The Dissertation Committee for Thomas Jonathan Benton certifies that this is the approved version of the following dissertation:

MUSICAL EXPERTISE AS A SCAFFOLD FOR NOVICE PROGRAMMING

Committee:

Joan E. Hughes, Supervisor

Flavio Azevedo

Matthew Berland

Paul Resta

Jason Rosenblum

MUSICAL EXPERTISE AS A SCAFFOLD FOR NOVICE PROGRAMMING

by

Thomas Jonathan Benton, B.S.M.E.; M.Ed.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2015

Acknowledgements

First, I would like to thank my advisor, Joan Hughes; her guidance and encouragement have been invaluable, not simply during this dissertation process but throughout my graduate school career. I would also like the thank the other members of my committee – Flavio Azevedo, Matthew Berland, Paul Resta, and Jason Rosenblum – for their time and valuable input.

I would like to thank my parents for their tireless support, Marni Hamilton for her endless encouragement, and Tina and Michael Lobel for their kind hospitality.

Musical Expertise as a Scaffold for Novice Programming

Thomas Jonathan Benton, Ph.D.

The University of Texas at Austin, 2015

Supervisor: Joan E. Hughes

This study addresses the role of musical expertise on novice computer programming. Engaging novices with computer programming is one of the great challenges of computer science education. Although there is extensive research focusing on constructionist approaches to programming education and creative entry points to programming, little research addresses the topic of how musical expertise informs an unstructured programming activity. To answer this question I focused on the role of participant talk during programming, patterns in participant programming, and evidence of computational thinking in participants' final Scratch projects.

For this interpretivist study, I worked with a dozen novice programmers from a variety of musical backgrounds: classical musicians, jazz musicians, composers, and non-musicians. Each participant worked on a free-form musical project in the Scratch programming environment. I collected data including participant talk, screen recordings of participant programming, and participants' final Scratch projects.

Overall, musical participants more readily took to the numeracy involved in programming music in Scratch. Also, musical participants were able to use musical concepts and techniques as jumping-off points for programming challenges. Considering my results by participant group, composers stood out in a number of ways: working the longest, testing their programs the most often, adding Scratch objects the slowest, removing the most Scratch objects, creating projects of the greatest nested depth, and unanimous use of operators and random numbers. Non-musicians, on the other hand, worked for the shortest amount of time, added the fewest Scratch objects, and created projects of the lowest nested depth.

In addition to adding to the body of research around chunking and tinkering, this study reinforces the importance of context and comfort in an introduction to computer programming. Composition may be an especially rich area to leverage, given the design-like programming activity of the composers here. Future research projects could resemble this one while focusing on younger learners, explicit musical concepts like those invoked by participants, or alternative performing arts framings such as theater or dance.

Table of Contents

List of Tables xi

List of Figures xii

List of Illustrations xvi

Chapter 1: Introduction 1

    Purpose and Significance of the Study 3

    Overview of the Dissertation 6

Chapter 2: Review of Literature 7

    Constructionism 7

    Computer Programming 10

    Novice Programmers 12

        Characteristics of Novice Programmers 12

        Novice & Expert Programmers 14

        Musicians as Novice Programmers 16

    Computational Thinking 16

    Approaches to Programming Learning and Instruction 20

        Lecture-Based Programming Instruction 20

        Constructionist Programming Instruction and Learning 20

            Tinkering 21

            Design 22

            Games 23

            Arts and Crafts 24

            Music 24

    Open Questions 26

Chapter 3: Methods 28

    Epistemology and Theoretical Perspective 28

    Methodology 29

    Participants 30

Materials 30
Procedure 31
Data 32
Data Analysis..32
Participant Pre-Surveys 33
Think Aloud Interviews..34
Screen-Captured Scratch Programming 35
Final Scratch Projects 36
Comparative Analysis 39
Trustworthiness 39
Assumptions and Limitations 40
Researcher Positionality 41
Chapter 4: Individual Results 43
Non-Musicians 43
Participant A 43
Participant B 46
Participant C 49
Jazz Musicians..52
Participant D 52
Participant E 55
Participant F..58
Classical Musicians 61
Participant G 61
Participant H 64
Participant I 67
Composers 70
Participant J 70
Participant K 73
Participant L 76

Chapter 5: Group Results 80
Musical Concepts as Scaffolds 80
Musical Questions 81
Musical Assertions 81
Musical Intentions 83
Musical Talk by Participant Group 84
Summary 85
Patterns in Programming Process by Participants' Musical Backgrounds...86
Summary 93
Differences In Final Projects by Musical Background 94
Distinct Programs 94
Maximum Computational Thinking Depth 95
Computational Thinking Concepts 95
User Interaction 96
Loops 97
Conditional Statements 99
Parallelism 100
Sequences 101
Operators 103
Boolean Logic 104
Random Numbers 105
Variables 107
Lists 109
Overall Use of Computational Thinking Concepts 110
Summary 112
Musical Background and Novice Programming 113
Chapter 6: Discussion 114
Chunking 114
Tinkering 115
Context & Comfort 116

Composers and Design 117
Practical Implications 118
Future Research 119
Conclusion 120

Appendix A: Research Matrix 122

Appendix B: Direct Recruitment Letter 123

Appendix C: Consent Form 124

Appendix D: Pre-Survey 126

Appendix E: Scratch Sample Programs & Reference Sheet 127

Appendix F: Computational Thinking Coding 130

References 140

List of Tables

Table 1: Subject and Type Distribution of Participant Utterances........................81

Table 2: Musical Utterances by Participant Group 85

List of Figures

Figure 1: Computational Thinking Diagram Example to Illustrate Calculating Average Depth Metric 36
Figure 2: Sample Scratch Project with Three Operable Programs....................37
Figure 3: Computational Thinking diagram for top left sample program. 38
Figure 4: Computational Thinking diagram for bottom left sample program.......38
Figure 5: Computational Thinking diagram for right sample program. 38
Figure 6: Participant A's Final Scratch Project....................44
Figure 7: CT Diagram of Participant A's Final Scratch Project 45
Figure 8: Timeline of Participant A's addition and removal of Scratch objects ...46
Figure 9: Participant B's Final Scratch Project 47
Figure 10: CT Diagram of Participant B's Final Scratch Project....................48
Figure 11: Timeline of Participant B's addition and removal of Scratch objects .49
Figure 12: Participant C's Final Scratch Project 50
Figure 13: CT Diagram of Participant C's Final Scratch Project....................51
Figure 14: Timeline of Participant C's addition and removal of Scratch objects .52
Figure 15: Participant D's Final Scratch Project....................53
Figure 16: CT Diagram of Participant D's Final Scratch Project 54
Figure 17: Timeline of Participant D's addition and removal of Scratch objects .55
Figure 18: Participant E's Final Scratch Project 56
Figure 19: CT Diagram of Participant E's Final Scratch Project....................57
Figure 20: Timeline of Participant E's addition and removal of Scratch objects..58
Figure 21: Participant F's Final Scratch Project....................59
Figure 22: CT Diagram of Participant F's Final Scratch Project 60

Figure 23: Timeline of Participant F’s addition and removal of Scratch objects..61
Figure 24: Participant G’s Final Scratch Project..62
Figure 25: CT Diagram of Participant G’s Final Scratch Project 63
Figure 26: Timeline of Participant G’s addition and removal of Scratch objects .64
Figure 27: Participant H’s Final Scratch Project..65
Figure 28: CT Diagram of Participant H’s Final Scratch Project 66
Figure 29: Timeline of Participant H’s addition and removal of Scratch objects .67
Figure 30: Participant I’s Final Scratch Project 68
Figure 31: CT Diagram of Participant I’s Final Scratch Project............................69
Figure 32: Timeline of Participant I’s addition and removal of Scratch objects...70
Figure 33: Participant J’s Final Scratch Project 71
Figure 34: CT Diagram of Participant J’s Final Scratch Project............................72
Figure 35: Timeline of Participant J’s addition and removal of Scratch objects ..73
Figure 36: Participant K’s Final Scratch Project..74
Figure 37: CT Diagram of Participant K’s Final Scratch Project 75
Figure 38: Timeline of Participant K’s addition and removal of Scratch objects .76
Figure 39: Participant L’s Final Scratch Project 77
Figure 40: CT Diagram of Participant L’s Final Scratch Project...........................78
Figure 41: Timeline of Participant L’s addition and removal of Scratch objects..79
Figure 42: Programming Duration 86
Figure 43: Program Tests Per Minute 87
Figure 44: Net Scratch Objects Added...88
Figure 45: Net Scratch Objects Added Per Minute 89
Figure 46: Types of Net Scratch Objects Added..90

Figure 47: Types of Net Scratch Objects Added as a Percentage of All Net Scratch Objects Added 91
Figure 48: Total Objects Removed Across All Programming Time 92
Figure 49: Objects Removed as a Percentage of All Objects Added 93
Figure 50: Distinct Programs in Final Projects 95
Figure 51: Maximum Depth of Final Scratch Projects 95
Figure 52: Number of Instances of *User Interaction*, by participant 96
Figure 53: Average Depth of *User Interaction*, by participant 97
Figure 54: Number of Instances of *Loops*, by participant 98
Figure 55: Average Depth of *Loops*, by participant 98
Figure 56: Number of Instances of *Conditional Statements*, by participant 99
Figure 57: Average Depth of *Conditional Statements*, by participant 100
Figure 58: Number of Instances of *Parallelism*, by participant 101
Figure 59: Average Depth of *Parallelism*, by participant 101
Figure 60: Number of Instances of *Sequences*, by participant 102
Figure 61: Average Depth of *Sequences*, by participant 102
Figure 62: Number of Instances of *Operators*, by participant 103
Figure 63: Average Depth of *Operators*, by participant 104
Figure 64: Number of Instances of *Boolean Logic*, by participant 105
Figure 65: Average Depth of *Boolean Logic*, by participant 105
Figure 66: Number of Instances of *Random Numbers*, by participant 106
Figure 67: Average Depth of *Random Numbers*, by participant 107
Figure 68: Number of Instances of *Variables*, by participant 108
Figure 69: Average Depth of *Variables*, by participant 108
Figure 70: Number of Instances of *Lists*, by participant 109

Figure 71: Average Depth of *Lists*, by participant 110

Figure 72: CT Concepts Used, Instance Across All Participants 111

Figure 73: User Interaction Program 111

Figure 74: Number of Instances and Type of CT Concepts Used, by participant112

List of Illustrations

Illustration 1: LOGO Example 1 8
Illustration 2: LOGO Example 2 8
Illustration 3: LOGO Example 3 9
Illustration 4: Simple Programming Examples, Java on left, Scratch on right 11

Chapter 1: Introduction

This study focuses on how musical experience and expertise may impact the experiences of novice programmers. This study is motivated by a longstanding personal interest in music as well as by my experience in research focusing on how students learn about programming. I will begin with a personal story that depicts these two worlds intersecting for me and out of which this study grew.

Several years ago I took a course offered by the College of Music's electronic music studio called Experimental Music Performance Interfaces. In the course, we explored an array of approaches to controlling a computer music application, ranging from audio and motion sensors to video game controllers. These approaches had little in common aside from being dramatic departures from the musical keyboard or even the computer keyboard and mouse. For instance, our final course project was done in collaboration with students from the department of dance; we developed an application that generated live audio and video based on data from real-time motion generated by Wii controllers in the hands and strapped on the ankles of dancers.

The software tool with which we explored these ideas was MAX/MSP, a visual programming environment designed for the development of interactive musical and visual applications. The application we created in MAX/MSP parsed and smoothed the copious motion data we received from the Wii controllers (in the form of four streams of integers) and then fed these more manageable data streams into mathematical transformers that would turn them into something suitable for manipulating audio samples and driving several synthesizer modules. Similar processes generated abstract visual animations and controller playback of prerecorded video elements. The process of learning MAX/MSP nearly perfectly engaged me in terms of balancing meaningful challenges with rewarding, enjoyable payoffs, and I continue to create projects with it to this day.

This dissertation is motivated in particular by two observations that came in the wake of the class and have never been far from my mind. First, after the course I realized that not only had I learned my way around MAX/MSP, but I had also begun to intuit some important programming fundamentals as well. Prior to that course, I had never learned anything more about programming than was necessary to write occasional bits of oafishly effective but inelegant and inefficient code, and while our MAX/MSP instruction had been practical and application-focused, I had nevertheless developed a sense of why it was called “object-oriented programming” and of how to take advantage of this in writing code. In a Java course I took several semesters later, it was a very pleasant surprise to realize that working in the context of an extremely specialized, niche language (that certainly looked nothing like Java or any other “serious” textual programming language) had developed my intuitions sufficiently as to shed light on some of these larger concepts. The question this left me with was whether or not I had simply been very engaged because I enjoy music or if my experience as a musician had somehow informed my learning in the domain of computer programming.

A possible connection between music and programming leads into something I observed as we batted ideas around with our partners in the dance department. At the beginning of the collaboration I spent a great deal of time figuring out how to articulate musical ideas in code, but as the semester (and my proficiency) progressed I would sometimes find myself starting with the code rather than the music; that is, fiddling with some MAX/MSP object (a random number generator or mathematical transform, for instance) and experimenting with what kind of music I could wring out of it. The results, pleasant listening or not, were certainly sounds that I could never have conceptualized on my own and in some cases presented appealing new musical ideas to explore.

Using ruthlessly relentless machines to drive musical experimentation certainly predates accessible computer technology as we know it – in 1965 the American composer Steve Reich created a musical piece by running two copies of an identical tape loop at two slightly different speeds, such that the short audio loop (from a recording of a San Francisco street preacher) slowly drifted in and out of sync with itself over the course of seventeen minutes. While not traditionally musical, the result is a powerful and even mind-bending listening experience. Almost fifty years later, Reich has composed numerous acclaimed pieces for soloists and ensembles that use similar time-shifting techniques, the import of which might have never been realized had he not devoted some hours to playing with a pair of reel-to-reel machines (Reich, 2002).

The ready accessibility of computers, and of novice-friendly programming platforms in particular, open wide the doors to this kind of experimentation. The tinkering and discovery and surprise of this computationally-aided creativity could surely fill many dissertations; the corner I will explore in this study examines how musical proficiency may usefully inform programming learning and practice. This dissertation is guided by an overarching research question: Do different kinds of musical backgrounds play a role in novice programming?

PURPOSE AND SIGNIFICANCE OF THE STUDY

First, this study attempts to expand the body of research around novice computer programming. There is a large body of research focusing on programming in fun and creative contexts, including storytelling, game design, robotics, and crafting. While music serves as a valuable entry point to developing a variety of digital literacy skills, research around music and learning programming (particularly in the open-ended fashion of this study) is somewhat limited. This research could inform future work on introductory programming instruction as well as provide insight into the design of more interesting and powerful digital tools for musicians and other artists.

In addition to contributing to research around novice programming, this study may point towards another entry point to the discipline of computer programming. The ability to program a computer has become a valuable commodity in countless professional fields (Guzdial, 2008), to say nothing of the search for compelling pathways into computer and information technology (Knobelsdorf & Schulte, 2008). However, while introductory programming courses are widely available in secondary (and some primary) schools, participation continues to be dampened by perceptions of programming as formidably difficult, boring, and/or the province of a limited demographic of students (Margolis, Estrella, Goode, Holme, & Nao, 2008). Fortunately, the current state of programming education offers many points of entry with the potential to dispel these objections. Engaging visual programming environments allow novices to sidestep the arcane and potentially intimidating syntax associated with traditional programming languages (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008) and many of these same environments are optimized for creative expression in a variety of media (Peppler & Kafai, 2005).

Much of the research on engaging and creative entry points to programming focuses on game development or media-rich storytelling; while both of these areas often involve music, relatively little research has specifically addressed music as a context for learning programming or computational concepts. Young people have the opportunity to engage with music from a variety of perspectives, from school ensembles to Garageband, and this experience could potentially be leveraged to provide novices exciting, engaging, and scaffolded introductions to programming. In addition to its potential for engagement, music is rich with information and information-processing in its own right (Edwards, 2011), and this capacity has great potential for interaction with programming. The merit of approaching a traditionally technical field from a creative or artistic perspective is supported by calls for the expansion of STEM (Science, Technology, Engineering, and Math) education to STEAM with the addition of arts and design; furthermore, of the seven overarching computing ideas that drive the College Board and National Science Foundation's Computer Science (CS) Principles course (Astrachan, Barnes, Garcia, Paul, Simon, & Snyder, 2011), three implicitly or explicitly make the case for creativity and the arts as fundamental to computing, boldfaced below:

1. **Computing is a creative human activity.**
2. Abstraction reduces information and detail to focus on concepts relevant to understanding and solving problems.
3. Data and information facilitate the creation of knowledge.
4. Algorithms are tools for developing and expressing solutions to computational problems.
5. **Programming is a creative process that produces computational artifacts.**
6. Digital devices, systems, and the networks that interconnect them enable and foster computational approaches to solving problems.
7. **Computing enables innovation in other fields, including science, social science, humanities, arts, medicine, engineering, and business.** (Astrachan et al., p. 398)

The appeal and value of programming in accessible contexts combined with the recognition of art and creativity's place in programming education motivate this study of the intersection of music and programming.

OVERVIEW OF THE DISSERTATION

The literature review that follows examines the research threads that contribute to this study. First I discuss my two foundations, constructionism and computer programming. This is followed by an overview of research on novice programmers and then a discussion of computational thinking, an important framework for content knowledge in this study. Finally, I will review a myriad of approaches to computer programming instruction. This includes lecture-based models and a variety of constructionist approaches, including music-focused ones.

The methods section begins by explaining the constructivist epistemology and interpretivist theoretical perspective that guide this qualitative study. These justify my choice of a descriptive methodology, and then I outline the study in detail, which involved participants working on musical projects in the Scratch programming language. Primary data sources were think-aloud interviews, recordings of participant programming, and final Scratch projects. A constant comparative analysis approach was used to compare the themes that emerged from analysis of these data sources. Finally, I explain my positionality in this project as a researcher.

Next, I present the findings of the study, for individuals and groups, and discuss these findings. The dissertation will conclude with a summary and final remarks.

Chapter 2: Review of Literature

The review of the literature begins with discussions of constructionism and computer programming, two foundational concepts in this study. Following this I will present research on novice programmers. Next I will discuss computational thinking, a framework for content knowledge that I am using in this study. Finally, I will review several approaches to programming instruction. These include lecture-based instruction and a number of constructionist approaches to programming instruction.

CONSTRUCTIONISM

This study is grounded in constructionist learning theory. Constructionism builds on the constructivist theory that knowledge is constructed rather than simply received and constructionist pedagogies are related to problem-based learning, inquiry learning, and other constructivist approaches in the sense of using authentic and (ideally) meaningful tasks as entry points to content and concepts. Constructionism builds on constructivism in connecting learning with the creation of physical or digital artifacts (Harel & Papert, 1991). Though constructionism is often summarized as "learning by making," this description neglects the sense of play and exploration to which constructionism lends itself and which is a vital element of this study.

