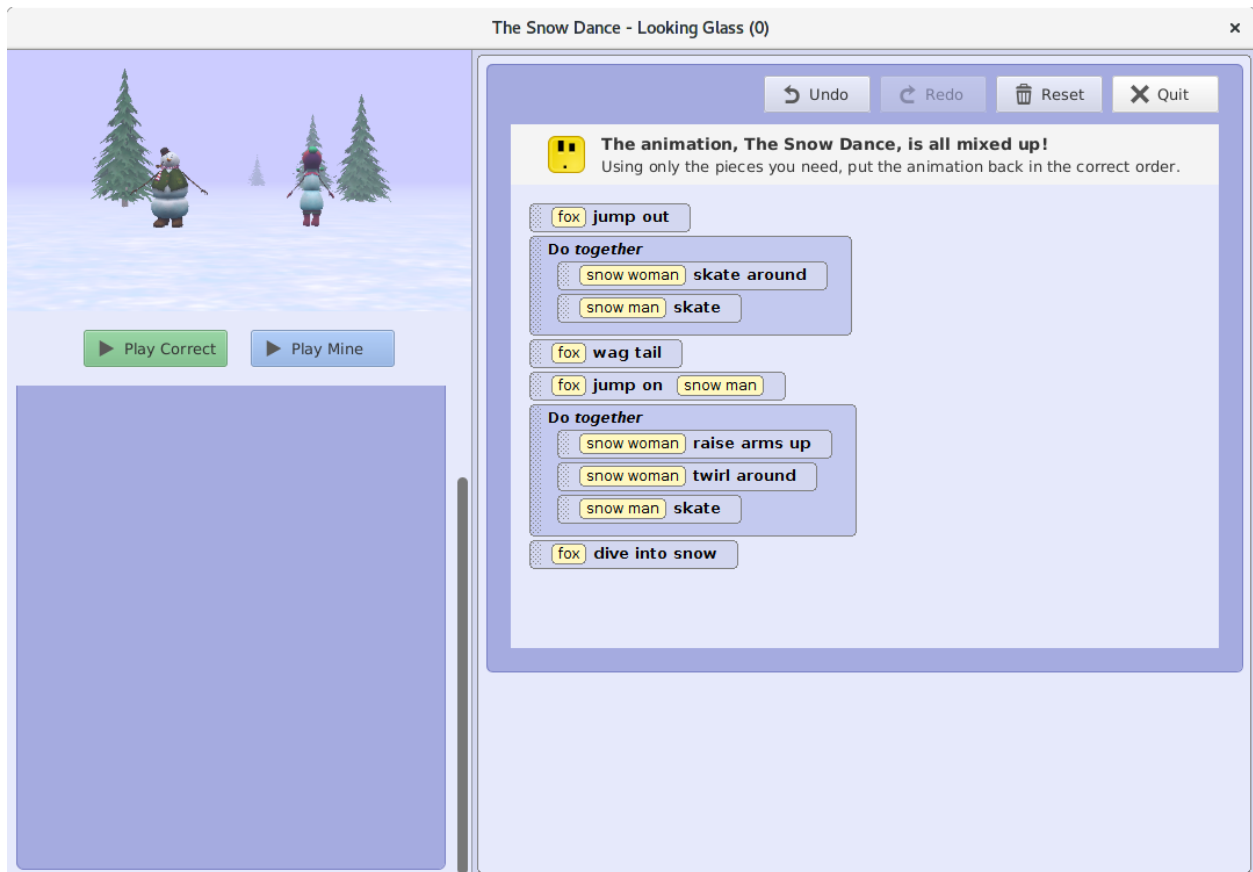


Figure D.5: The *Snow Dance* (5) puzzle reinforces the *Do Together* construct and demonstrates that blocks may contain more than two statements.



Figure D.6: The *Messed Up Magic* (6) puzzle enable learners to practice using the *Do Together* programming construct.

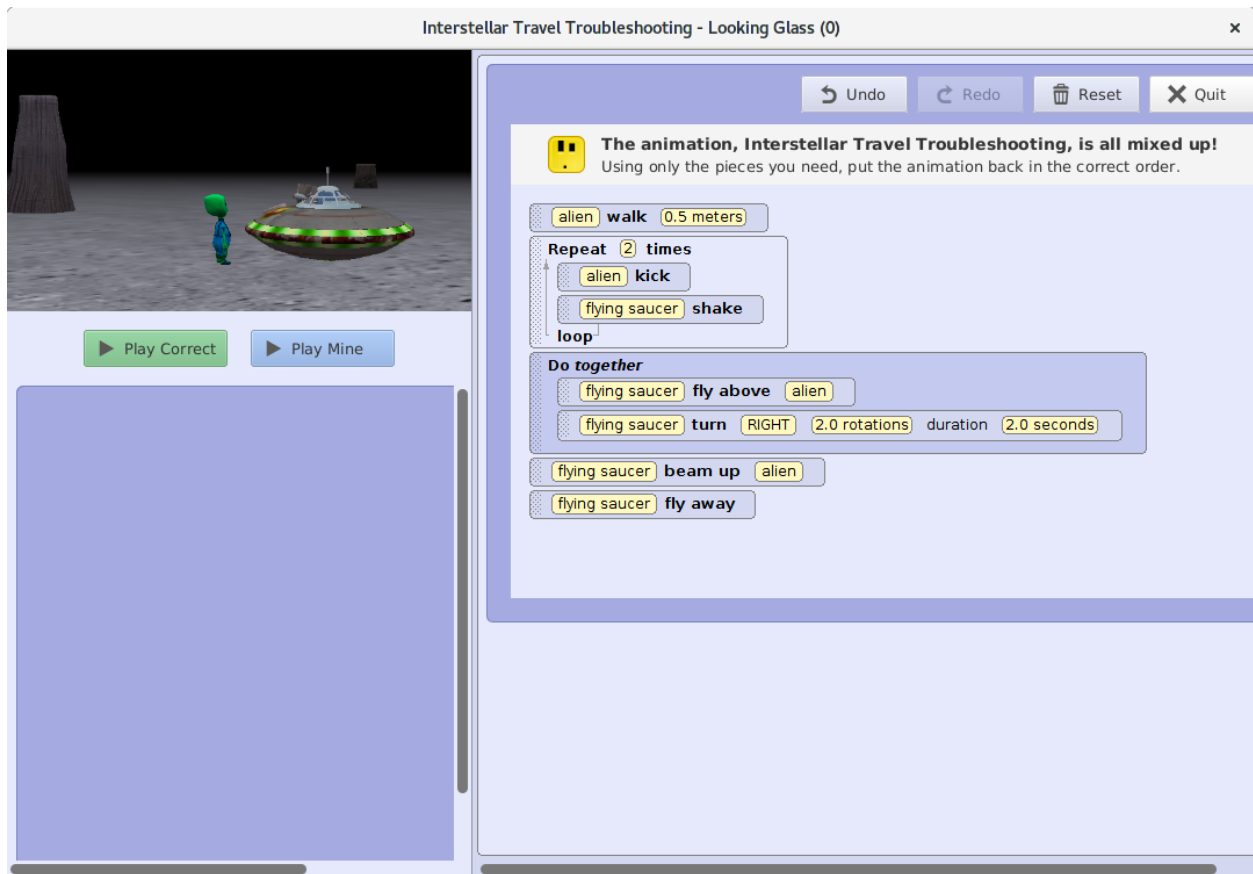


Figure D.7: In the *Interstellar Travel Troubleshooting* (7) puzzle, learners practice using both *Repeat* and *Do Together* programming constructs within the same program. Note: This puzzle was later renamed *Space Mechanic* in the version released in Looking Glass.

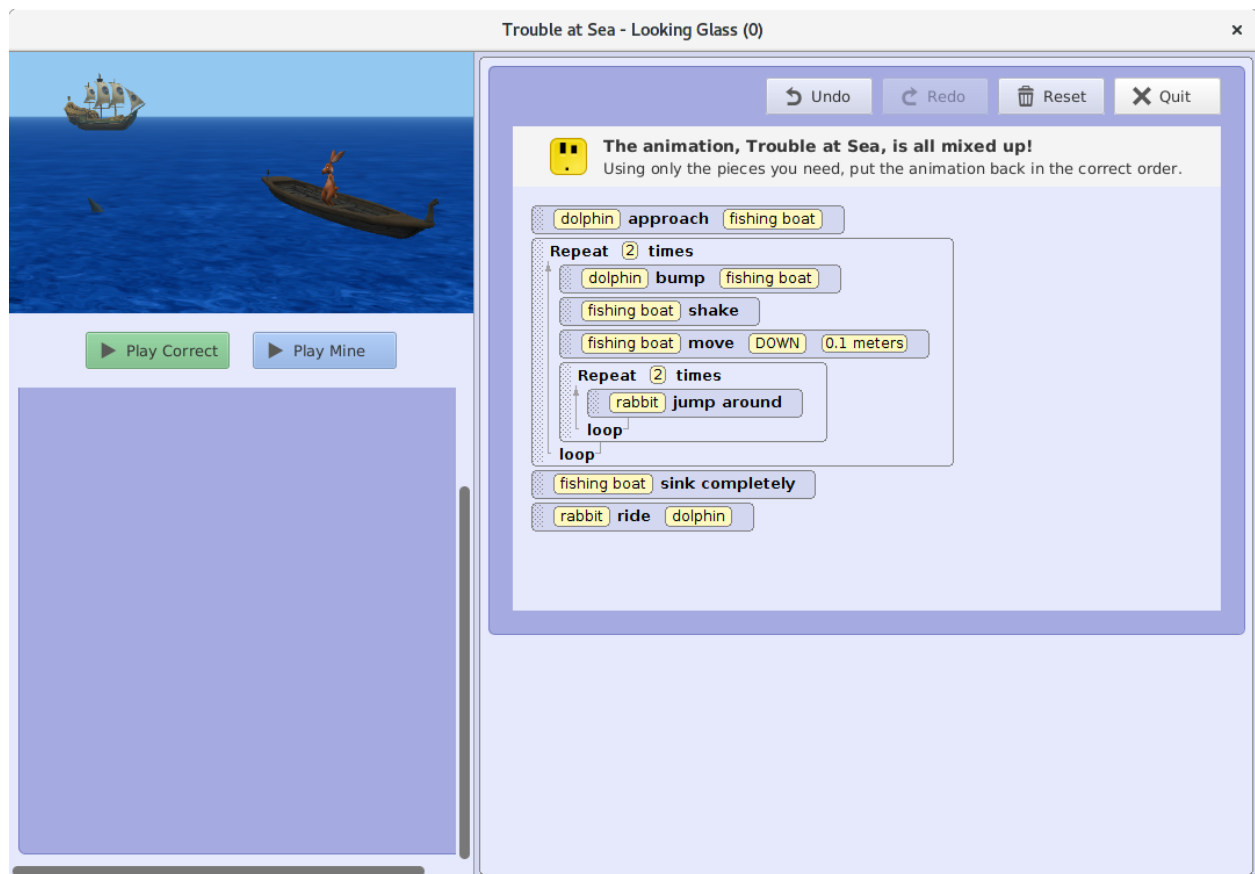


Figure D.8: The *Trouble at Sea* (8) puzzle introduces learners to nesting a *Repeat* block inside of a *Repeat* block.

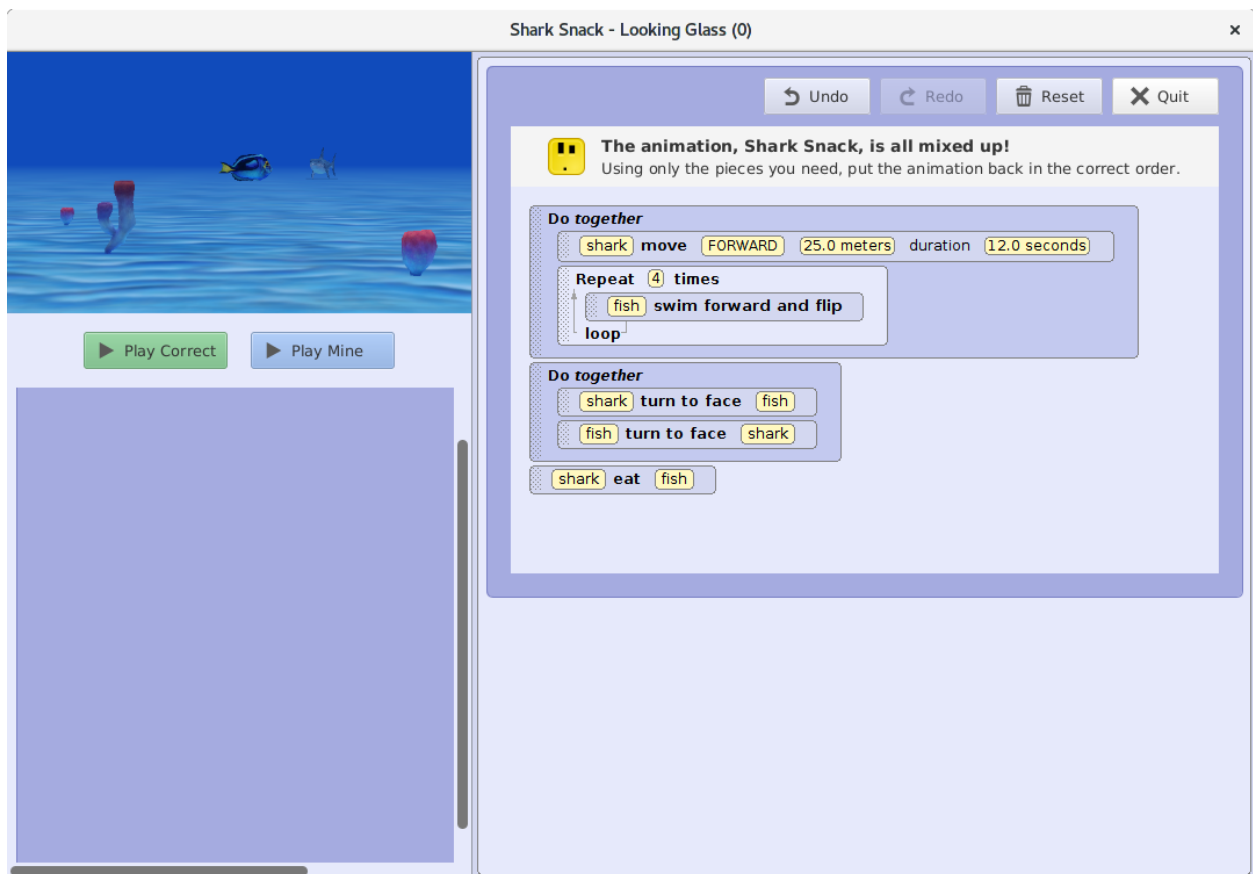


Figure D.9: The *Shark Snack* (9) puzzle introduces learners to nesting a *Repeat* block inside of a *Do Together* block.

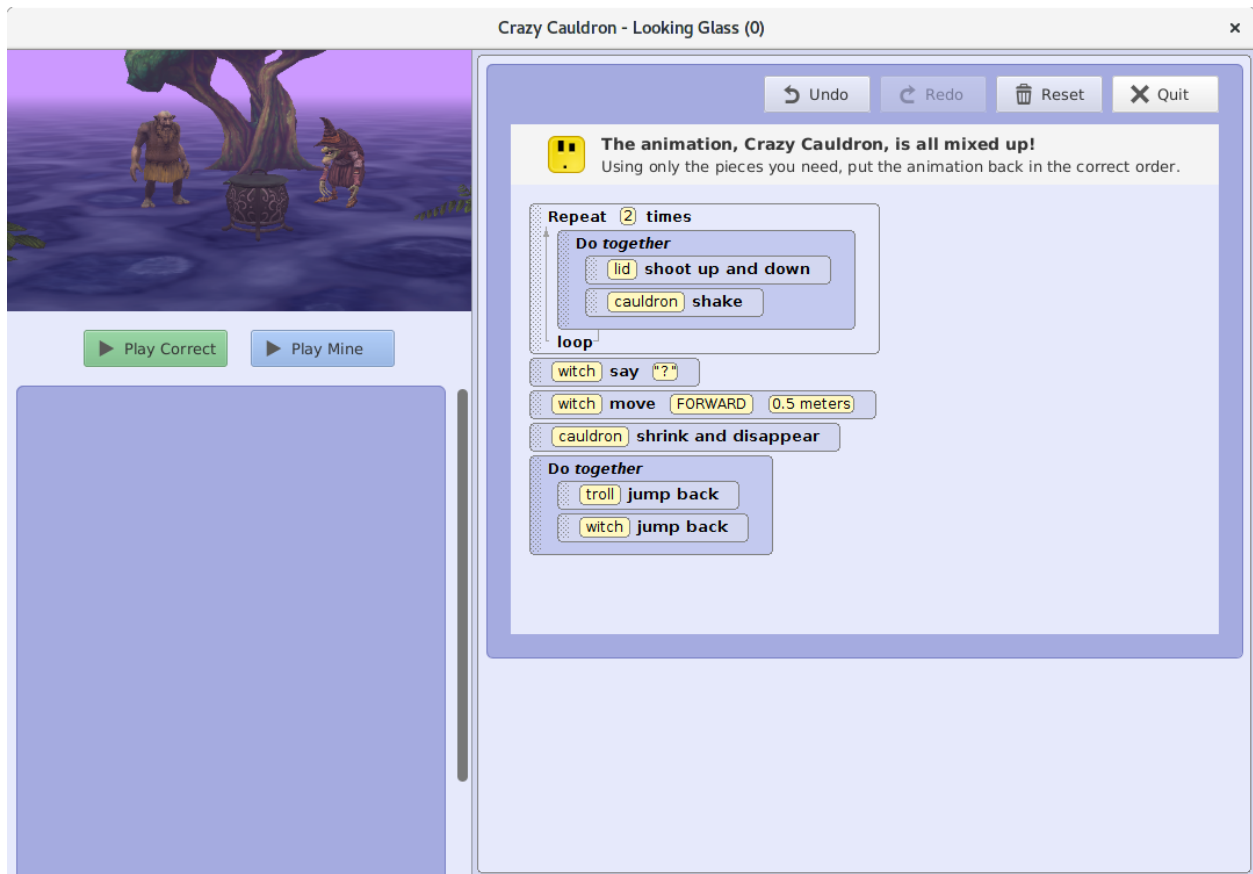


Figure D.10: The *Crazy Cauldron* (10) puzzle enables learners to practice nesting a *Do Together* block inside of a *Repeat* block.

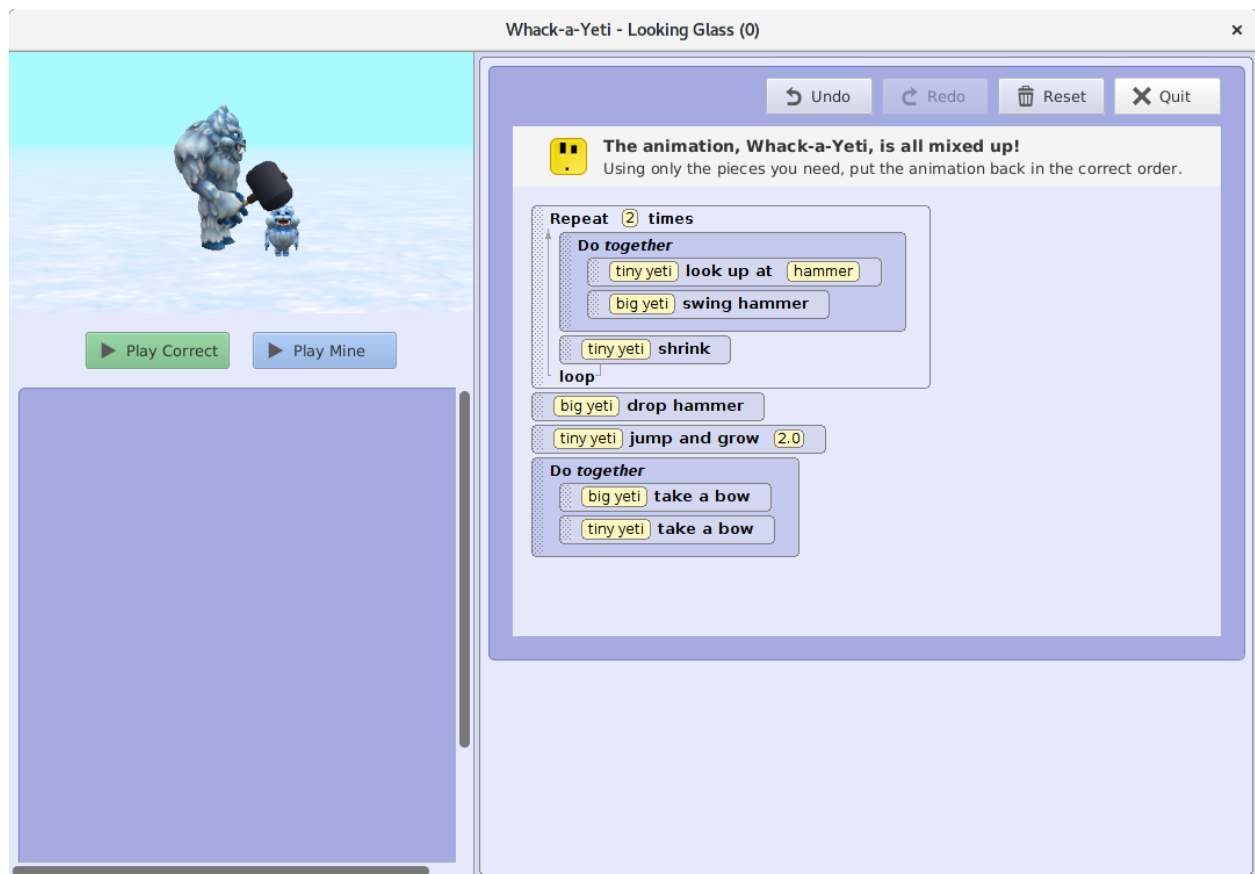


Figure D.11: The *Whack-a-Yeti* (11) puzzle introduces learners to nesting a *Do Together* block inside of a *Repeat* block.

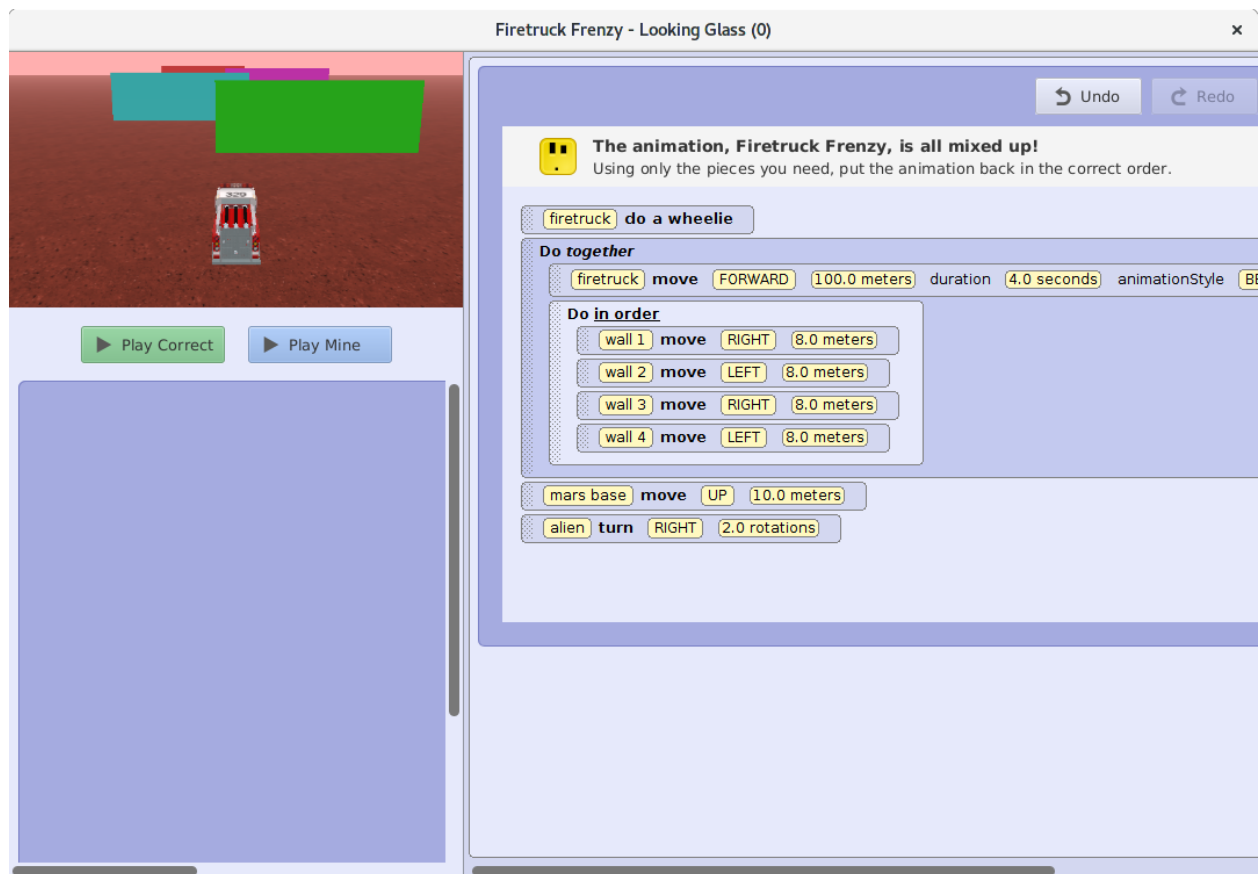


Figure D.12: The *Firetruck Frenzy* (12) puzzle introduces learners to nesting a *Do in Order* block inside of a *Do Together* block.



Figure D.13: The *Air Traffic* (13) puzzle enables learners to practice nesting a *Do in Order* block inside of a *Do Together* block.

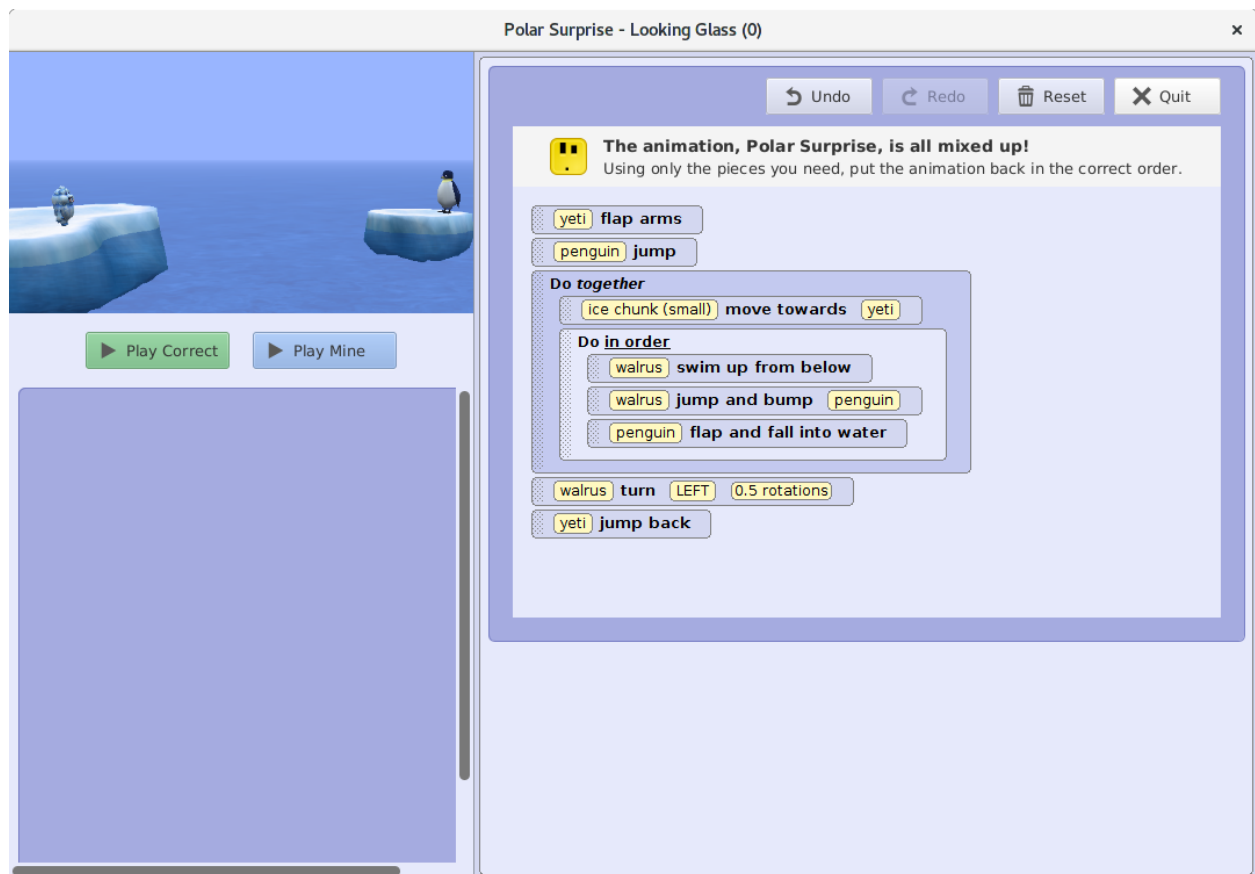


Figure D.14: The *Polar Surprise* (14) puzzle further reinforces nesting a *Do in Order* block inside of a *Do Together* block.

Appendix E

Summative Evaluation I Materials

Figures E.1–E.24 are the materials used in the summative evaluation presented in Chapter 3.

COMPUTING HISTORY SURVEY

1. How old are you? _____
2. What is your current grade in school? _____
3. What is your gender? (choose **one**)
 - a) Female
 - b) Male
 - c) Not specified
4. What kind of school do you go to? (choose **one**)
 - a) Public school
 - b) Private school
 - c) Home-schooled
5. Have you ever coded or written a computer program? (choose **one**)
 - a) Don't know
 - b) Yes
 - c) No
6. How would you characterize your coding or computer programming experience? (choose **one**)
 - a) No coding or programming experience.
 - b) I have coded or programmed a few times as part of an activity.
 - c) I enjoy coding or programming in my free time.
 - d) I'm not sure.
 - e) Other: _____
7. Have you ever used the following computer coding or programming software? (circle **all** that apply)
 - a) Hopscotch
 - b) LEGO Mindstorms
 - c) Looking Glass or Alice
 - d) Robotics
 - e) Scratch
 - f) Programming languages (examples: Javascript, Python, C++, Java, Visual Basic, C#)
 - g) None
 - h) Other Coding or Programming Software: _____
8. Have you participated in the following coding or programming activities? (circle **all** that apply)
 - a) Hour of code. (code.org)
 - b) CoderDojo. (coderdojo.com)
 - c) Coding or programming at school as part of a classroom activity.
 - d) A coder or programmer camp or after school workshop.
 - e) Coding or programming at an event (examples: scouting, academy of science, science center)
 - f) I code or program a computer at home.
 - g) Looking Glass research study. (Not including today)
 - h) None – I have never participated in a programming activity.
 - i) Other: _____
9. Have you spent more than **three** hours coding or programming a computer in your life?? (choose **one**)
 - a) Don't know
 - b) Yes
 - a) No

Figure E.1: Pre-study computing history survey.

E.1 Training Phase

E.1.1 Control Condition: Tutorials

For this part of the study, please follow the on-screen tutorial instructions.

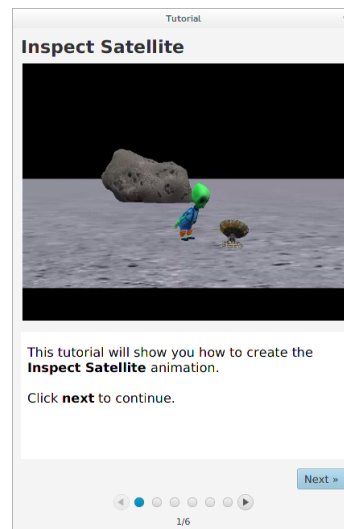


Figure E.2: Training task instruction sheet for the control condition.

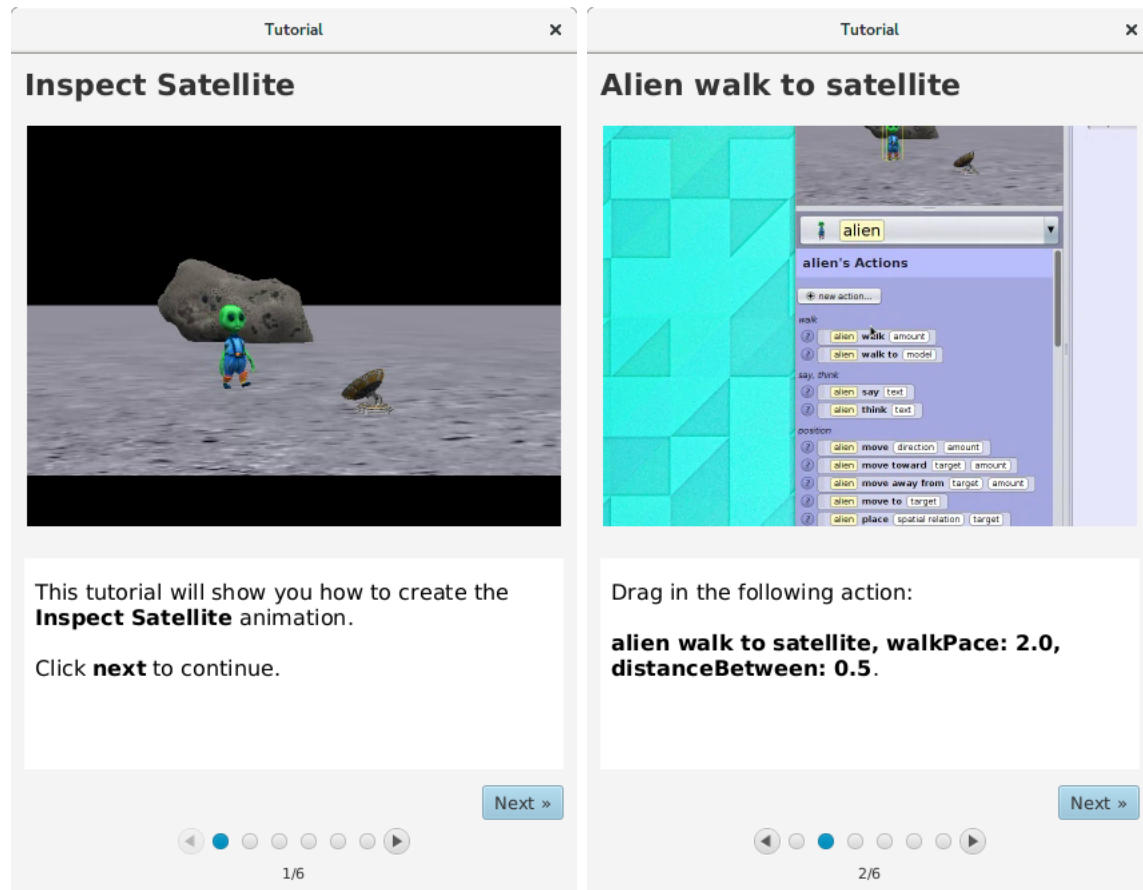


Figure E.3: Tutorial steps window.

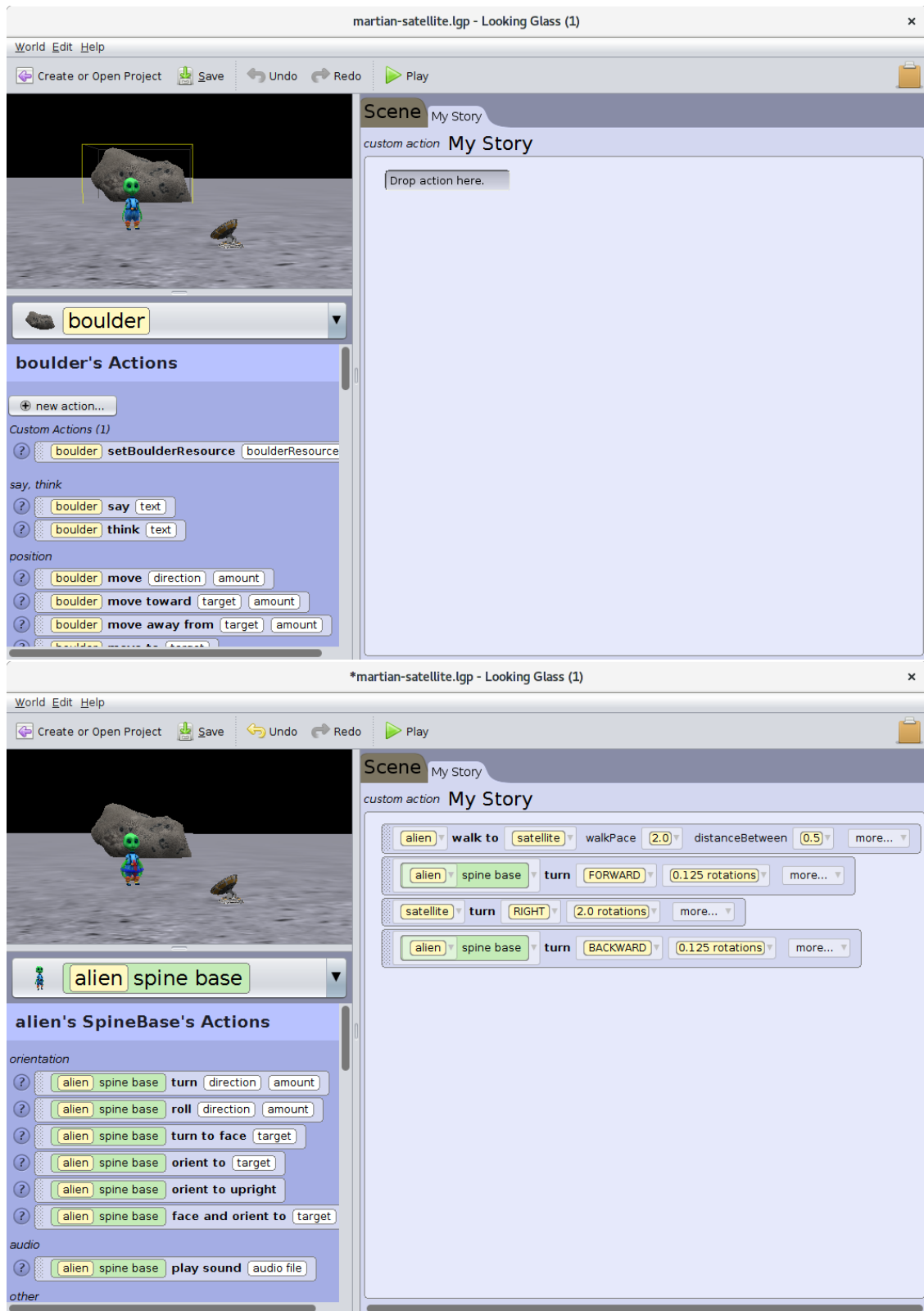


Figure E.4: Control training familiarization task. Initial state of each tutorial (*top*); completed state of tutorial (*bottom*).

E.1.2 Experimental Condition: Puzzles

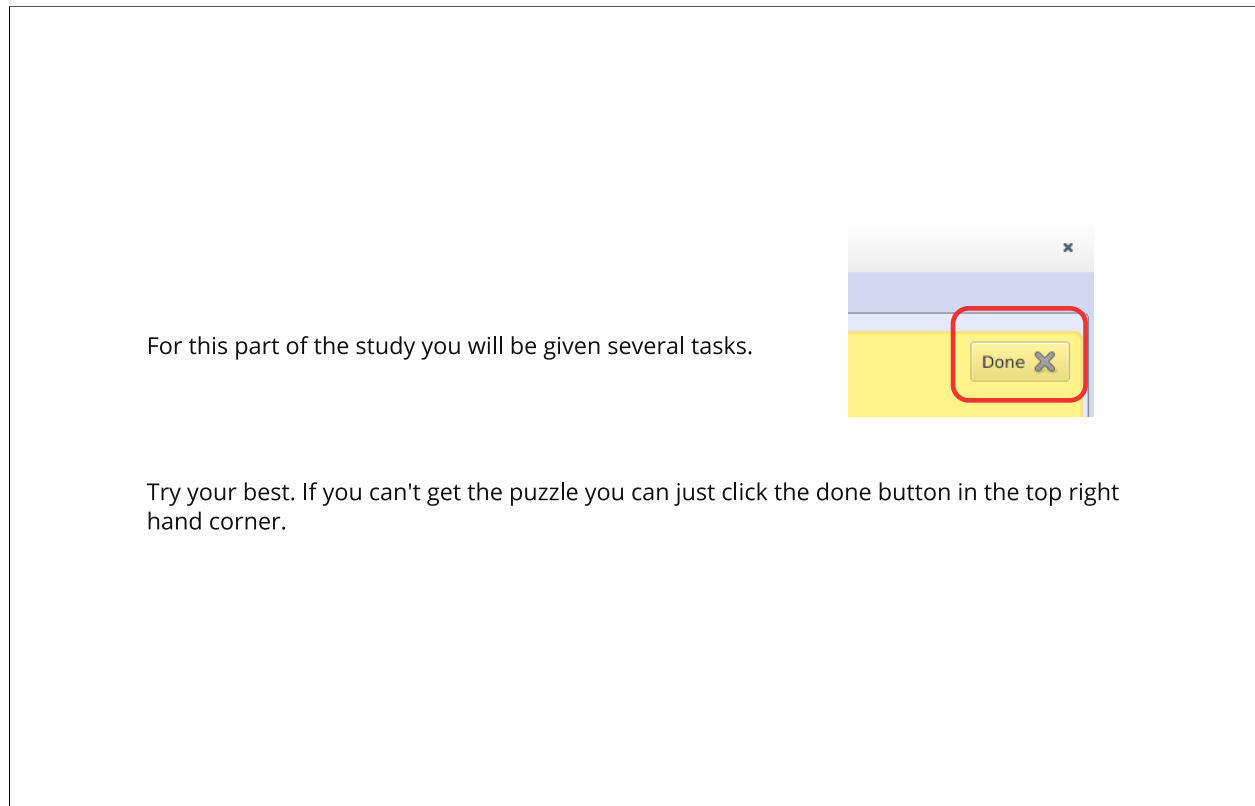


Figure E.5: Training task instruction sheet for the experimental condition.

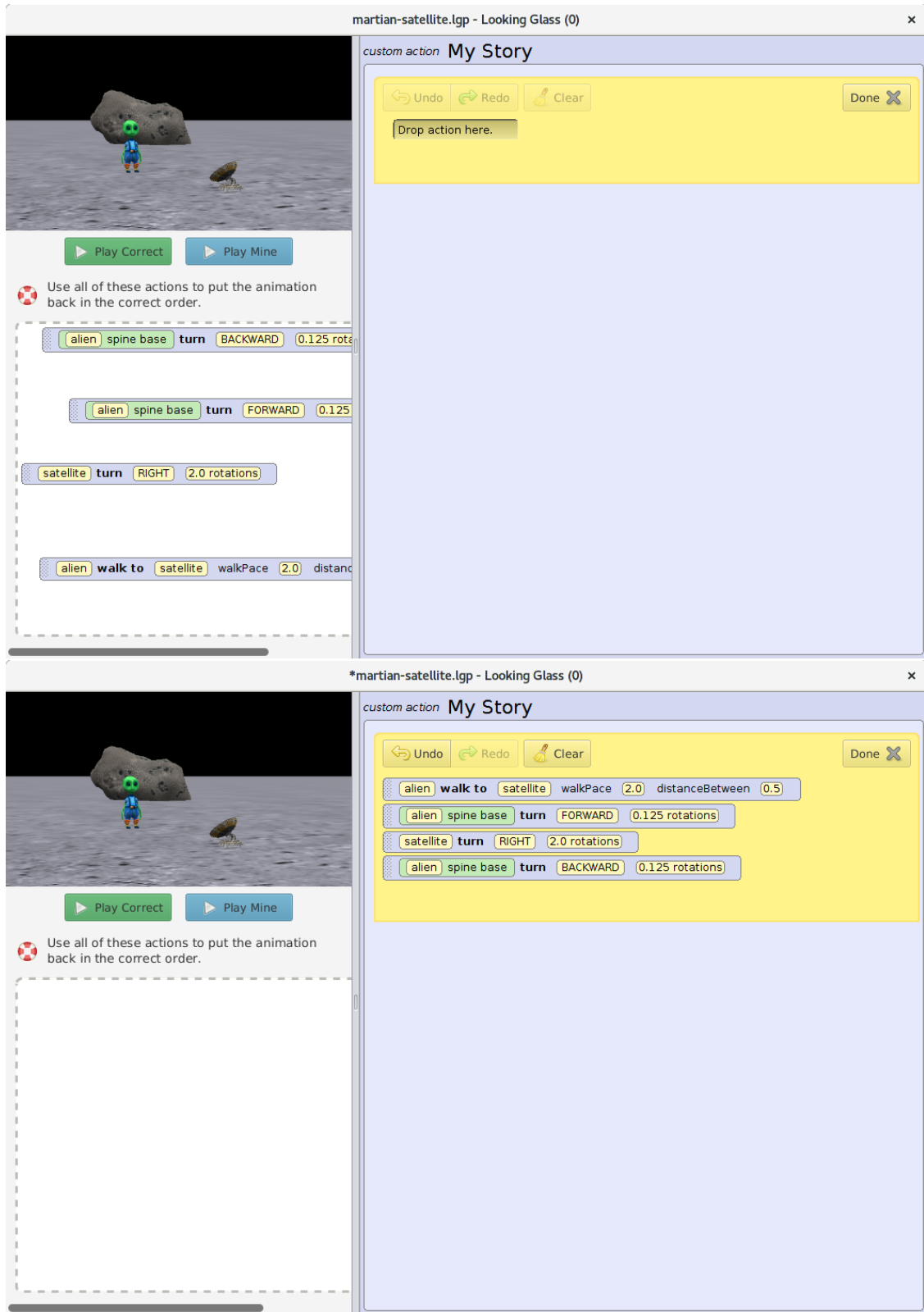
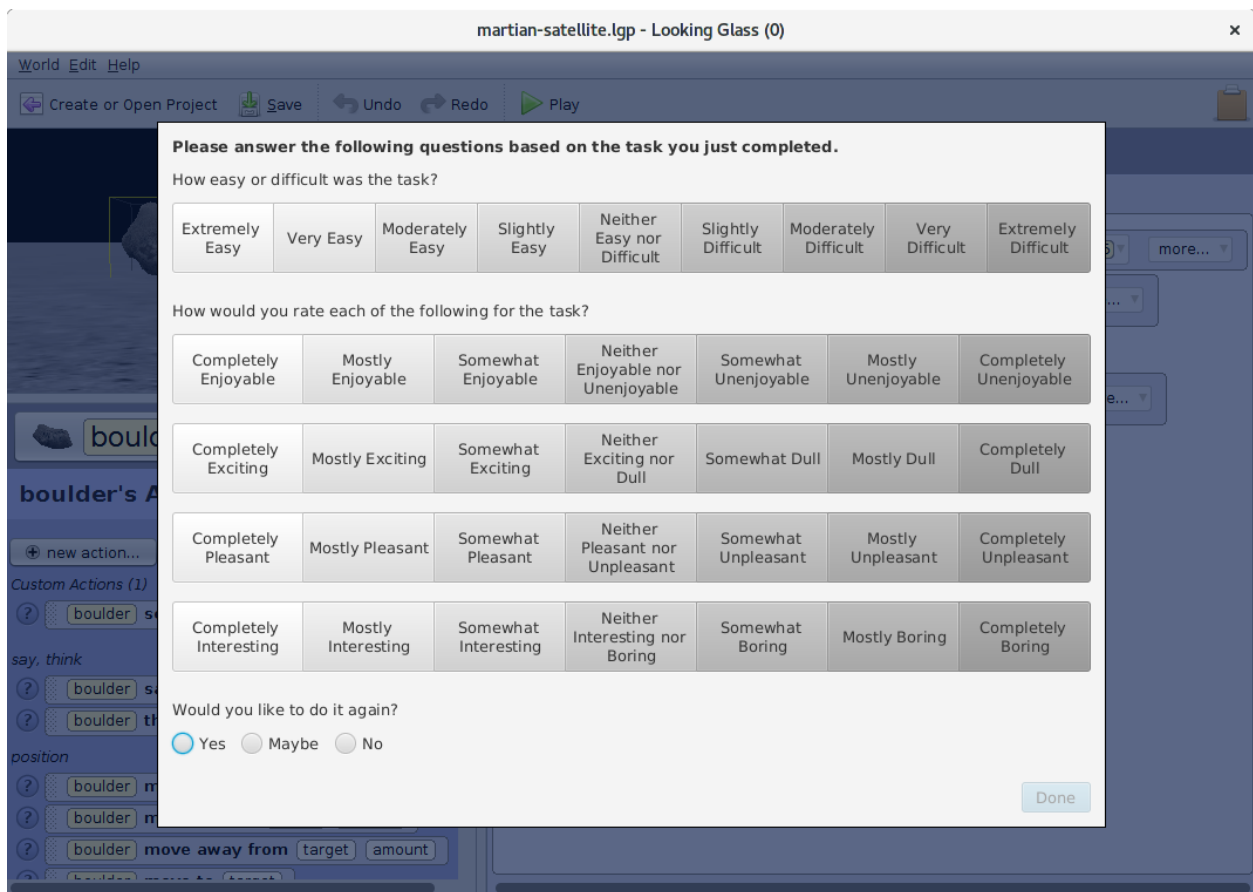


Figure E.6: Experimental training familiarization task. Initial state of each puzzle (*top*); completed state of puzzle (*bottom*).

E.1.3 Training Tasks



The screenshot shows a survey window titled "Please answer the following questions based on the task you just completed." overlaid on a game interface. The game interface has a menu bar with "World", "Edit", and "Help". Below it is a toolbar with icons for "Create or Open Project", "Save", "Undo", "Redo", and "Play". The background of the game shows a dark, rocky landscape with a "boulder" object. On the left, there is a "Custom Actions (1)" panel with a "boulder" object and a "say. think" action. Below that, there is a "position" panel with a "boulder" object and a "move away from" action. The survey window contains the following questions and options:

Please answer the following questions based on the task you just completed.

How easy or difficult was the task?

Extremely Easy	Very Easy	Moderately Easy	Slightly Easy	Neither Easy nor Difficult	Slightly Difficult	Moderately Difficult	Very Difficult	Extremely Difficult
----------------	-----------	-----------------	---------------	----------------------------	--------------------	----------------------	----------------	---------------------

How would you rate each of the following for the task?

Completely Enjoyable	Mostly Enjoyable	Somewhat Enjoyable	Neither Enjoyable nor Unenjoyable	Somewhat Unenjoyable	Mostly Unenjoyable	Completely Unenjoyable
Completely Exciting	Mostly Exciting	Somewhat Exciting	Neither Exciting nor Dull	Somewhat Dull	Mostly Dull	Completely Dull
Completely Pleasant	Mostly Pleasant	Somewhat Pleasant	Neither Pleasant nor Unpleasant	Somewhat Unpleasant	Mostly Unpleasant	Completely Unpleasant
Completely Interesting	Mostly Interesting	Somewhat Interesting	Neither Interesting nor Boring	Somewhat Boring	Mostly Boring	Completely Boring

Would you like to do it again?

☒ Yes ☐ Maybe ☐ No

Done

Figure E.7: Training task survey. After each training task, participants complete this survey.



Figure E.8: Training task 1. Completed control (*top*) and experimental (*bottom*) tasks.

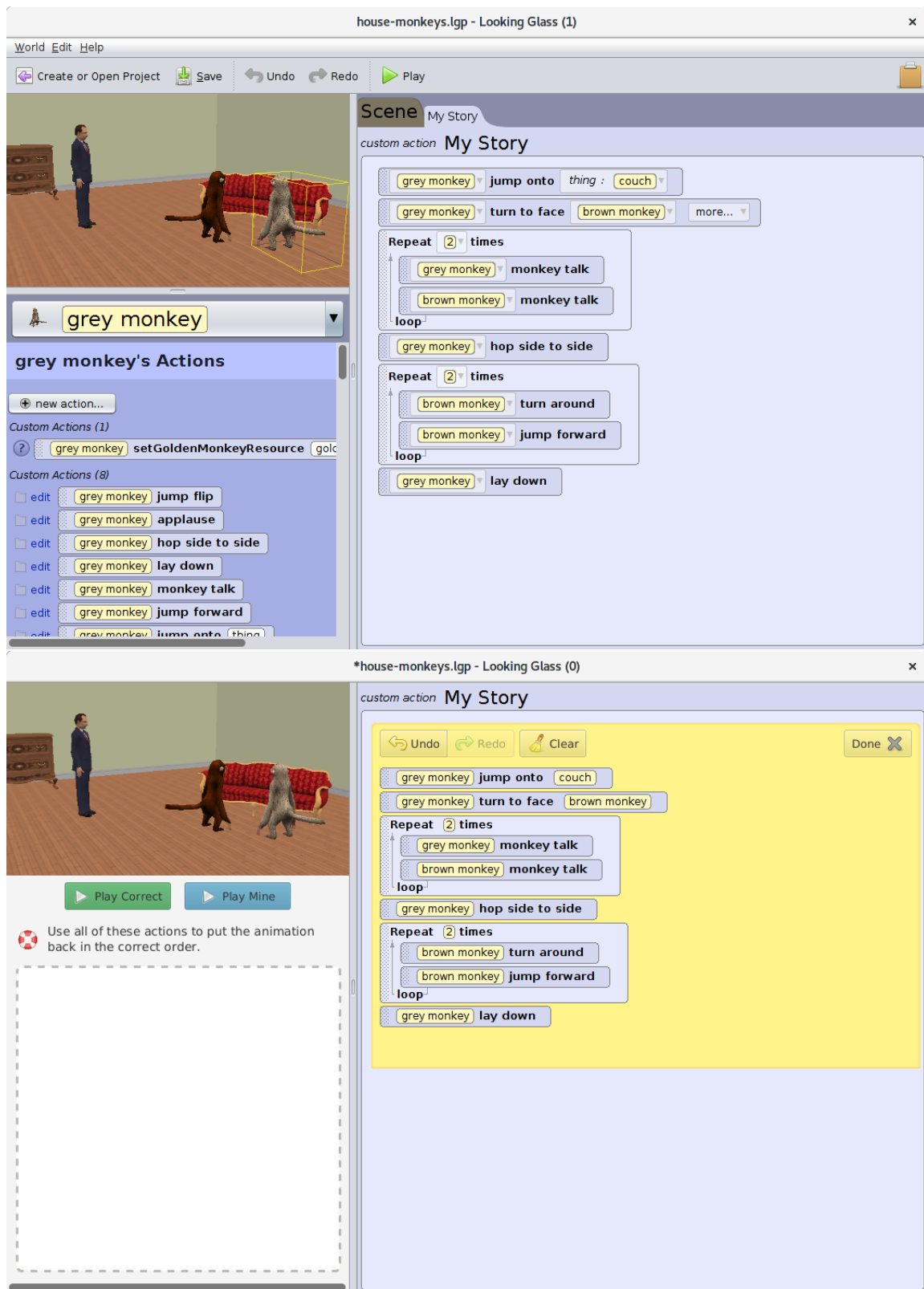


Figure E.9: Training task 2. Completed control (*top*) and experimental (*bottom*) tasks.

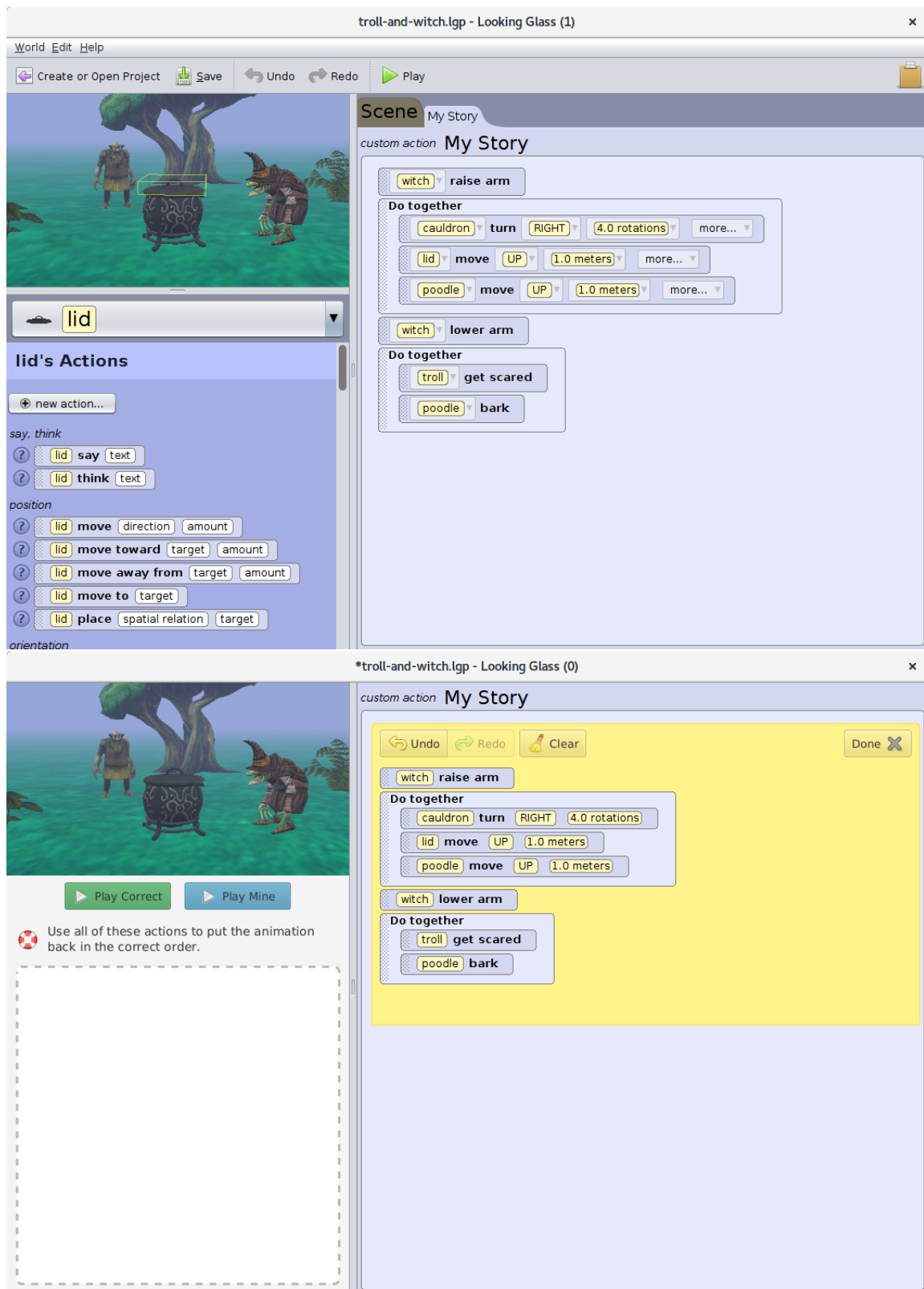


Figure E.10: Training task 3. Completed control (*top*) and experimental (*bottom*) tasks.

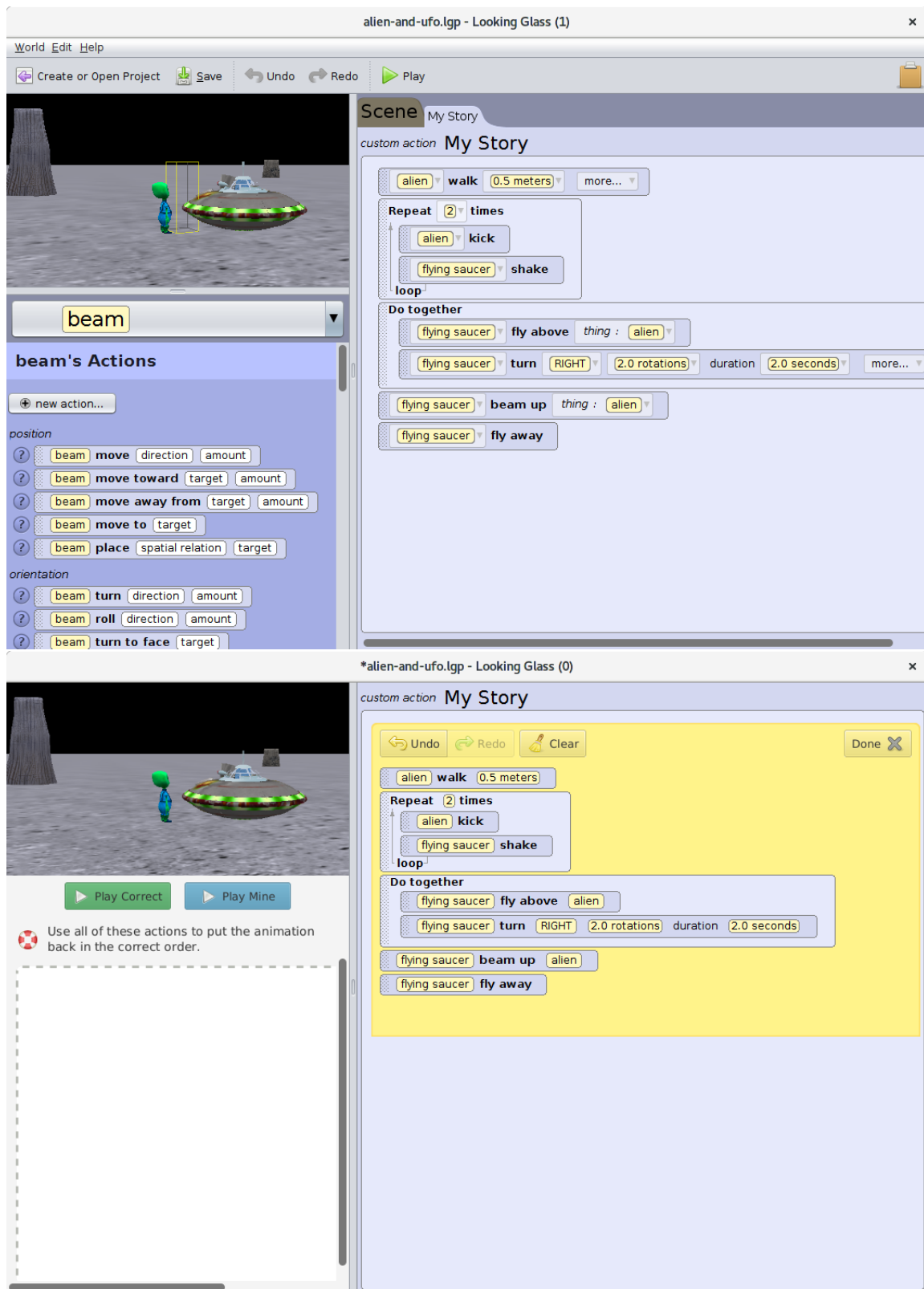


Figure E.11: Training task 4. Completed control (*top*) and experimental (*bottom*) tasks.

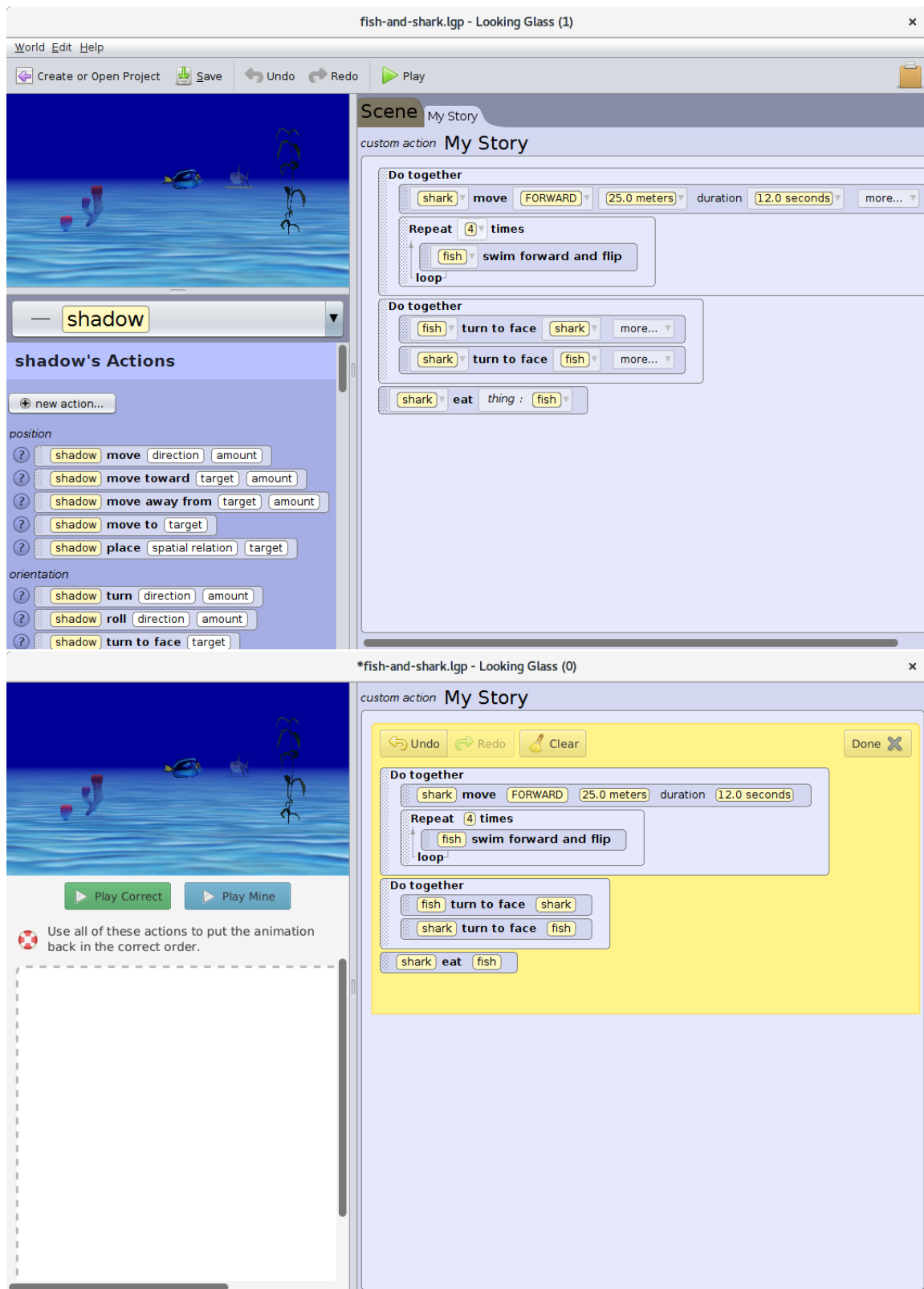


Figure E.12: Training task 5. Completed control (*top*) and experimental (*bottom*) tasks.