Most research around constructionism focuses on the mental-model-making prevalent in mathematics and science learning. Papert's (1980) research and writings on the LOGO programming language provide an early exemplar of constructionist learning in action. In LOGO, users control a small icon called the "turtle" using a series of basic terminal commands. For instance, a command as simple as `FORWARD 100 RT 90 FORWARD 100` would move the turtle as follows:

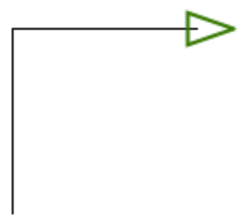


Illustration 1: LOGO Example 1

Used as a drawing program, LOGO puts early mathematics to work; for instance, learning about angles is no longer a matter of identifying obtuse versus acute and memorizing factoids about triangles. Instead, angles become an observable concept with which a learner can experiment and interact through drawing on screen. With understanding, a learner can create a limitless range of shapes or images. LOGO's affordances extend far beyond instantiating concepts from geometry and arithmetic. An ambitious user might compose a command such as `REPEAT 20 [FORWARD 100 RT 70]` and create:

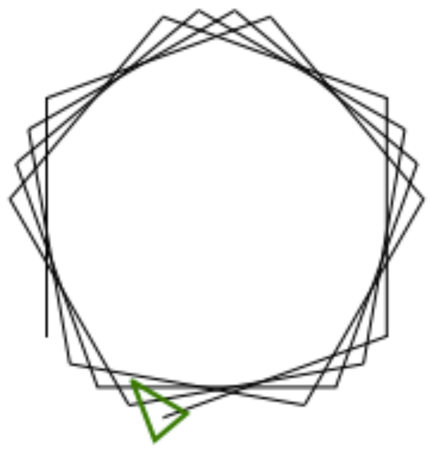


Illustration 2: LOGO Example 2

or even try out `REPEAT 20 [REPEAT 20 [FORWARD 100 RT 70] LT 70]` to produce:

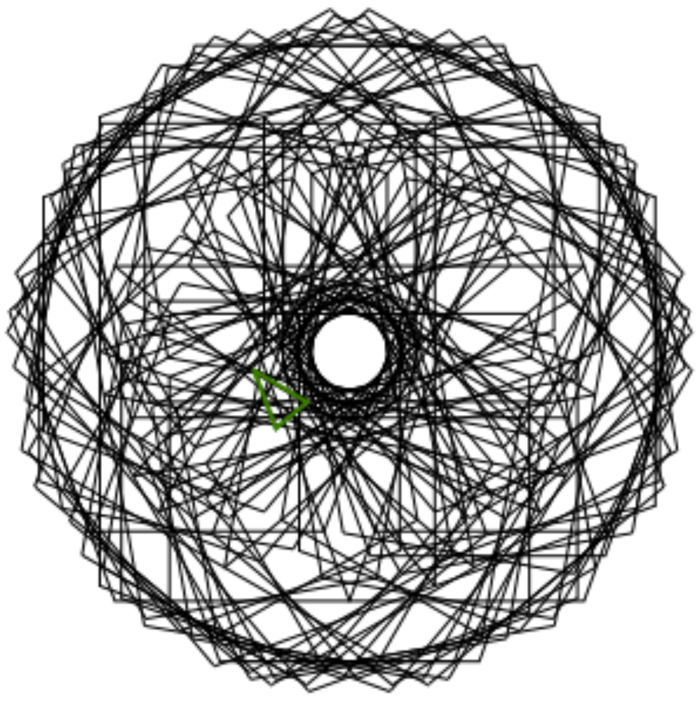


Illustration 3: LOGO Example 3

While certainly provoking some challenging geometrical thinking, these digital artifacts also challenge and encourage learners to begin developing mental models around processes, procedures, and concepts that undergird the practice of computer programming.

It is no surprise that constructionist learning approaches have arisen in parallel with computers and other digital tools. Computers' capacities for simulation are a powerful tool in science and complex systems learning, allowing learners to create, test, and interact with digital artifacts that model phenomena from the observable world (Resnick, 1997; Wilensky & Resnick, 1999).

Scratch, the programming language used in this study, is a constructionist descendant of LOGO and fosters understanding through playful trial and error (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010). Constructionism will be further discussed in specific contexts as the review continues.

COMPUTER PROGRAMMING

Computer programming could be succinctly described as creating instructions with which a computer might solve a problem or execute a task. Larry Wall, creator of the Perl programming language, expands upon this definition with an elegant analogy:

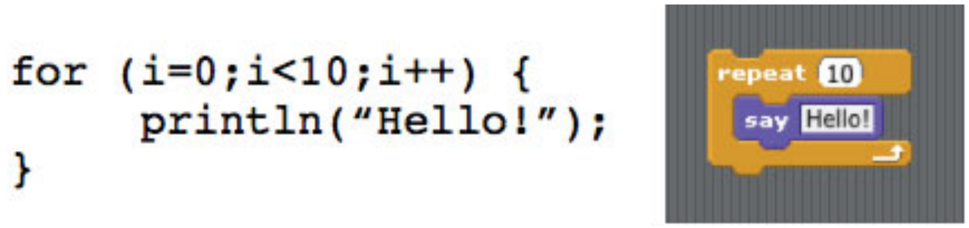
> Computer programming is really a lot like writing a recipe. If you've read a recipe you know what the structure of a recipe is. It's got some things up at the top that are your ingredients, and below there are directions for how to deal with those ingredients. A very similar thing happens in a computer program. You list the things that you're going to be dealing with and then you have some instructions that say what to do with those ingredients. (Big Think, 2011)

While this explanation provides an excellent bird's eye view of programming, questions may still remain: How does a programmer write such a recipe? What goes into writing one?

Programs are written in a programming language. For our purposes, a programming language is simply a language that a computer understands. More accurately, programs written in a programming language are first translated into a special machine language that computers understand, though this step is beyond the scope of this discussion.

Like any language, programming languages have syntax and grammar. Some have overarching rules governing structure and composition (ingredients at the top, as Wall says), while some do not. In the former category are languages like Java or C, designed for writing entire recipes. On the other extreme is a language like LOGO, which simply executes directions as the programmer gives them. Type `FORWARD 100 RT 90 FORWARD 100` – the turtle moves, turns, moves, and waits for its next instruction.

While the programming languages mentioned thus far are “written” in a literal sense, some adopt entirely different paradigms. In Illustration 4, the left example is written in Java. The example on the right is in the Scratch programming language, a visual programming environment optimized for simple animations, graphics, and sound (Maloney, Resnick, Rusk, Silverman, and Eastmond, 2010). Their outputs are essentially identical. The Java example uses that language’s prescribed conventions to describe a task (displaying “Hello!” ten times). Alternately, Scratch provides the programmer a library of blocks instantiating various computational concepts; the act of programming is not so much literally writing code as it is arranging these blocks in an appropriate way to obtain the desired output.



```
for (i=0;i<10;i++) {
     println(“Hello!”);
}
```

```
repeat 10
  say Hello!
```

Illustration 4: Simple Programming Examples, Java on left, Scratch on right

The mention of computational concepts brings us to a second important point. Returning to our recipe analogy, it would not be entirely fair to say that recipes are written in “plain English.” In addition to physical building blocks such as ingredients and kitchen tools, recipes include a host of culinary processes and procedures that may confound the beginning cook with a single word. *How does one sweat an onion? What in the world is a chiffonade?* Computer programming includes its own library of similarly contextually-rich processes and concepts. These processes and concepts are expressed differently in different programming languages; for instance, in the above examples we see *looping* in action. In Java, a loop is set up using the first line of code in the example above. Scratch uses a repeat block. In this study and review, these processes and concepts are referred to as computational thinking concepts and are discussed in much greater detail later in this chapter. For a preview and short explanation of the computational thinking concepts under consideration in this study, please visit Appendix F.

NOVICE PROGRAMMERS

As this study focuses on novice computer programmers, this next section will review research on this particular population of learners. I will discuss general characteristics of novice programmers as well as differences between novice and expert programmers.

Characteristics of Novice Programmers

A wide breadth of research has examined characteristics potentially impacting the experience of the novice programmer, most considering factors correlated with performance in introductory programming courses and/or programming self-efficacy.

Factors influencing programming success, typically based on performance in an introductory programming course, lean towards experience in specific content areas. Numerous studies have links between programming performance and standardized mathematics scores (Barfield, LeBold, Salvendy, and Shodja, 1983; Bergin and Reilly, 2005; Byrne and Lyons, 2001; Pillay and Jugoo, 2005; Wilson and Shrock, 2001) as well as chemistry (Barfield, LeBold, Salvendy, and Shodja, 1983), physics and biology (Bergin and Reilly, 2005), and science courses generally (Byrne and Lyons, 2001). While aptitude in mathematics is often interpreted as a proxy for a suite of abstract problem-solving skills vital to programming, other studies have refuted the importance of abstraction (Bennedsen and Caspersen, 2006) and general cognitive skills (Barfield, LeBold, Salvendy, and Shodja, 1983).

Research about the impact of content areas outside of science and mathematics is primarily focused on language. While proficiency in the language of classroom programming instruction understandably has some bearing on course performance (Pillay, 2005), this may be a general artifact of classroom instruction, as Wong, Ceung, and Chen (1998) show that English fluency plays little role in the competency of professional programmers working in an English-based syntax.

While areas within the arts and humanities have been successfully leveraged for topic-specific programming interventions (as will be discussed later), the impact of experience in (rather than enthusiasm for) the arts and humanities on novice programming in general is unexplored.

Learner characteristics associated with programming self-efficacy are most often those associated with personal history and experience. Based on a survey of students in an introductory Java course, Askar and Davenport (2009) found programming self-efficacy strongly connected with computing experience as well as family computer usage (primarily the usage by siblings and mothers, with computer usage among fathers having virtually no impact). The choice of student major (computer programming versus electronics or industrial engineering) played a small role, though all students were comparably qualified in mathematics and other prior coursework. Other studies (Hasan, 2003; Wiedenbeck, 2005) support the importance of learners' perceptions of their own computer experience. Interestingly, specific aptitudes and domain expertise appear to play little role in novice programmers' self-efficacy beliefs, despite their importance in predicting programming success. Looking at computing and programming self-efficacy in specific contexts, Ortiz and Webster (2011) showed a positive correlation between task-specific self-efficacy and computing self-efficacy, with the strength of this correlation increasing with the novelty of the task. This is a promising notion in considering computing and programming activities based around the relatively novel arts and humanities as this study does; while these fields are not traditionally associated with programming aptitude, self-efficacy in these context domains may positively influence learners' self-efficacy around programming.

Novice & Expert Programmers

Across many fields, key differences between novices and experts are closely linked to the organization of information: how situations are mined for patterns, how knowledge is stored for retrieval, and how information is evaluated on the basis of context (Bransford, Brown, and Cocking, 2000). This theme holds true for novice and expert programmers; much of the research in this area discusses chunking, the practice by which programmers organize knowledge in a meaningful fashion (Graci, 1992). For instance, a more novice programmer comfortable with loops might spend time implementing various multiple loop approaches to a problem until the correct nested loop solution is discovered. Meanwhile, for the expert programmer the nested loop could be a freestanding chunk all on its own, promptly recognized as the problem solution and executed.

Novice programmers do not simply organize information into meaningful chunks less often than experts. They organize chunks more often around natural language as opposed to programming concepts (McKeithen, Reitman, Reuter, & Hirtle, 1981) and face difficulty in transitioning from conceptual understanding to practical implementation of most programming concepts (Butler & Morgan, 2007).

Learning to usefully organize information like an expert is beset by its own set of questions. In comparing learners working with procedural and object-oriented programming languages, Wiedenbeck, Ramalingam, Sarasamma, and Corritoe (1999) did not find either group consistently evincing superior programming understanding. (Procedural programming is fundamentally based on stepwise instructions while object-oriented programming focuses on combining data and functions into modular objects.) On assessments using shorter programs, object-oriented programming learners performed better on questions about program function, while procedural programming learners performed better on questions about control and data flow; the two groups performed similarly on questions about specific operations within the programs. On assessments using longer and more advanced programs, procedural programming learners performed better across all types of question. Pirolli and Recker (1994) highlight the formidable metacognitive demands of successful programming learning, including self-monitoring of knowledge and self-motivated generation of concrete explanations of abstract concepts.

Research on the novice-expert divide in programming does not stop at chunking. It touches heavily on comfort with abstraction (Ye, 1996) and correctness of mental models (Ma, Ferguson, Roper, & Wood, 2011; Ramalingam, LaBella, & Wiedenbeck, 2004) as well. All three of these topics are inexorably connected; comfort with degrees of abstraction is a prerequisite for larger and more concept-intensive chunks and correctly organized chunks are integral to an effective mental model.

The challenges of novice information organization are an important aspect of this study. No strangers to chunking, experts in musical domains may organize and internalize enormous amounts of information. Jazz improvisers carry libraries of melodic approaches to a myriad of harmonic situations, to be deployed on the fly (Brown, 1991), and expert instrumentalists utilize combinations of cognitive and motor chunking to quickly learn or even memorize highly complex musical passages (Chaffin & Imreh, 2002). The question of how musical knowledge scaffolds programming learning views music not only as an engaging context for learning programming, but as a context in which learners and/or performers often organize information.

Musicians as Novice Programmers

In part, this study focuses on what sets musicians apart from other novice programmers. A large body of research has focused on the cognitive impacts of musical trains. Multiple reviews of the literature highlight consistent short-term increases in spatial, verbal, and memory abilities during the first two years of musical training in children (Costa-Giomi, 2014; Miendlarzewska & Trost, 2014). However, these results tend to dissipate by the third year of musical training and the authors point out that the web of factors related to both inclination to pursue musical training and cognitive abilities becomes too complex to unravel by this point. Research from neurology echoes these findings around early musical training, identifying increased brain activity in certain cognitive centers in child and, in some cases, adult musicians (Steele, Bailey, Zatorre, & Penhune, 2013; Zuk, Benjamin, Kenyon, & Gaab, 2014).

While the impact of unrelated and uncontrollable factors makes studying the cognitive impact of musical training in adults a formidable challenge, this question has been approached from other angles. Johnson-Laird (2002) presents a model of jazz improvisation as a mentally algorithmic process and Amitani and Hori (2002) have developed tools to support the computational foundations of musical composition.

COMPUTATIONAL THINKING

This study utilizes computational thinking concepts to address the challenge of describing knowledge or understanding about programming in a way that is not wedded to the conventions of a particular programming language; rather, these concepts are typically integral to computer science and recur in language after language. The route towards situating abstract computational knowledge like this begins with the computational literacy and computational thinking frameworks.

Computational literacy, as introduced by DiSessa (2000), uses print literacy as a jumping-off analogy with which to expand computational know-how from a grab bag of skills to something larger and more abstract. For instance, most of us would likely agree that literacy does not stop at reading and writing; a literate person should be able to organize an effective argument and read text critically, for instance. The contemporary print literate individual can not only decode symbols into words and sentences, but can use text as a medium in which to share and evaluate ideas and to ask and answer questions about the world.

diSessa defines computational literacy as a similarly broad set of skills, focused around taking advantage of computer technology's affordances to articulate ideas and engage with the world in a variety of ways. He describes general literacy as built upon three pillars: material, cognitive, and social. This framework may also be applied to thinking about computational literacy. The material aspect of computational literacy describes the ability to use programming languages and navigate digital environments. A learner that understands the structure and syntax of the Java loop in Illustration 4 is demonstrating material computational literacy. The cognitive aspect of computational literacy describes the ability to wield the computer as a thinking tool and to frame problems in ways that take advantage of its computational capabilities. Planning the overall structure and flow of a program (before coding it in a particular language) requires cognitive computational literacy. Finally, the social aspect of computational literacy describes the ability to communicate about computation in a way that is comprehensible to other practitioners and learners. For instance, two programmers discussing the efficiency (or inefficiency) of the program planned in the previous cognitive literacy example would be demonstrating social computational literacy.

All three elements of computational literacy are of interest in this study. Creating a functional program in Scratch will demand a certain level of material computational fluency. The cognitive aspect of computational literacy is of particular interest in terms of how participants' musical projects are impacted by Scratch's computational affordances. Finally, how students narrate and explain the process of creating their projects may involve social aspects of computational literacy.

More recently, computational thinking has become a framework for discussing computational knowledge and understanding in even more abstract terms, moving us much closer to identifiable computational fluencies in the process. Wing (2006) asserts that computational thinking "involves solving problems, designing systems, and understanding human behavior, by drawing on concepts fundamental to computer science." Computational literacy is fundamentally about programming, whereas computational thinking explicitly does not involve a material component. It emphasizes conceptualizing computational solutions to problems, though not necessarily turning those solutions into executable code. With this in mind, some educational initiatives have focused on developing learners' computational thinking independent of computers (Taub, Ben-Ari, & Armoni, 2009). Wing goes so far as to identify a variety of everyday activities, from choosing a line at the supermarket to packing a school backpack, where smart practitioners are in fact exhibiting computational concepts whether they know it or not. It is in explaining these everyday activities that more discrete and definable computational concepts come to light. Sitting under the fundamental umbrellas of abstraction and automation, these concepts ranging from topics closely tied to computer programming (parallel processing, recursion) to more abstractly applicable practices (modeling or decomposition).

Computational thinking has been concretized in a variety of ways beyond the above. For instance, Basawapatna, Koh, Repenning, Webb, and Marshall (2011) described how novice programmers could describe how agent interaction algorithms (referred to as Computational Thinking Patterns) that they had learned developing games might be put to work in a science simulation. That is, the student recognized a fundamental programming pattern underneath the gaming context in which they were working.

The computational thinking concepts considered in this study are derived from prior Scratch research. A study focusing on the use of Scratch blocks explicitly related to programming (as opposed to graphics or other media manipulation) in over 500 Scratch projects identified the following concepts, in order of frequency of use: user interaction, loops, conditional statements, communications and synchronization, boolean logic, variables, and random numbers (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008). A more recent framing of student Scratch work includes computational concepts, practices, and perspectives, in ascending order of abstraction. Concepts in this case included sequences, loops, parallelism, events, conditionals, operators, variables, and lists (Brennan & Resnick, 2012).

As this study also utilizes Scratch, the prior two studies (Brennan & Resnick, 2012; Maloney, Peppler, Kafai, Resnick, & Rusk, 2008) will provide the basis for the computational concepts under consideration here.

Approaches to Programming Learning and Instruction

This study focuses on the experience of learning programming in the context of creating a relatively free-form musical project. The sections that follow will present an overview of existing approaches to programming instruction, from lecture-based models to constructionist approaches. I will discuss tinkering, a guiding idea in this study, as well as several specific content contexts for constructionist programming learning (music included).