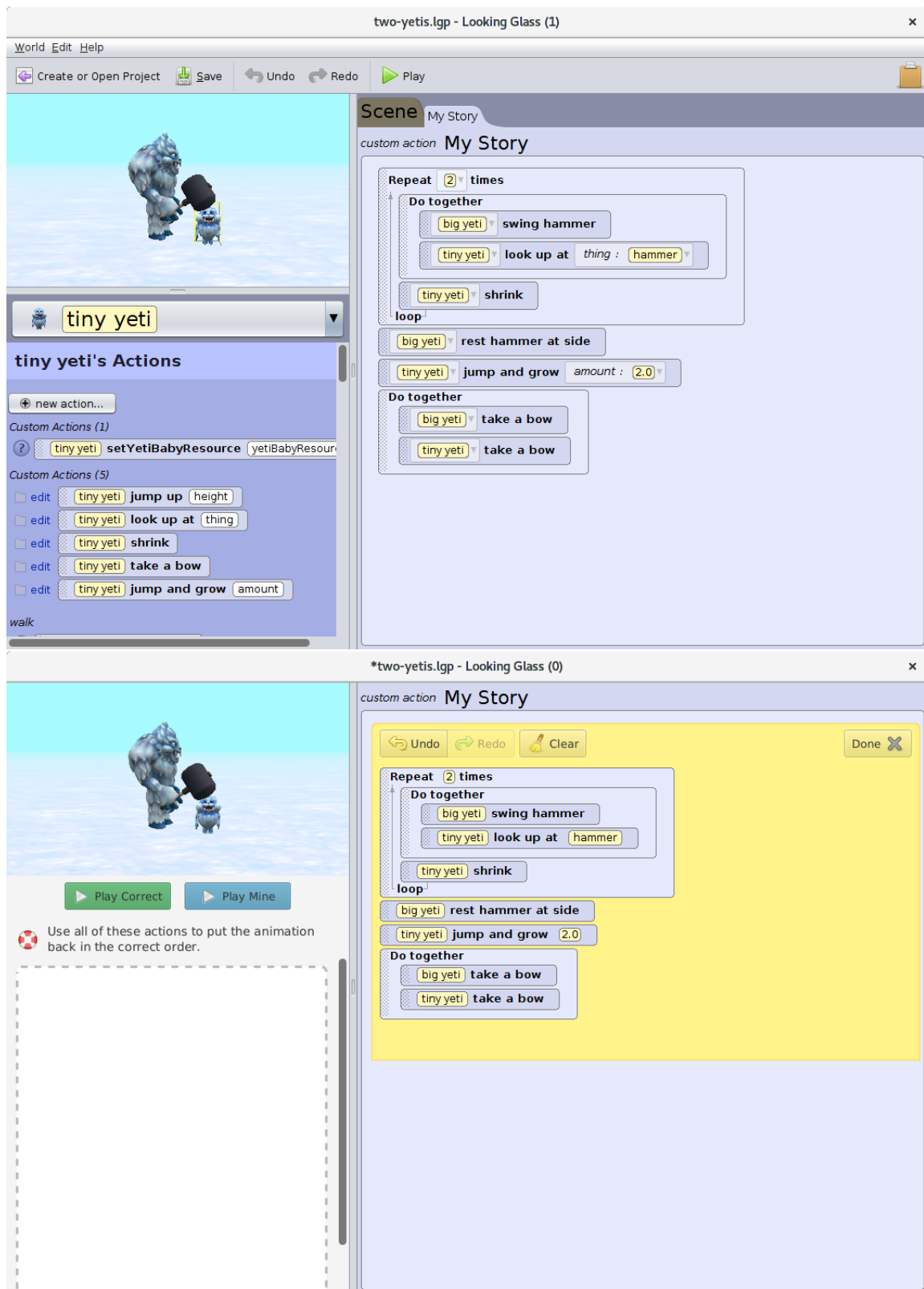


Figure E.13: Training task 6. Completed control (*top*) and experimental (*bottom*) tasks.

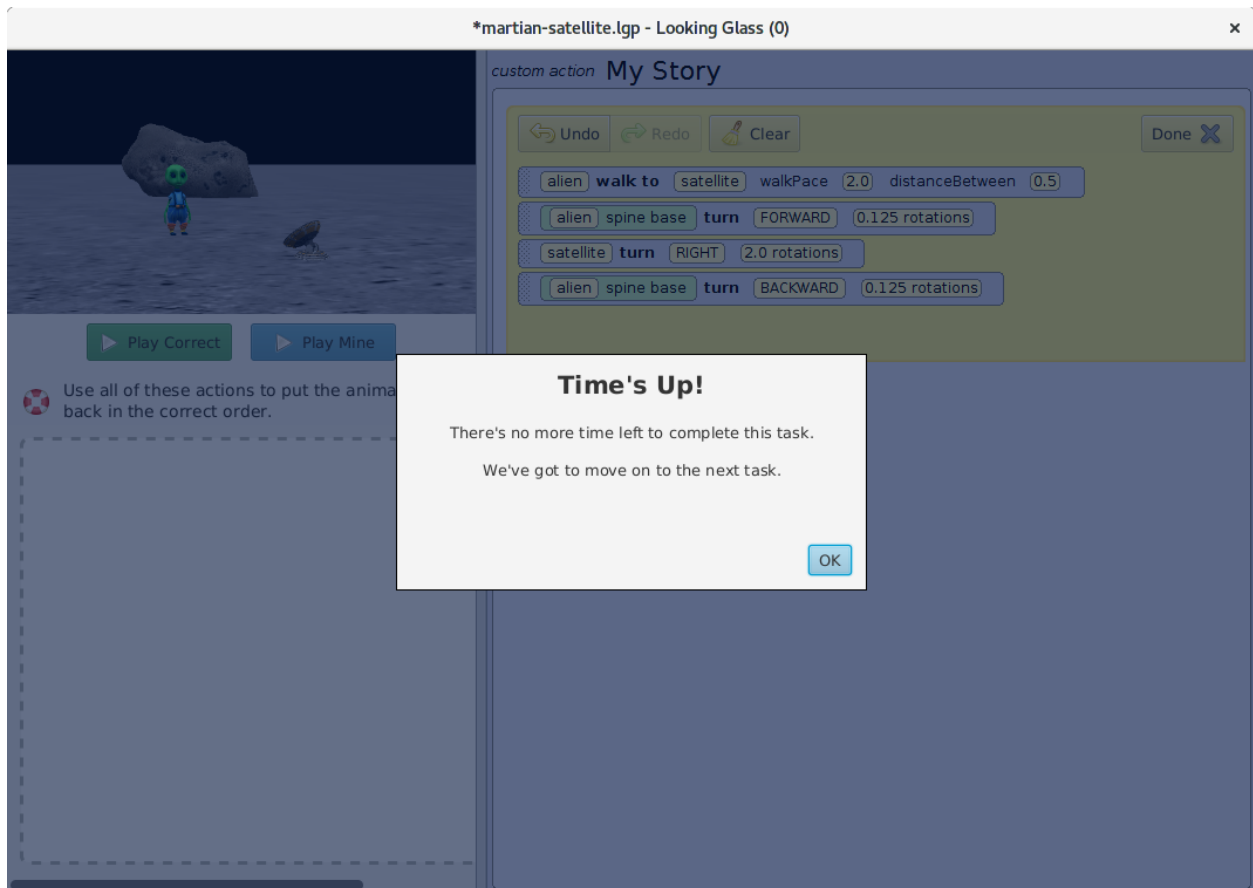


Figure E.14: This dialog is shown to participants if they have not completed the training task within the allotted time.

LOOKING GLASS SURVEY

For each of the following statements, please indicate how true it is for you, using the following scale:

	1	2	3	4	5	6	7
	not at all true			somewhat true			very true
	(not at all true) 1	2	3	(some- what true) 4	5	6	(very true) 7
While I was completing the tasks I was thinking about how much I enjoyed it.							
I did not feel nervous at all about completing the tasks.							
I felt that it was my choice to complete the tasks.							
I think I am pretty good at completing the tasks.							
I found completing the tasks interesting.							
I felt tense while completing the tasks.							
I think I did pretty well completing the tasks, compared to others.							
Completing the tasks was fun.							
I felt relaxed while completing the tasks.							
I enjoyed completing the tasks very much.							
I didn't really have a choice about completing the tasks.							
I am satisfied with my performance at completing the tasks.							
I was anxious while completing the tasks.							
I thought completing the tasks was boring.							
I felt like I was doing what I wanted to do while I was working on completing the tasks.							
I felt pretty skilled at completing the tasks.							
I thought completing the tasks was very interesting.							
I felt pressured while completing the tasks.							
I felt like I had to complete the tasks.							
I would describe completing the tasks as very enjoyable.							
I completed the tasks because I had no choice.							
After completing the tasks for awhile, I felt pretty competent.							

Figure E.15: After completing all training tasks, participants complete the *Task Evaluation Questionnaire*.

E.2 Transfer Phase

For this part of the study you will be given several tasks.

For each task please note:

1. The app will not tell if your answer is correct.
2. When you think you have the answer, click the Done button in the top right corner.
3. Try your best. If you can't get it to match exactly that's okay. Get it as close as you can.

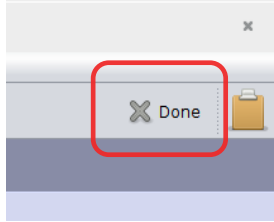
A screenshot of a mobile application interface. It features a light gray header bar with a small 'x' icon in the top right corner. Below the header is a white rectangular area containing a gray 'X' icon followed by the text 'Done'. This 'Done' button is enclosed in a red rectangular box. To the right of the 'Done' button is a small orange folder icon. The background of the app is a solid light blue color.

Figure E.16: Transfer task instruction sheet.

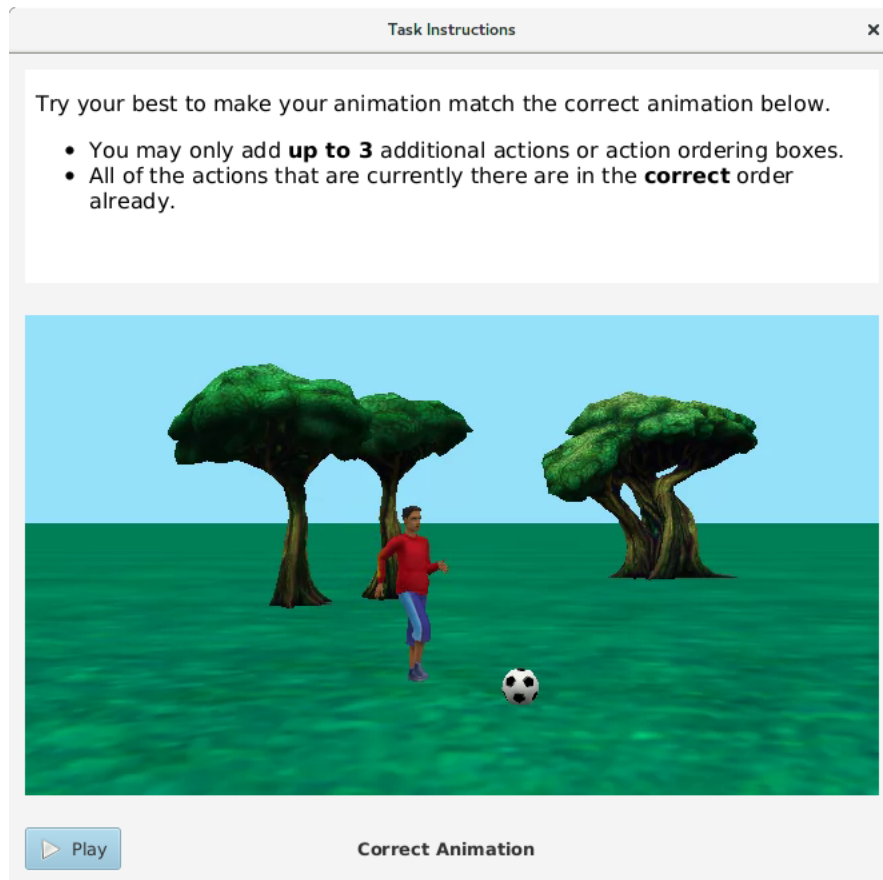


Figure E.17: Onscreen transfer task instructions. This window shows the correct output for each transfer task.

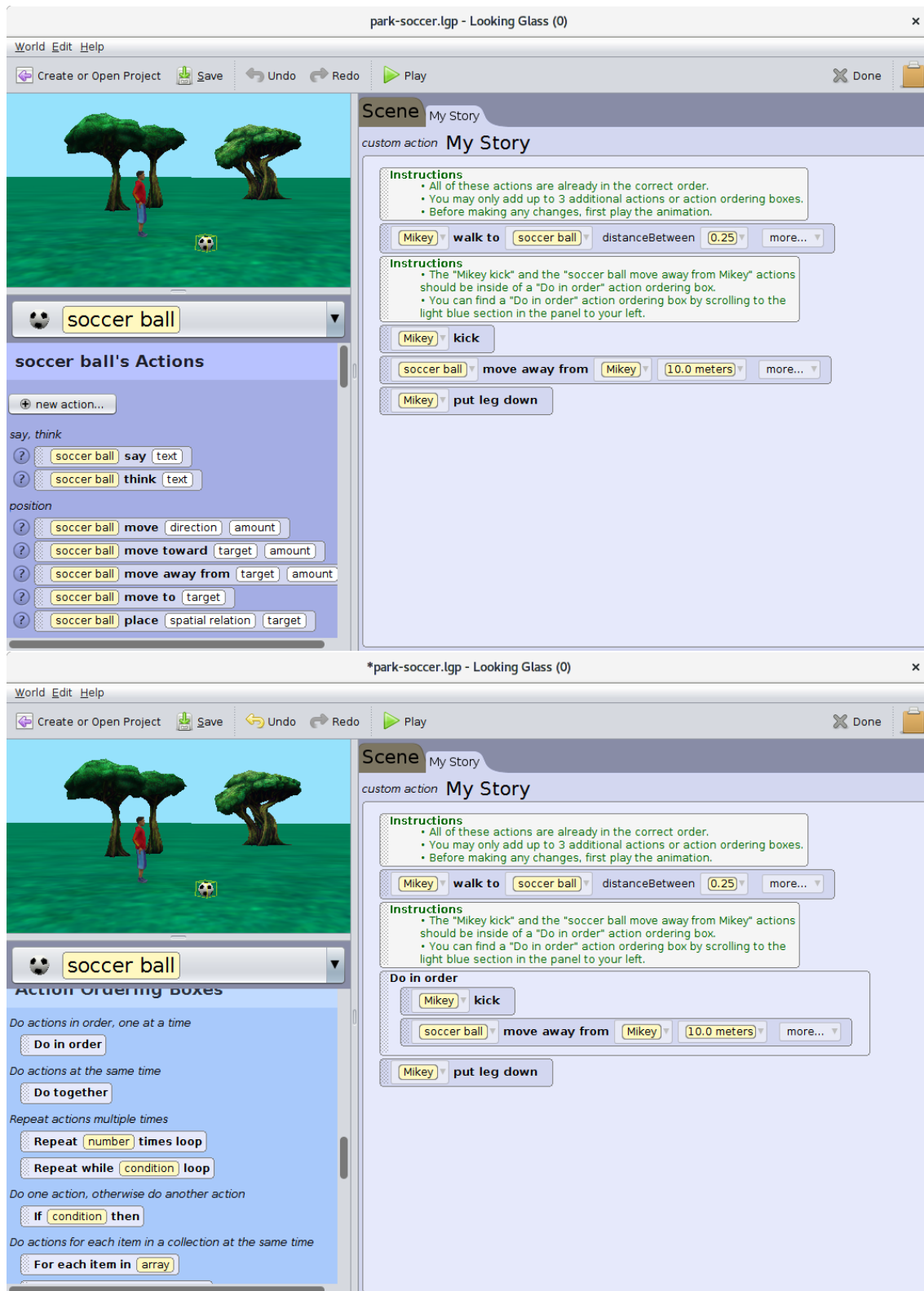


Figure E.18: Transfer familiarization task. Initial state of each transfer task (*top*); completed state of transfer task (*bottom*).

E.2.1 Transfer Tasks

The screenshot shows a software window titled "*park-soccer.lgp - Looking Glass (0)". A survey dialog box is overlaid on the interface. The dialog box contains the following sections:

Please answer the following questions based on the task you just completed.

How easy or difficult was the task?

Extremely Easy	Very Easy	Moderately Easy	Slightly Easy	Neither Easy nor Difficult	Slightly Difficult	Moderately Difficult	Very Difficult	Extremely Difficult
----------------	-----------	-----------------	---------------	----------------------------	--------------------	----------------------	----------------	---------------------

How would you rate each of the following for the task?

Completely Enjoyable	Mostly Enjoyable	Somewhat Enjoyable	Neither Enjoyable nor Unenjoyable	Somewhat Unenjoyable	Mostly Unenjoyable	Completely Unenjoyable
Completely Exciting	Mostly Exciting	Somewhat Exciting	Neither Exciting nor Dull	Somewhat Dull	Mostly Dull	Completely Dull
Completely Pleasant	Mostly Pleasant	Somewhat Pleasant	Neither Pleasant nor Unpleasant	Somewhat Unpleasant	Mostly Unpleasant	Completely Unpleasant
Completely Interesting	Mostly Interesting	Somewhat Interesting	Neither Interesting nor Boring	Somewhat Boring	Mostly Boring	Completely Boring

Would you like to do it again?

☐ Yes ☐ Maybe ☐ No

Done

Figure E.19: Transfer task survey. After each transfer task, participants complete this survey.

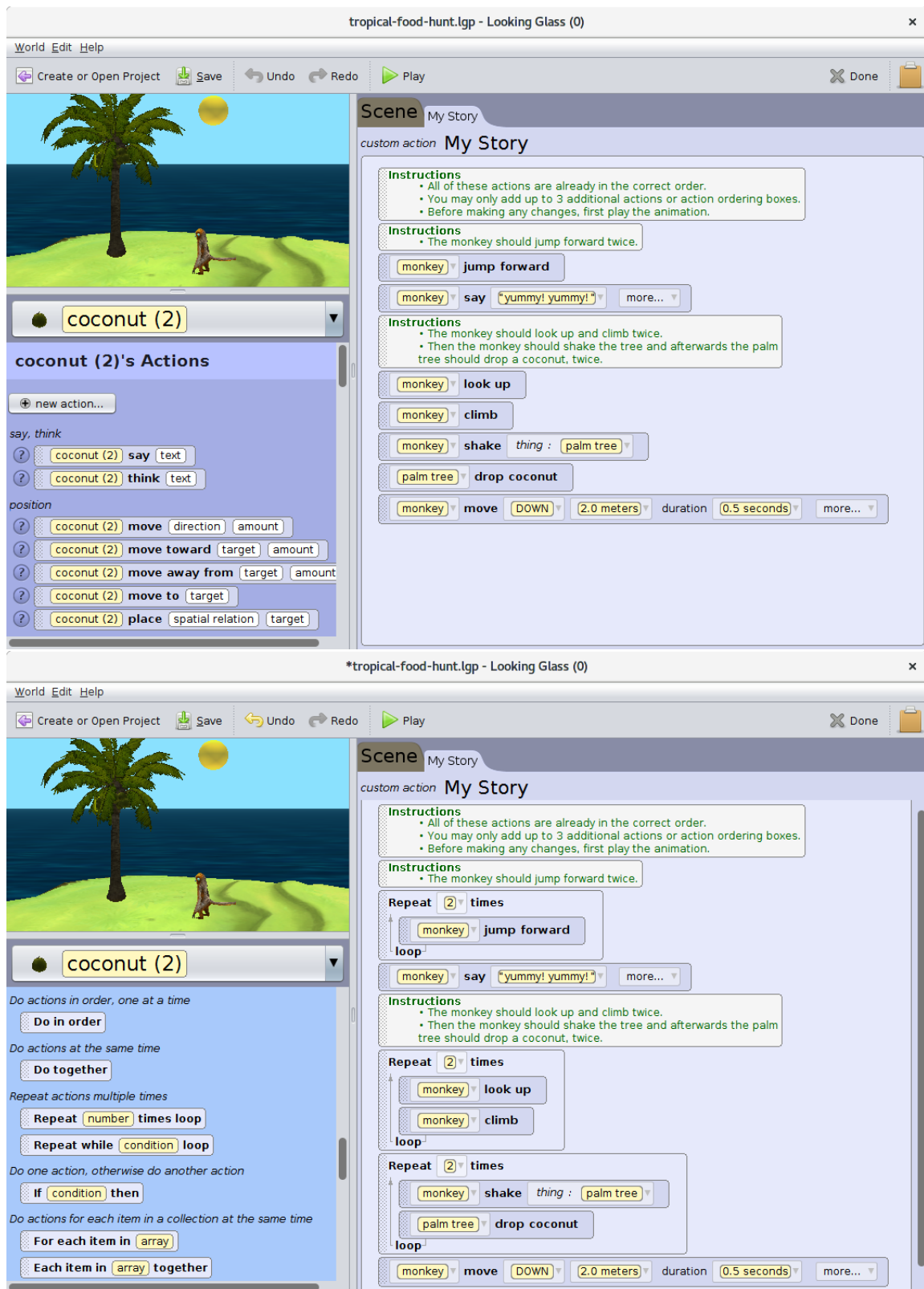


Figure E.20: *Repeat* transfer task. Initial state (*top*) and completed state (*bottom*).



Figure E.21: *Do Together* transfer task. Initial state (*top*) and completed state (*bottom*).

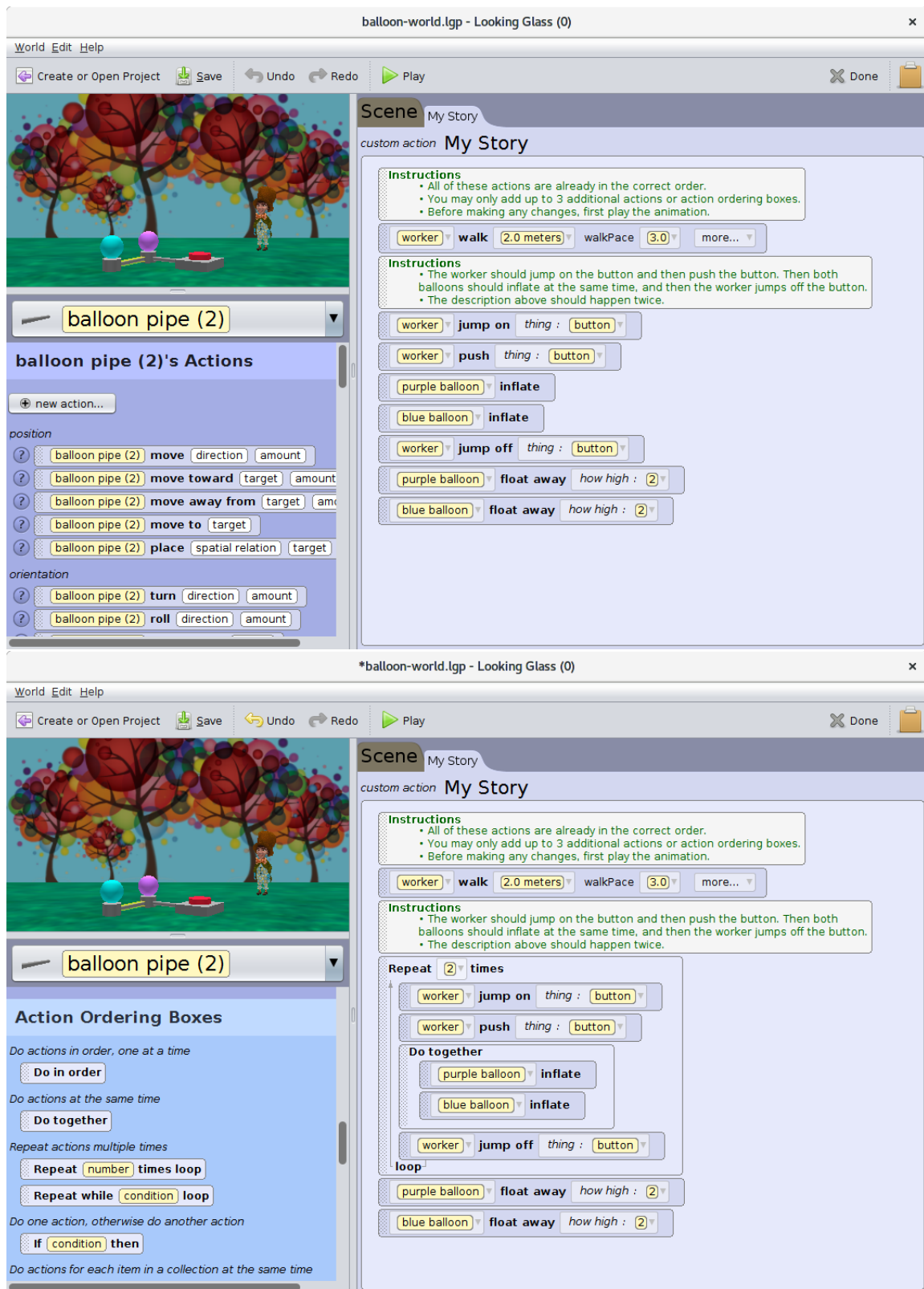


Figure E.22: *Repeat { Do Together }* transfer task. Initial state (*top*) and completed state (*bottom*).

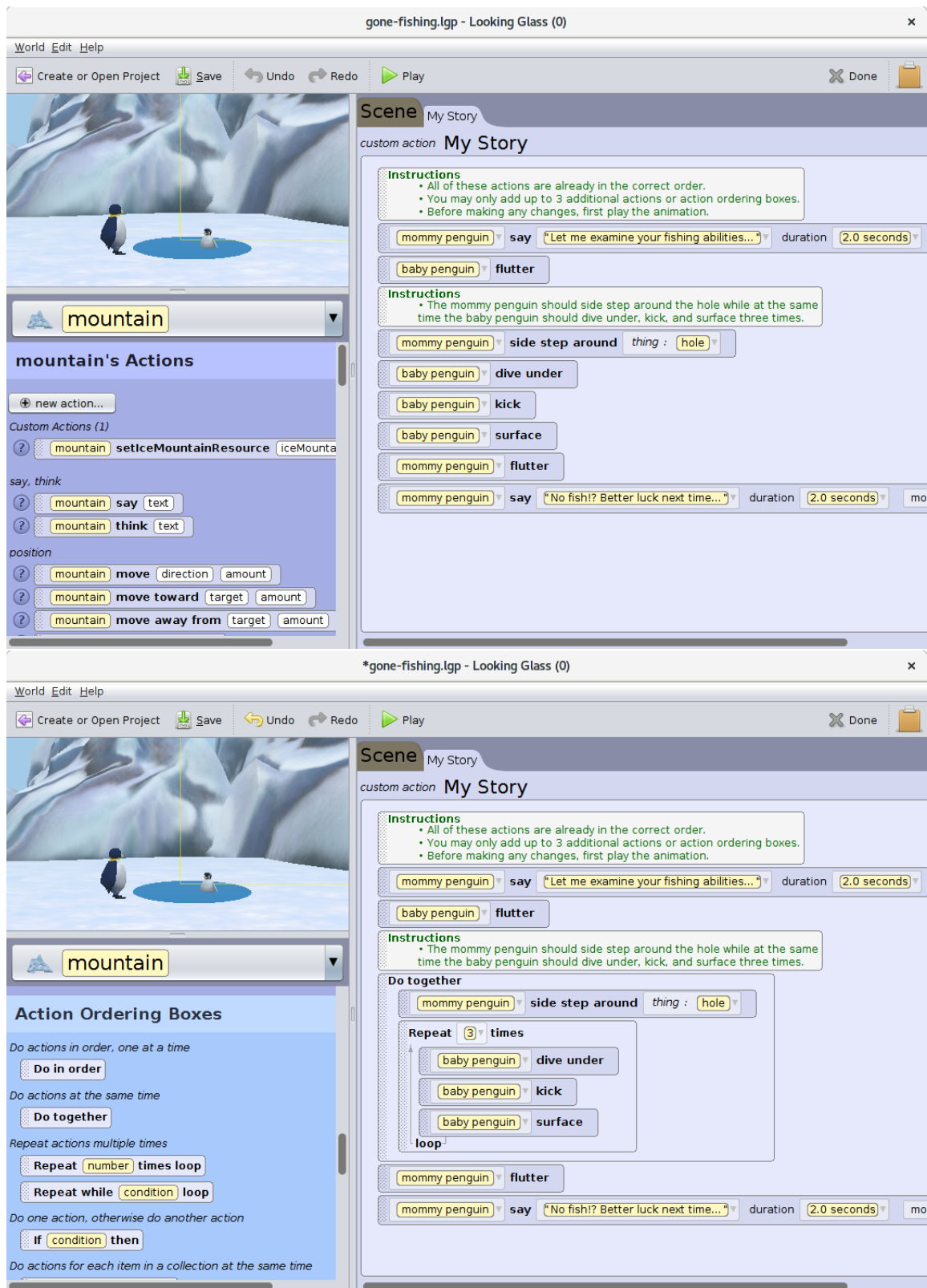


Figure E.23: *Do Together { Repeat }* transfer task. Initial state (*top*) and completed state (*bottom*).

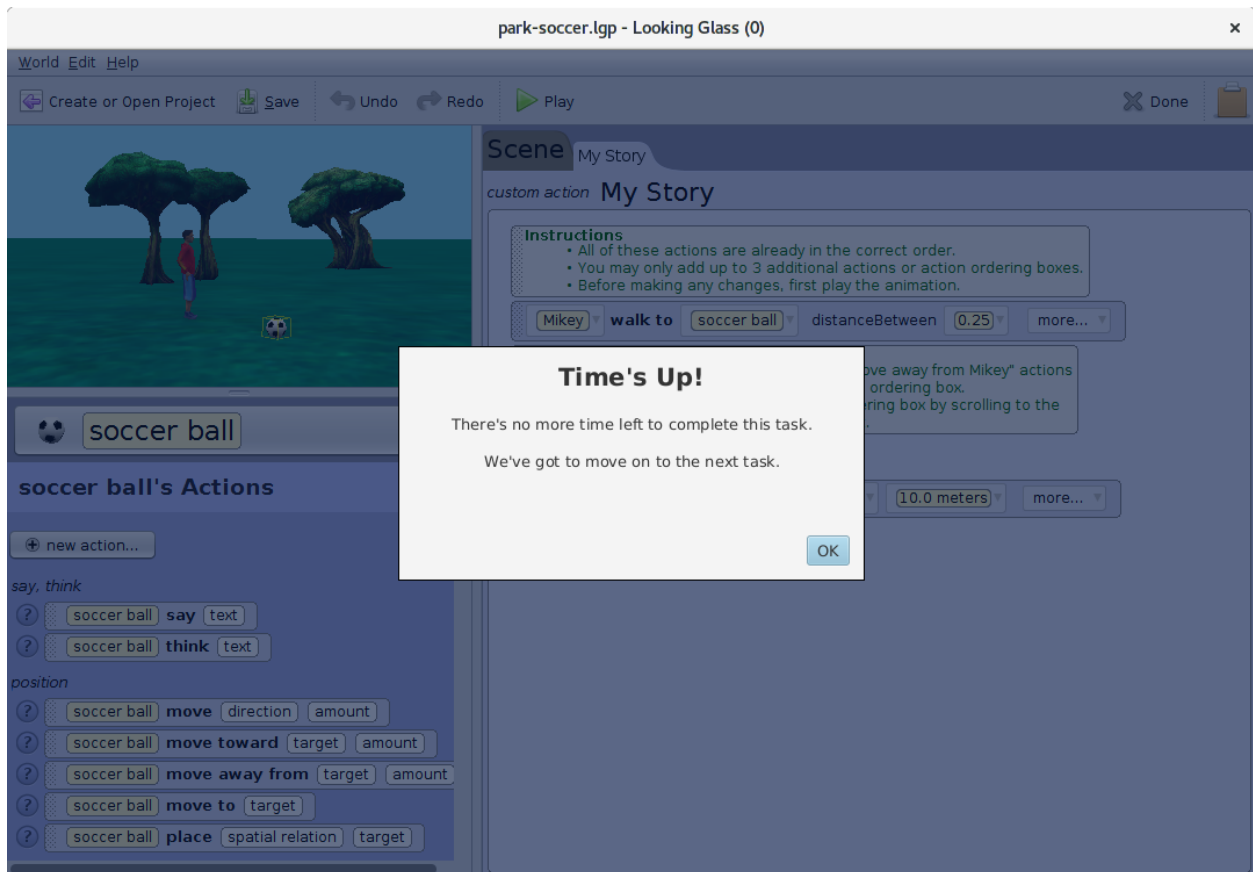


Figure E.24: This dialog is shown to participants if they have not completed the transfer task within the allotted time.

Appendix F

Summative Evaluation II Materials

Figures F.1–F.36 are the materials used in the summative evaluation presented in Chapter 4.

Programming Experience Survey

1. How old are you? _____
2. What is your current grade in school? _____
3. What is your gender?
 - ☐ Female
 - ☐ Male
 - ☐ Not specified
4. What kind of school do you go to?
 - ☐ Public school
 - ☐ Private school
 - ☐ Home-schooled
5. Have you ever participated in a Looking Glass or Animated Movie study?
 - ☐ Don't Know
 - ☐ Yes
 - ☐ No
6. Have you ever programmed a computer? Have you ever coded before? Have you ever coded or written a computer program?
 - ☐ Don't know
 - ☐ No
 - ☐ Yes
7. Has anyone ever taught you about programming or coding?
 - ☐ Don't know
 - ☐ No
 - ☐ Yes
8. Have you ever taken a class or attended a camp or workshop where you learned to code or program?
 - ☐ Don't know
 - ☐ No
 - ☐ Yes
9. Have you spent more than 3 hours total coding or programming before?
 - ☐ Don't know
 - ☐ No
 - ☐ Yes

Figure F.1: Pre-study computing history survey.

Followup

1. Have you ever participated in a Looking Glass or Animated Movie study?
 - ☐ Don't Know
 - ☐ Yes
 - ☐ No
2. Have you ever programmed a computer? Have you ever coded before? Have you ever coded or written a computer program?
 - ☐ Don't know
 - ☐ No
 - ☐ Yes
 - ↳ Where did you code? (example: class, camp)
3. Has anyone ever taught you about programming or coding?
 - ☐ Don't know
 - ☐ No
 - ☐ Yes
 - ↳ Who taught you? (example: teacher, parent)
4. Have you ever taken a class or attend a camp or workshop where you learned to code or program?
 - ☐ Don't know
 - ☐ No
 - ☐ Yes
 - ↳ How long did it last? (example: 1 day, 1 semester)
5. What types of things have you coded or programmed before? Leave blank if you don't know. (example: played a game on code.org, created animation in Scratch, programmed a robot)

Figure F.2: Follow-up pre-study computing history survey completed by a researcher during the follow-up interview.

LIST OF PROGRAMMING SOFTWARE

- code.org
- Scratch
- App Inventor
- LEGO Mindstorms
- Alice
- Looking Glass
- Hopscotch
- Greenfoot
- Eclipse
- Visual Studio
- Netbeans

LIST OF PROGRAMMING LANGUAGES

- Javascript
- Python
- C++
- Java
- Visual Basic
- C#
- Ruby
- Processing
- Node.js

LIST OF PROGRAMMING ACTIVITIES

- Hour of Code
- CoderDojo
- Programming class
- Programming activity in class
- Afterschool programming activity
- Programming workshop (example: Boy Scouts or Girl Scouts)
- Programming camp
- Programming a robot
- Making a Minecraft Mod

Figure F.3: Reference sheet for completing the two pre-study surveys.