Lecture-Based Programming Instruction

Traditional models of beginning programming instruction are typically based around lectures and relatively small-scale programming assignments (Linn & Dalbey, 1985). Some courses may begin with a focus on the programming language itself, beginning with syntax and expanding from there, while others might organize around programming concepts and let students come to understand the important details of the language as they go (Kranch, 2012). In all these cases, students face at least three potential challenges: a demanding emphasis on syntax (Jenkins, 2001), a lack of engaging context for programming assignments (Linn, 1985), and the rush to cover a prescribed amount of material (Sleeman, Putnam, Baxter, & Kuspa, 1987). As such, these courses can leave students with relatively poor understandings of programming (Linn & Dabley, 1985; Soloway, Erlich, Bonar, & Greenspan, 1982) while providing them little or no opportunity to think deeply about design or problem-solving (Pea & Kurland, 1984).

Constructionist Programming Instruction and Learning

The affinity between computer programming and constructionist learning is well articulated by Knuth:

> It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand

something until after teaching it to a computer, i.e., express it as an algorithm. (Knuth, 1975)

While this study is not explicitly focused on algorithmic thinking, Knuth's statement nevertheless echoes the lessons of LOGO. To "teach" a computer to draw a square or triangle or hexagon, the learner must understand the geometry of the shape in question. This certainly does not apply only to situations in which the computer is a tool with which to explore content outside of programming. The same could be said for the nested loop in Illustration 3, the final LOGO example. Suffice it to say that there are no better routes to understanding computing concepts than to execute them on a computer, and in both cases the computer does not simply give feedback in terms of success or failure, but provides endless opportunities for theory-testing. Perhaps a learner will stumble on a pleasantly surprising "failure" along the way and then attempt to unravel where it came from!

The following sections review several specific areas under the heading of constructionist programming instruction and learning. Tinkering is not explicitly a style of instruction, but rather a style of programming practice marked by learning through exploration. I will contrast this with a discussion around design-focused learning and will also discuss approaches to programming instruction in several important contexts: game design, arts, and music.

Tinkering

In the context of programming, tinkering can be described as a tightly coupled cycle of learning and making. Tinkering is preceded in the literature by bricolage, a term coined by Levi-Strauss (1968) to describe a "science of the concrete" in primitive societies, in which objects at hand are rearranged and renegotiated until a suitable theory of the moment is reached. Turkle and Papert (1990) use bricolage to frame a discussion of the styles of young programmers. They discovered that many programmers did not adhere to the methodical planner approach that might be presented in a programming course. Rather, the *bricoleurs* rarely planned more than a step or two ahead and it was typically through a cycle of corrections and intuition that they reached their goal or destination.

Common themes in research on tinkering in programming are testing (Brandt, Guo, Lewenstein, Dontcheva, & Klemmer, 2009), trial and error (Dorn & Guzdial, 2010), and a lack of specific goals (Petre & Blackwell, 2007). Hancock (2003) offers that tinkering often involves making something work without directly understanding how. Petre and Blackwell (2007) describe children who identify their tinkering programming simply as play, not as programming at all. While most studies of tinkering have been observational and interpretive, data-mining and learning analytics have been used to identify tinkering as a particular phase of a student programming activity, bookended by defined phases of exploration and refinement (Berland, Martin, Benton, Smith, & Davis, 2013).

This picture of tinkering functions as a framework for my discussion of programming practice; themes such as testing and goal focus will be used to derive metrics with which to compare the programming of different participants.

Design

Design, meanwhile, could be considered the apotheosis of the planner described by Turkle and Papert above. While the *bricoleur* rambles around, testing ideas, backtracking as necessary, all with perhaps no particular goal in mind, the designer is generating problem definition statements and evaluating design concepts (Dym et al., 2007). Instruction based around design, such as Problem-Based Learning (Savery &

Duffy, 1995) and Challenge-Based Instruction (Martin, Rivale, & Diller, 2007), has found great utility in training engineering and other design professionals. Tolerance for uncertainty and ambiguity are often elements of these instructional approaches and are important to the contemporary designer (Dym et al., 2007); however, this embrace of uncertainty is frequently guided by heuristics and a history of best practices and, as such, is very different than the exploration and experimentation associated with tinkering as Turkle and Papert discuss it.

Games

Game design has been an especially fruitful area for both encouraging programming and observing programming learning in action. Gaming as an engaging motivator has been the foundation of many interventions around programming education and conceptual reframing of computer science education (Bayliss & Strout, 2006; Coleman, Krembs, Labouseur, & Weir, 2005; Leutenegger & Edgington, 2004; Roden & LaGrande, 2013; Xu, Blank, & Kumar, 2008).

Most research focusing specifically on computational thinking involves game development. As discussed earlier in this review, gaming was used as a forum for looking at computational concepts in the form of Computational Thinking Patterns (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011). Scratch readily lends itself to game design and many of the student programs discussed in the papers from which this study derives its computational thinking concepts were in fact games (Brennan & Resnick, 2012; Maloney, Peppler, Kafai, Resnick, & Rusk, 2008). Recent initiatives have even gone beyond simply using gaming as an engaging entry point to programming and have proposed game design curriculums explicitly guided by computational thinking concepts (Repenning, Webb, & Ioannidou, 2010).

Arts and Crafts

Similarly, mediums related to art and craft have primarily been leveraged to engage learners with programming. The Lilypad Arduino combines a small programmable microcontroller with conductive thread to allow relatively novice programmers to integrate sensors, LEDs, and other electronic elements into clothing and textile products (Beuchley, Eisenberg, Catchen, & Crockett, 2008). This is representative of a larger movement in bringing the fruits of accessible programming into the physical world (House, Malloy, & Buckley, 2010).

The development of computational proficiencies and computer science concepts has been studied in a small number of tightly focused artistic contexts. The Storytelling Alice programming environment supports the creation of animated stories by providing users pre-created characters, scenery, and animations as well as by basing tutorials around storyboard examples, acquainting learners with concepts such as objects and methods in the process (Kelleher & Pausch, 2007).

Music

Music provides a number of entry points and approaches towards learning computer programming. While composers have experimented with algorithm-based music since computer programming has been a possibility (Jacob, 1996; McCormack, 1996), algorithmic music has more recently been used as an introduction to programming and computing (Peterson & Hickman, 2008) and has been highlighted as a context for identifying computational thinking concepts in music itself rather than only in its production (Edwards, 2011). The relatively recent practice of *live coding* has provided a novel new approach towards exploring computer programming through music (Blackwell & Collins, 2005; Bown, Eldrige, & McCormack, 2009; Brown, 2007; Brown & Sorensen, 2009; Sorensen & Gardner, 2010). In a live coding performance, music is created in the moment by editing code in real time, rather than executing an existing program.

Reaching into the physical world, Sawyer et al. (2013) showed students successfully creating novel musical instruments using sensors, interface devices, and a visual programming environment. Ensembles such as the Princeton Laptop Orchestra, which brings together students of all musical and technical skill levels, similarly encourage programming learning in the pursuit of a creative musical goal (Wang, Trueman, Smallwood, & Cook, 2008). Utilizing an approach called "computational music remixing," EarSketch challenges learners to make music in a hip-hop vein by manipulating loops and samples using specific computational approaches in Python (Sawyer et. al, 2013).

In spite of all this, research that specifically focuses on leveraging musical knowledge for programming learning is somewhat more limited. Working with high school musicians, Meyers, Cole, Korth, and Pluta (2009) used structural concepts from contemporary music as successful entry points to programming concepts in a short introductory programming course. Similarly, Ruthman et al. (2010) demonstrate a wide variety of computational concepts in action such as loops, initialization, variables, and modularization in a hybrid computer science and music course. Both of these studies demonstrate that musical concepts can be successfully used as a foothold in acquainting students with programming concepts. However, both of these examples depend upon instructors' guiding student in making these musical-programming connections. The literature does not address the impact of this same musical know-how on a less defined programming task.

OPEN QUESTIONS

While the review of the literature has converged on several examples of music as a context for learning programming, these studies provide learners explicit connections between musical concepts and programming concepts. The literature does not address what musicians of different backgrounds potentially bring to a less structured experience of learning to program. I believe that investigating this is worthwhile in light of the many accessible programming languages and opportunities available to musical and non-musical learners. This leads to my overarching research question:

Do different kinds of musical backgrounds of learners play a role in novice programming?

I will address this larger question through three actionable research questions:

1. *How do musical concepts emerge as a scaffold for novice programmers?*
2. *What kind of patterns do learners from a variety of musical backgrounds exhibit in their programming processes?*
3. *How did the final projects of learners from a variety of musical backgrounds differ?*

While research has explicitly connected musical concepts with computational concepts (Meyers, Cole, Korth, & Pluta, 2009), I am interested in what role an understanding of musical concepts may play without that explicit connection. How expertise in a separate domain may deal with the challenge of organizing programming knowledge and structuring programming tasks is one specific perspective guiding this question. I will answer this by analyzing participant talk while they work on an open-ended musical programming project in Scratch.

The umbrella of "musician" encompasses an enormous range of musical study and practice. I will compare the programming processes of musicians from a variety of backgrounds by analyzing participants' Scratch projects using metrics based on a "programming learning by tinkering" perspective.

Finally, I will look at the impact of different participant backgrounds by analyzing participants' completed Scratch projects, this time based on their usage of computational thinking concepts.

Chapter 3: Methods

This section will explain the epistemological and theoretical perspectives guiding this qualitative research study, justify the use of an interpretive methodology, and describe the study's participants, materials, data sources, and analysis. It will also present the steps taken to ensure qualitative rigor and my perspective on the study’s generalizability. Finally, I will explain my own positionality as a researcher and the role of my personal experiences and perspectives in shaping this study.

Epistemology and Theoretical Perspective

Epistemologies describe theories of knowledge and, especially important in the practice of research, ways of knowing, informing the theoretical perspective and methodology that follow (Crotty, 1998). In particular they communicate important assumptions that inform how claims and conclusions are drawn from data and analysis (Koro-Ljungberg, Yendol-Hoppey, Smith & Hayes, 2009).

As this study focused on the experiences of the participants in learning and creating, I took a constructivist epistemological perspective. Broadly, this perspective asserts that reality is socially constructed (Lincoln & Guba, 1995); in a more practical research context, constructivism places the participants' own constructions of meaning at the forefront and attempts to reach understanding through interpretation of those meanings (Charmaz, 2000). Implicit in constructivism's participant focus is an acknowledgement of the reality in which the study exists. From the experiences and beliefs of the researchers and participants to the dynamic relationships between those same parties to the qualities of the research site, these factors all inform the research at every level, and the researcher's conclusions must be qualified with this context in mind (Charmaz, 2000).

An important element of this study is its focus on the experience of novice programmers of different backgrounds while they are actually programming, not grappling with programming concepts delivered in a lecture or book format; furthermore, the programming activity focused on the creation of a musical project in Scratch. As such, this study was heavily informed by the learning theory of constructionism, which posits that learning can often be effectively transmitted when it is embedded in the creation of physical or digital artifacts (Papert, 1991).

METHODOLOGY

This study adhered to an interpretivist descriptive framework. This is because my most substantive questions revolved around describing what role different musical backgrounds may play in novice programming. In short, my emphasis will be on describing rather than explaining the experiences of the participants (Charmaz, 2006). The study focuses closely on context and a search for patterns within that context, and utilizes complex data requiring careful interpretation. These are all hallmarks of an interpretivist research paradigm (Glesne, 2011).

The interpretation and discussion of my results are literally descriptive. This includes description of the projects that participants have created along with selected stories of the participants' programming experiences. I will also describe and attempt to explain any meaningful patterns I observe in participant programming or projects.

Finally, while formulating theory was not a goal of this study, my constructivist epistemology demands that I acknowledge that any conclusions I draw are as much (or more) constructed than emergent (Charmaz, 2006).

Participants

Participants were 12 adults with no programming experience. Three participants were drawn from each of four different groups: non-musicians, classical musicians, jazz musicians, and composers. To establish a baseline of expertise, participants in the classical, jazz, and composer groups either had a college degree in the subject or were currently professionally active in that area. Participants were recruited directly, based on my knowledge of their expertise and their availability.

Along with the consent form that appears in Appendix C, prospective participants were also given the pre-survey that appears in Appendix D. Placement in one of the groups described above was based on a simple rubric. Selecting a college major from the options for Question 5 placed students in the Classical ('Instrumental Performance'), Jazz ('Jazz'), or Composition ('Composition') groups. If the participant did not select a college major, I looked toward their personal description of their current musical activities in Question 6. Professional classical musicians, jazz musicians, and composers were placed in the corresponding group. These groups are not meant to be uniform representations of background and knowledge in a particular area, but to represent a reasonable diversity of musical experience. The survey's other questions provided additional data for potential discussion and further reinforced the participant groups.

Materials

Participants worked with Scratch, a constructionist programming language from the MIT Media Lab that is freely available and very popular as an introduction to programming. It is often used for creating animations and games and includes a number of musical elements that can be used to create static pieces of music, virtual instruments, or more sophisticated generative music. Scratch is a visual environment in which discrete programming elements are literally dragged and snapped together; it also encourages rapid development and execution of projects in progress (Maloney, Resnick, Rusk, Silverman, and Eastmond, 2010). Affordances such as these support accessibility for novice programmers.

PROCEDURE

I worked with participants at a quiet location of their convenience, primarily their homes. We worked with Scratch on a laptop computer provided by me. The session began with a short (~20 min) introduction to the Scratch environment, featuring succinct example programs utilizing Scratch objects from the Sound, Control, Sensing, Operators, and Variables panels, as well as keyboard and mouse input. All example programs introduced during this introduction appear in a handout provided to participants (see Appendix E). After giving participants the handout, I created each simple program in Scratch, executed it, and reiterated the short description appearing on the sheet. In addition to providing participants with information about Scratch's capabilities, this handout acquainted them with the simple method by which a Scratch program is assembled. Participants had access to the handout for the remainder of the session.

Participants had the remainder of the two hours or 90 minutes (whichever was shorter) to work on a project of their choosing, guided by the requirement that their final project make sound and offer some degree of interaction with the user (that is, a "virtual musical instrument"). I gently encouraged each participant to work for at least 45 minutes, though some pronounced their project completed or simply that they were finished sooner than that point. QuickTime Player's screen recording feature was used to capture participants' programming as well as audible discussion between the participant and me.

DATA

The data collected in this dissertation includes: a pre-survey, audio recording of participant talk, video recording of their programming, and participants' final Scratch projects.

The aforementioned pre-survey (Appendix D) collected information about participants' musical proficiencies and experiences. This data was used to generate groupings for later analysis. The survey concluded with a final question about programing experience to officially record participants' experience levels.

Participants' thought processes were obtained using a think-aloud protocol during the programming session. Participants were continually encouraged to explain their actions and to articulate their thoughts and ideas, using prompts along the lines of:

"What is this piece you're working on right now?"

"Is this what you were hoping it would do? How is it different from what you had in mind?"

This process of having participants constantly report what they are thinking, doing and feeling is often used in social science to capture processes rather than simply beginning and end points (Ericsson & Simon, 1980).

In addition to collecting participants' final Scratch programs, I also captured video of their programming in progress. This allowed me to access to participants' projects at any point in development.

DATA ANALYSIS

Analysis is described for my four main data sources.

Participant Pre-Surveys

Pre-survey results were used to create groupings for use in later analysis. All participants fulfilled my criteria for inclusion in a participant group, described in detail as follows.

None of the non-musician participants read music or played an instrument. That said, all three were theater and/or dance artists and described music as playing a significant role in their art. I chose participants who fit this description such that while they did not have musical backgrounds (around which most of my research questions are based), they did have precedent for creatively engaging with music.

All musical participants read music and played at least one instrument. Most improvised on an instrument (all participants in the jazz group, two in the composer group, and one in the classical group) and most also composed music (all participants in the composer group, two in the jazz group, and one in the classical group).

Most musical participants had studied their area of expertise at the college level. All members of the classical group had undergraduate degrees in instrumental performance. Two of the three participants in the jazz group had degrees in that area (one undergraduate and one doctorate). Two of the three composers had degrees in composition (one undergraduate and one masters level). Various participants also held additional or advanced degrees in music theory, music education, and ethnomusicology.

Most musical participants divided their professional time between some combination of performing (or composing) and teaching. Two participants (one in the jazz group and one in the composer group) had unrelated full-time jobs and did no teaching, but continued to perform and/or compose.

Think Aloud Interviews

Interviews were transcribed and analyzed using an approach described by Burnard (1991) for analyzing semi-structured interviews. First, the text was organized into distinct utterances. Utterances are defined by Walker and Whitaker (1990) as assertions, commands, questions, or prompts. I followed Burnard's prescription that utterances be related to the task at hand; that is, the programming activity. This excluded some incidental and uncodeable talk as well as overarching discussion about the task as a whole (both unsolicited and in response to questions from me). Some of the talk about the programming task explicitly addressed my research questions and, as such, appears in my own discussion of the study.

I used an open coding approach (Berg, 1989) to freely code individual utterances, generating succinct descriptions of the content of each utterance without any final coding scheme in mind. The next step in Burnard's approach involves creating overarching themes by grouping together related or similar codes. In doing this I arrived at nine themes, each a combination of two elements: type (Question, Assertion, and Intention) and subject (Operation, Programming, Scratch, Musical, and Aesthetic). Because all utterances are related to the task at hand and relatively direct, this simple and direct scheme describes them appropriately.

In this scheme, Questions are relatively straightforward. Assertions include self-narration as well as statements of immediate action ("I am going to change this value to do x"), whereas Intentions are more abstract and less discrete ("I would like to try using a Scratch *repeat* object somehow.") and reflected a possible intended action in the future. More extensive description of subjects and examples of subject/type combinations follow:

Operation: Related to what a program should do in "plain language" (not musical or programming talk).

Operational Intention: *"I want to do something based on holding down the mouse."*

Programming: Related to programming concepts and Scratch objects that are not Scratch-specific.

Programming Question: *"Does* repeat *mean indefinitely? Is it going to stop?"*

Scratch: Related specifically to the Scratch programming environment.

Scratch Question: *"So everything here* [in the programming window] *can just be floating wherever?"*

Musical: Related to musical concepts or utilizing musical language.

Musical Question: *"So what does 0.2 beats mean?"*

Aesthetic: A creative but not technical or musical description.

Aesthetic Assertion: *"Now it sounds like some kind of animal. I love it."*

These themes were used to examine how musical knowledge and talk informed this novice programming experience.

Screen-Captured Scratch Programming

To look at how projects evolved over time, I logged how Scratch objects were added to or removed from participants' programs. While I am interested in programming processes in terms of working steadily towards a goal versus wandering and experimenting with Scratch's features, I do not attempt to clearly define "planners" and "tinkerers." Rather, I sought any patterns that emerged on the continuum between those two extremes. From the logs of participant programming, I chose metrics that utilized the available data to describe participant programming from several different perspectives: Scratch objects added and removed, program tests, and use of Scratch objects by type.

Final Scratch Projects

Final Scratch projects were coded for the use of a variety of computational thinking concepts. These concepts were among those identified in a study of 500 student Scratch projects and include *user interaction*, *loops*, *conditional statements*, *Boolean logic*, *variables*, and *random numbers* (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008), and *sequences*, *parallelism*, *operators*, and *lists* (Brennan & Resnick, 2012). Descriptions of each of these concepts appear in Appendix F, alongside an example Scratch program illustrating the concept in implementation.