F.1 Training Phase

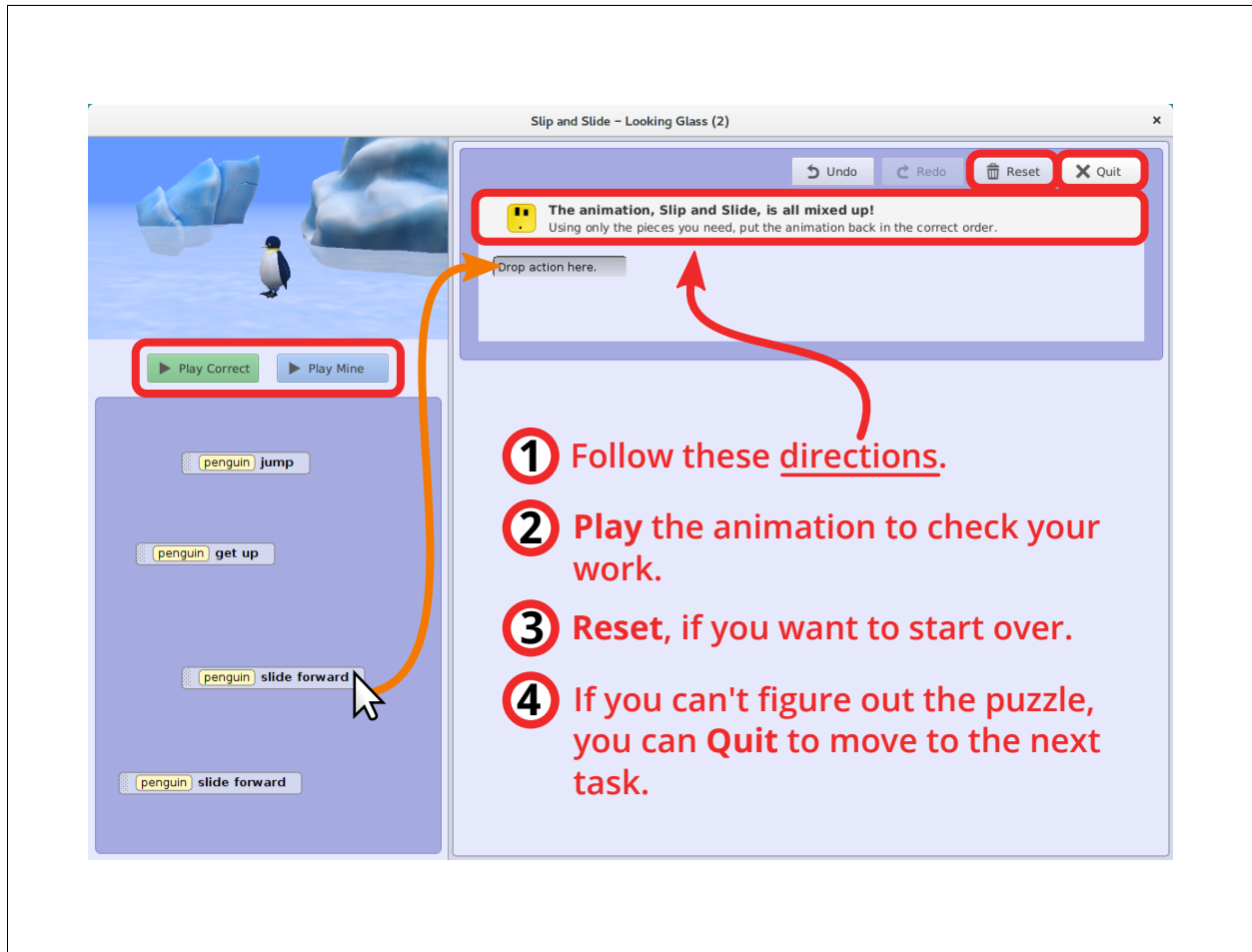


Figure F.4: Instruction sheet for the training phase. Participants were able to reference this sheet for all training tasks.

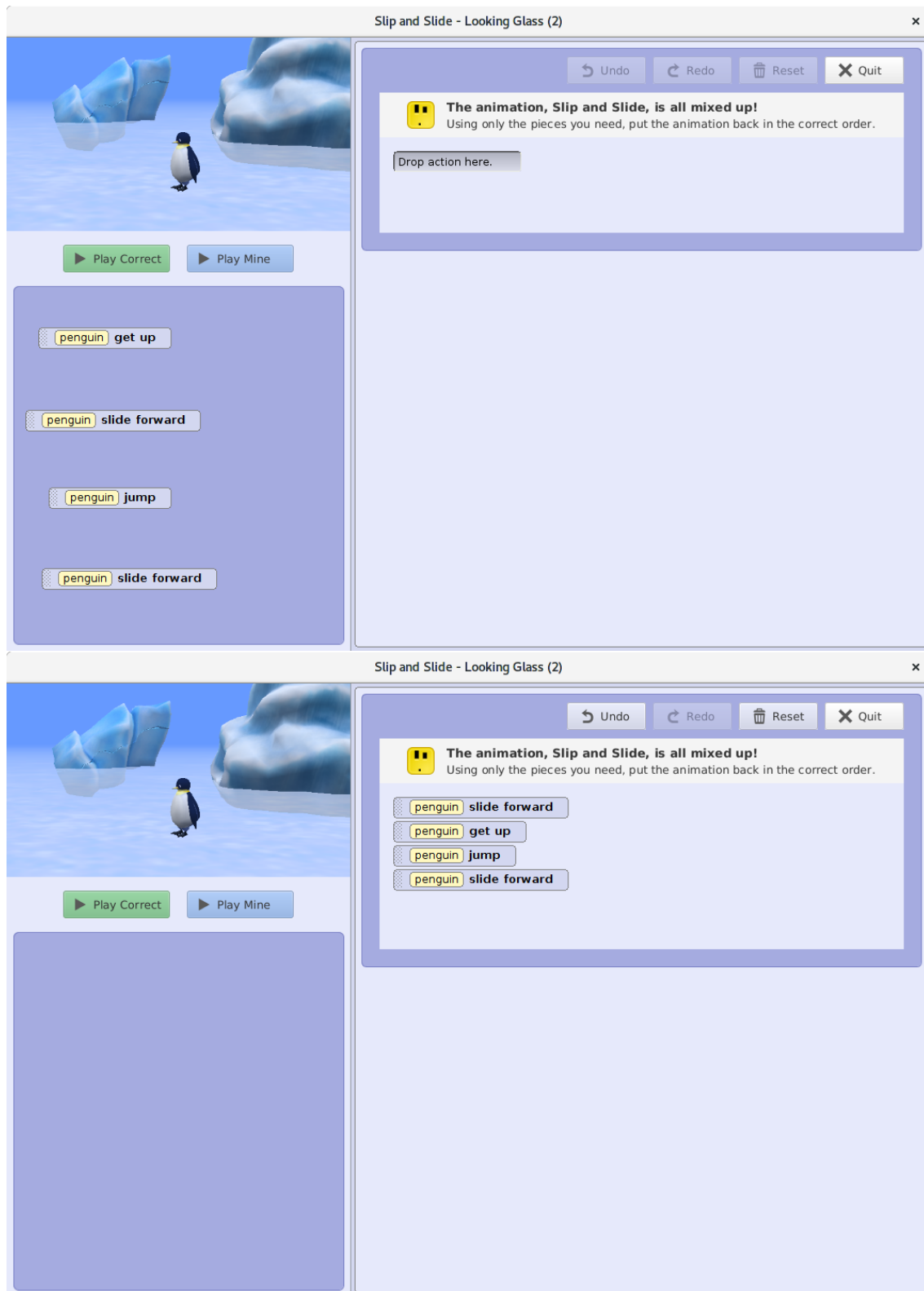


Figure F.5: Training familiarization task. Initial state (*top*); completed state (*bottom*).

Slip and Slide - Looking Glass (2)

Undo
Redo
Reset
Quit

The animation, Slip and Slide, is all mixed up!
Using only the pieces you need, put the animation back in the correct order.

penguin slide forward

penguin get up

In completing this task I invested:

Very, Very Low Mental Effort	Very Low Mental Effort	Low Mental Effort	Rather Low Mental Effort	Neither Low nor High Mental Effort	Rather High Mental Effort	High Mental Effort	Very High Mental Effort	Very, Very High Mental Effort
------------------------------	------------------------	-------------------	--------------------------	------------------------------------	---------------------------	--------------------	-------------------------	-------------------------------

How easy or difficult was this task?

Extremely Easy	Very Easy	Moderately Easy	Slightly Easy	Neither Easy nor Difficult	Slightly Difficult	Moderately Difficult	Very Difficult	Extremely Difficult
----------------	-----------	-----------------	---------------	----------------------------	--------------------	----------------------	----------------	---------------------

Done

Figure F.6: Training task survey. After each training task, participants complete this survey.

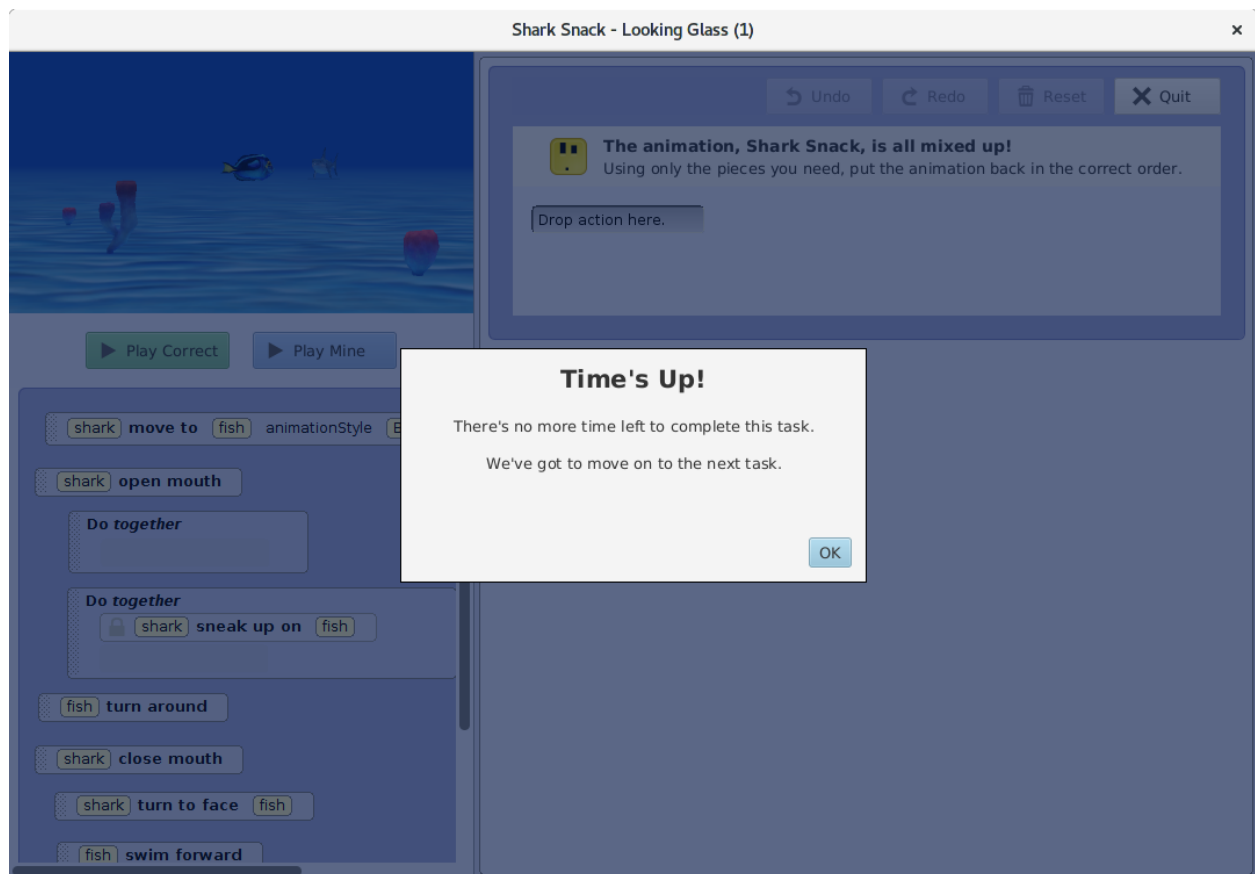


Figure F.7: This dialog is shown to participants if they have not completed the training task within the allotted time.

F.1.1 Control Training Tasks: No Distractors

None of the training tasks in the control condition contain distractors. Notice in Figures F.8–F.13 that in the completed state there are no statements in the unused statement bin on the left.

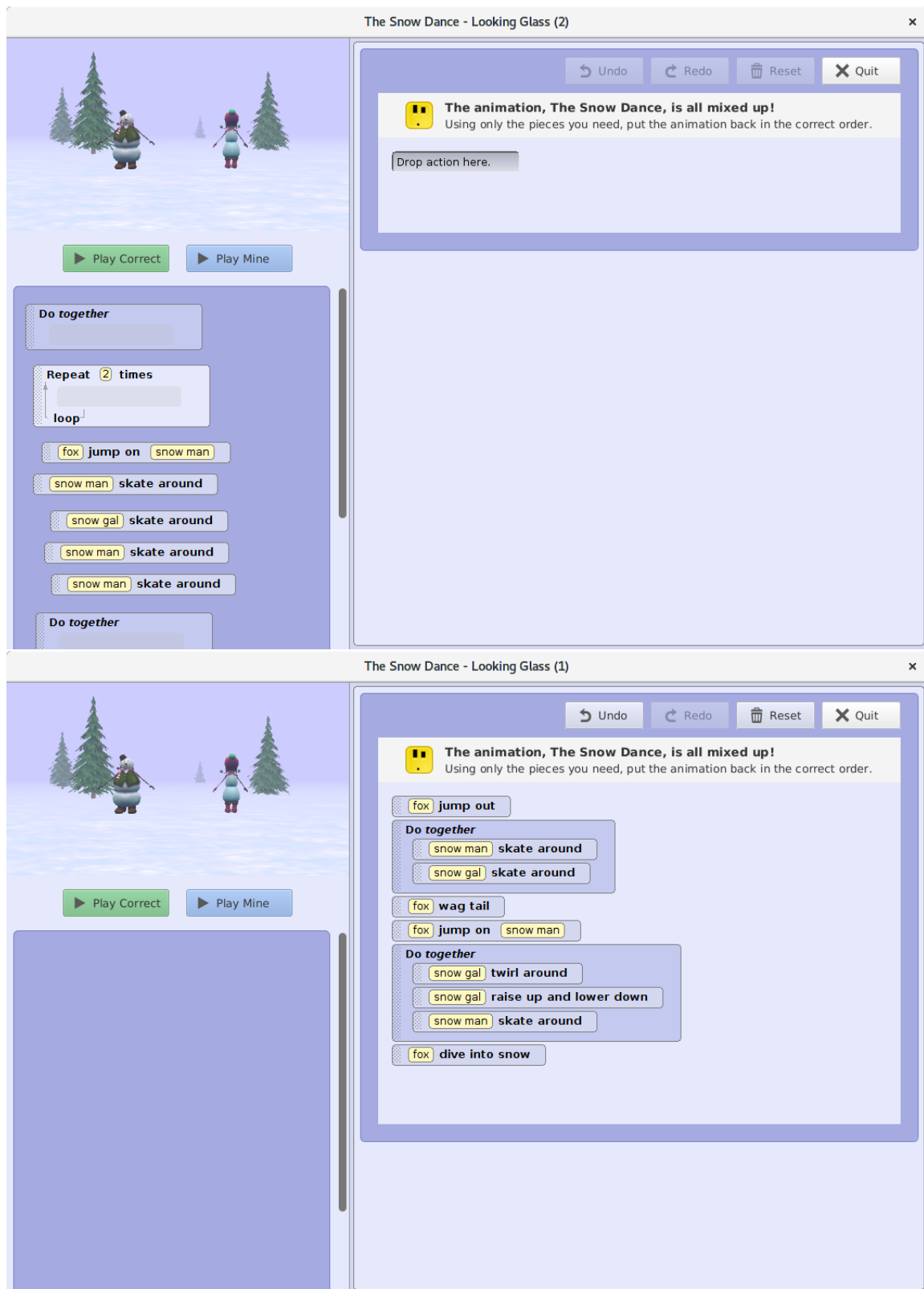


Figure F.8: Control training task 1. Initial state (*top*); completed state (*bottom*).

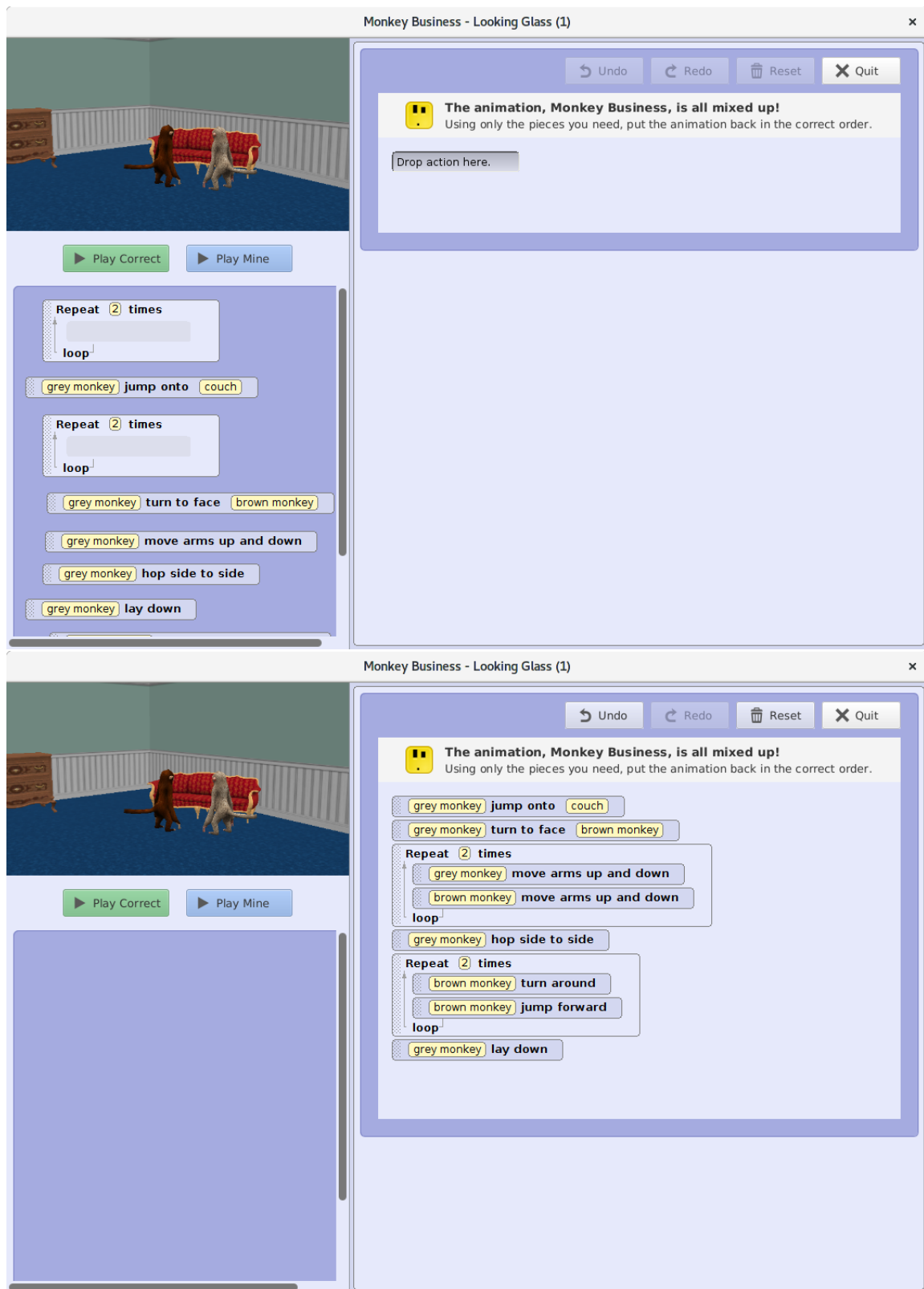


Figure F.9: Control training task 2. Initial state (*top*); completed state (*bottom*).

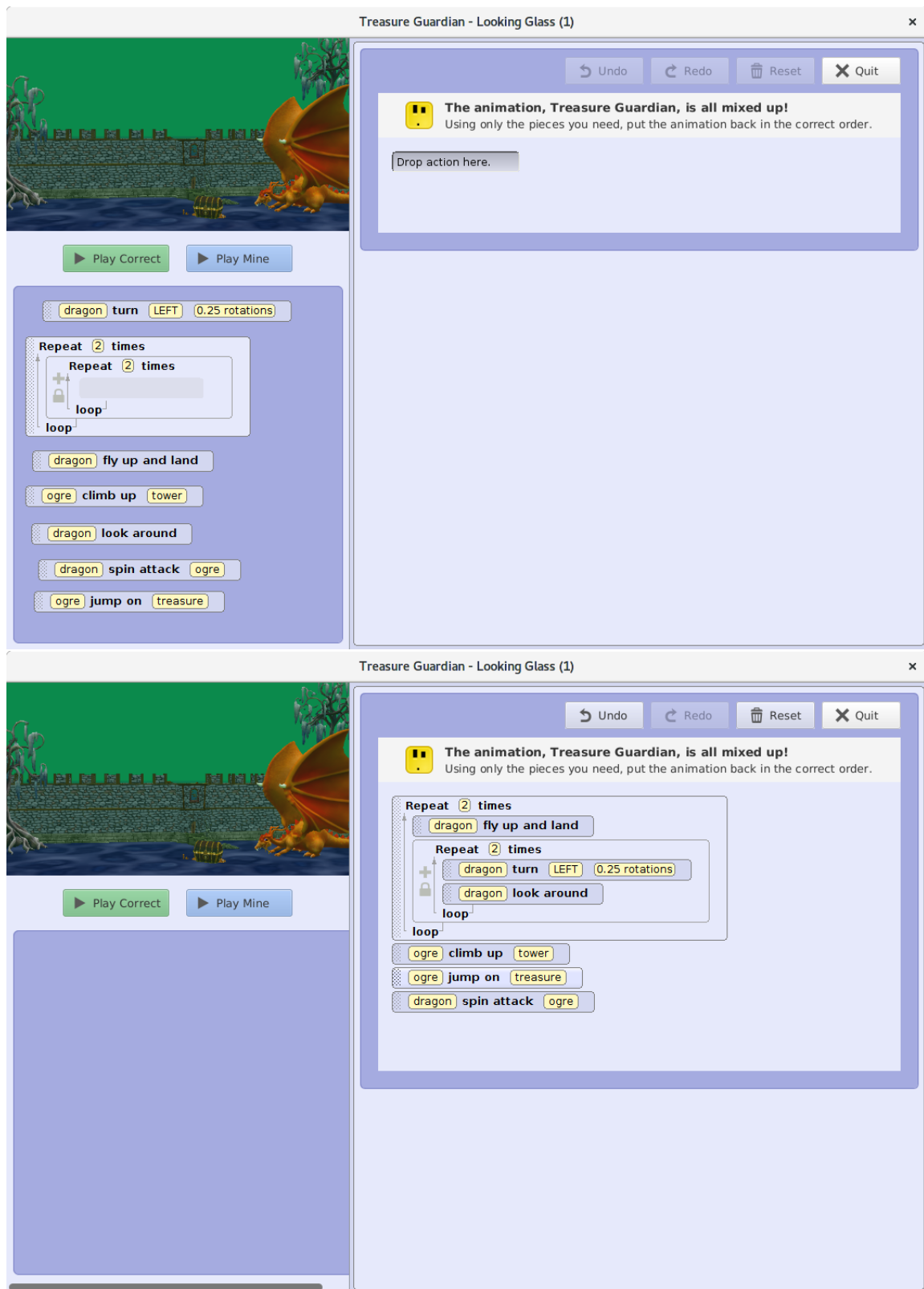


Figure F.10: Control training task 3. Initial state (*top*); completed state (*bottom*).

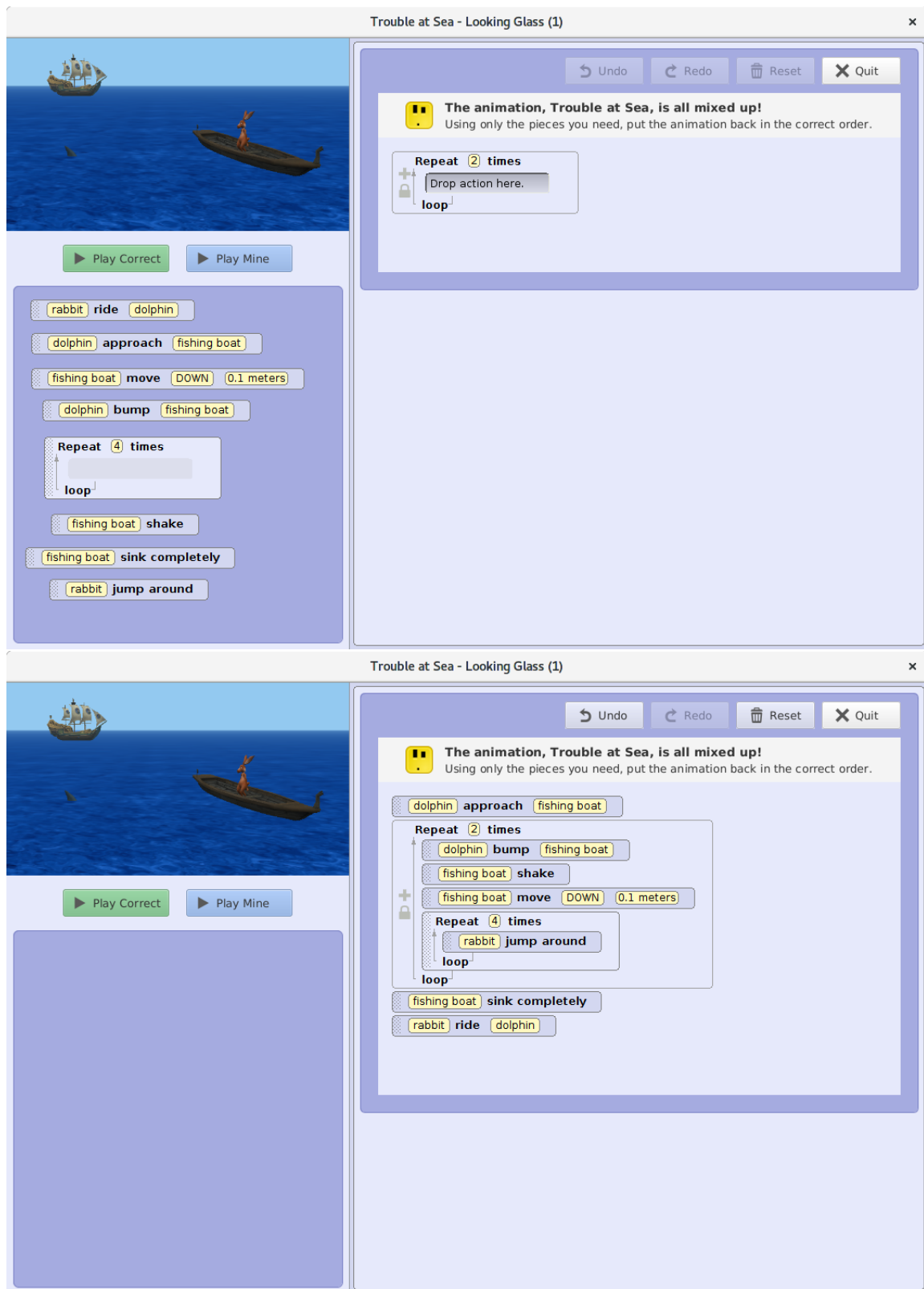


Figure F.11: Control training task 4. Initial state (*top*); completed state (*bottom*).

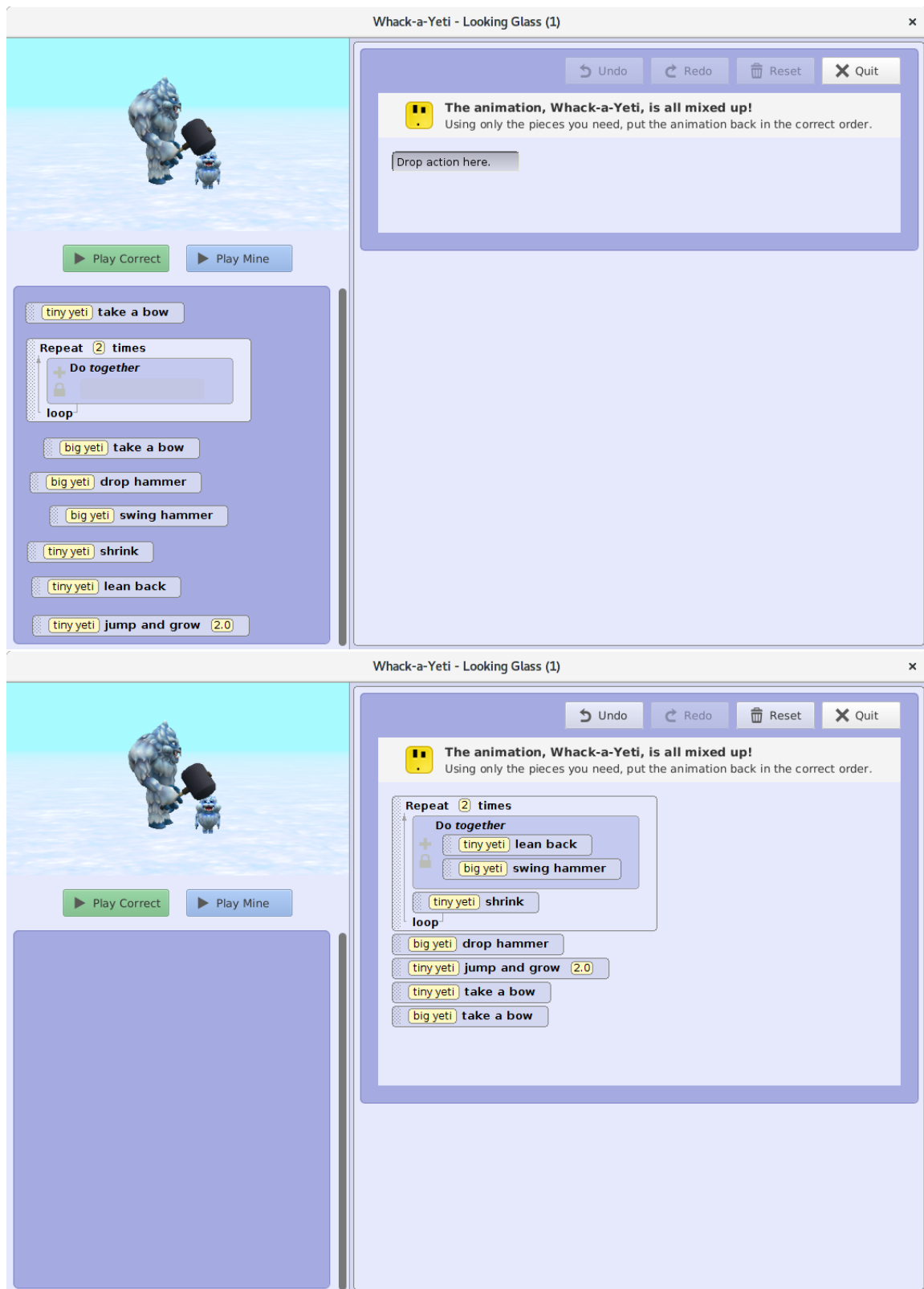


Figure F.12: Control training task 5. Initial state (*top*); completed state (*bottom*).

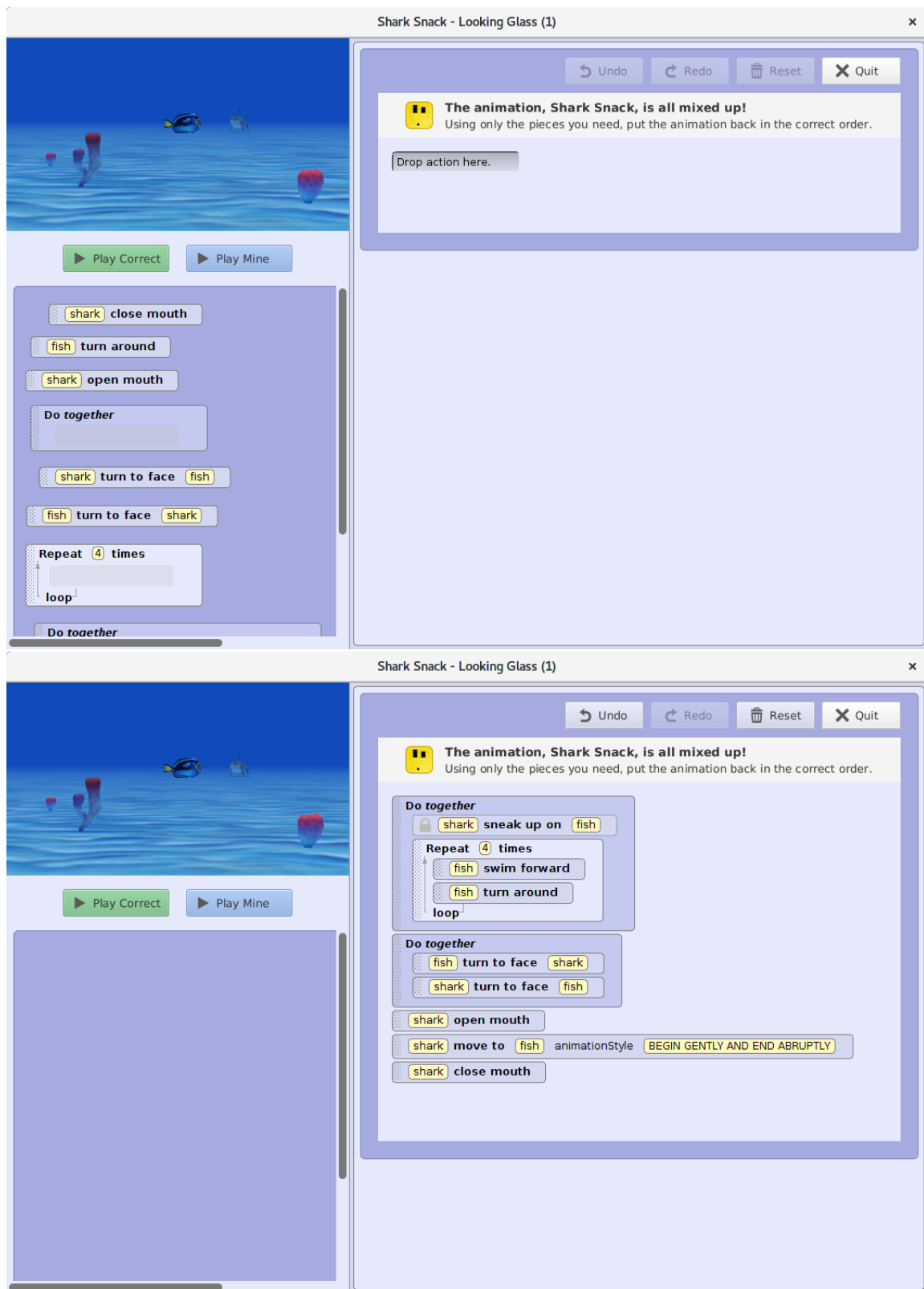


Figure F.13: Control training task 6. Initial state (*top*); completed state (*bottom*).