In addition to simply counting the occurrence of computational thinking concepts, I considered where these concepts occurred in projects; this is represented by my "average depth" metric, which looks at the nestedness of various CT concepts in a given project. For a given project, I simply add the depths for all instances of a particular concept and divide by the number of instances of that concept. For instance, in the example of a computational thinking diagram that follows in Figure 1, there are three CT concepts in action: two instances of *user interaction* and a single instance each of *conditional statement* and *loop*. *User interaction* appears twice; once at depth level one and once at level three, for an average of two ((3+1)/2). *Conditional statement* and *loop* each appear once, receiving average depths of two (2/1) and three (3/1), respectively.

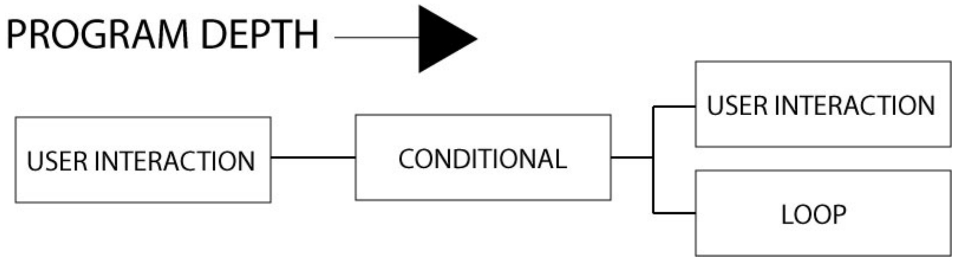
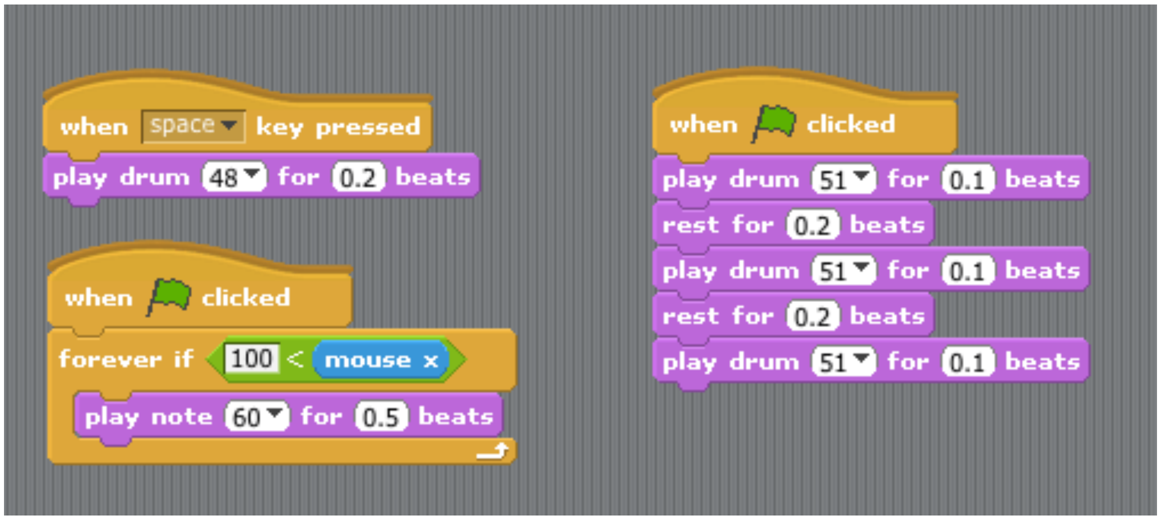


Figure 1: Computational Thinking Diagram Example to Illustrate Calculating Average Depth Metric

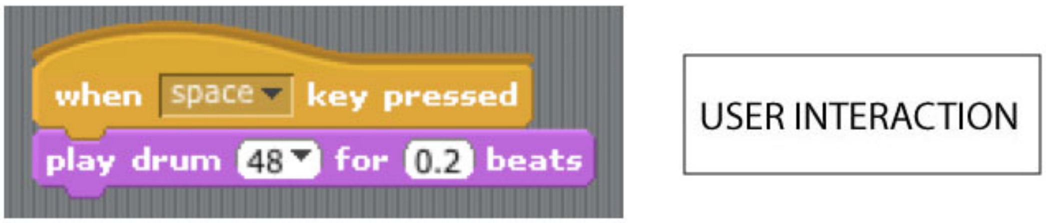
This section of analysis also examined the number of discrete programs in a particular project as well as maximum and average depth of participant projects. I define a “discrete program” as an independent, operable piece of Scratch code. Figure 2 below shows an example of a Scratch project containing three individual programs. While contained in the same project, each chunk of Scratch code is complete and fully operable on its own.



```
when space key pressed
play drum 48 for 0.2 beats

when clicked
forever if 100 < mouse x
    play note 60 for 0.5 beats

when clicked
play drum 51 for 0.1 beats
rest for 0.2 beats
play drum 51 for 0.1 beats
rest for 0.2 beats
play drum 51 for 0.1 beats
```

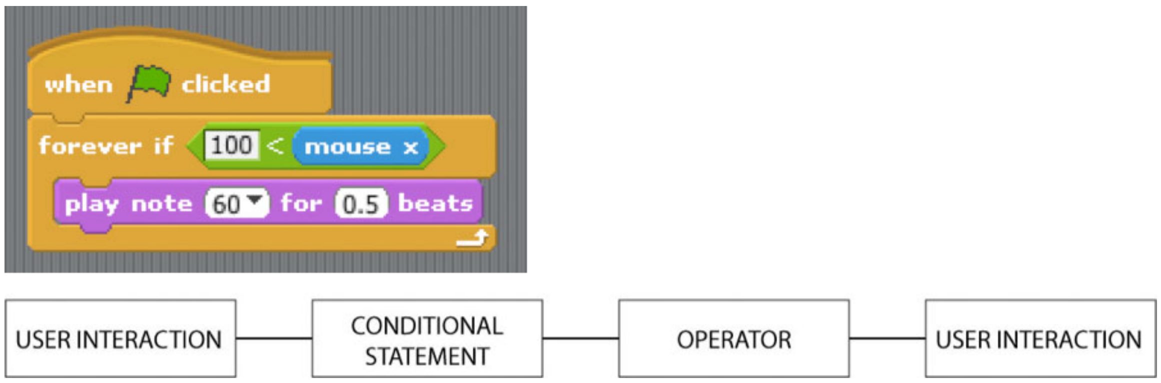
Figure 2: Sample Scratch Project with Three Operable Programs

Project depth refers to computational thinking diagrams and uses the same definition of depth as above. Each program in a given project will have a depth value; the maximum project depth is simply the maximum of these and the average is the average. For instance, the three distinct programs (or chunks) that appear in the project in Figure 2 are represented by the computational thinking diagrams that follow (Figures 3-5). The program on the top left (Figure 3) is represented by a single instance of user interaction and has a depth of one. The program on the bottom left (Figure 4) incorporates several CT elements and has a depth of four. The program on the right (Figure 5) has a depth of two. Maximum depth for this particular project is four and the average is 2.3 ([4 + 2 + 1] /3).



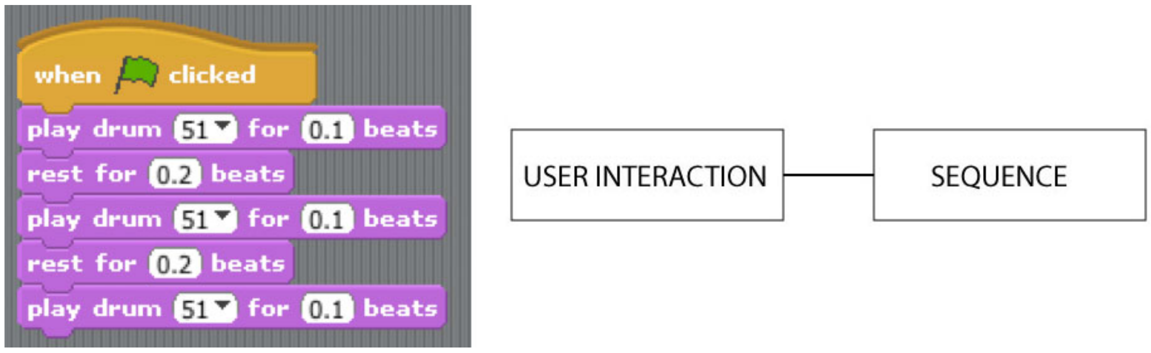
```
when [space] key pressed
play drum (48) for (0.2) beats
```

Figure 3: Computational Thinking diagram for top left sample program.



```
when [flag] clicked
forever if <(100) < (mouse x)>
    play note (60) for (0.5) beats
```

Figure 4: Computational Thinking diagram for bottom left sample program.



```
when [flag] clicked
play drum (51) for (0.1) beats
rest for (0.2) beats
play drum (51) for (0.1) beats
rest for (0.2) beats
play drum (51) for (0.1) beats
```

Figure 5: Computational Thinking diagram for right sample program.

Comparative Analysis

The answers to my research questions are based on an interpretive analysis of themes derived from the data. In discussing chunking, tinkering, constructionism, and computational thinking, I have described the theoretical frames that guide the study design and data analysis (Hewitt-Taylor, 2001). Data, analysis, and sample answers to each question, which served as a guide for me during the dissertation, are consolidated in the Research Matrix in Appendix A.

My research questions deal with the processes and products of participants from a variety of musical backgrounds and utilize a constant comparative analytic approach.
In inductive analysis, patterns "emerge from the data rather than being imposed on them prior to data collection and analysis" (Patton, 1990, p 360). I used an inductive approach to look for patterns across all the data.

TRUSTWORTHINESS

I worked to ensure trustworthiness and qualitative rigor in a number of ways. The use of multiple data sources (interviews and Scratch projects) triangulated emerging findings from these sources. While the coding process was entered into with a number of broad codes in mind, I allowed the coding process to be fundamentally guided by the data and arrived at a final coding scheme that I had not expected in the beginning. I have carefully considered my own biases and experiences, which appear below; these were kept carefully in mind during data collection as well as analysis. Consultation with my chair and other committee members has helped address specific issues that arose during the course of the study.

Assumptions and Limitations

Like any other study, this one includes assumptions in place from the very beginning. Grouping participants as I have may be reasonably descriptive in the broadest sense but does not begin to capture the diversity of musical backgrounds and experiences with which such a system of categorization would potentially have to contend. For instance, a “classical” musician specializing in 20th century music may regularly deal with mental musical challenges that an instrumentalist specializing in Renaissance music would not even begin to understand. This holds true for the jazz and composer groups as well, with some active practitioners working within well-established traditions while others explore the outer limits of novelty and complexity.

I also assume that all participants entered this study as similar programming novices. I chose Scratch for this study specifically to accommodate participants’ lack of concrete programming experience and to dispel the unease with syntax and semicolons that may accompany this lack of experience. That said, this study does not deeply examine participants’ levels of digital literacy and general (if unconscious) engagement with computational thinking concepts. For instance, while I would not consider using Wordpress to be programming experience, it is a content management *system* and engagement with it may foster elements of systems and computational thinking. Providing all participants with careful explanations of Scratch objects in concept as well as implementation was integral here. I also carefully considered this aspect of participant comfort and confidence in my discussion.

An important limitation to note regarding this study’s generalizability is the participant population and study environment. All musical participants were skilled adult musicians electing to take part in a potentially enjoyable programming experience of their own volition. Insights from these participants may not be easily applicable to young learners with far less musical experience in a classroom environment.

I worked to be very present and aware of my role during data collection to provide each participant a comparable experience. While some participant questions were self-directed or simply rhetorical, others were very much directed at me, and I did my best to address them consistently. Generally, when a participant had reached an impasse in articulating an idea in Scratch, I would suggest a particular object to try. Several participants rescaled mouse x and y values and in some cases I helped them with the appropriate arithmetic. The only instances in which I offered any creative input was in cases where participants reached an "I'm not sure what to do next" point relatively early in the process (less than 30 minutes); I would suggest a family of Scratch objects they hadn't explored, with some context relating to what the participant had already worked on.

RESEARCHER POSITIONALITY

One of the primary assumptions of my constructivist epistemology is that the research is impacted by the beliefs, experiences, and values of the researcher. As mentioned in the introduction, this study is entirely inspired by a meaningful personal experience. As such, it was vitally important that I assess the lens through which my interpretations and conclusions emerged. Here I present the elements of my background and beliefs that most clearly informed this study.

My backgrounds in the core elements of this study (programming and music) contain similar blends of formal and informal education. Prior to the course described in the introduction, I had taken two programming courses, a semester of Pascal in high school and a semester of Fortran while working on my undergraduate degree in mechanical engineering. In a job following college I used Fortran a great deal but exclusively for coding numerical calculations. Following the electronic music course, I did not slow down in working on recreational projects in MAX/MSP and, perhaps as a result of that, was very pleasantly surprised at how easily I took to JAVA in a course I took several semesters later. During graduate school I worked on projects using a wide variety of programming languages and environments and in all cases have had reasonable success with diving into projects and learning as needed. In short, my own programming experience certainly biases me towards the merit of tightly coupled learning and practice.

My experience in music follows a similar path, though beginning with substantially more formal school experience. I played in the school band through junior high and high school, learning an instrument and to read music, and studied music theory with a private instructor during high school. While I put music on a relative hiatus during my undergraduate years, upon graduating I dove into a rich, informal musical education as a semi-professional musician in Austin. This was an extremely valuable but very holistic education. I'm hard pressed to isolate and identify specific things that I learned during this time; rather, I can only say that my voice as an improviser, composer, and instrumentalist became more sophisticated as well as more personal over time.

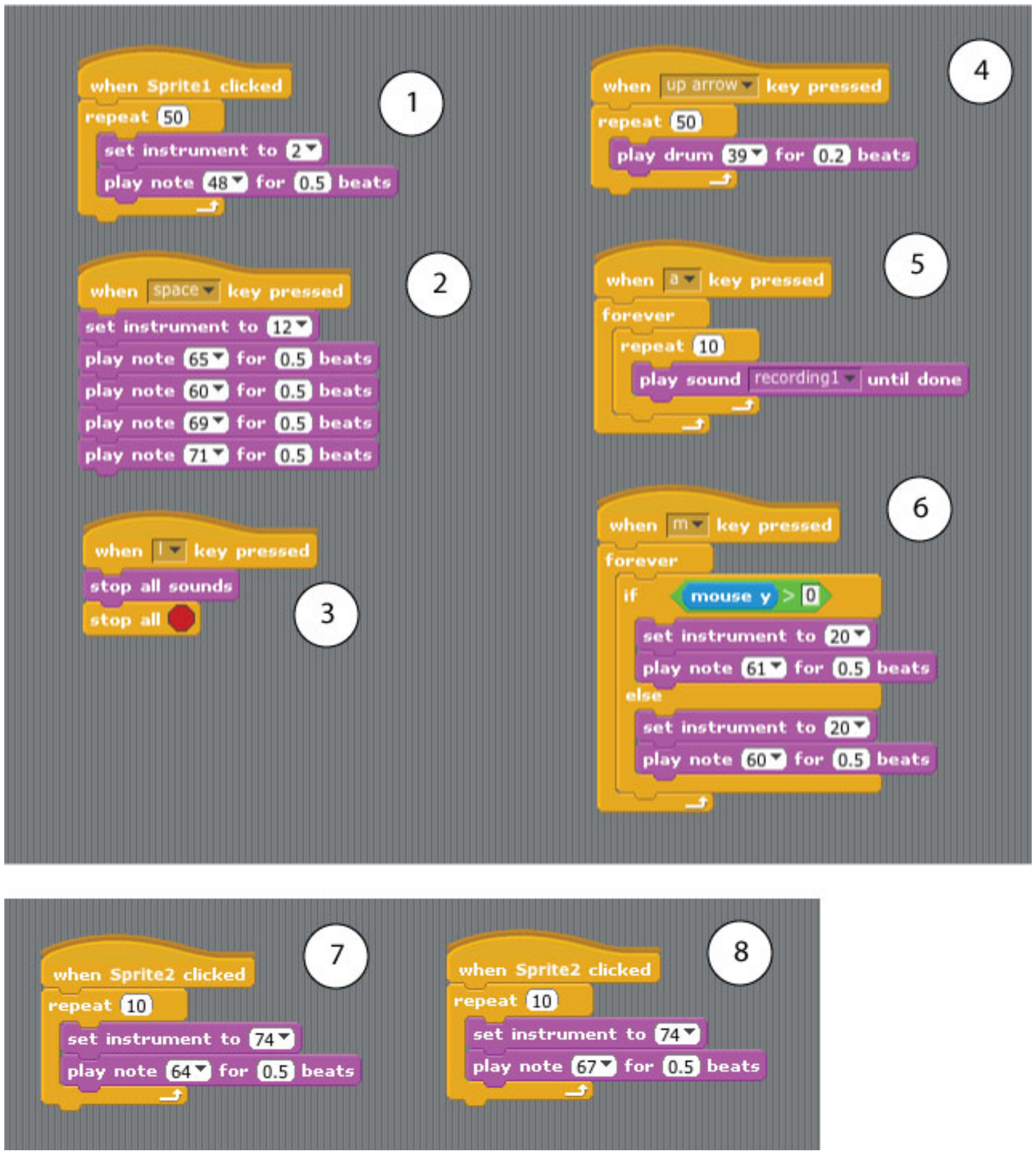
Chapter 4: Individual Results

This chapter shares Scratch projects, computational thinking diagrams, and time-based visualizations of programming for each participant. Computational thinking diagrams are explained in Chapter 3; the time-based visualizations of participant programming track the addition and removal of Scratch objects by type as well as the creation of sounds and additional sprites. They also show program tests (vertical lines) and continuous tests during which projects were edited (horizontal bars above the graph).

NON-MUSICIANS

Participant A

Seen in Figure 6, this project primarily relies on key presses and mouse clicks to create a variety of sounds. Element include a repeated single note (1), a short melody played once (2), a repeated drum hit (program 4), and a repeated recording of crumpling paper (5). The main stage also includes a set of objects that repeats a single note based on the position of the mouse (6) as well as an abort key (3). The participant created a second sprite to allow for a second instrument sample (since each sprite accommodates one instrument at a time, a discovery made in the course of data collection). Code associated with the second sprite repeats two notes (programs 7 & 8).


```
1
when Sprite1 clicked
repeat 50
    set instrument to 2
    play note 48 for 0.5 beats
```

```
2
when space key pressed
set instrument to 12
play note 65 for 0.5 beats
play note 60 for 0.5 beats
play note 69 for 0.5 beats
play note 71 for 0.5 beats
```

```
3
when l key pressed
stop all sounds
stop all
```

```
4
when up arrow key pressed
repeat 50
    play drum 39 for 0.2 beats
```

```
5
when a key pressed
forever
    repeat 10
        play sound recording1 until done
```

```
6
when m key pressed
forever
    if mouse y > 0
        set instrument to 20
        play note 61 for 0.5 beats
    else
        set instrument to 20
        play note 60 for 0.5 beats
```

```
7
when Sprite2 clicked
repeat 10
    set instrument to 74
    play note 64 for 0.5 beats
```

```
8
when Sprite2 clicked
repeat 10
    set instrument to 74
    play note 67 for 0.5 beats
```

Figure 6: Participant A’s Final Scratch Project

A computational thinking diagram of Participant A’s project, seen in Figure 7, primarily shows *user interactions* triggering single or looped instrumental sounds. The two programs associated with the second sprite (programs 7 and 8) are both activated by the same trigger, introducing *parallelism*. Program 6 uses a *conditional statement*, *operator*, and additional instance of *user interaction* to play a note depending on the position of the mouse.