F.1.2 Experimental Training Tasks: Distractors

All of the training tasks in the experimental condition contain distractors. Notice in Figures F.14–F.19 that in the completed state there are ‘leftover’ statements in the unused statement bin on the left.

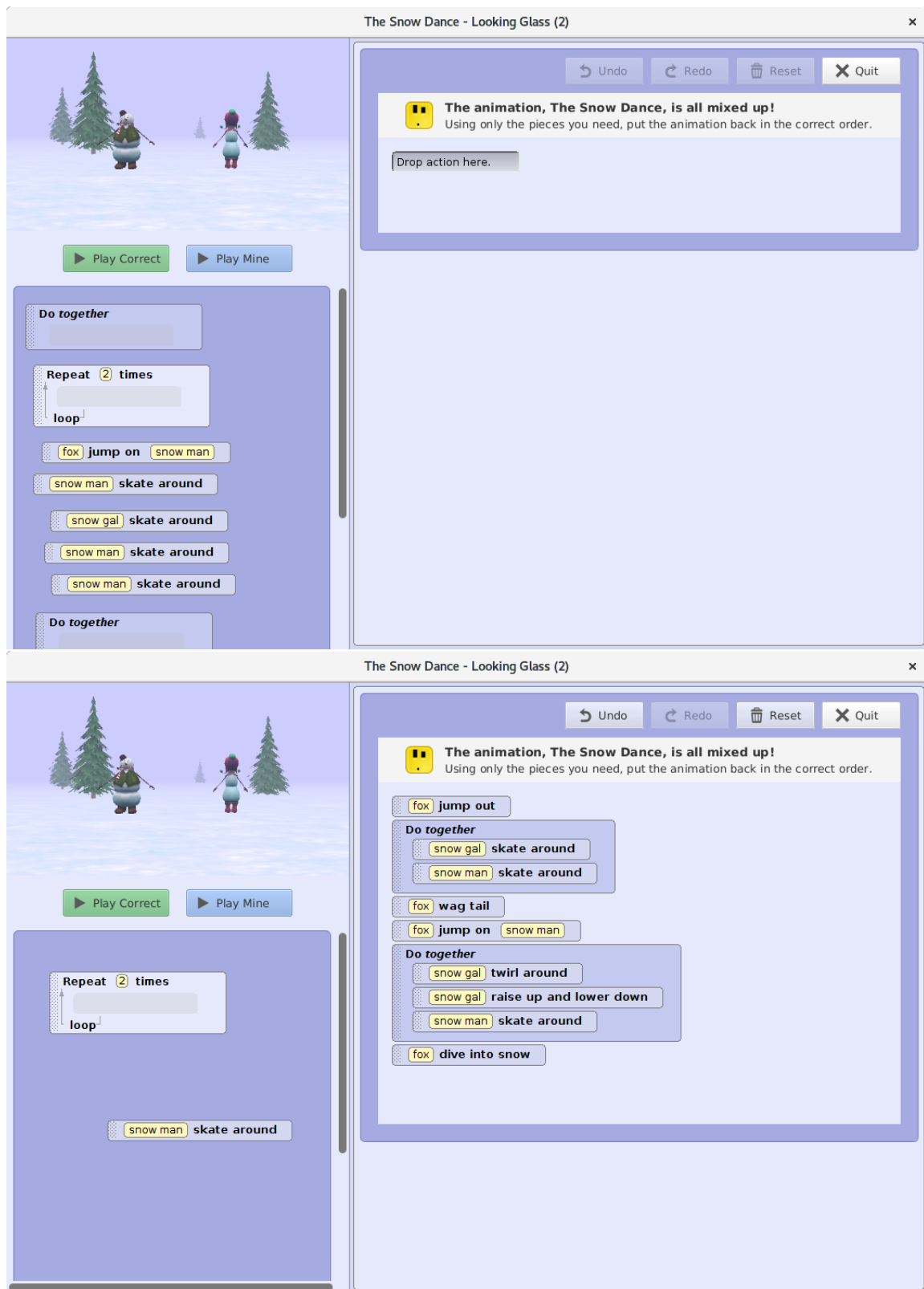


Figure F.14: Distractors training task 1. Initial state (*top*); completed state (*bottom*).

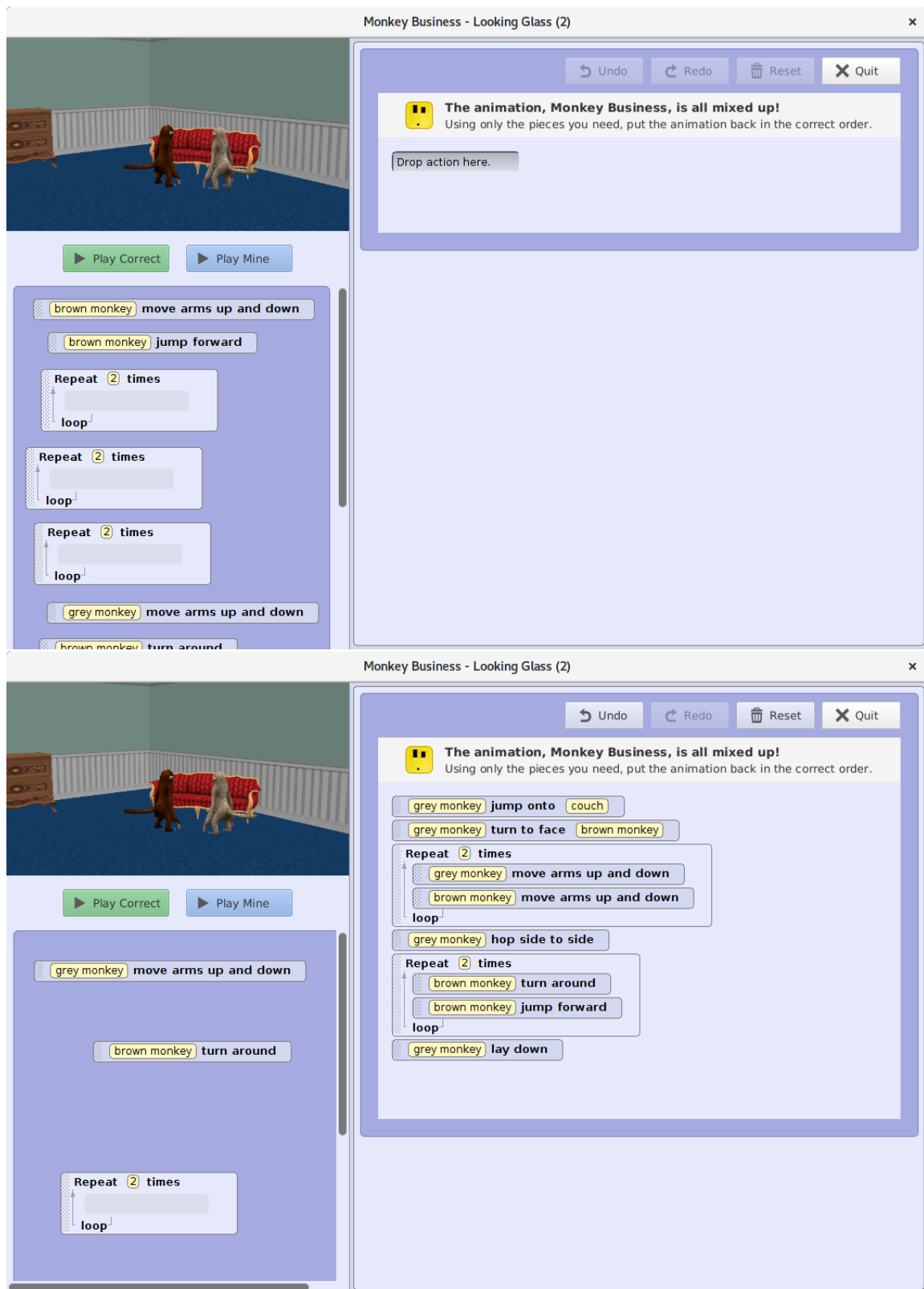


Figure F.15: Distractors training task 2. Initial state (*top*); completed state (*bottom*).



Figure F.16: Distractors training task 3. Initial state (*top*); completed state (*bottom*).

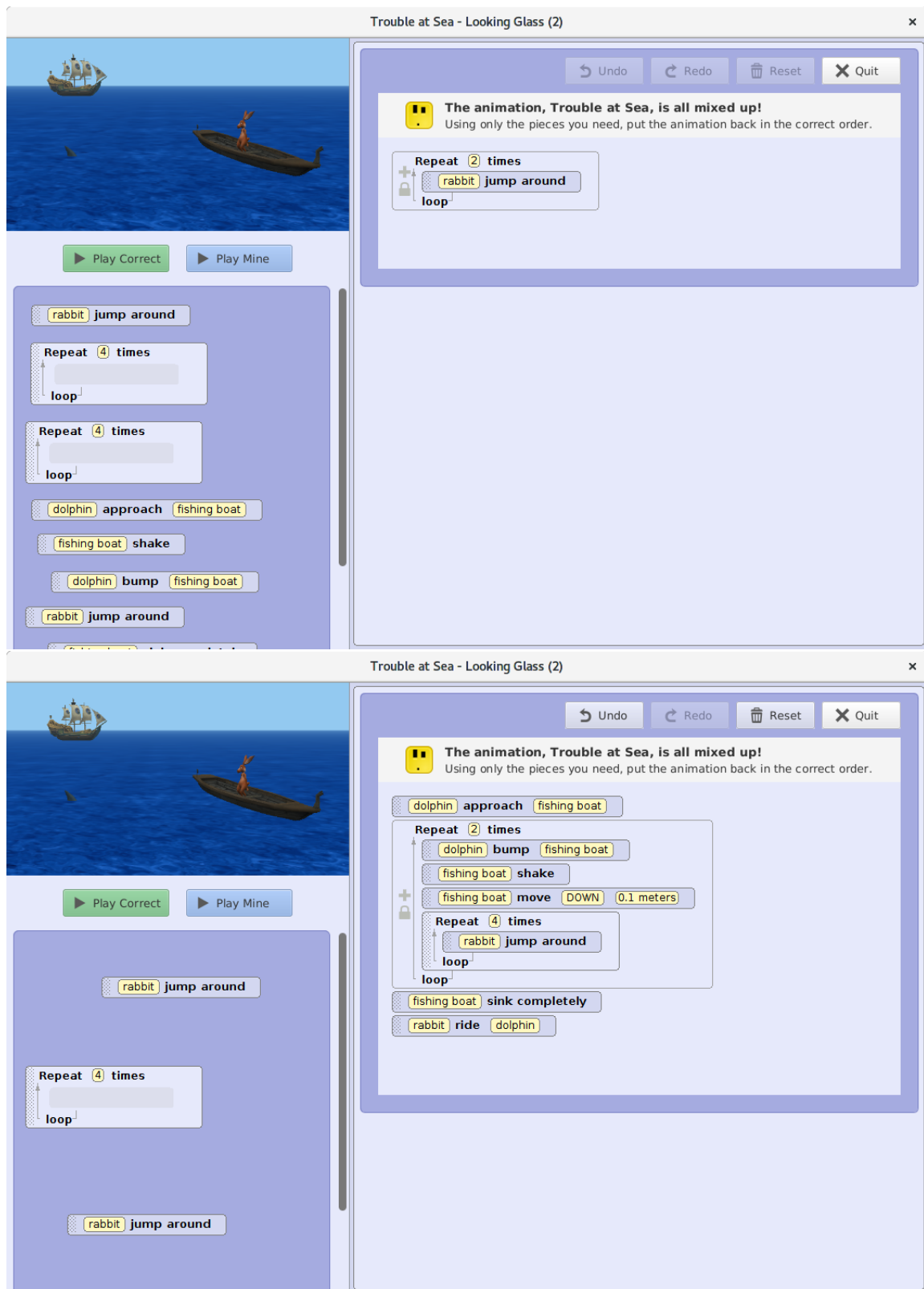


Figure F.17: Distractors training task 4. Initial state (*top*); completed state (*bottom*).

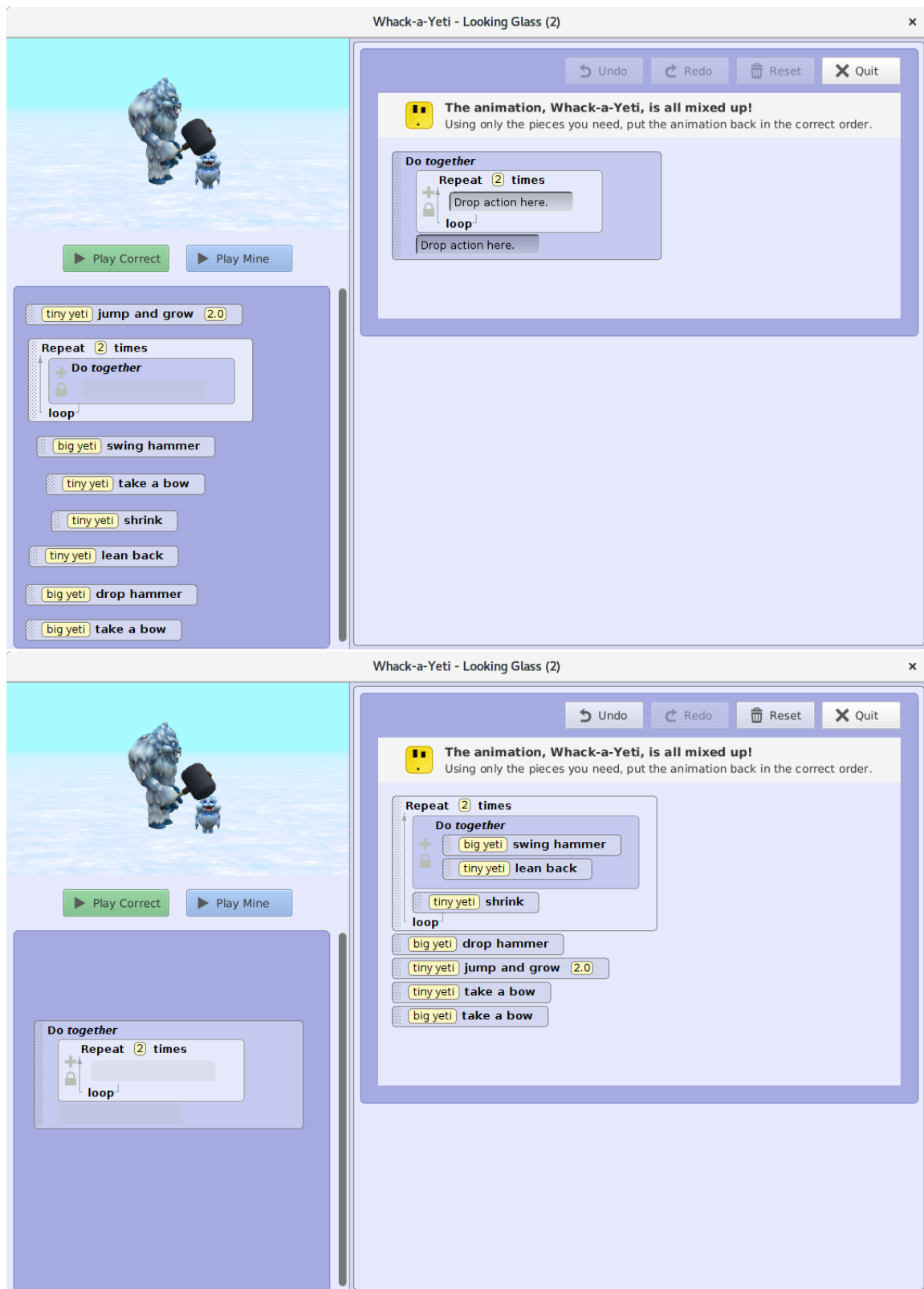


Figure F.18: Distractors training task 5. Initial state (*top*); completed state (*bottom*).

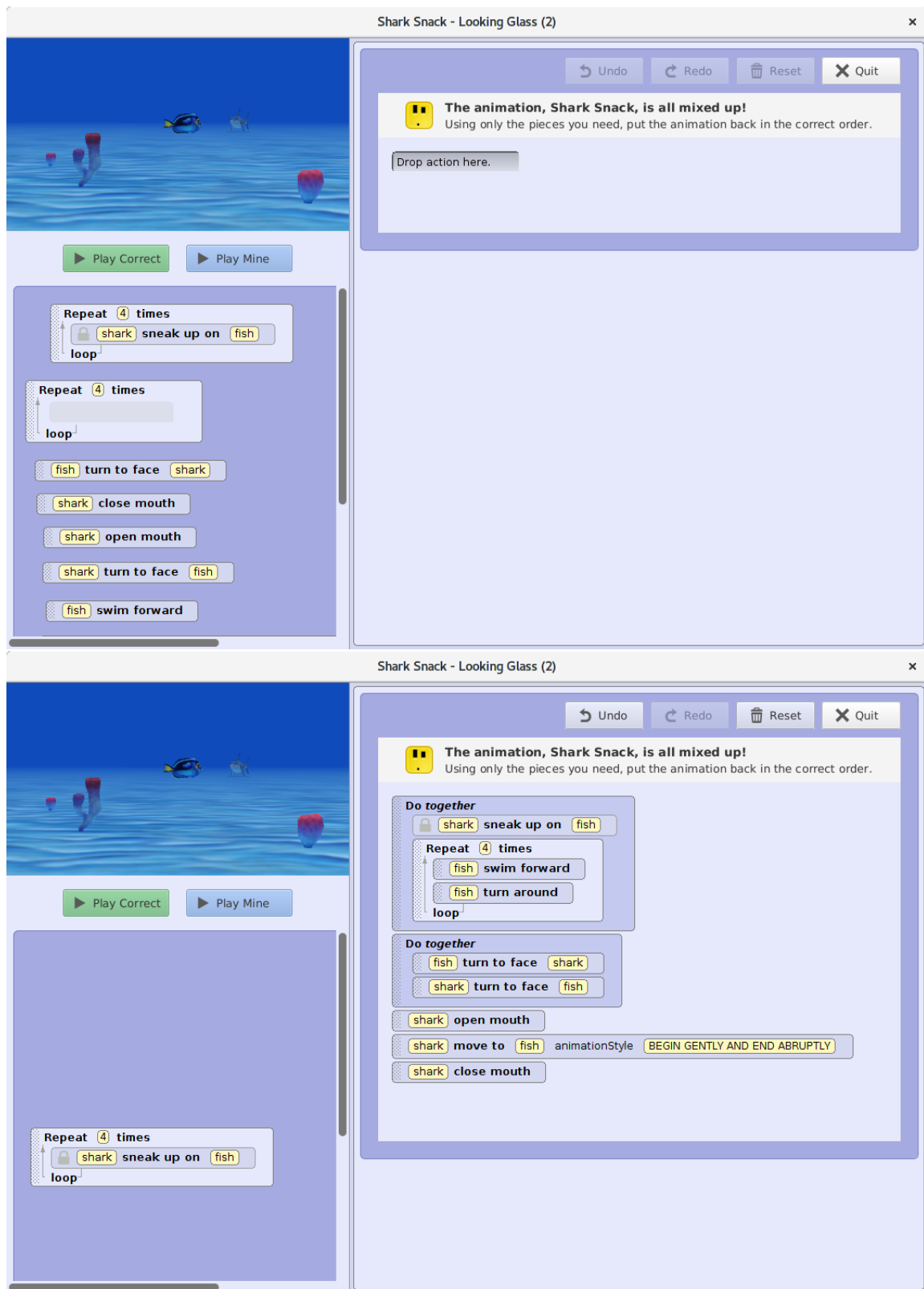


Figure F.19: Distractors training task 6. Initial state (*top*); completed state (*bottom*).

F.1.3 Post Training Task Surveys

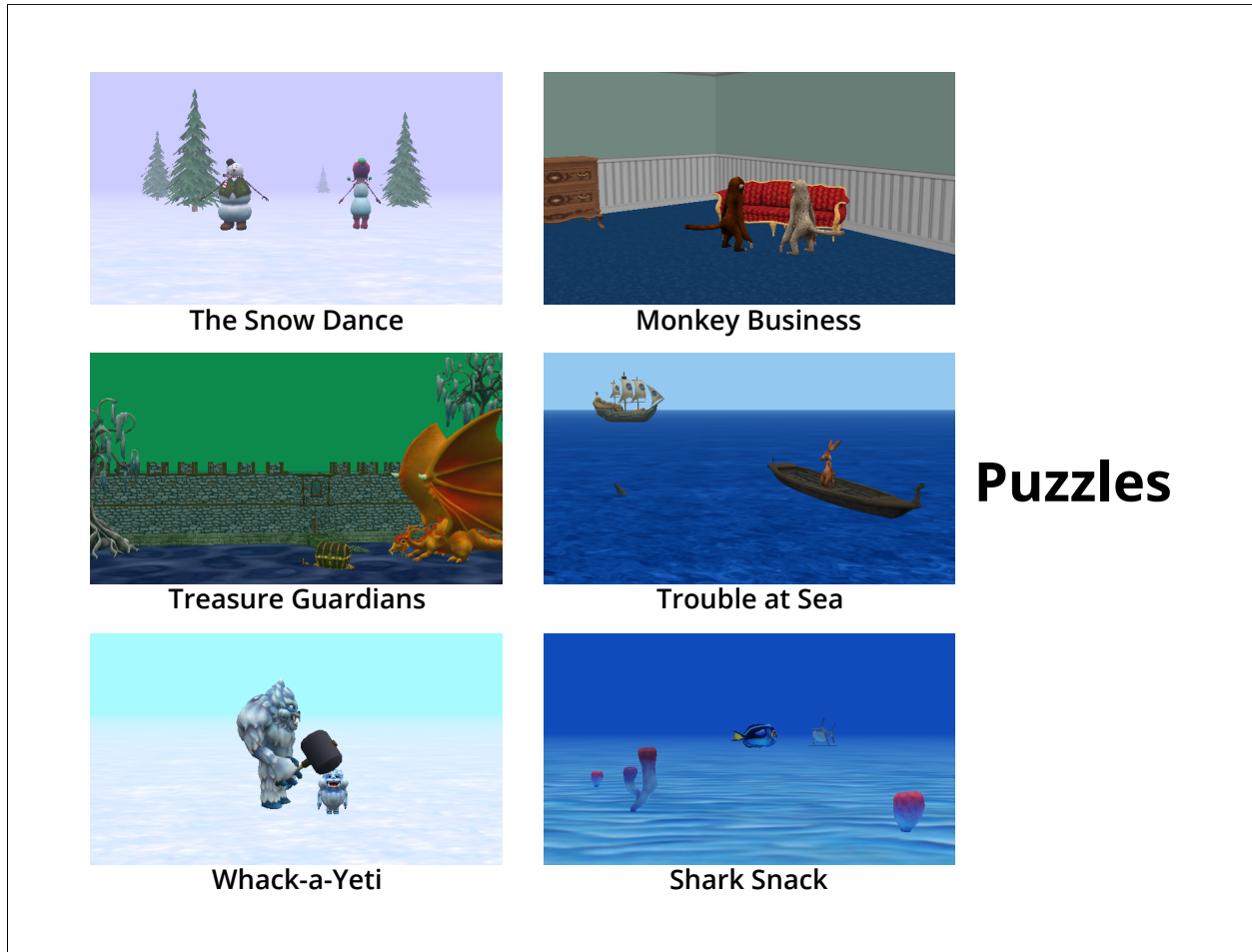


Figure F.20: Before completing the following training phase surveys, we gave participants this reminder sheet.

The following questions refer to **only** to the **last 6 puzzles** that you completed.

Circle the answer that best represents your **overall** impression of these puzzles.

In completing these **puzzles** I invested:

Very, Very Low Mental Effort	Very Low Mental Effort	Low Mental Effort	Rather Low Mental Effort	Neither Low nor High Mental Effort	Rather High Mental Effort	High Mental Effort	Very High Mental Effort	Very, Very High Mental Effort
------------------------------	------------------------	-------------------	--------------------------	------------------------------------	---------------------------	--------------------	-------------------------	-------------------------------

How easy or difficult were these **puzzles**?

Extremely Easy	Very Easy	Moderately Easy	Slightly Easy	Neither Easy nor Difficult	Slightly Difficult	Moderately Difficult	Very Difficult	Extremely Difficult
----------------	-----------	-----------------	---------------	----------------------------	--------------------	----------------------	----------------	---------------------

Figure F.21: Overall cognitive load survey for all training tasks.

Instructions:
*All of the following questions refer **only** to the **last 6 puzzles** that you just finished. Answer each question by choosing a number that best represents how you felt.*

	not at all the case					somewhat the case					completely the case
The topics covered in the activity were very complex.	0	1	2	3	4	5	6	7	8	9	10
The activity covered program code that I perceived as very complex.	0	1	2	3	4	5	6	7	8	9	10
The activity covered concepts and definitions that I perceived as very complex.	0	1	2	3	4	5	6	7	8	9	10
The instructions and/or explanations during the activity were very unclear.	0	1	2	3	4	5	6	7	8	9	10
The instructions and/or explanations were, in terms of learning, very ineffective.	0	1	2	3	4	5	6	7	8	9	10
The instructions and/or explanations were full of unclear language.	0	1	2	3	4	5	6	7	8	9	10
The activity really enhanced my understanding of the topic(s) covered.	0	1	2	3	4	5	6	7	8	9	10
The activity really enhanced my knowledge and understanding of computing / programming.	0	1	2	3	4	5	6	7	8	9	10
The activity really enhanced my understanding of the program code covered.	0	1	2	3	4	5	6	7	8	9	10
The activity really enhanced my understanding of the concepts and definitions.	0	1	2	3	4	5	6	7	8	9	10

Figure F.22: CS CLCS for the training phase.

F.2 Transfer Phase



Figure F.23: Mechanics instruction sheet for the transfer phase. Participants were able to reference this sheet for all transfer tasks.



Figure F.24: Transfer mechanics familiarization task. Initial state (*top*); completed state (*bottom*).

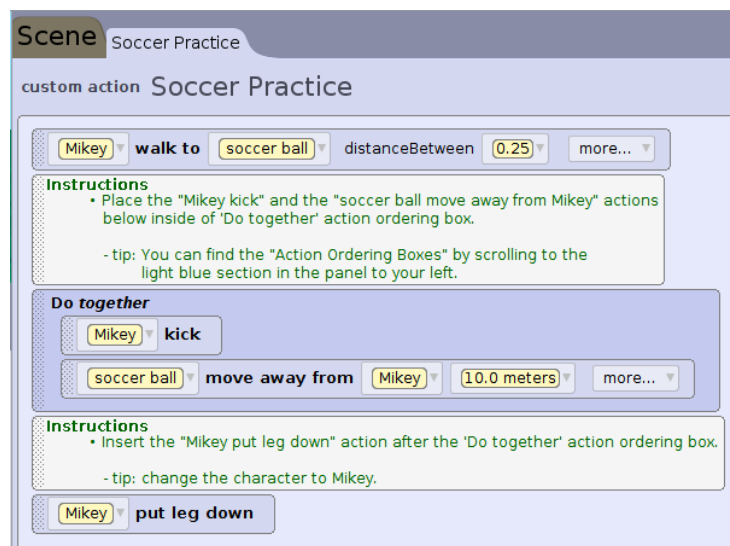


Figure F.25: Transfer mechanics familiarization task solution. Participants are given this solution sheet to correct their mistakes in the mechanics familiarization task before moving on to the second familiarization task.

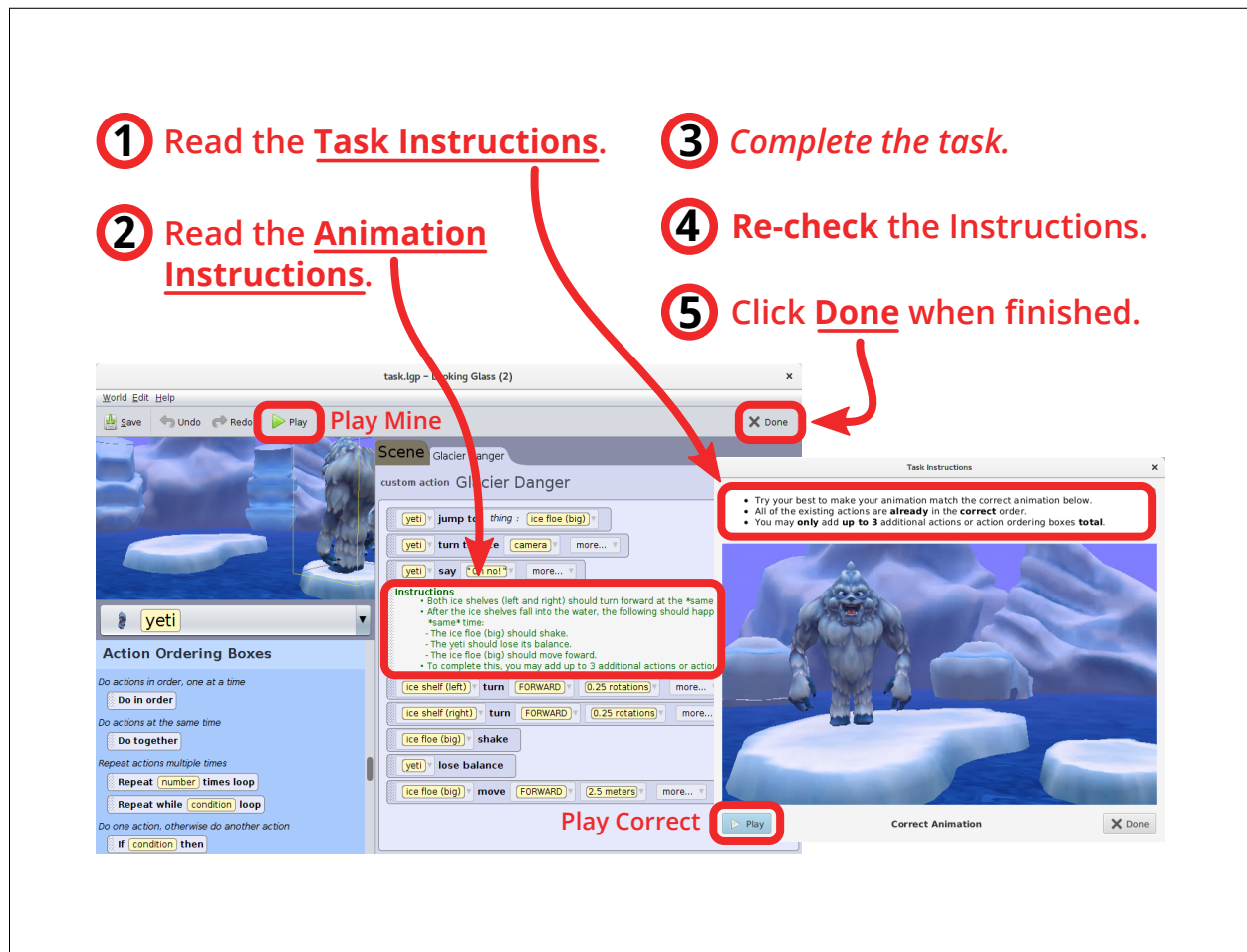


Figure F.26: Transfer task instruction sheet. Participants were able to reference this sheet for all transfer tasks.

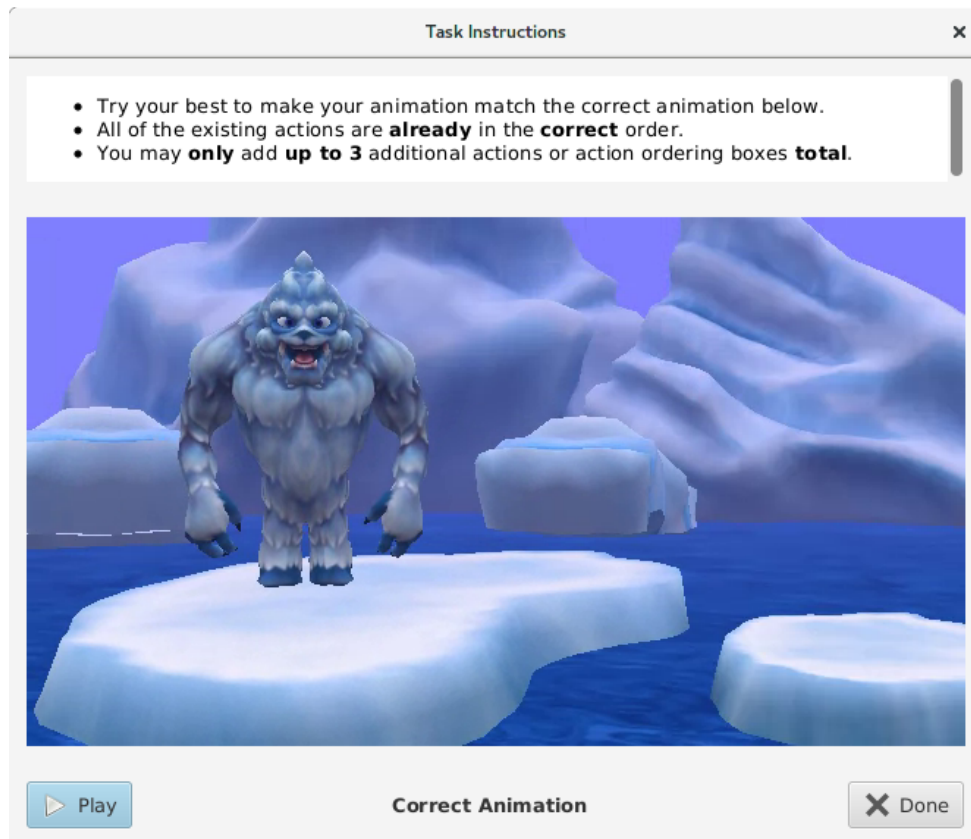


Figure F.27: Onscreen transfer task instructions. This window shows the correct output for each transfer task. This window is shown for all transfer tasks.



Figure F.28: Transfer instructions familiarization task. Initial state (*top*); completed state (*bottom*).