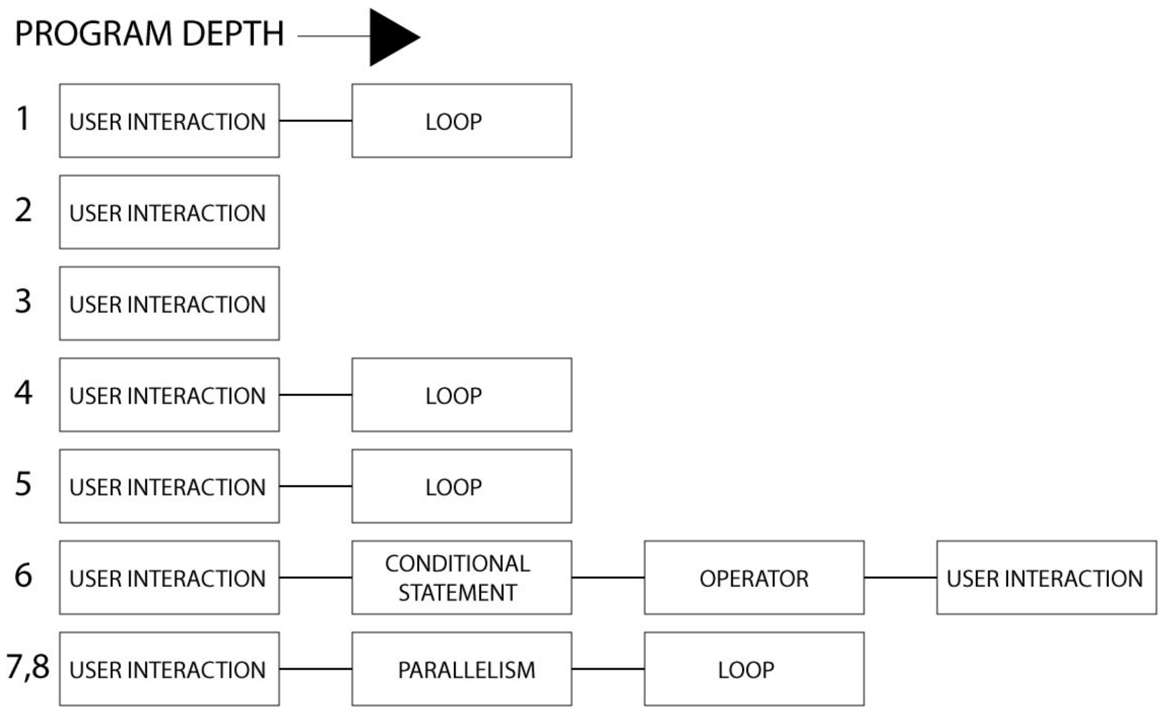


Figure 7: CT Diagram of Participant A's Final Scratch Project

Participant A's programing process, seen in Figure 8, begins with the addition of a series of primarily sound and control objects. This is followed by a period of addition and removal and then another of addition alone. Participant A is one of several participants to create a second sprite; this is followed by the relatively quick addition of two identical chunks. Participant A created three sounds, though only used one in her final project.

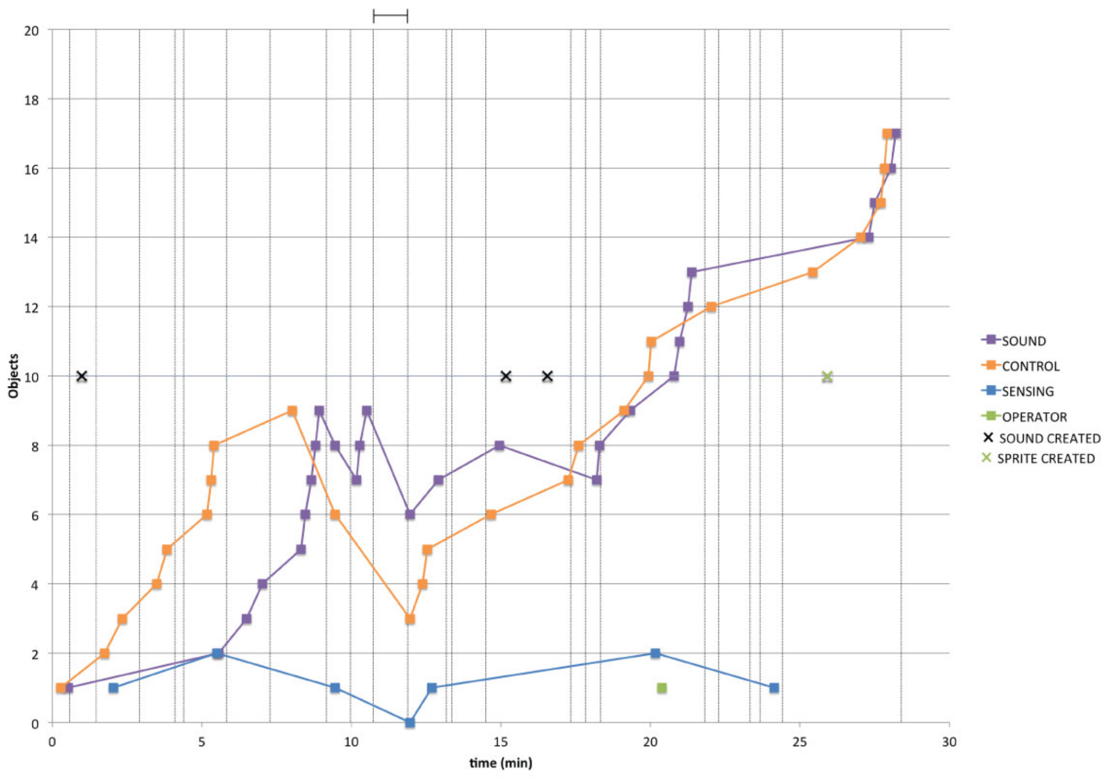
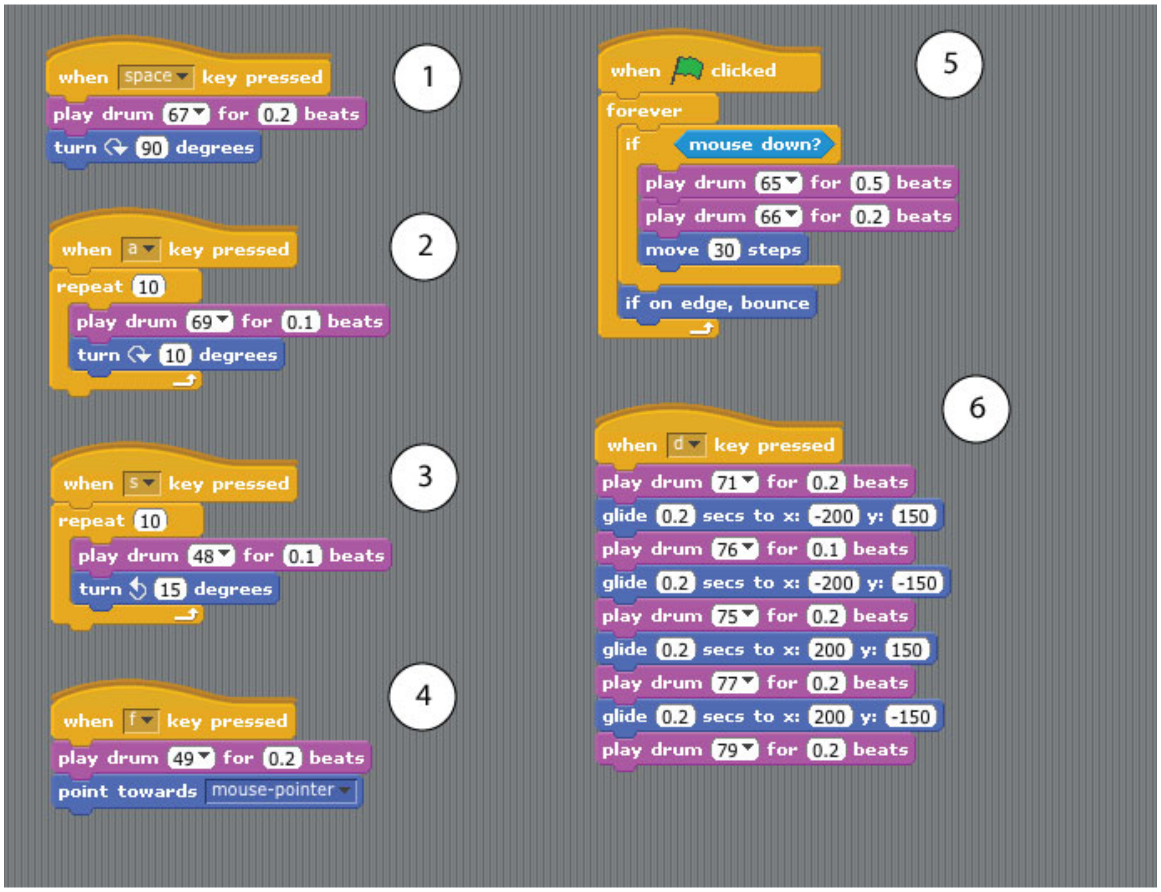


Figure 8: Timeline of Participant A's addition and removal of Scratch objects

Participant B

While several participants explored Scratch's motion objects, Participant B based her project in Figure 9 around them (perhaps unsurprisingly, this was one of the non-musician choreographer participants). Key presses were used to turn the sprite 90° instantaneously (1) or 100° in 10° increments (2, 3). These actions could effectively interfere with one another; activating programs 2 and 3 would cause the sprite to rapidly oscillate back and forth, for instance. The sprite could also be instantly oriented towards the mouse (4). All of these programs could also work in conjunction with a larger one that marched the sprite forward when the mouse was pressed and turned the sprite around if it reached the edge of the stage (5). The final program smoothly moved the sprite from its current position to each corner of the screen (6). Each program punctuated the sprite's motion with a percussion accent.



```
when [space] key pressed
play drum (67) for (0.2) beats
turn ↻ (90) degrees

when [a] key pressed
repeat (10)
    play drum (69) for (0.1) beats
    turn ↻ (10) degrees

when [s] key pressed
repeat (10)
    play drum (48) for (0.1) beats
    turn ↺ (15) degrees

when [f] key pressed
play drum (49) for (0.2) beats
point towards [mouse-pointer]

when ⚑ clicked
forever
    if <mouse down?>
        play drum (65) for (0.5) beats
        play drum (66) for (0.2) beats
        move (30) steps
    if on edge, bounce

when [d] key pressed
play drum (71) for (0.2) beats
glide (0.2) secs to x: (-200) y: (150)
play drum (76) for (0.1) beats
glide (0.2) secs to x: (-200) y: (-150)
play drum (75) for (0.2) beats
glide (0.2) secs to x: (200) y: (150)
play drum (77) for (0.2) beats
glide (0.2) secs to x: (200) y: (-150)
play drum (79) for (0.2) beats
```

Figure 9: Participant B's Final Scratch Project

Participant B's computational thinking diagram primarily shows *user interaction* elements triggering single and looped sounds and movement actions (similar to Participant A's triggering single and looped sounds). Also similar to Participant A, we see one chunk using a *conditional statement* to base output on the state of the mouse.

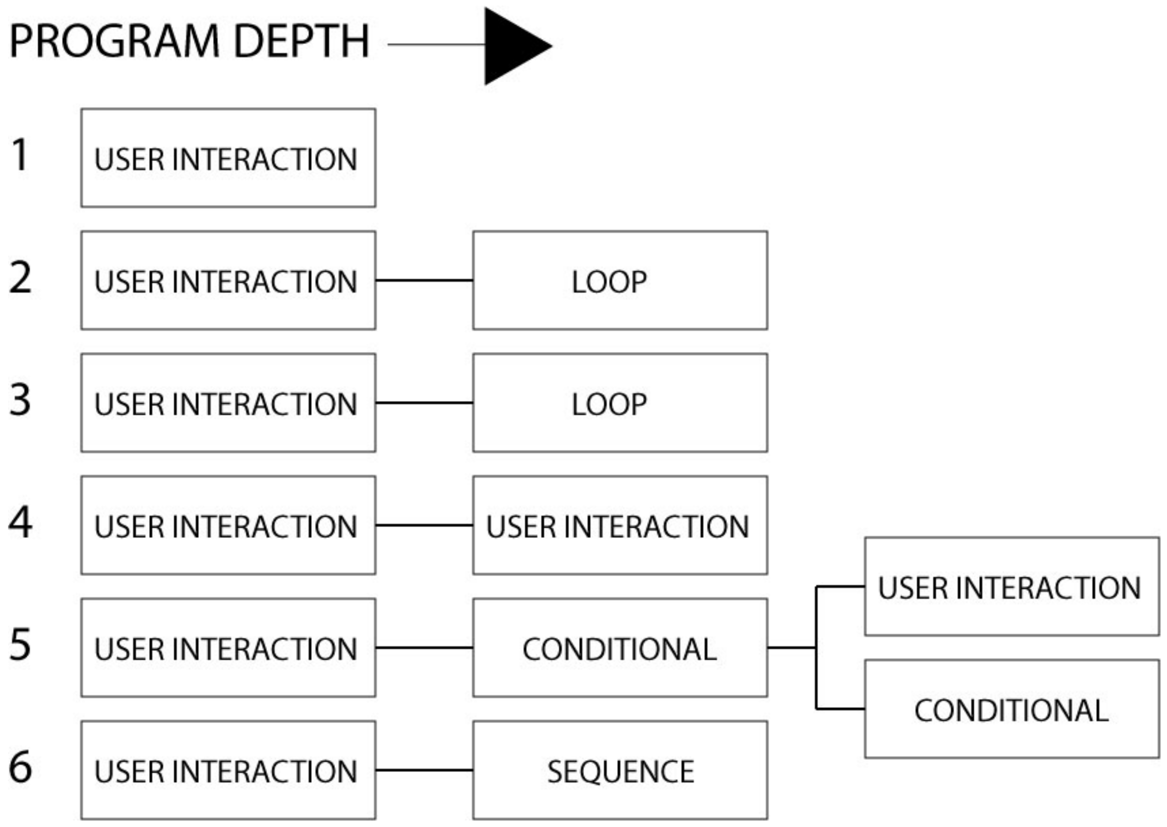


Figure 10: CT Diagram of Participant B's Final Scratch Project

Seen in Figure 11, Participant B steadily added control, sound, and motion objects with few removals.

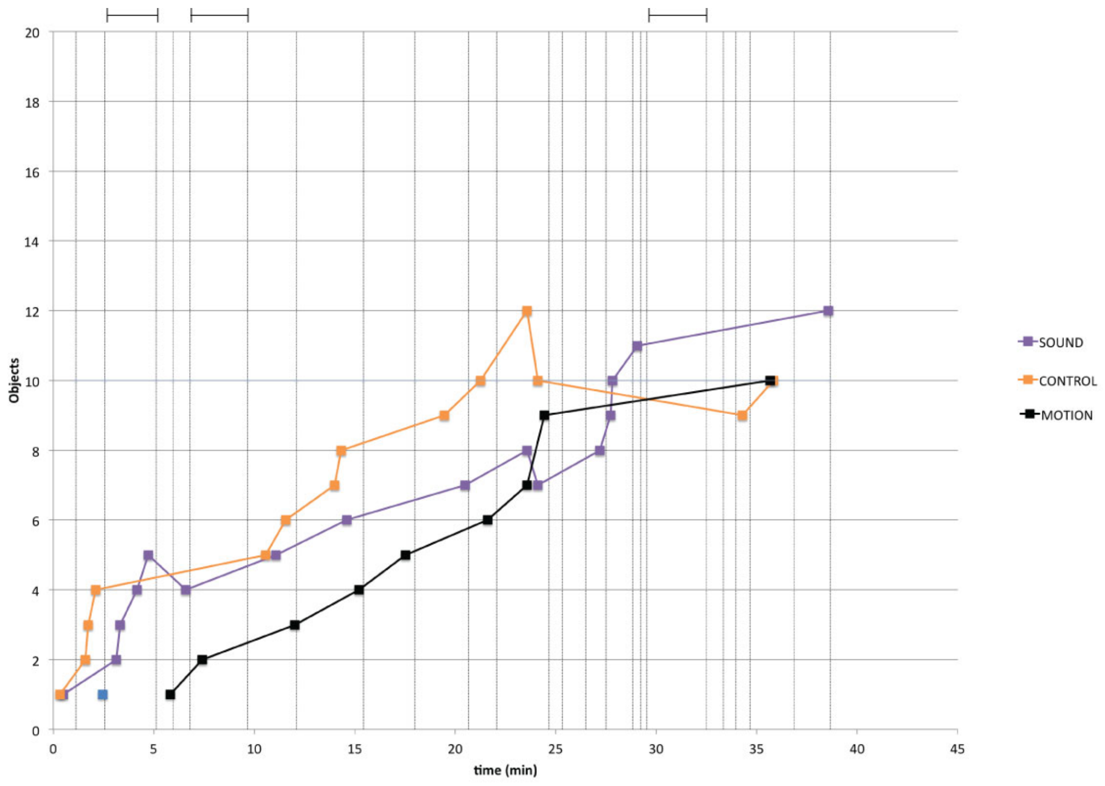
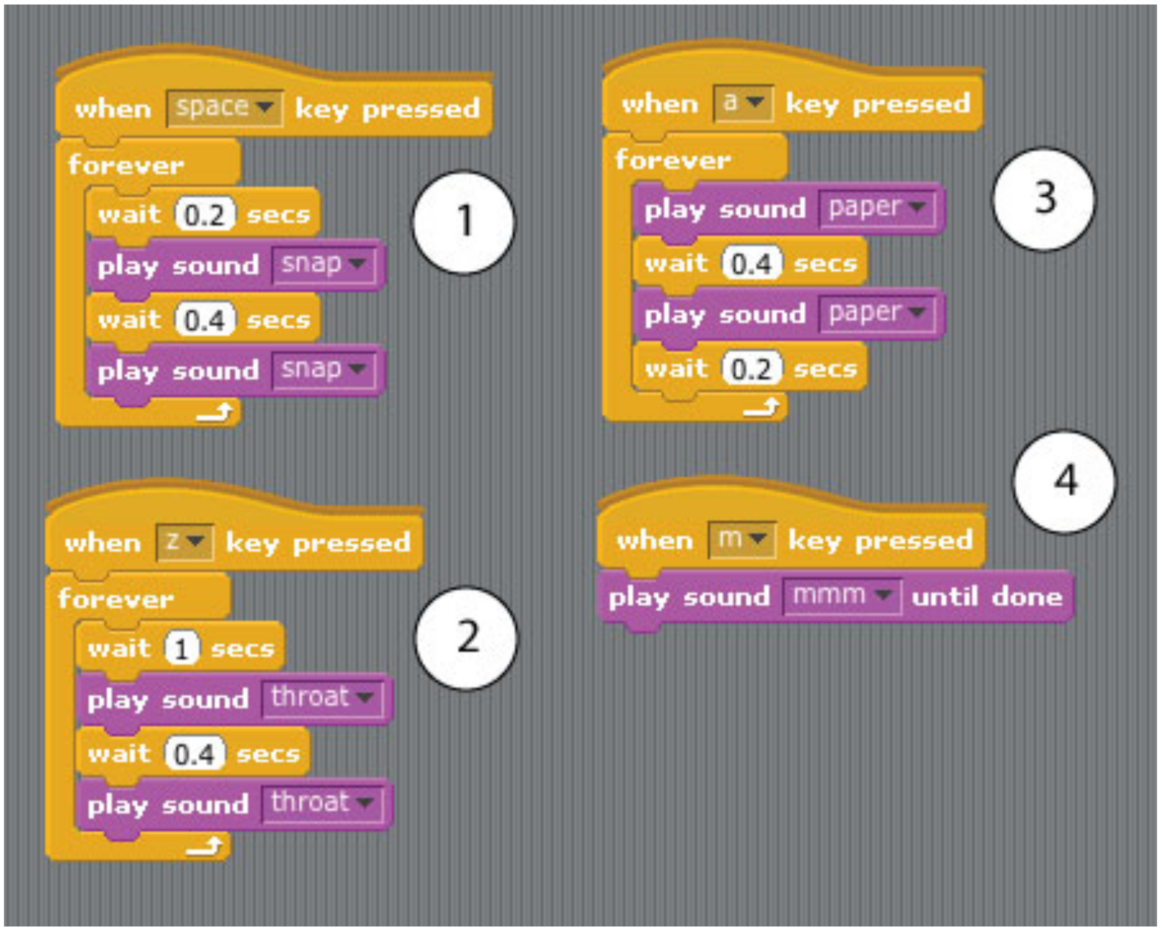


Figure 11: Timeline of Participant B's addition and removal of Scratch objects

Participant C

Seen in Figure 12, this project uses four key pressed to trigger different audio recordings (finger snaps, paper crunches, and throat clearing). The first three repeat forever and use `wait` objects to create rhythmic interplay between the samples. The fourth simply plays a humming recording a single time.



```
when space key pressed
forever
    wait 0.2 secs
    play sound snap
    wait 0.4 secs
    play sound snap
```

```
when z key pressed
forever
    wait 1 secs
    play sound throat
    wait 0.4 secs
    play sound throat
```

```
when a key pressed
forever
    play sound paper
    wait 0.4 secs
    play sound paper
    wait 0.2 secs
```

```
when m key pressed
play sound mmm until done
```

Figure 12: Participant C's Final Scratch Project

Participant C's project is relatively straightforward from the perspective of its computational thinking elements, seen in Figure 13. In all four programs, *user interaction* triggers an audio sample; three of the four programs use Scratch `wait` objects to introduce *sequences* to their outputs.

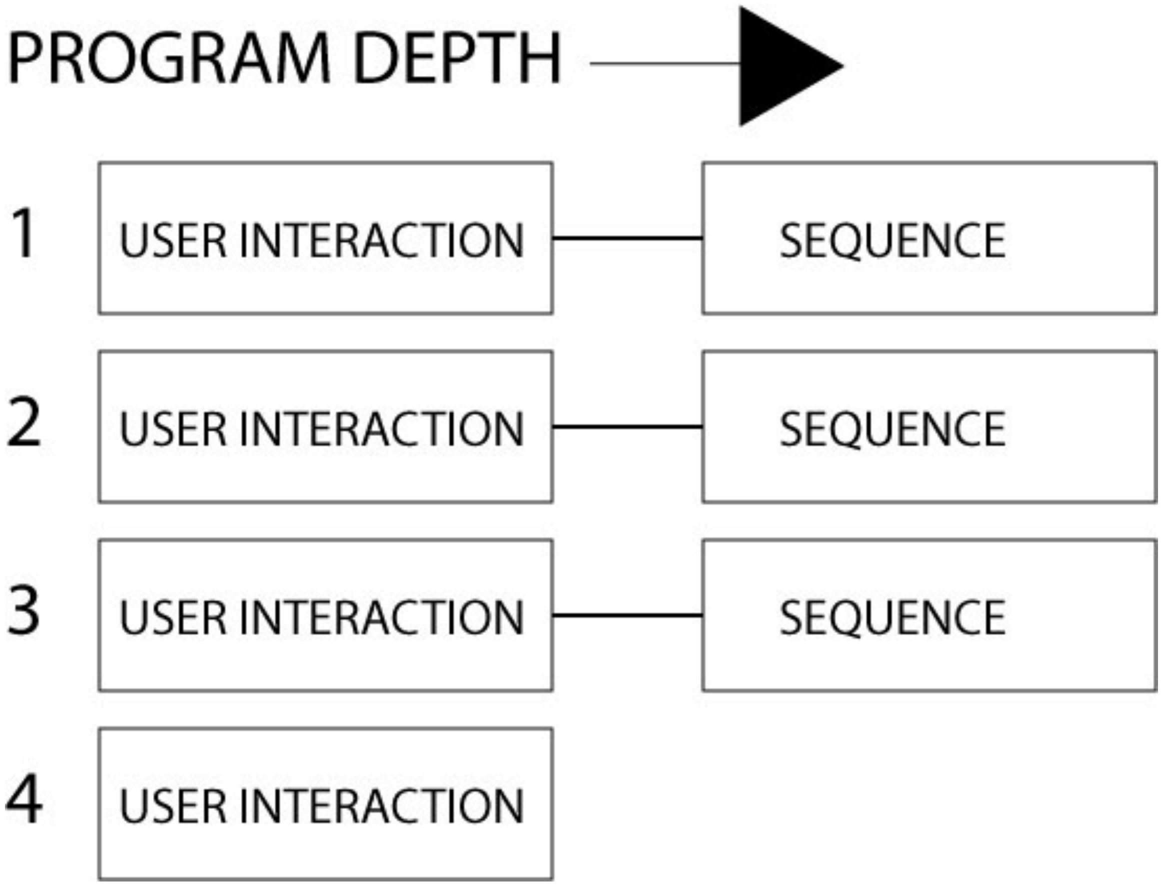


Figure 13: CT Diagram of Participant C’s Final Scratch Project

Seen in Figure 14, Participant C worked the longest of the non-musician participants. She removed more objects than the other non-musician participants, while still ending up with roughly as many (or more) Scratch objects. She created multiple sounds early in the programming process and used all of them as well as experimenting with (and discarding) a series of blue sensing objects. She utilized four extended program tests while adding Scratch objects and editing her project’s timing parameters.

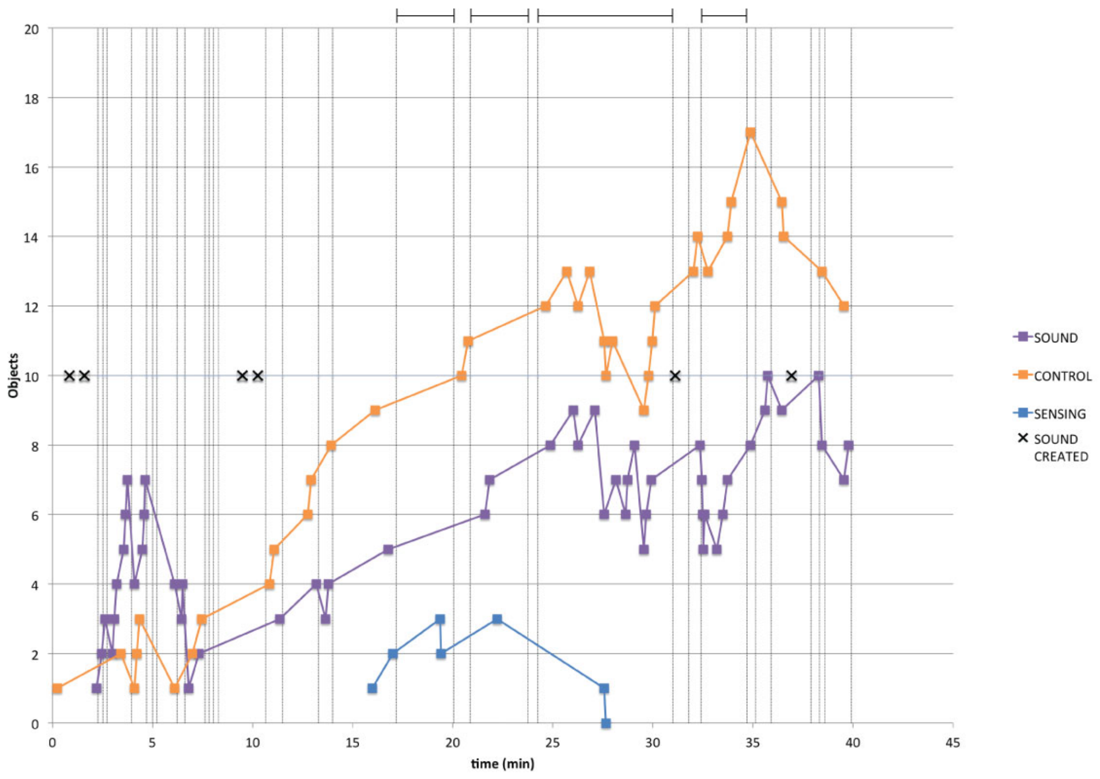
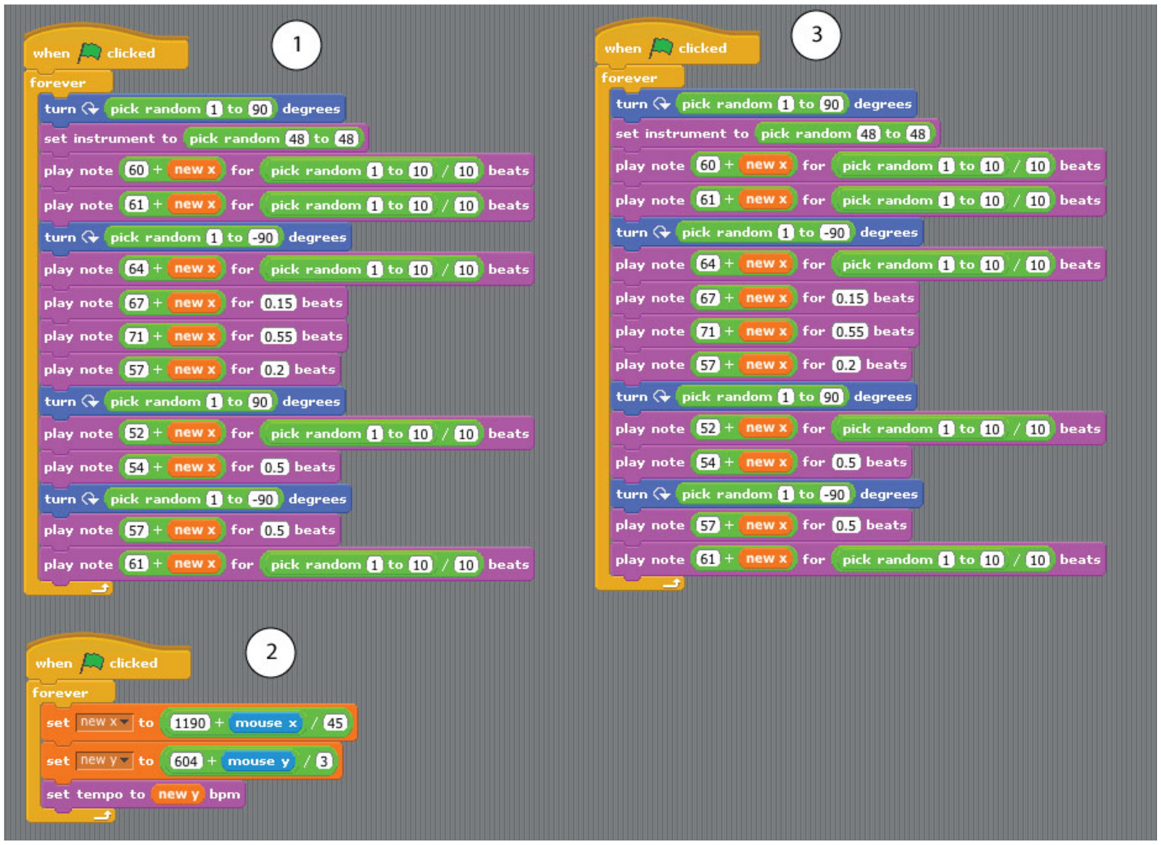


Figure 14: Timeline of Participant C's addition and removal of Scratch objects

JAZZ MUSICIANS

Participant D

Seen in Figure 15, this project creates many effects with little activation from the user. Each of its three programs are triggered by a flag click. One continually rescales mouse x and y positions from 0-127 and sets the tempo to the rescaled y variable (2). Another executes a lengthy series of commands: rotating the sprite a random amount and playing a sequence of notes, all modified by the new x variable and many for random durations (1); this program is identical to the final one (3), though the two different thanks to their random elements.



```
1
when clicked
forever
  turn (pick random 1 to 90) degrees
  set instrument to (pick random 48 to 48)
  play note (60 + new x) for ((pick random 1 to 10) / 10) beats
  play note (61 + new x) for ((pick random 1 to 10) / 10) beats
  turn (pick random 1 to -90) degrees
  play note (64 + new x) for ((pick random 1 to 10) / 10) beats
  play note (67 + new x) for 0.15 beats
  play note (71 + new x) for 0.55 beats
  play note (57 + new x) for 0.2 beats
  turn (pick random 1 to 90) degrees
  play note (52 + new x) for ((pick random 1 to 10) / 10) beats
  play note (54 + new x) for 0.5 beats
  turn (pick random 1 to -90) degrees
  play note (57 + new x) for 0.5 beats
  play note (61 + new x) for ((pick random 1 to 10) / 10) beats

2
when clicked
forever
  set new x to ((1190 + mouse x) / 45)
  set new y to ((604 + mouse y) / 3)
  set tempo to (new y) bpm

3
when clicked
forever
  turn (pick random 1 to 90) degrees
  set instrument to (pick random 48 to 48)
  play note (60 + new x) for ((pick random 1 to 10) / 10) beats
  play note (61 + new x) for ((pick random 1 to 10) / 10) beats
  turn (pick random 1 to -90) degrees
  play note (64 + new x) for ((pick random 1 to 10) / 10) beats
  play note (67 + new x) for 0.15 beats
  play note (71 + new x) for 0.55 beats
  play note (57 + new x) for 0.2 beats
  turn (pick random 1 to 90) degrees
  play note (52 + new x) for ((pick random 1 to 10) / 10) beats
  play note (54 + new x) for 0.5 beats
  turn (pick random 1 to -90) degrees
  play note (57 + new x) for 0.5 beats
  play note (61 + new x) for ((pick random 1 to 10) / 10) beats
```

Figure 15: Participant D's Final Scratch Project

Participant D's computational thinking diagram, seen in Figure 16, is almost entirely buried within *user interaction* and *parallelism* elements; it is activated by a single key press. The project utilizes many *operator*, *random number*, and *variable* elements, many nested within each other. *User interaction* (in the form of mouse position) is nested within *operators* to change two of the variables.

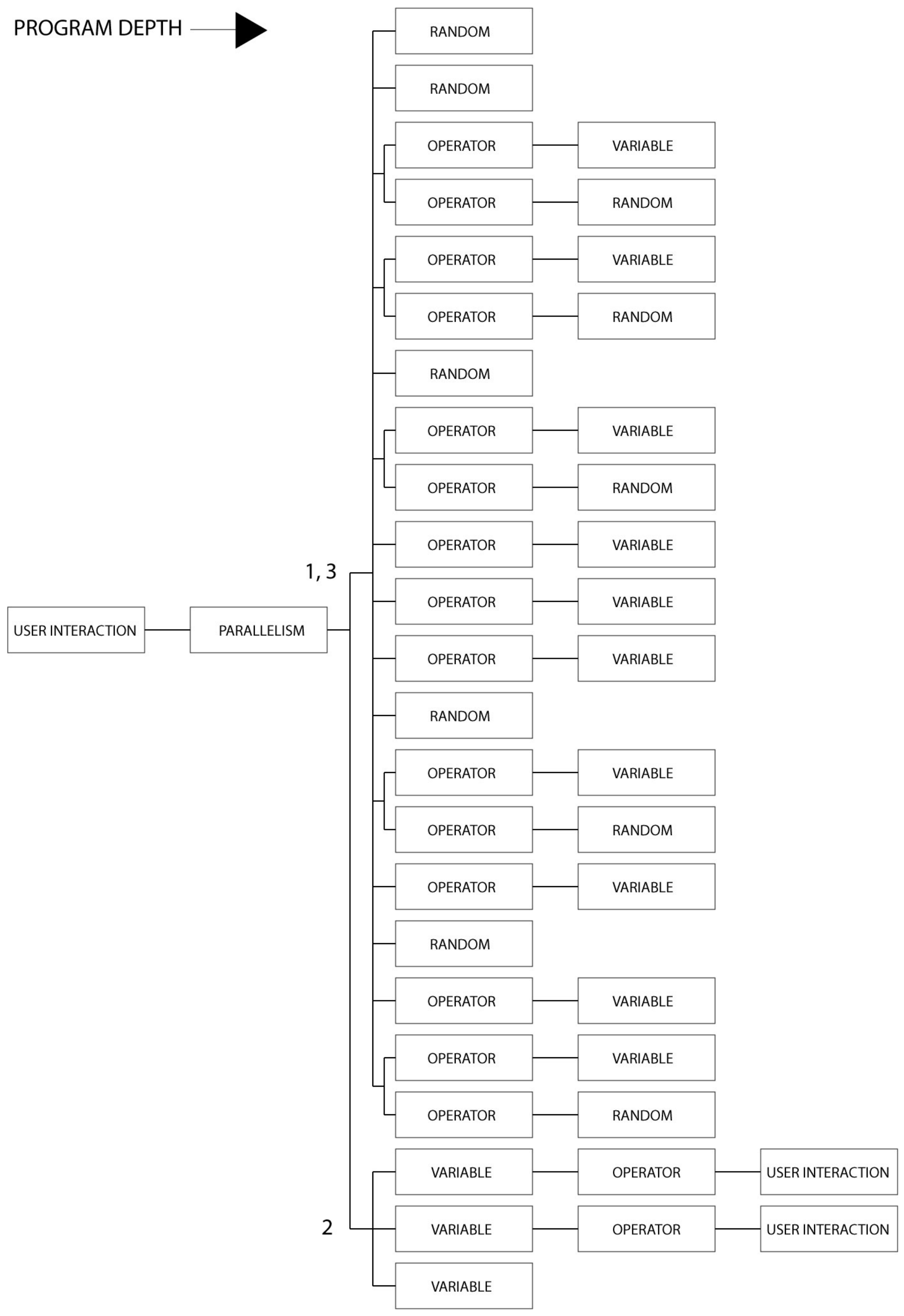


Figure 16: CT Diagram of Participant D’s Final Scratch Project

Seen in Figure 17, Participant D added many Scratch objects of many types. In some cases the participant copied large chunks of code; in others he added many variable or operator objects extremely quickly. Participant D removed very few objects relative to additions. He experimented with adding several motion objects near the end of his programming time.