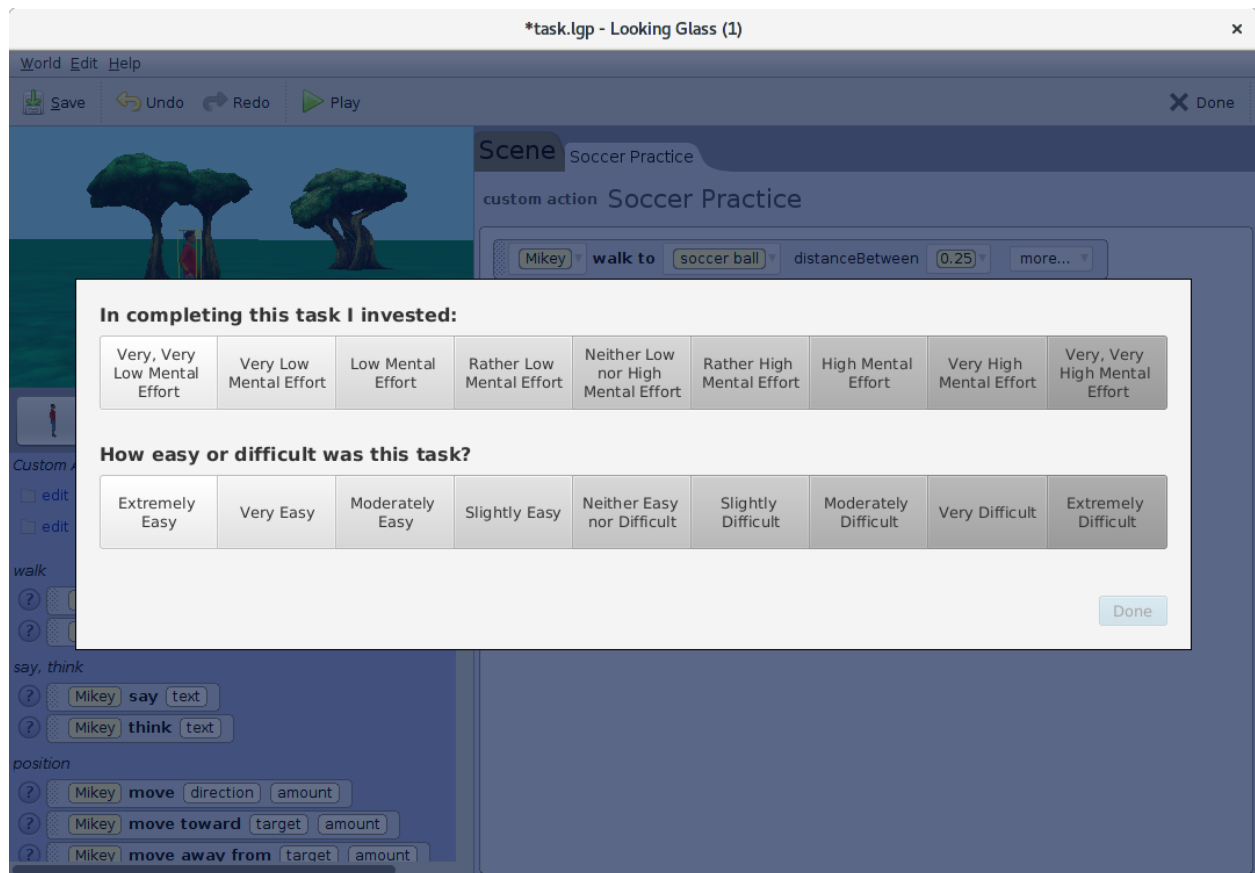


Figure F.29: Transfer task survey. After each transfer task, participants complete this survey.

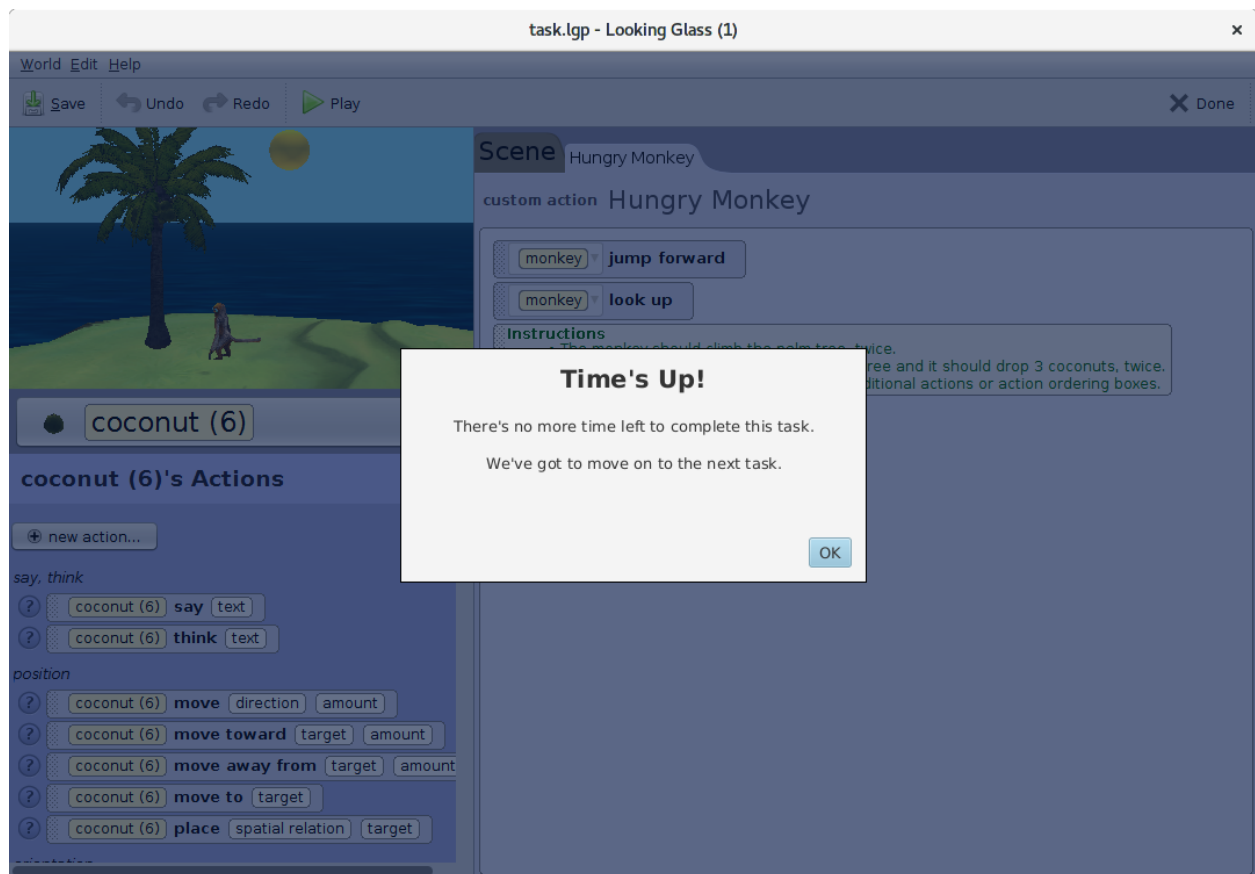


Figure F.30: This dialog is shown to participants if they have not completed the transfer task within the allotted time.

F.2.1 Transfer Tasks

Figures F.31–F.33 show initial and completed state of each transfer task.

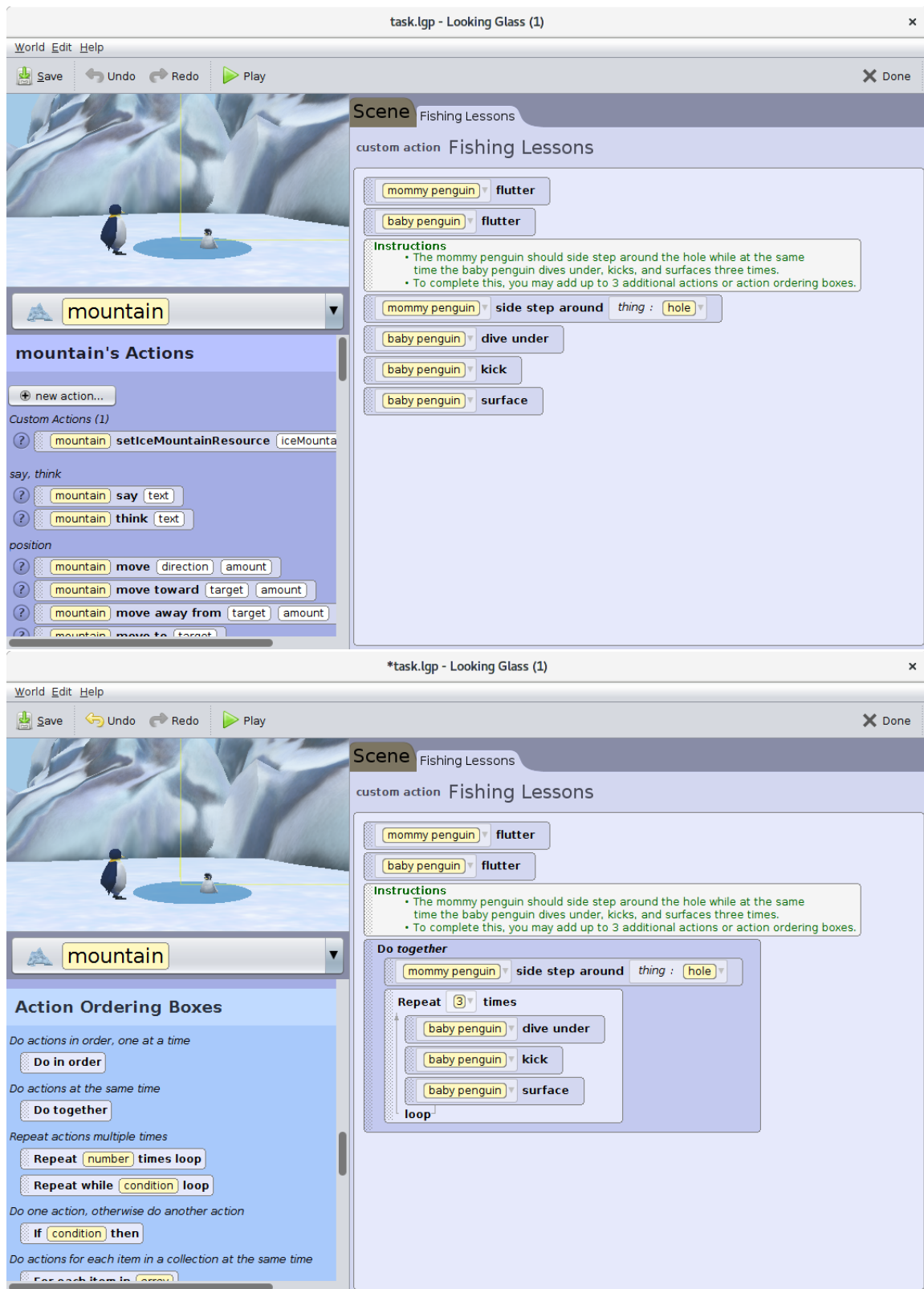


Figure F.31: *Do Together { Repeat }* transfer task. Initial state (*top*) and completed state (*bottom*).

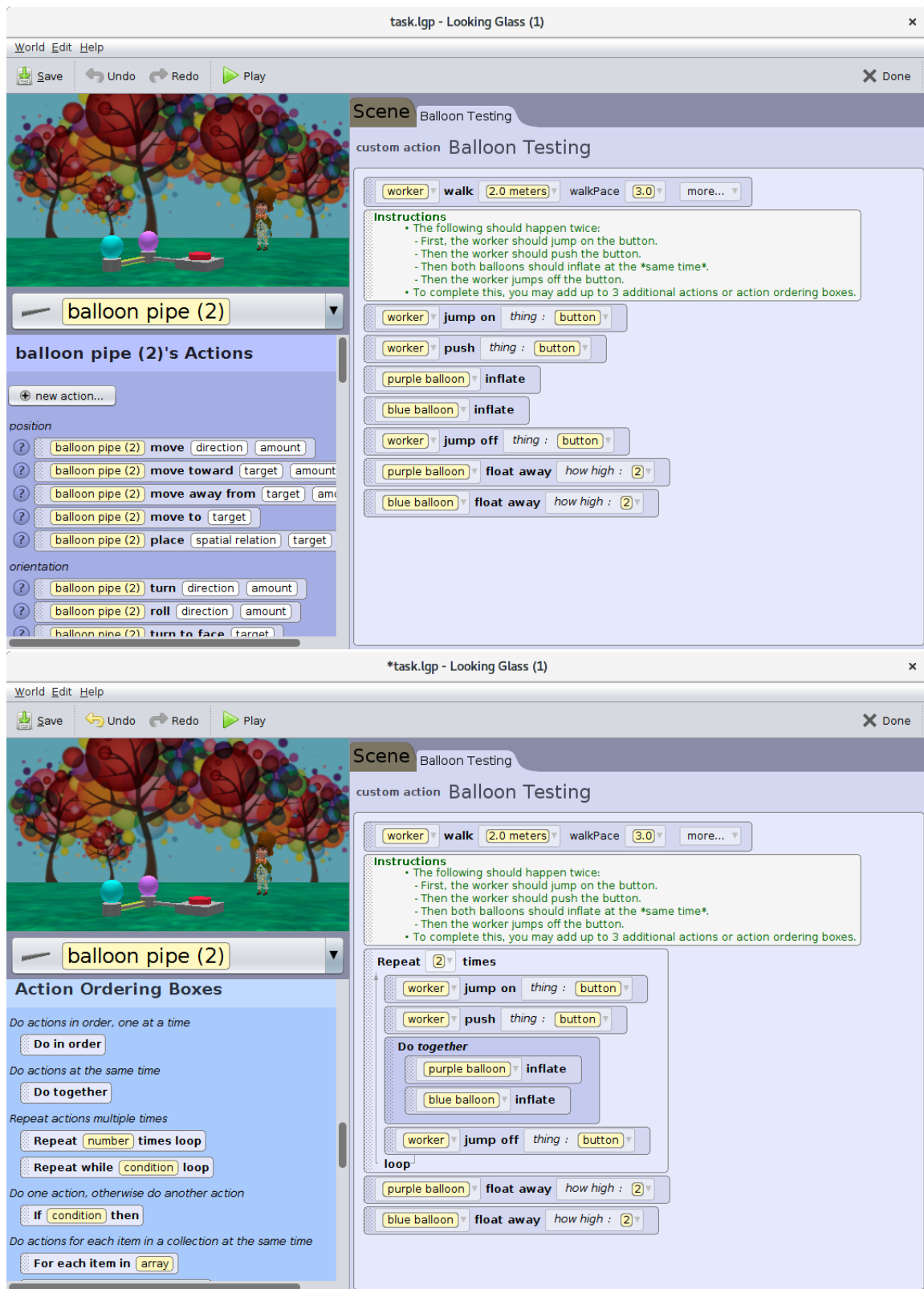


Figure F.32: *Repeat { Do Together }* transfer task. Initial state (*top*) and completed state (*bottom*).

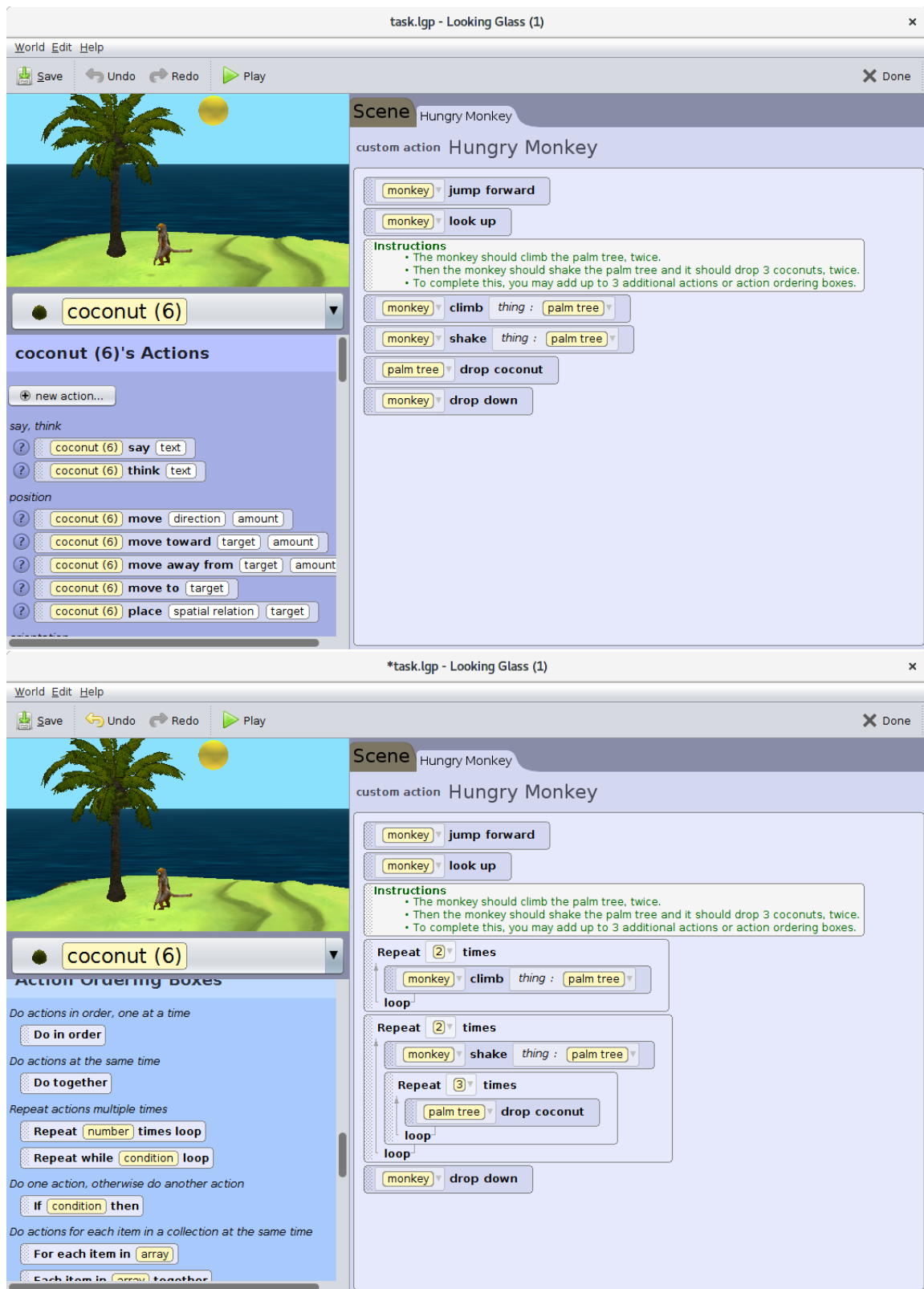


Figure F.33: *Repeat { Repeat }* transfer task. Initial state (*top*) and completed state (*bottom*).

F.2.2 Post Transfer Task Surveys

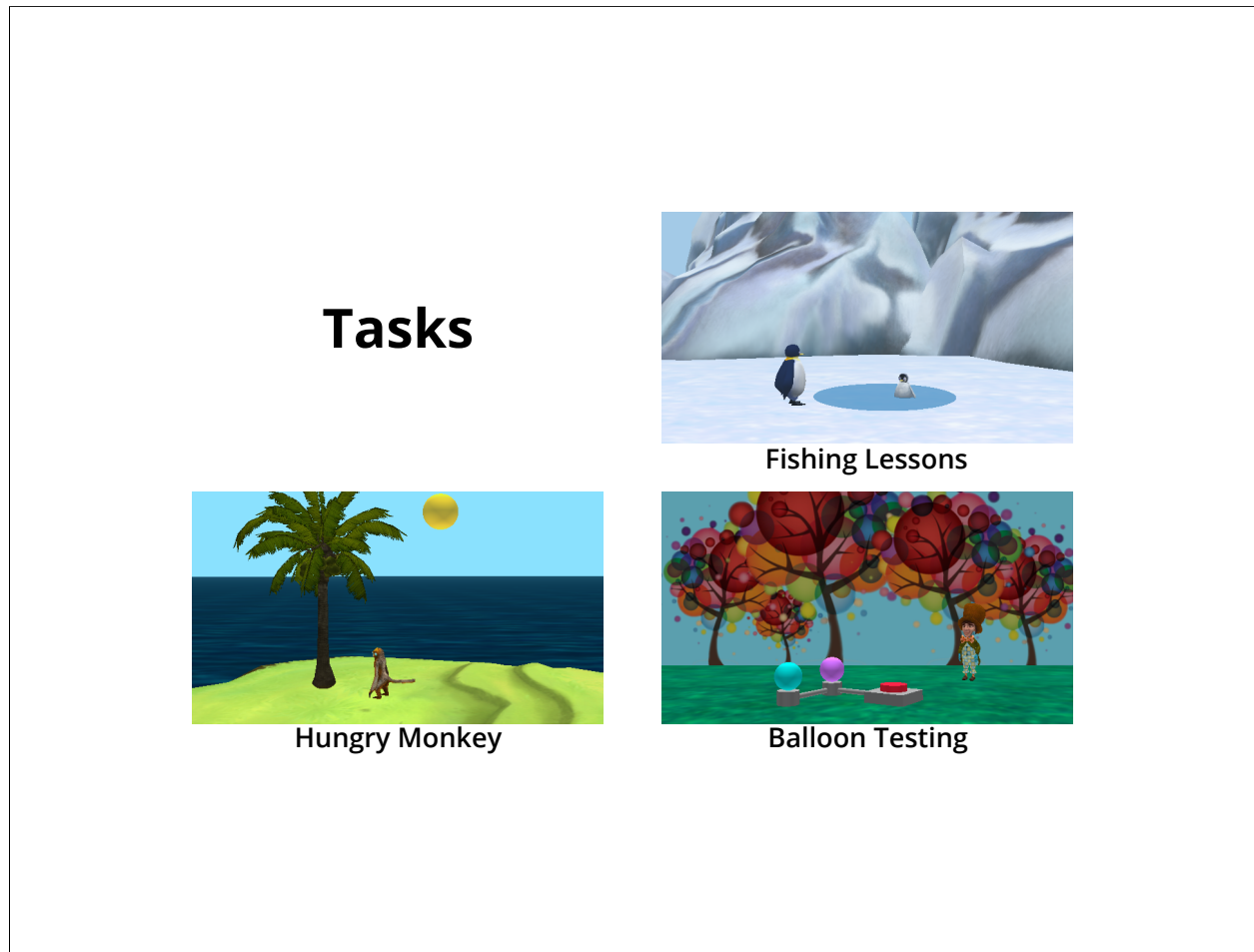


Figure F.34: Before completing the following transfer phase surveys, we gave participants this reminder sheet.

The following questions refer to **only** to the **last 3 tasks** that you completed.

Circle the answer that best represents your **overall** impression of these tasks.

In completing these **tasks** I invested:

Very, Very Low Mental Effort	Very Low Mental Effort	Low Mental Effort	Rather Low Mental Effort	Neither Low nor High Mental Effort	Rather High Mental Effort	High Mental Effort	Very High Mental Effort	Very, Very High Mental Effort
------------------------------	------------------------	-------------------	--------------------------	------------------------------------	---------------------------	--------------------	-------------------------	-------------------------------

How easy or difficult were these **tasks**?

Extremely Easy	Very Easy	Moderately Easy	Slightly Easy	Neither Easy nor Difficult	Slightly Difficult	Moderately Difficult	Very Difficult	Extremely Difficult
----------------	-----------	-----------------	---------------	----------------------------	--------------------	----------------------	----------------	---------------------

Figure F.35: Overall cognitive load survey for all transfer tasks.

Instructions:
*All of the following questions refer **only** to the **last 3 tasks** that you just finished. Answer each question by choosing a number that best represents how you felt.*

	not at all the case					somewhat the case					completely the case
The topics covered in the activity were very complex.	0	1	2	3	4	5	6	7	8	9	10
The activity covered program code that I perceived as very complex.	0	1	2	3	4	5	6	7	8	9	10
The activity covered concepts and definitions that I perceived as very complex.	0	1	2	3	4	5	6	7	8	9	10
The instructions and/or explanations during the activity were very unclear.	0	1	2	3	4	5	6	7	8	9	10
The instructions and/or explanations were, in terms of learning, very ineffective.	0	1	2	3	4	5	6	7	8	9	10
The instructions and/or explanations were full of unclear language.	0	1	2	3	4	5	6	7	8	9	10
The activity really enhanced my understanding of the topic(s) covered.	0	1	2	3	4	5	6	7	8	9	10
The activity really enhanced my knowledge and understanding of computing / programming.	0	1	2	3	4	5	6	7	8	9	10
The activity really enhanced my understanding of the program code covered.	0	1	2	3	4	5	6	7	8	9	10
The activity really enhanced my understanding of the concepts and definitions.	0	1	2	3	4	5	6	7	8	9	10

Figure F.36: CS CLCS for the transfer phase.

Appendix G

Summative Evaluation III Materials

Figures G.1–G.24 are the materials used in the summative evaluation presented in Chapter 5. Additionally, Section G.1 contains the semi-structured interviews used in the evaluation.

COMPUTING HISTORY SURVEY

1. How old are you? _____
2. What is your current grade in school? _____
3. What is your gender? (choose **one**)
 - a) Female
 - b) Male
 - c) Not specified
4. What kind of school do you go to? (choose **one**)
 - a) Public school
 - b) Private school
 - c) Home-schooled
5. Have you ever participated in a Looking Glass study or event? (choose **one**)
 - a) Don't Know
 - b) Yes
 - c) No
6. Have you ever coded or written a computer program? (choose **one**)
 - a) Don't know
 - b) Yes
 - c) No
7. Have you ever used the following computer coding or programming software? (circle **all** that apply)
 - a) code.org
 - b) Scratch
 - c) LEGO Mindstorms
 - d) Looking Glass or Alice
 - e) Hopscotch
 - f) Robotics
 - g) Programming languages (examples: Javascript, Python, C++, Java, Visual Basic, C#)
 - h) None
 - i) Other: _____
8. Have you participated in the following coding or programming activities? (circle **all** that apply)
 - a) Hour of code. (code.org)
 - b) CoderDojo. (coderdojo.com)
 - c) Coding or programming at school as part of a classroom activity.
 - d) A coder or programmer camp or after school workshop.
 - e) Coding or programming at an event (examples: scouting, academy of science, science center)
 - f) I code or program a computer at home.
 - g) None – I have never participated in a programming activity.
 - h) Other: _____
9. Have you spent more than **three (3)** hours coding or programming a computer in your life? (choose **one**)
 - a) Don't know
 - b) Yes
 - a) No

Figure G.1: Pre-study computing history survey.

G.1 Semi-Structured Interviews

G.1.1 Interview Materials

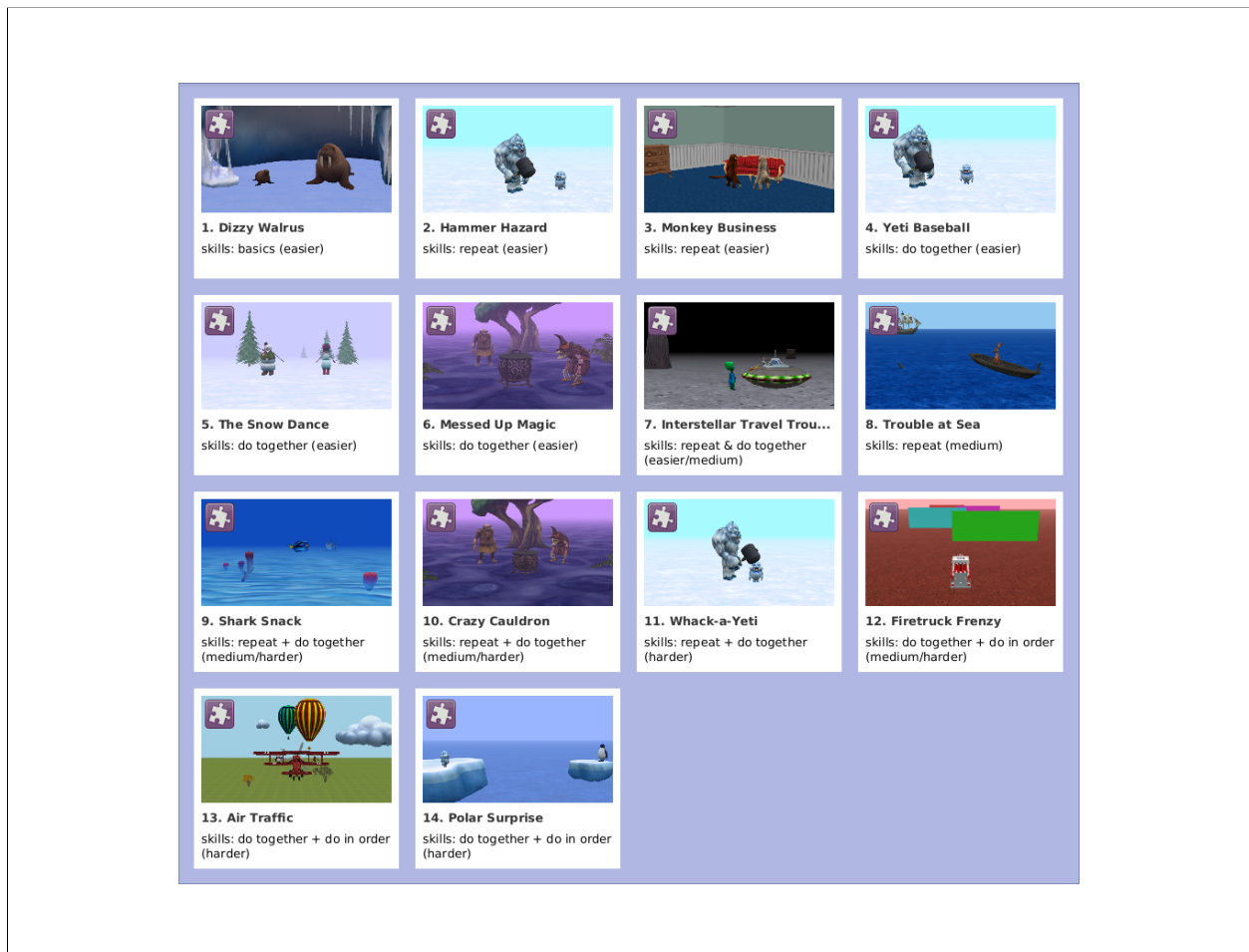


Figure G.2: During the semi-structured interviews, participants could refer back to this sheet to help remind them about the instructional tasks.

Novice	Advanced Beginner	Competent	Proficient	Expert
1	2	3	4	5

Extremely Easy	Very Easy	Moderately Easy	Slightly Easy	Neither Easy nor Difficult	Slightly Difficult	Moderately Difficult	Very Difficult	Extremely Difficult
1	2	3	4	5	6	7	8	9

Figure G.3: Several interview questions ask participants to rank their expertise or task difficulty. Participants are shown this sheet when they are asked these questions.

G.1.2 Pre-Study Interview

1. How good would you say you are at computer programming or coding?
2. *Show participant Figure G.3.* How would you rate your current programming or coding expertise? why?
3. *Show participant Figure G.3.* Do you feel that you can get to the next level, ____?
 - (a) *If yes,* How do you think you might do that?
4. How confident are you that you can program the things you want to create?

G.1.3 Post-Task Interview

1. How was it?
2. Did you enjoy doing the *tutorial/puzzle*? why?
3. What was the best part of the doing the *tutorial/puzzle*?
4. What was the worst part of the doing the *tutorial/puzzle*?
5. Do you feel the *tutorial/puzzle* was useful or helpful to you? why?
6. Was the experience of completing the *tutorial/puzzle* valuable to you in any way? why?
7. Did you learn anything new or did you acquire any new skills while doing the *tutorial/puzzle*?
 - (a) *If yes,* Did you know about x or use x before this, or is this your first time?
 - (b) Did it help you better understand x ? Do you feel you gained more experience with x ?

8. What different skills do you think you might have learned if you had done this animation as a *puzzle/tutorial* instead?
9. *Show participant Figure G.3.* How easy or difficult did you think this *tutorial/puzzle* was going to be before you started it? why?
10. *Show participant Figure G.3.* Now that you've finished it, how easy or difficult would you say this *tutorial/puzzle* actually was? why?
11. *Show participant Figure G.3.* How easy or difficult do you think this have been if you had done it instead as a *puzzle/tutorial*?
12. Was this *tutorial/puzzle* in any way frustrating or rewarding? why?
13. *Show participant Figure G.2.* You ranked your last *tutorial/puzzle* as _____. You rank this *tutorial/puzzle* as _____? What made the last one _____, but this one _____?
14. Why did you pick to do this animation as a *tutorial/puzzle* instead of a *puzzle/tutorial*?
15. Why did you pick this specific animation?
16. What do you think you will do next, a tutorial or puzzle? why?

G.1.4 Early Termination Interview

1. Why did you decide to go do something else?
2. *Show participant Figure G.3.* How easy or difficult did you think this *tutorial/puzzle* was going to be before you started it? why?
3. *Show participant Figure G.3.* How easy or difficult would you say this *tutorial/puzzle* actually was? why?

4. *Show participant Figure G.3.* How easy or difficult do you think this have been if you had done it instead as a *puzzle/tutorial*?
5. What do you think you will pick next, a tutorial or a puzzle? why?

G.1.5 Post-Study Interview

1. Which do you enjoy more, puzzles or tutorials? why?
2. What makes a tutorial experience different than the puzzle experience?
3. What makes a puzzle experience different than the tutorial experience?
4. Do you feel you learn different things from the tutorials and puzzles?
 - (a) *If yes*, What did you learn from the puzzles? What did you learn from the tutorials?
5. When is it better to to use tutorials on your own?
6. When is it better to to use puzzles on your own?
7. If a friend with *no* programming experience were to ask you for advice on how to get started using Looking Glass to make their own animations, what advice would you give them about when they should use tutorials and puzzles?
8. If a friend with *some* programming experience were to ask you for advice on how to get started using Looking Glass to make their own animations, what advice would you give them about when they should use tutorials and puzzles?
9. *Show participant Figure G.3.* Earlier you rated your expertise as ____? How would you rate your current programming or coding expertise now?