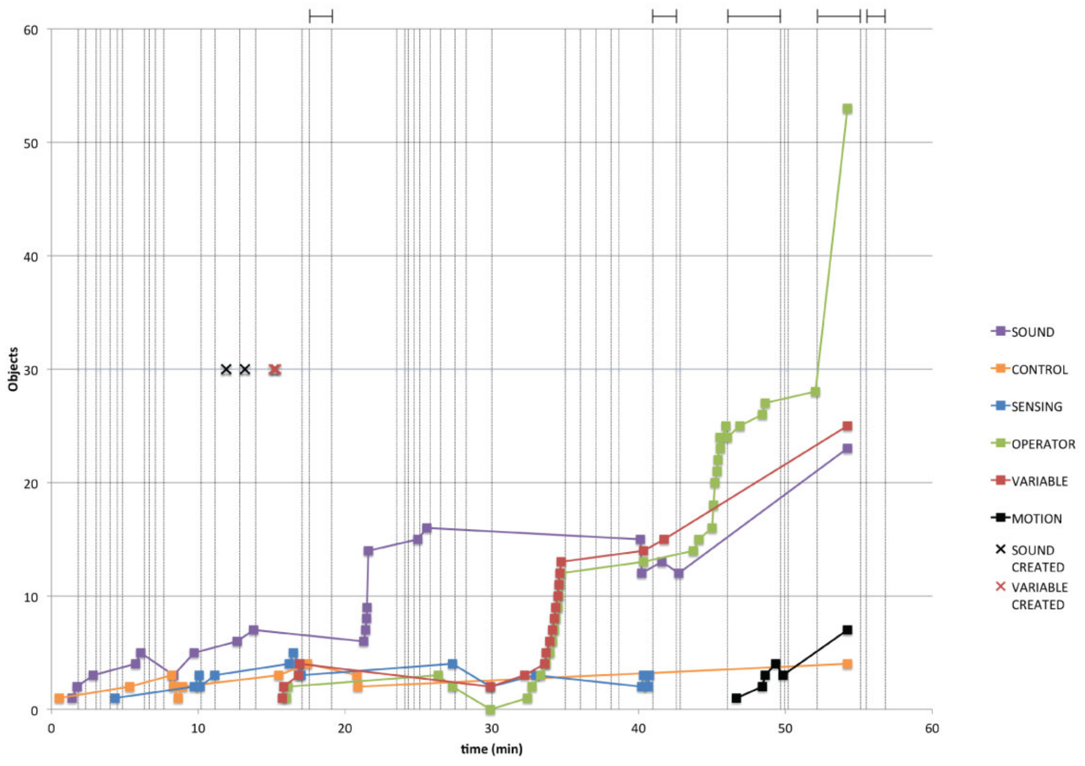
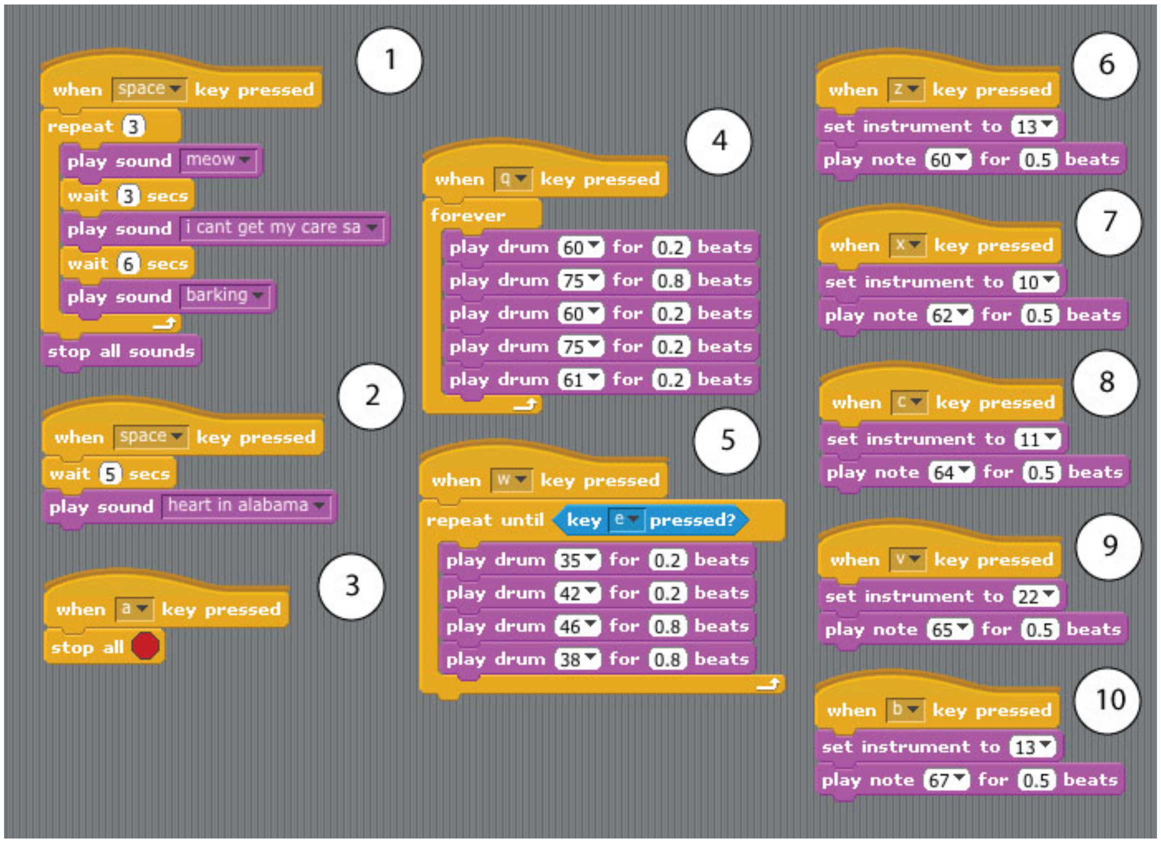


Figure 17: Timeline of Participant D’s addition and removal of Scratch objects

Participant E

This project, seen in Figure 18, performs a number of different tasks. The first program is triggered by the space bar and plays three different audio recordings in sequence (1). This repeats three times and then all sounds are stopped. The space key also triggers a fourth audio recording after a 5-second pause (2). Another key is used to stop all program operations (3). Programs 4 and 5 each repeat drum sequences forever, triggered by the q and w keys, respectively (5 is stopped by the e key, though the key must be held during the program's return to its beginning). The final five programs operate like a musical keyboard, each using a letter key to set an instrument and play a single note (6-10).



```
1
when space key pressed
repeat 3
    play sound meow
    wait 3 secs
    play sound i cant get my care sa
    wait 6 secs
    play sound barking
stop all sounds

2
when space key pressed
wait 5 secs
play sound heart in alabama

3
when a key pressed
stop all

4
when q key pressed
forever
    play drum 60 for 0.2 beats
    play drum 75 for 0.8 beats
    play drum 60 for 0.2 beats
    play drum 75 for 0.2 beats
    play drum 61 for 0.2 beats

5
when w key pressed
repeat until key e pressed?
    play drum 35 for 0.2 beats
    play drum 42 for 0.2 beats
    play drum 46 for 0.8 beats
    play drum 38 for 0.8 beats

6
when z key pressed
set instrument to 13
play note 60 for 0.5 beats

7
when x key pressed
set instrument to 10
play note 62 for 0.5 beats

8
when c key pressed
set instrument to 11
play note 64 for 0.5 beats

9
when v key pressed
set instrument to 22
play note 65 for 0.5 beats

10
when b key pressed
set instrument to 13
play note 67 for 0.5 beats
```

Figure 18: Participant E's Final Scratch Project

Participant E's computational thinking diagram, seen in Figure 19, primarily involves *user interaction* elements triggering an action or series of actions. Participant E also employs a pair of *sequences* triggered in *parallel* and a *loop* controlled by *user interaction*.

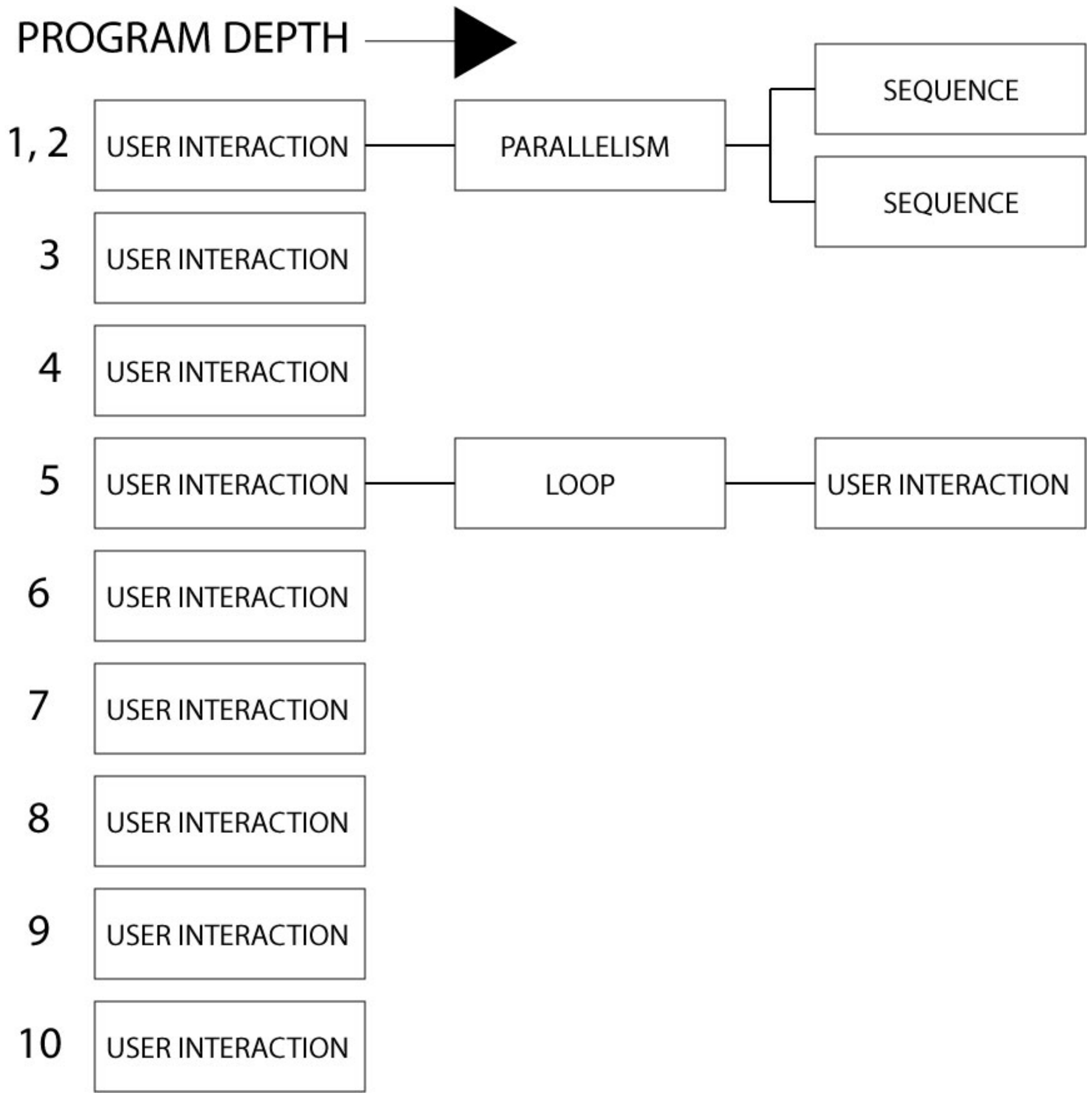


Figure 19: CT Diagram of Participant E's Final Scratch Project

In Figure 20, Participant E steadily adds sound and control objects with relatively few removals.

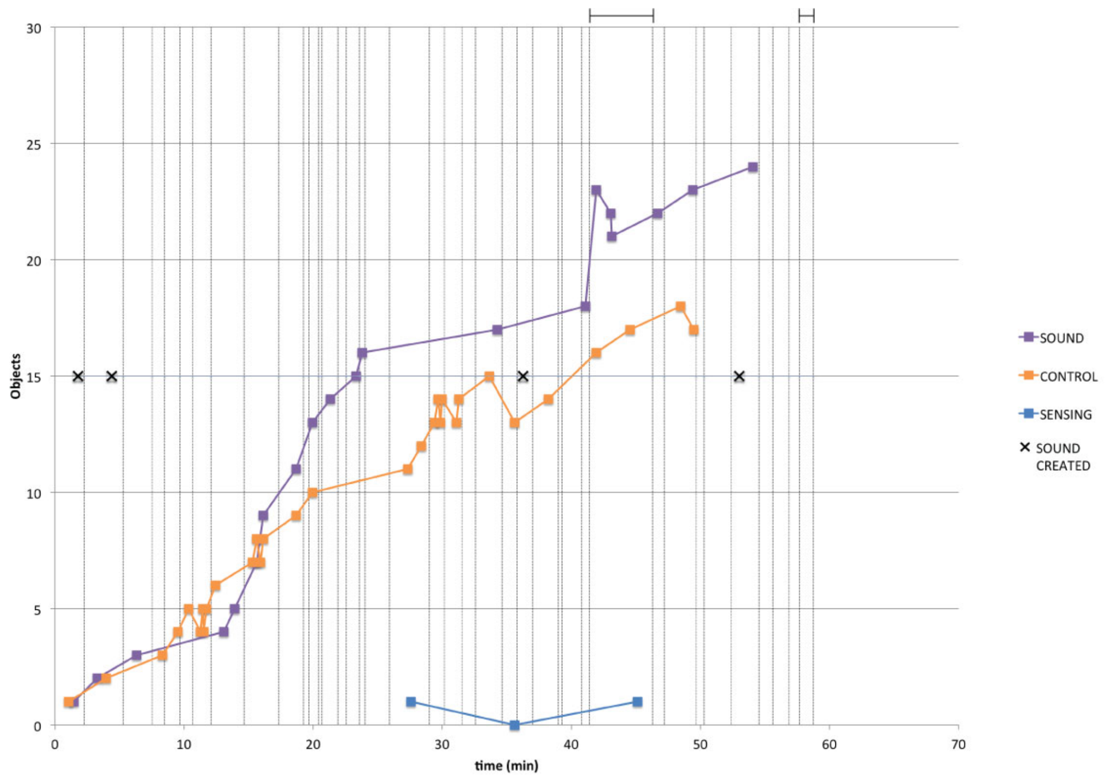
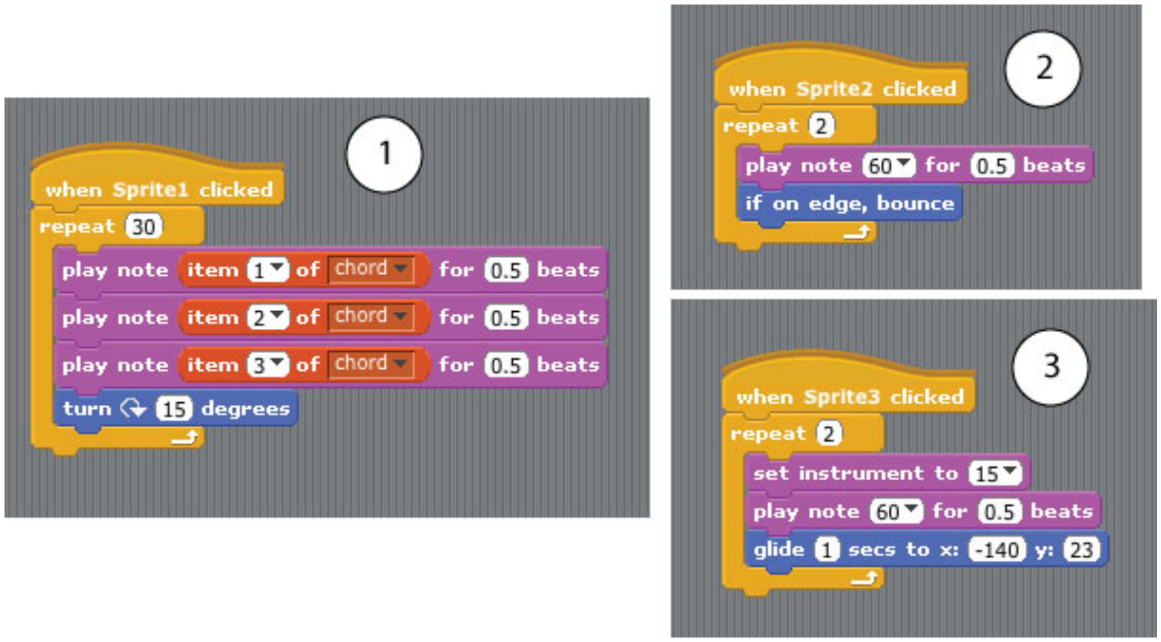


Figure 20: Timeline of Participant E's addition and removal of Scratch objects

Participant F

This project, seen in Figure 21, uses three discrete programs spread across three sprites. All programs are triggered by clicking their associated sprites. One plays three notes from a list and rotates its sprite 15 degrees (1). Another plays a single note twice (2). (The `if on edge, bounce` object is basically redundant as the sprite does not otherwise move). The last sets the sprite's instrument, plays a note, and smoothly moves the sprite to a particular point on the screen in 1 second (3). This sequence repeats, though only the note does so meaningfully.



```
when Sprite1 clicked
repeat 30
    play note item 1 of chord for 0.5 beats
    play note item 2 of chord for 0.5 beats
    play note item 3 of chord for 0.5 beats
    turn ↻ 15 degrees
```

```
when Sprite2 clicked
repeat 2
    play note 60 for 0.5 beats
    if on edge, bounce
```

```
when Sprite3 clicked
repeat 2
    set instrument to 15
    play note 60 for 0.5 beats
    glide 1 secs to x: -140 y: 23
```

Figure 21: Participant F's Final Scratch Project

The computational thinking diagram for Participant F, seen in Figure 22, shows a pattern of *loops* activated by *user interaction*. Nested within each loop is a *list*, *conditional statement*, or *sequence* element.