- (a) Do you feel that you can get to the next level, _____?
- i. *If yes*, How do you think you might do that?
10. *Show participant Figure G.2.* What was the most rewarding tutorial that you did today? What made it rewarding?
11. *Show participant Figure G.2.* What was the most rewarding puzzle that you did today? What made it rewarding?
12. *Show participant Figure G.2.* What was the most frustrating tutorial that you did today? What made it frustrating?
13. *Show participant Figure G.2.* What was the most frustrating puzzle that you did today? What made it frustrating?
14. *Show participant Figure G.2.* What was the easiest puzzle that you did today? What made it easy?
15. *Show participant Figure G.2.* What was the easiest tutorial that you did today? What made it easy?
16. *Show participant Figure G.2.* What was the hardest puzzle that you did today? What made it hard?
17. *Show participant Figure G.2.* What was the hardest tutorial that you did today? What made it hard?
18. What are you more likely to do on your own, the tutorials, puzzles, or both? why?
19. Is there anything else you might like to tell me?

G.2 Familiarization Tasks

Participants were asked to complete two familiarization tasks. Participants were randomly assigned to either complete the first one as a puzzle and the second as a tutorial or the first one as a tutorial and the second one as a puzzle.



Figure G.4: Completed familiarization task 1 as a tutorial (*top*) or puzzle (*bottom*).

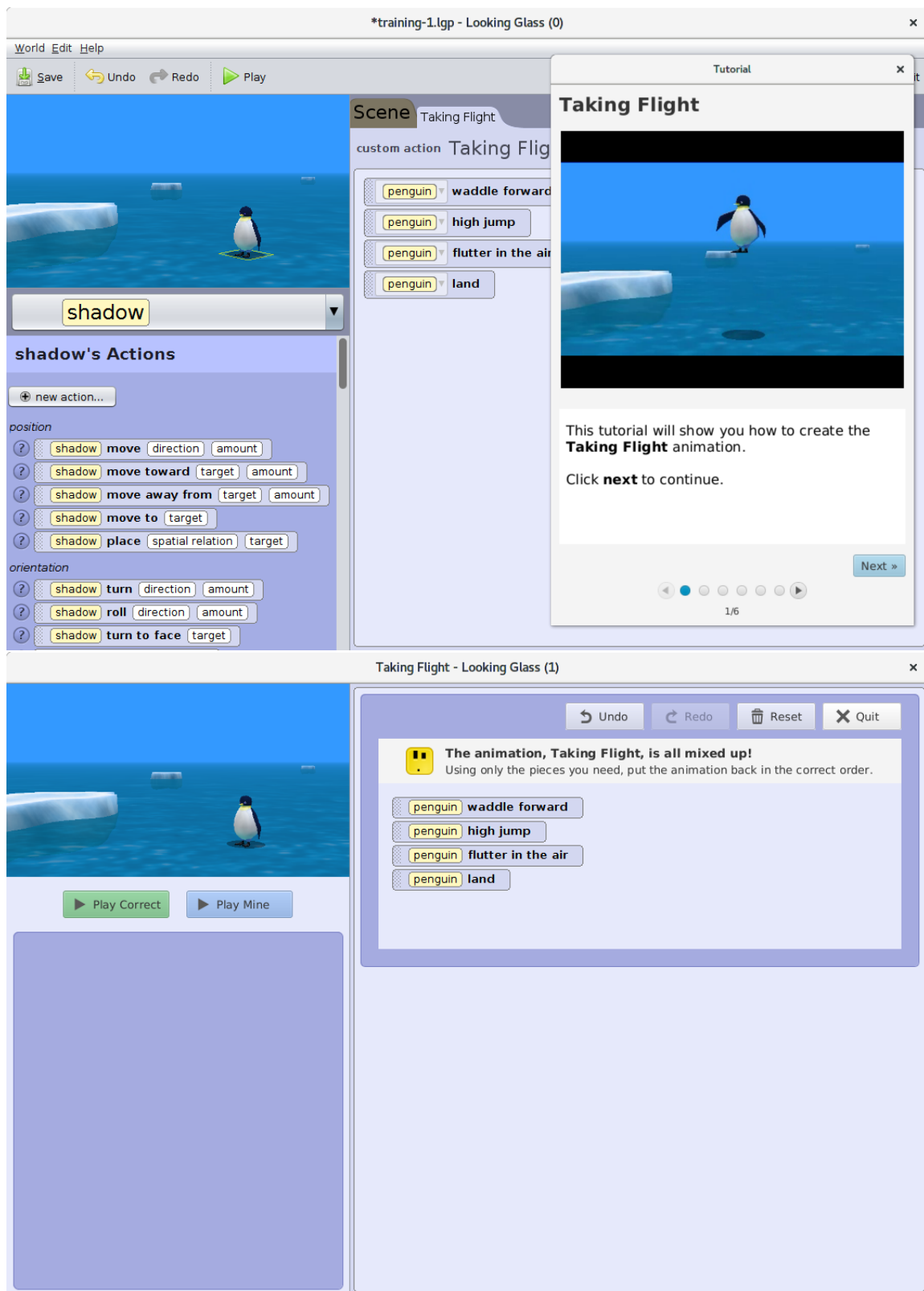


Figure G.5: Completed familiarization task 2 as a tutorial (*top*) or puzzle (*bottom*).

G.3 Instructional Tasks

After completing the two familiarization tasks, participants selected 6 instructional tasks to complete out of the 14. For each instructional task, participants decided whether to complete the task as a tutorial or as a puzzle.

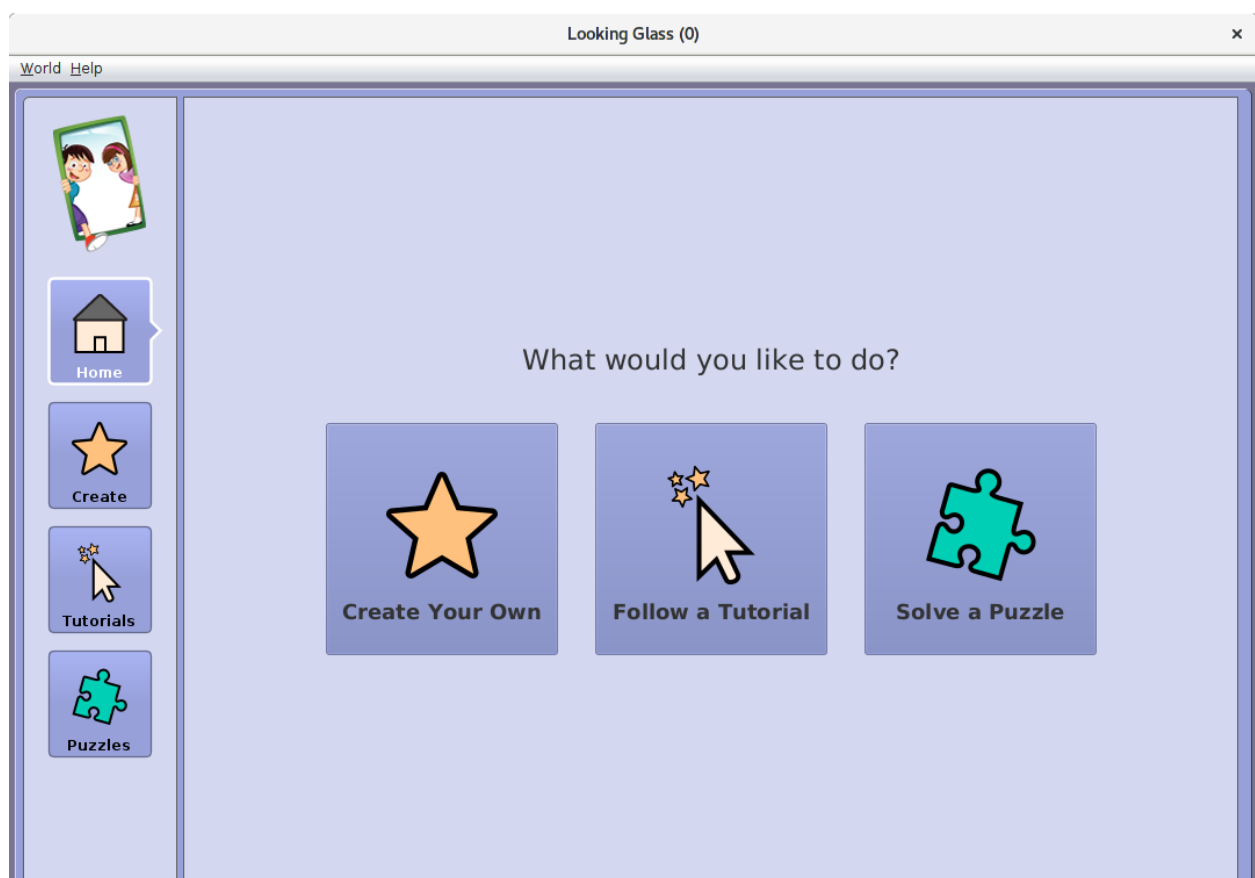


Figure G.6: Instructional task format selection. For each instructional task ($\times 6$) participants first select whether they want work on a tutorial or puzzle.

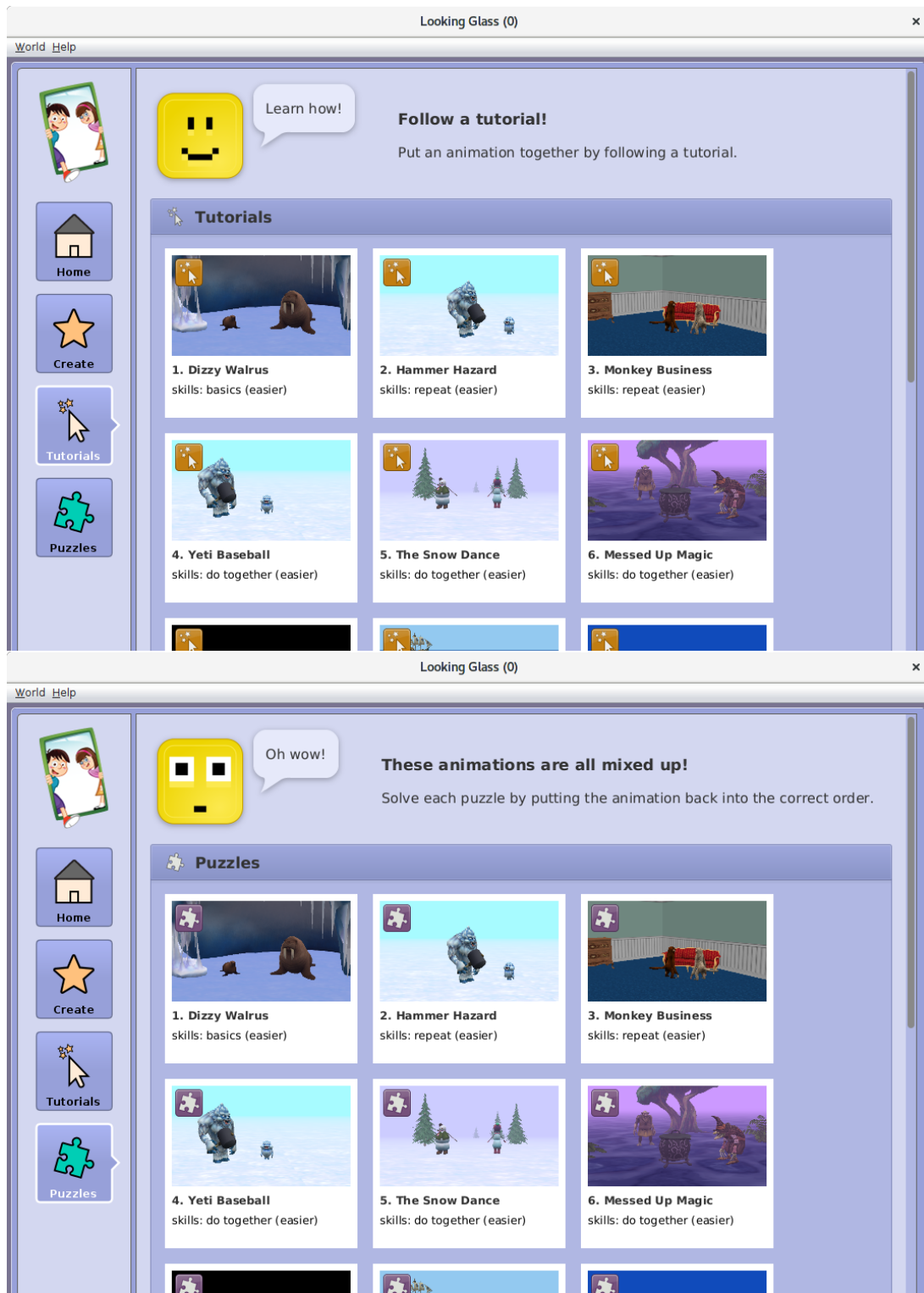


Figure G.7: After selecting a format in Figure G.6, participants then select an animation that they wish to complete as tutorial (*top*) or puzzle (*bottom*). Participants are free to change their format selection by selecting the *Tutorials* or *Puzzles* button in the left panel.

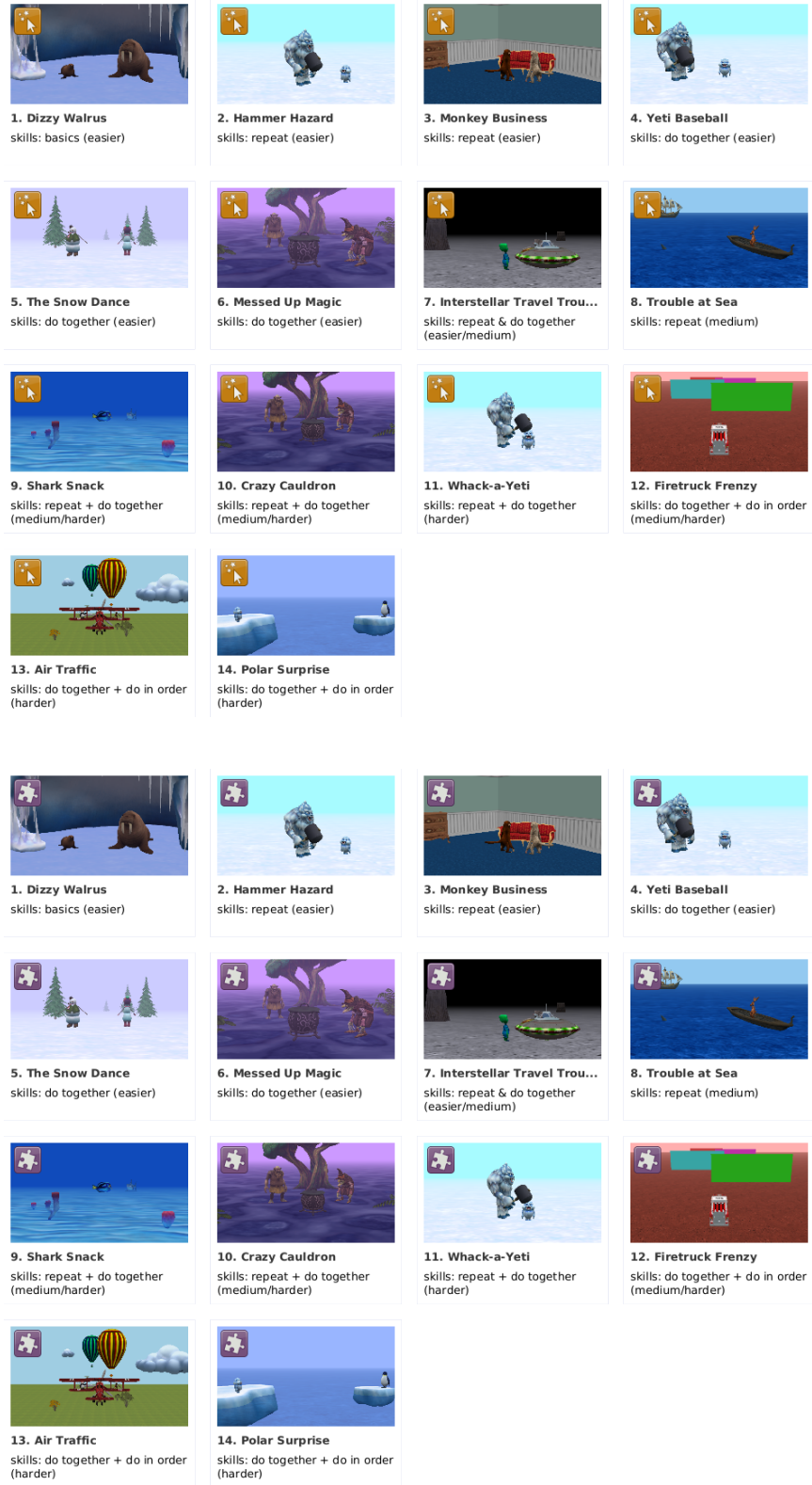


Figure G.8: List of the 14 instructional tasks as tutorials (*top*) and puzzles (*bottom*). Each task is numbered and lists its title, programming skills, and difficulty.



Figure G.9: Instructional task 1. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

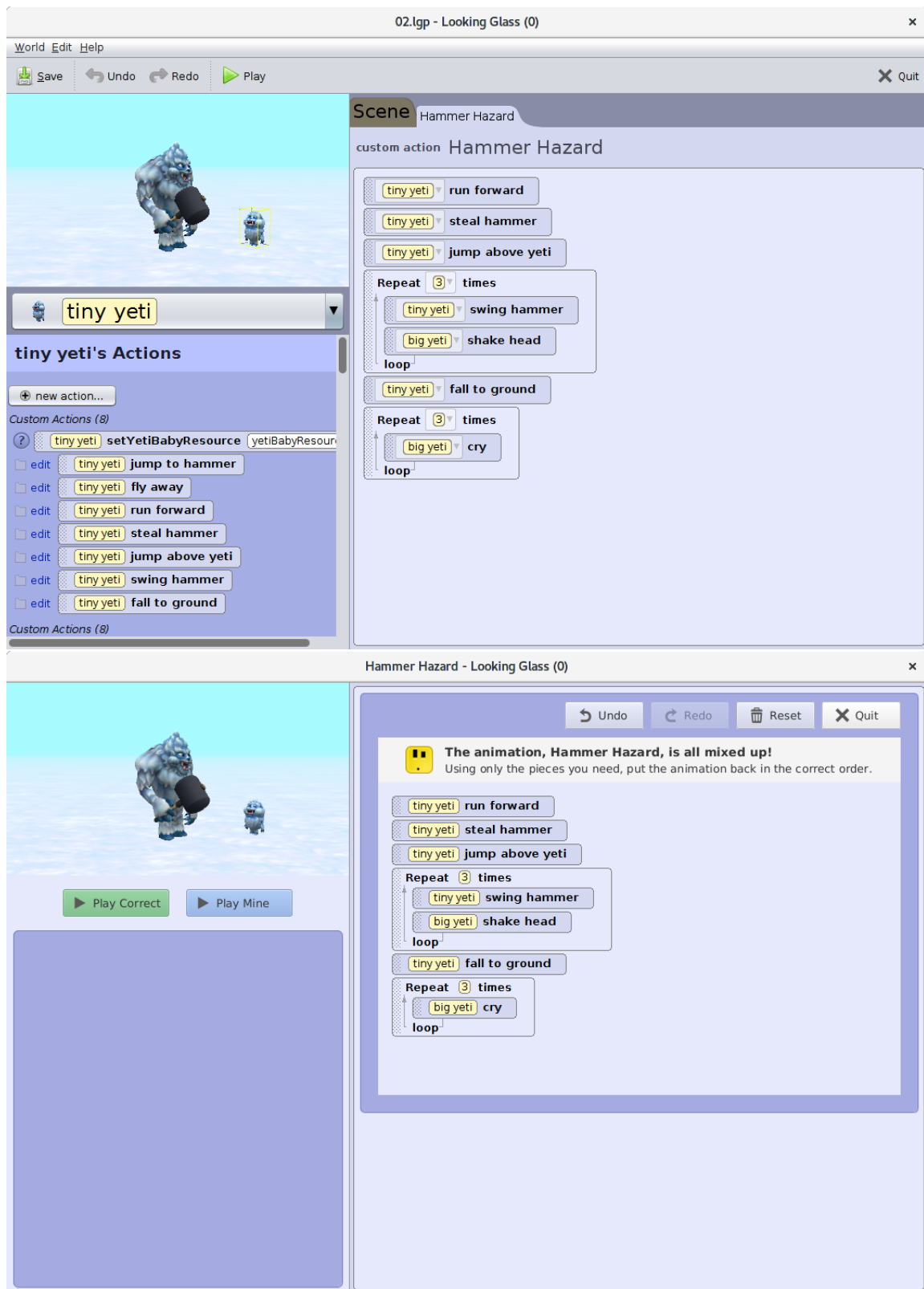


Figure G.10: Instructional task 2. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

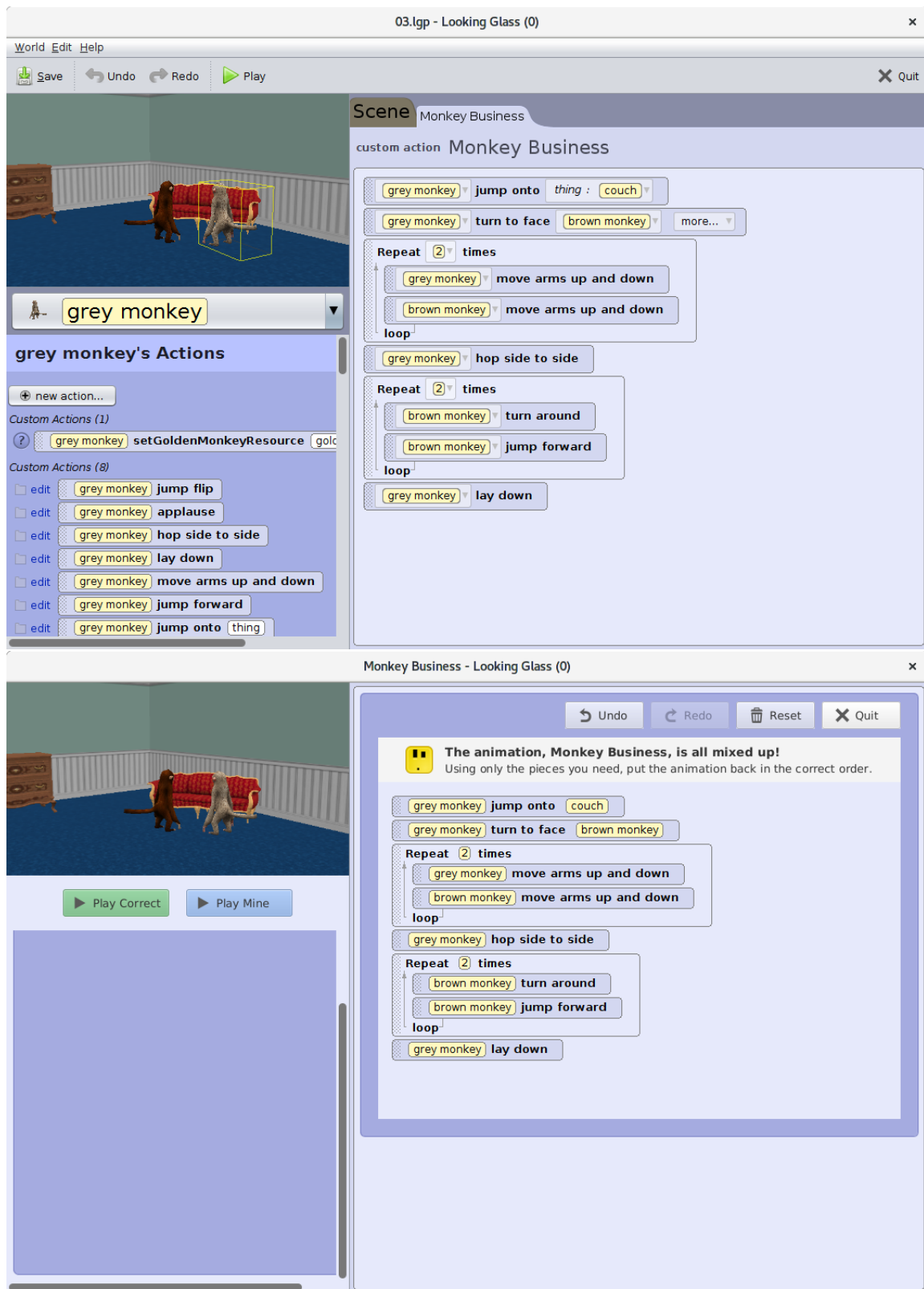


Figure G.11: Instructional task 3. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

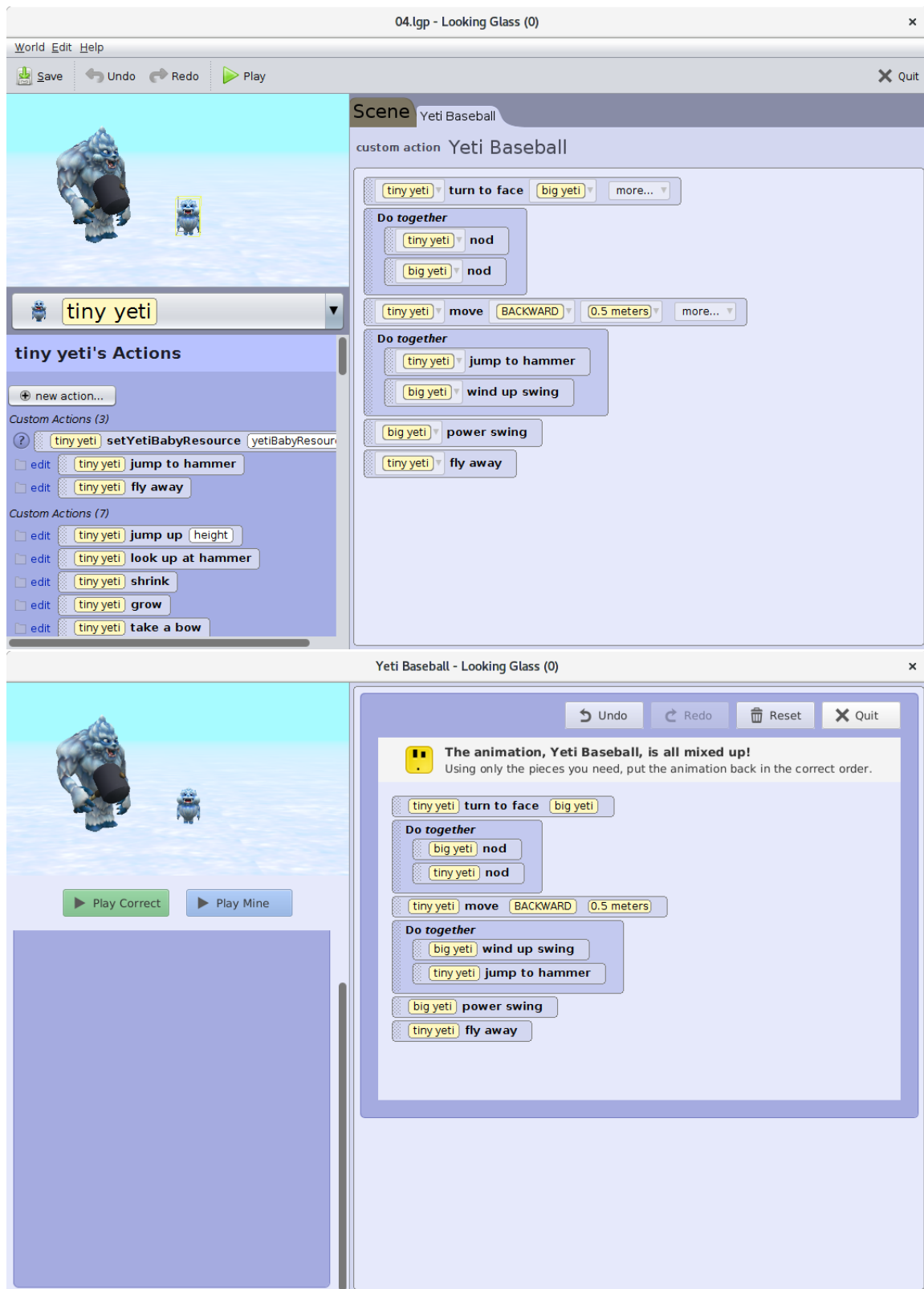


Figure G.12: Instructional task 4. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

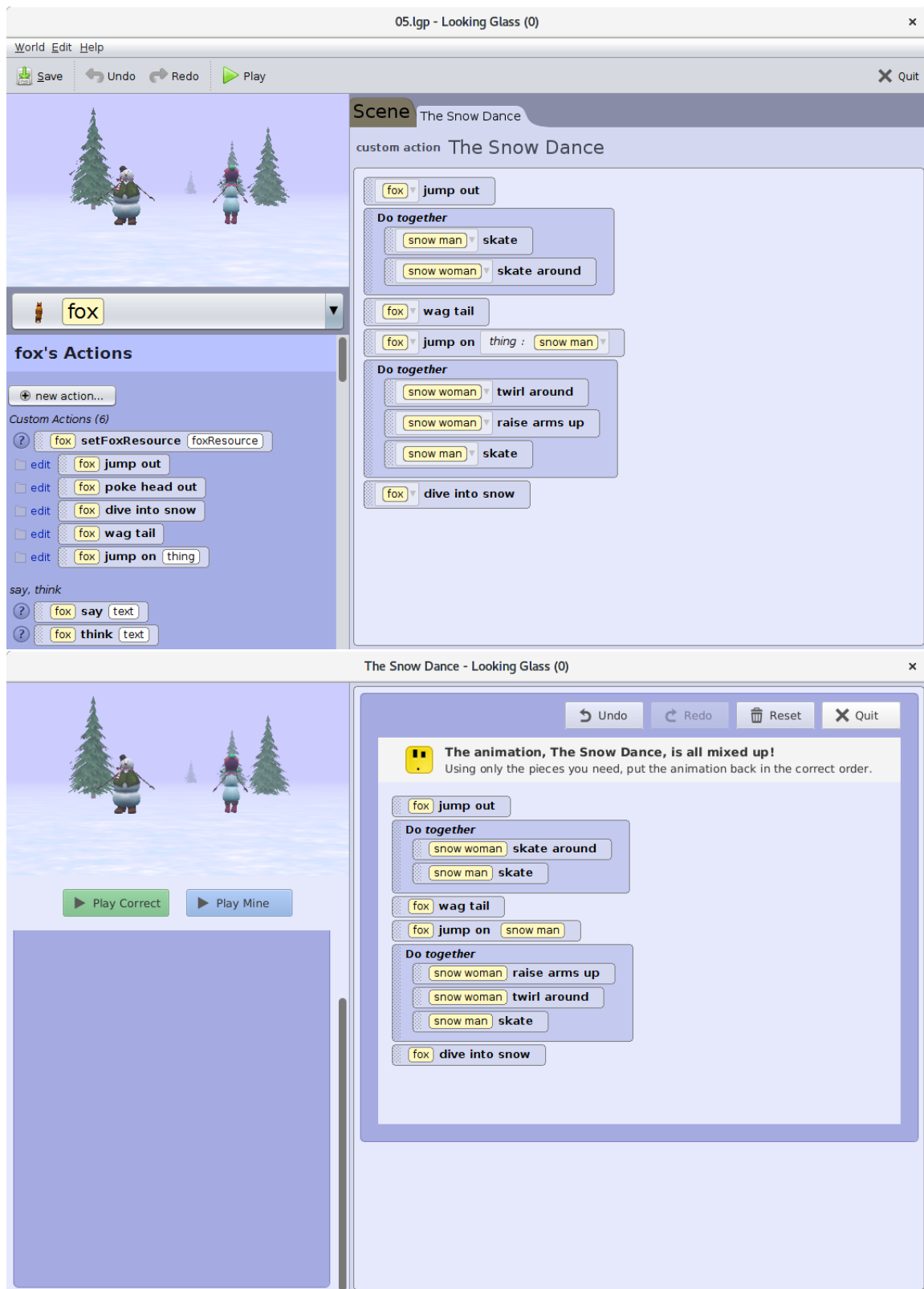


Figure G.13: Instructional task 5. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

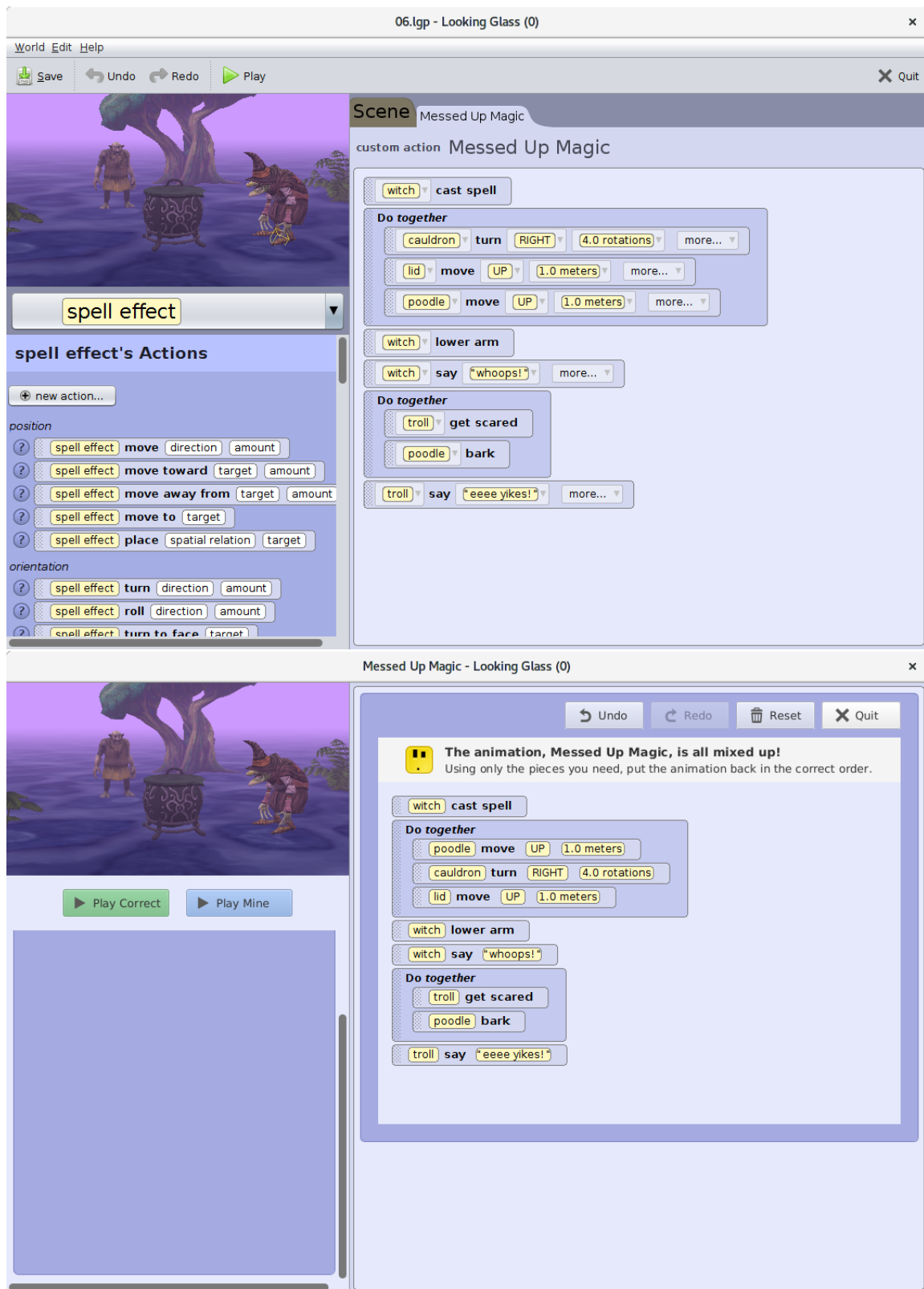


Figure G.14: Instructional task 6. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

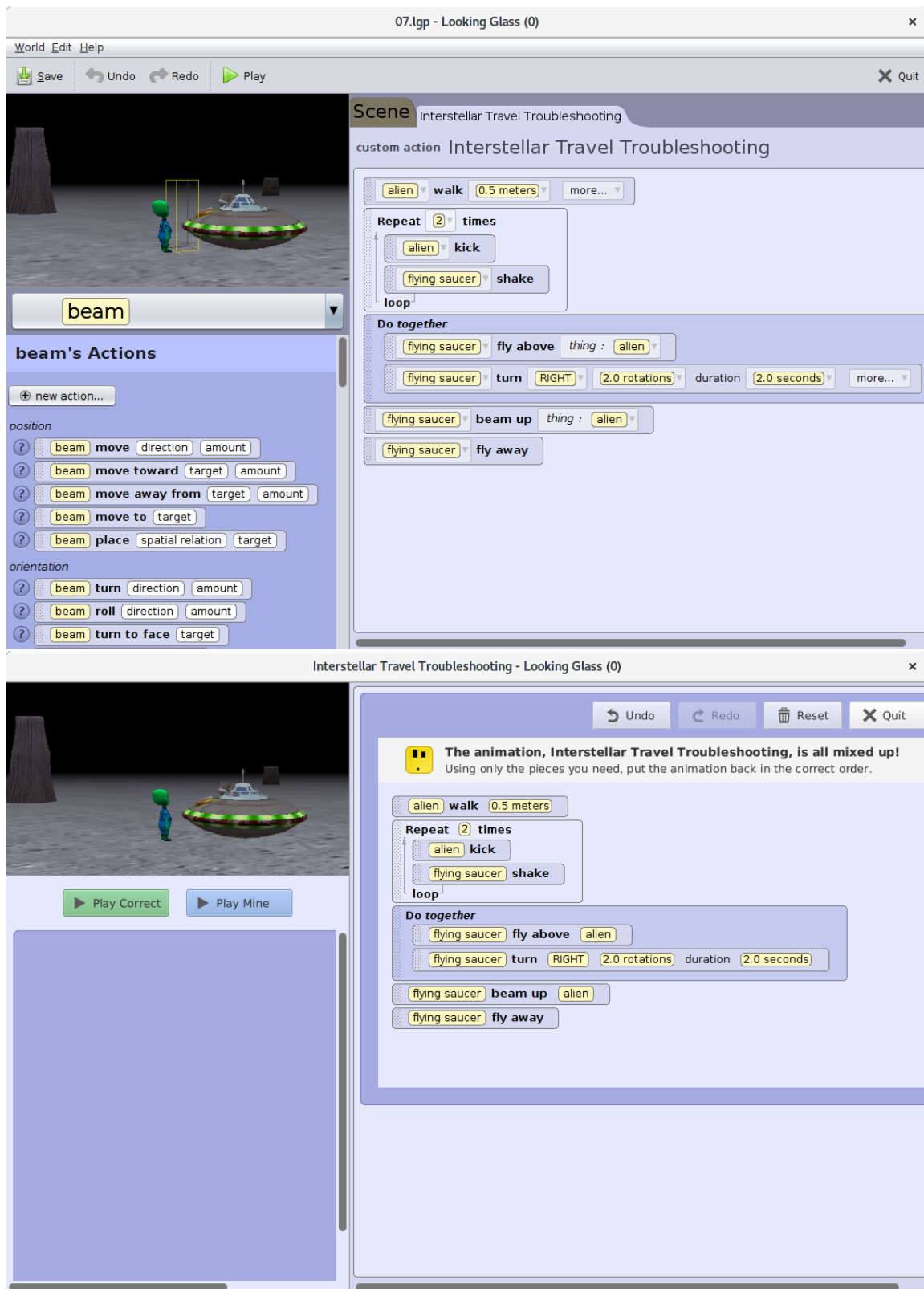


Figure G.15: Instructional task 7. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

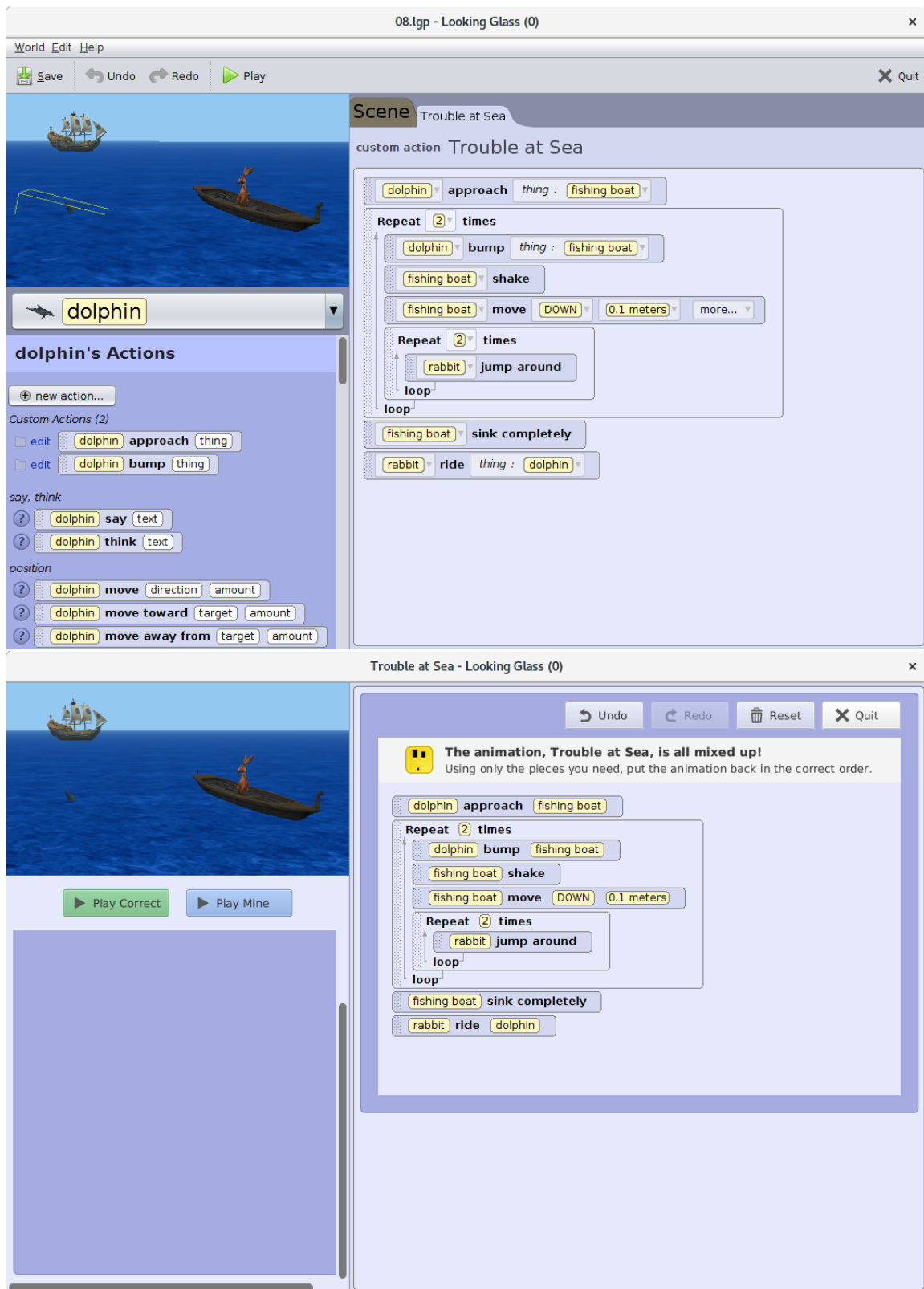


Figure G.16: Instructional task 8. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

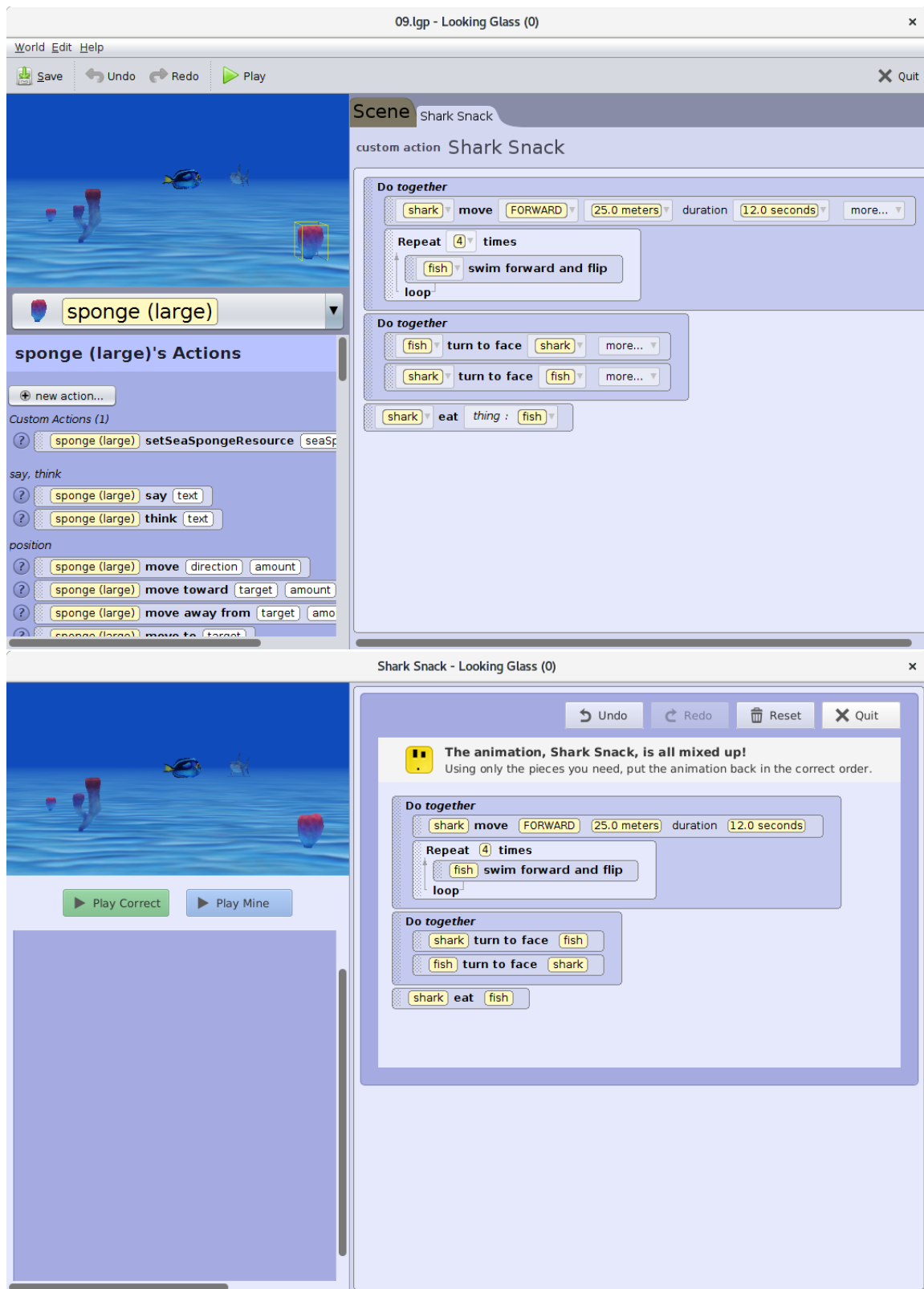


Figure G.17: Instructional task 9. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

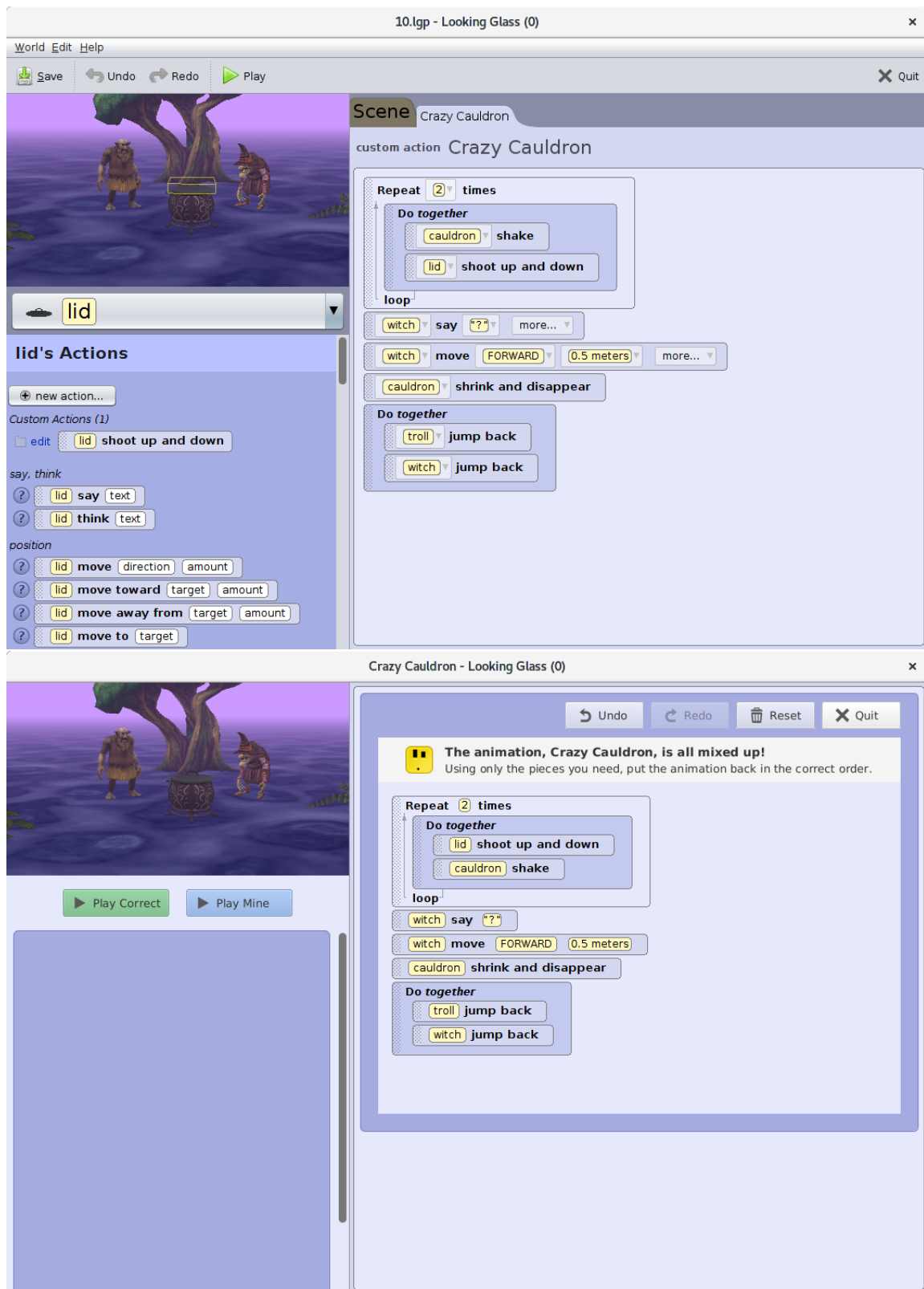


Figure G.18: Instructional task 10. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

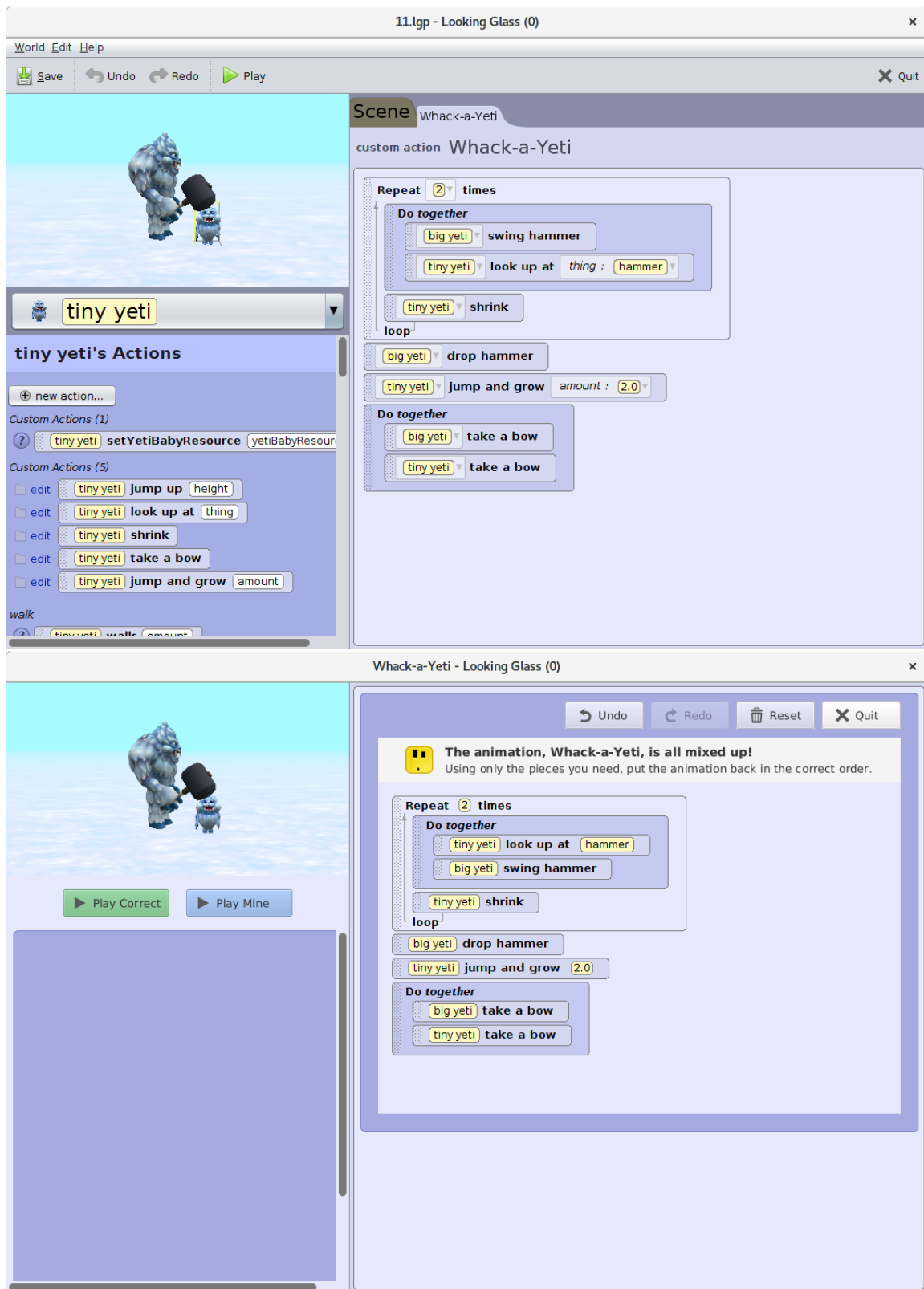


Figure G.19: Instructional task 11. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

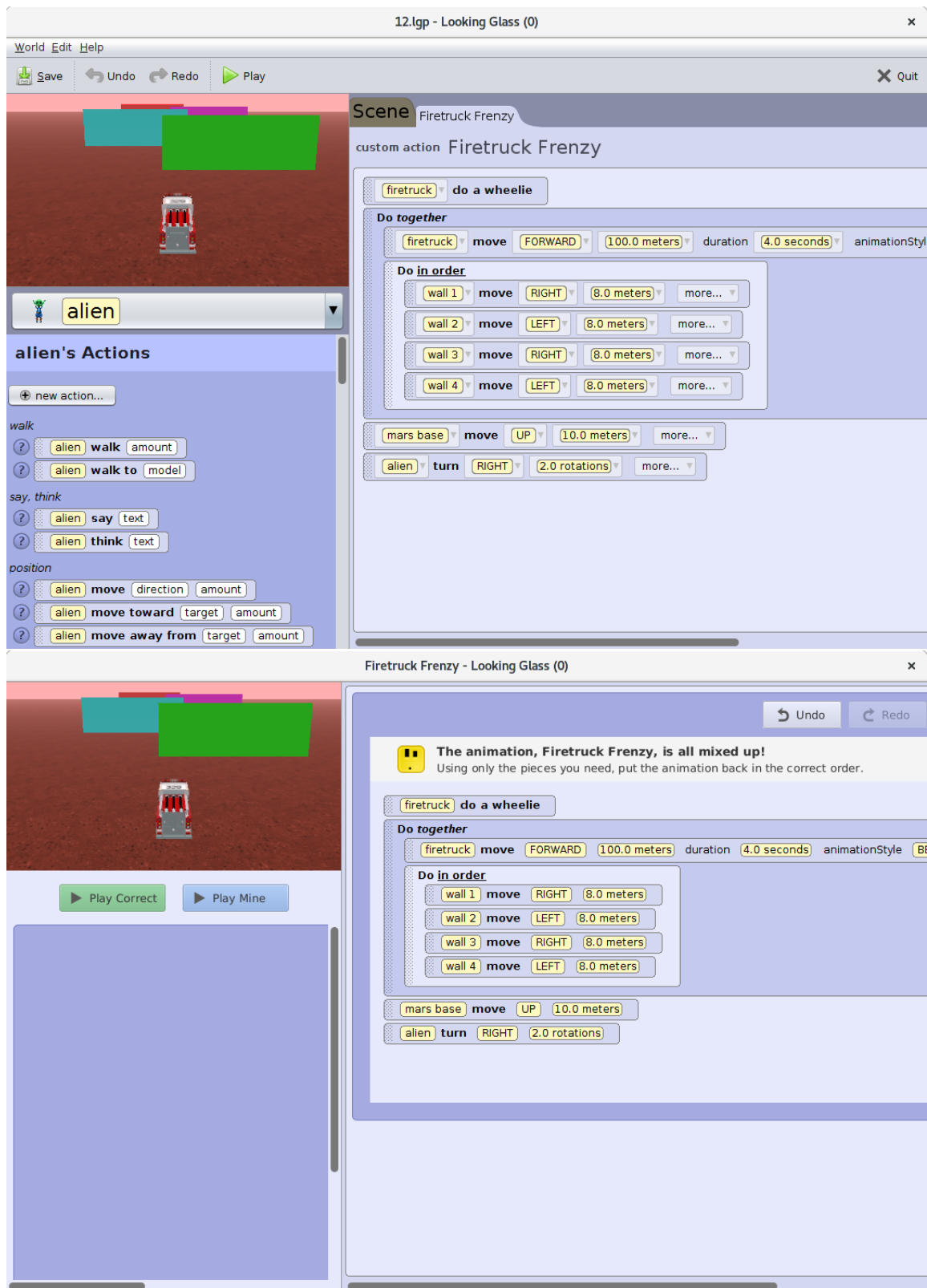


Figure G.20: Instructional task 12. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

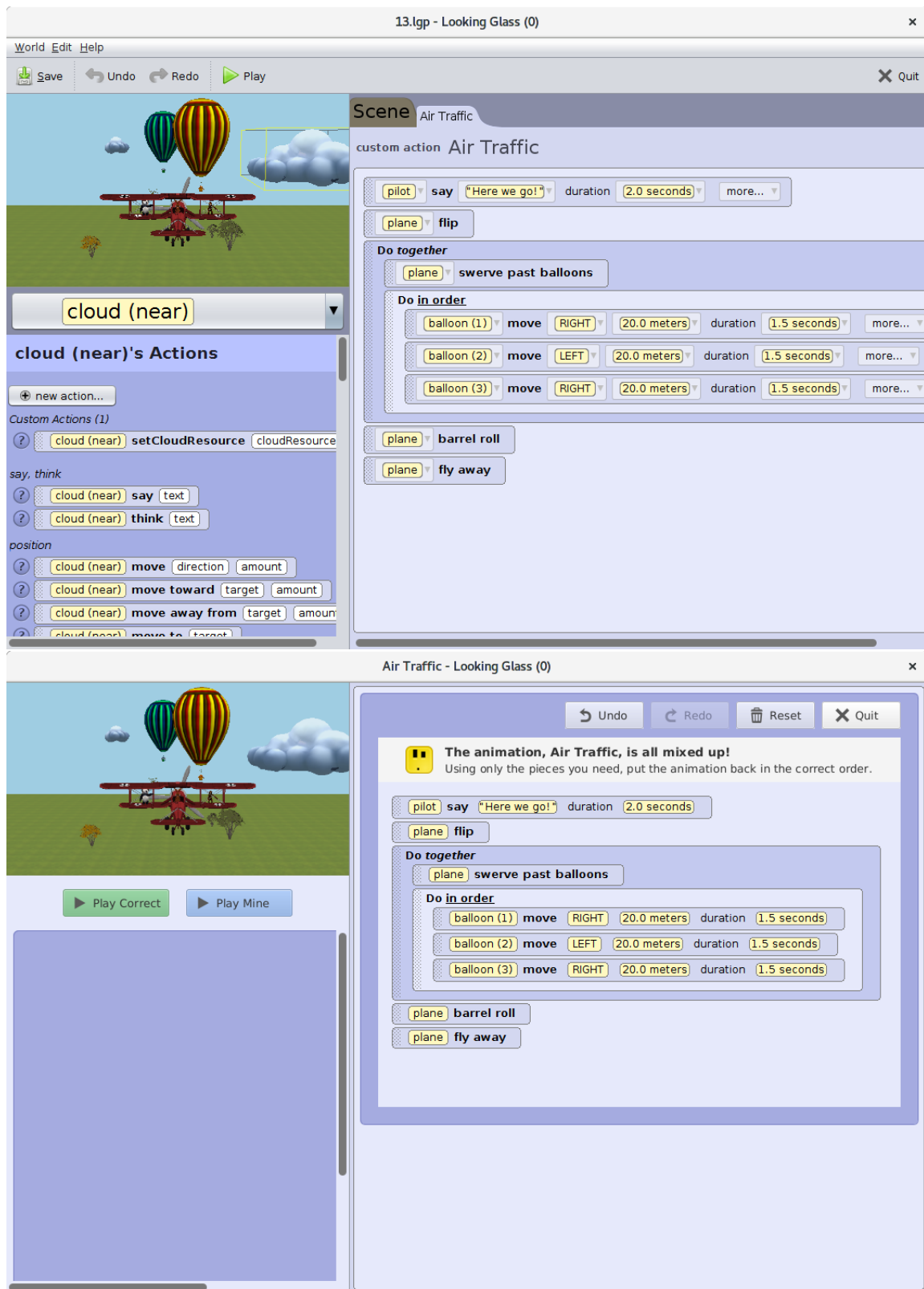


Figure G.21: Instructional task 13. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

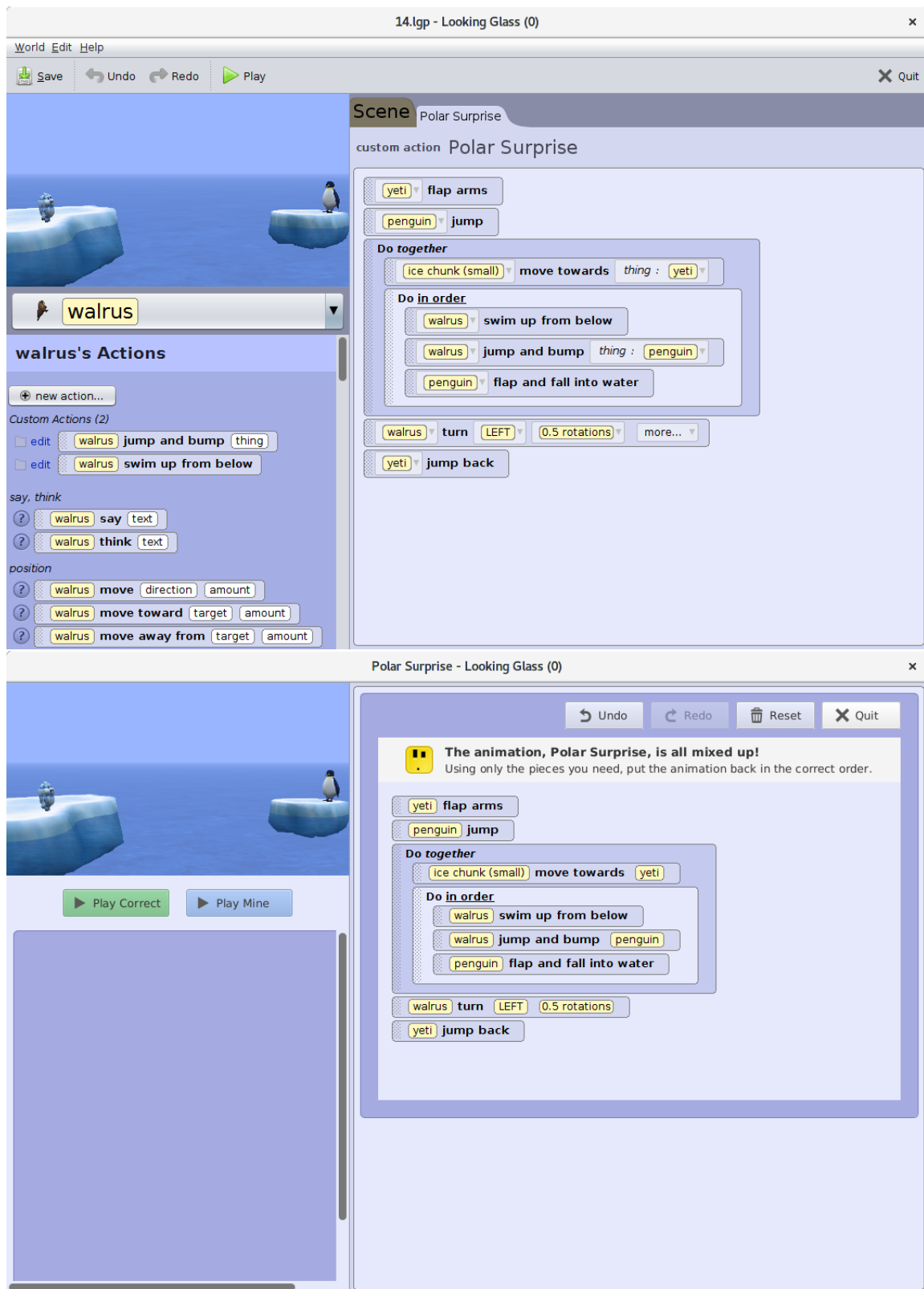


Figure G.22: Instructional task 14. Completed as a tutorial (*top*) or as a puzzle (*bottom*).

G.4 Post-Study Surveys

After completing the instructional tasks and after the post-study interview, we asked participants to complete two task evaluation questionnaires: one for puzzles and one for tutorials. The order in which the two surveys were completed was decided by each participant.

LOOKING GLASS **TUTORIAL** SURVEY

For each of the following statements, please indicate how true it is for you, using the following scale:

1 2 3 4 5 6 7
not at somewhat very
all true true true

	not at all true			somewhat true			very true
	1	2	3	4	5	6	7
While I was working on the tutorials I was thinking about how much I enjoyed it.							
I did not feel at all nervous about doing the tutorials .							
I felt that it was my choice to do the tutorials .							
I think I am pretty good at these tutorials .							
I found the tutorials very interesting.							
I felt tense while doing the tutorials .							
I think I did pretty well at the tutorials , compared to other students.							
Doing the tutorials was fun.							
I felt relaxed while doing the tutorials .							
I enjoyed doing the tutorials very much.							
I didn't really have a choice about doing the tutorials .							
I am satisfied with my performance at these tutorials .							
I was anxious while doing the tutorials .							
I thought the tutorials were very boring.							
I felt like I was doing what I wanted to do while I was working on the tutorials .							
I felt pretty skilled at these tutorials .							
I thought the tutorials were very interesting.							
I felt pressured while doing the tutorials .							
I felt like I had to do the tutorials .							
I would describe the tutorials as very enjoyable.							
I did the tutorials because I had no choice.							
After working at these tutorials for awhile, I felt pretty competent.							

Figure G.23: *Task Evaluation Questionnaire* for tutorials.

LOOKING GLASS **PUZZLE** SURVEY

For each of the following statements, please indicate how true it is for you, using the following scale:

1 2 3 4 5 6 7
not at somewhat very
all true true true

	not at all true			somewhat true			very true
	1	2	3	4	5	6	7
While I was working on the puzzles I was thinking about how much I enjoyed it.							
I did not feel at all nervous about doing the puzzles .							
I felt that it was my choice to do the puzzles .							
I think I am pretty good at these puzzles .							
I found the puzzles very interesting.							
I felt tense while doing the puzzles .							
I think I did pretty well at the puzzles , compared to other students.							
Doing the puzzles was fun.							
I felt relaxed while doing the puzzles .							
I enjoyed doing the puzzles very much.							
I didn't really have a choice about doing the puzzles .							
I am satisfied with my performance at these puzzles .							
I was anxious while doing the puzzles .							
I thought the puzzles were very boring.							
I felt like I was doing what I wanted to do while I was working on the puzzles .							
I felt pretty skilled at these puzzles .							
I thought the puzzles were very interesting.							
I felt pressured while doing the puzzles .							
I felt like I had to do the puzzles .							
I would describe the puzzles as very enjoyable.							
I did the puzzles because I had no choice.							
After working at these puzzles for awhile, I felt pretty competent.							

Figure G.24: *Task Evaluation Questionnaire* for puzzles.