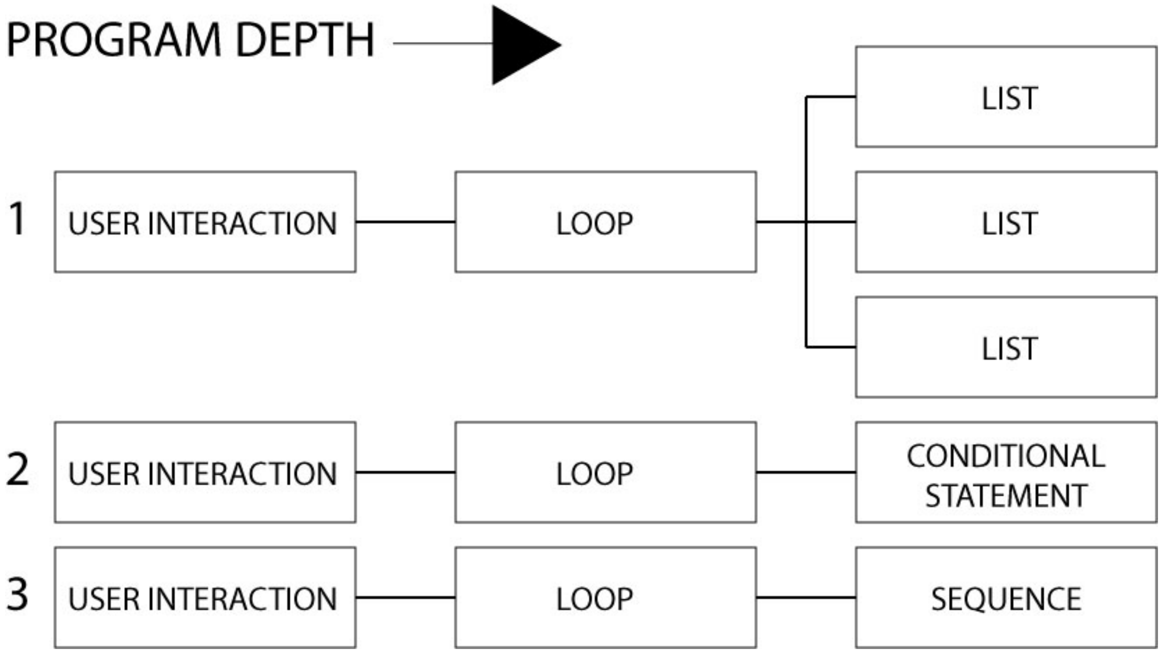


Figure 22: CT Diagram of Participant F's Final Scratch Project

As Figure 23 illustrates, despite adding the fewest overall objects Participant F added as wide a variety of Scratch objects as any participant. Participant F took a lengthy break to consider his work before adding motion objects near the very end of his programming.

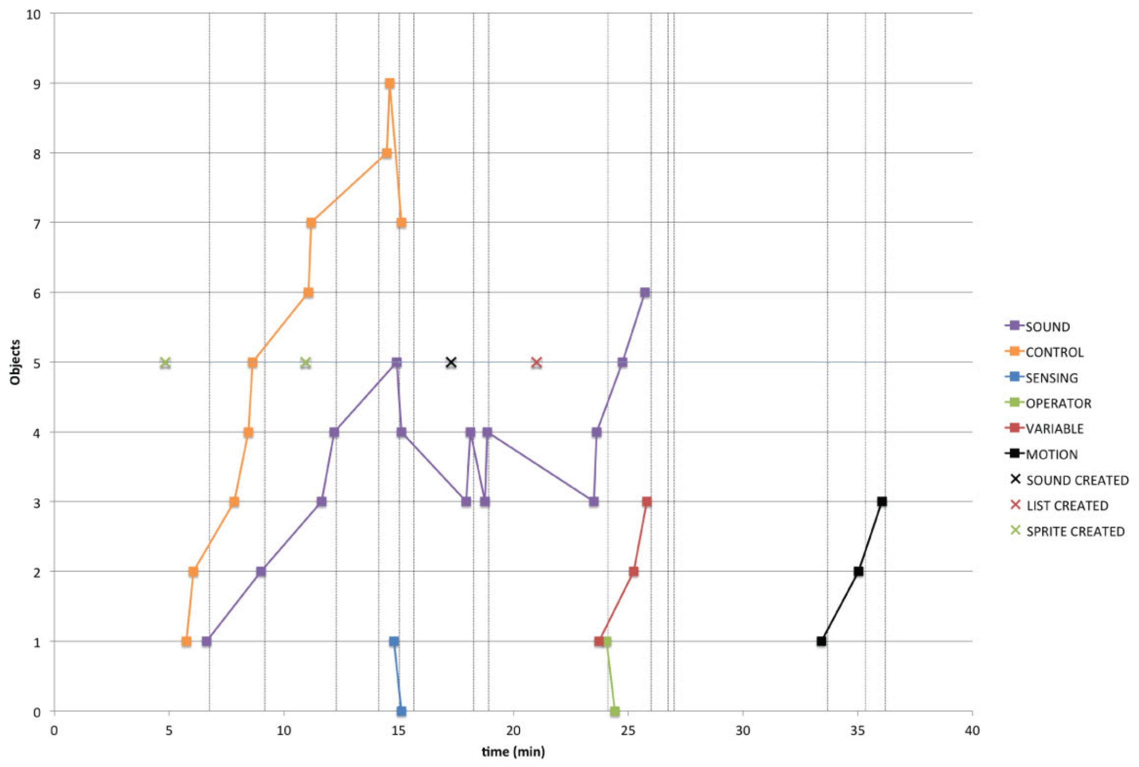
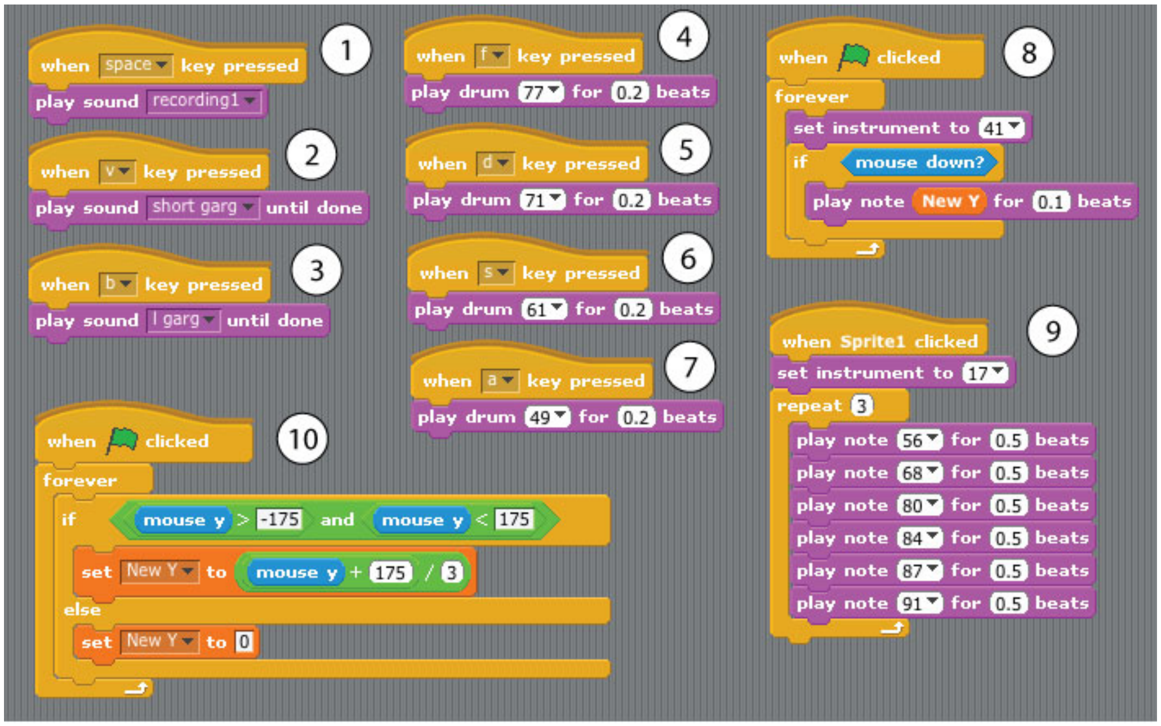


Figure 23: Timeline of Participant F's addition and removal of Scratch objects

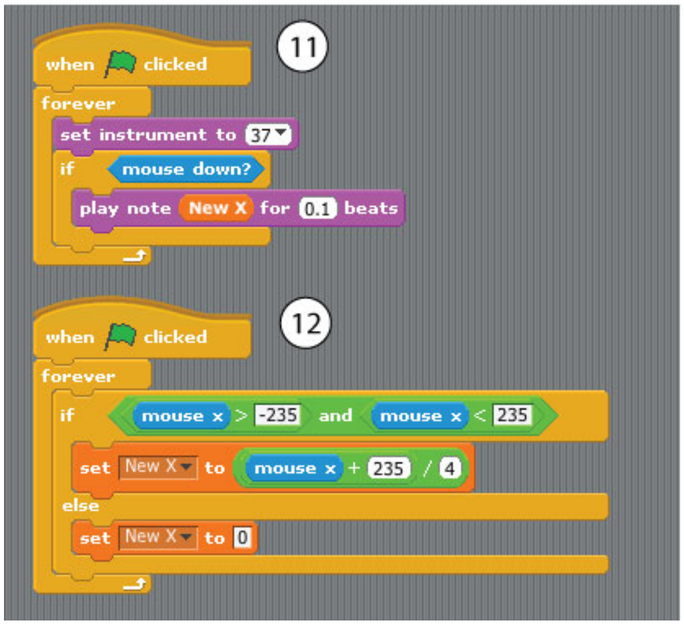
Classical Musicians

Participant G

This project's 11 programs are spread across two sprites, though the second sprite is not used for any interactivity purposes. Three use key presses to trigger audio recordings (1-3). Another three use key pressed to play particular drum sounds (4-7). One is activated by the green flag and continually sets a variable: if mouse y is within the Scratch window, the new variable is rescaled from 0-125, otherwise zero (10). Another, also activated by the green flag, plays a note equal to the variable when the mouse is pressed (8). The prior two programs are replicated for a second sprite, using a new variable (11, 12). Clicking Sprite 1 triggers a series of notes (9).



```
1
when space key pressed
play sound recording1

2
when v key pressed
play sound short garg until done

3
when b key pressed
play sound l garg until done

4
when f key pressed
play drum 77 for 0.2 beats

5
when d key pressed
play drum 71 for 0.2 beats

6
when s key pressed
play drum 61 for 0.2 beats

7
when a key pressed
play drum 49 for 0.2 beats

8
when clicked
forever
    set instrument to 41
    if mouse down?
        play note New Y for 0.1 beats

9
when Sprite1 clicked
set instrument to 17
repeat 3
    play note 56 for 0.5 beats
    play note 68 for 0.5 beats
    play note 80 for 0.5 beats
    play note 84 for 0.5 beats
    play note 87 for 0.5 beats
    play note 91 for 0.5 beats

10
when clicked
forever
    if mouse y > -175 and mouse y < 175
        set New Y to mouse y + 175 / 3
    else
        set New Y to 0
```



```
11
when clicked
forever
    set instrument to 37
    if mouse down?
        play note New X for 0.1 beats

12
when clicked
forever
    if mouse x > -235 and mouse x < 235
        set New X to mouse x + 235 / 4
    else
        set New X to 0
```

Figure 24: Participant G's Final Scratch Project

In Participant G's computational thinking diagram, seen in Figure 25, we see many *user interaction* elements triggering single actions (and one *loop*). Meanwhile, a single case of *user interaction* also activates two pairs of connected code; this utilizes *conditional statements*, *variables*, and *operators*, all based on more *user interaction*.

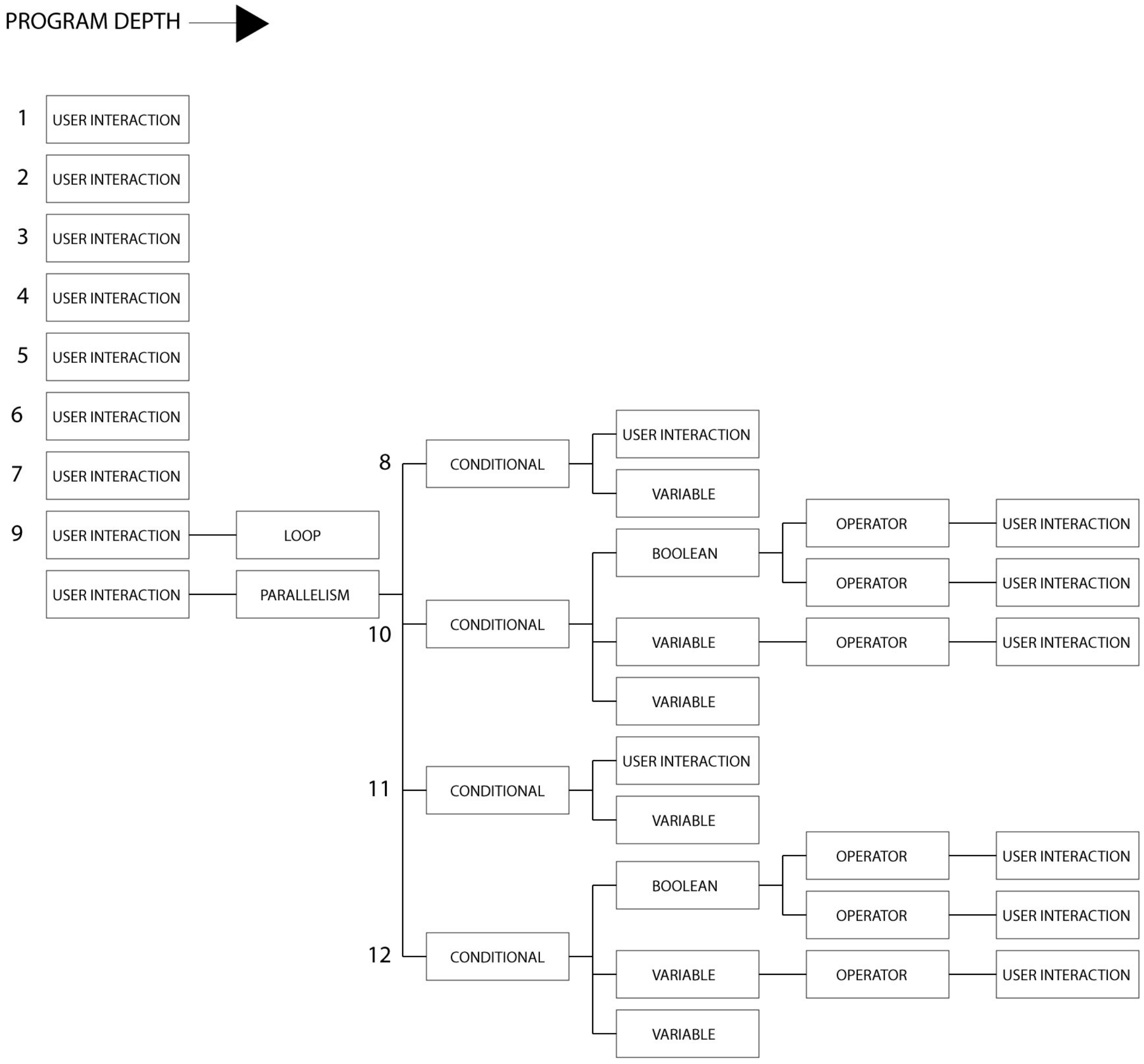


Figure 25: CT Diagram of Participant G’s Final Scratch Project

Seen in Figure 26, Participant G adds a wide variety of Scratch objects, primarily focusing on *control* and *sound* objects. He tests his program relatively seldom and removes the final 7-8 minutes of work before concluding his program.

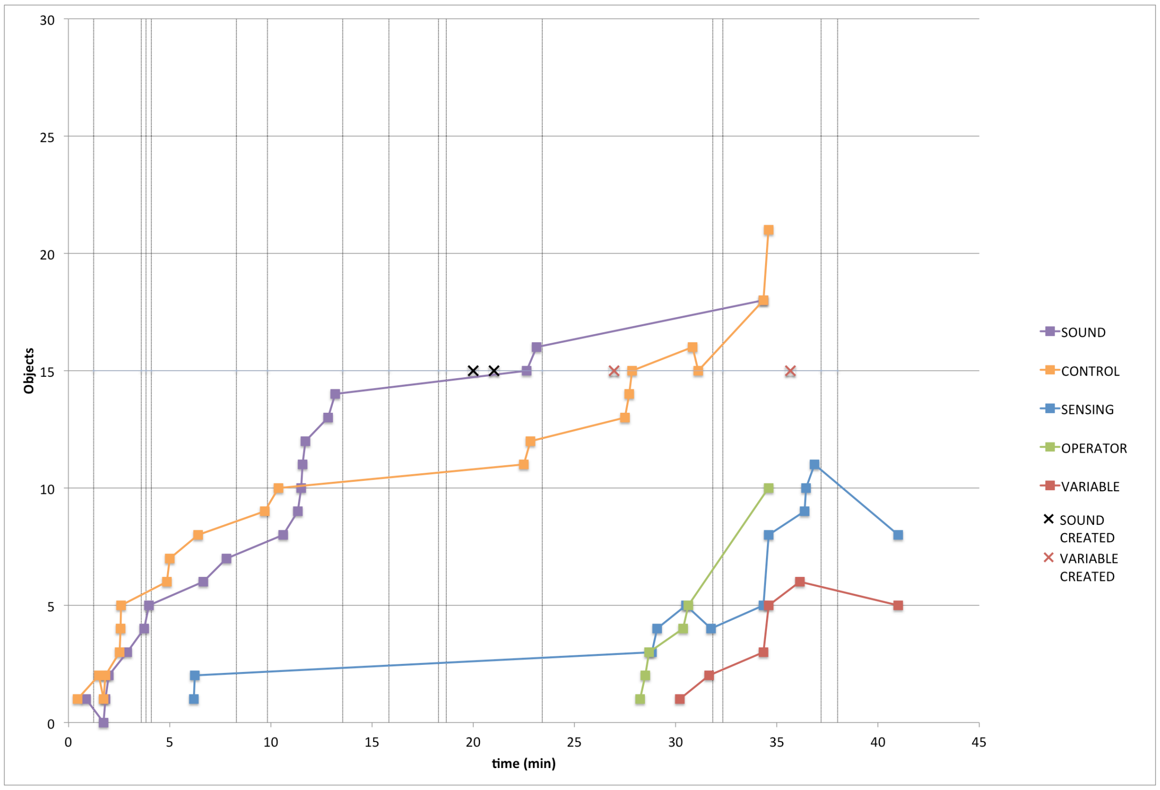
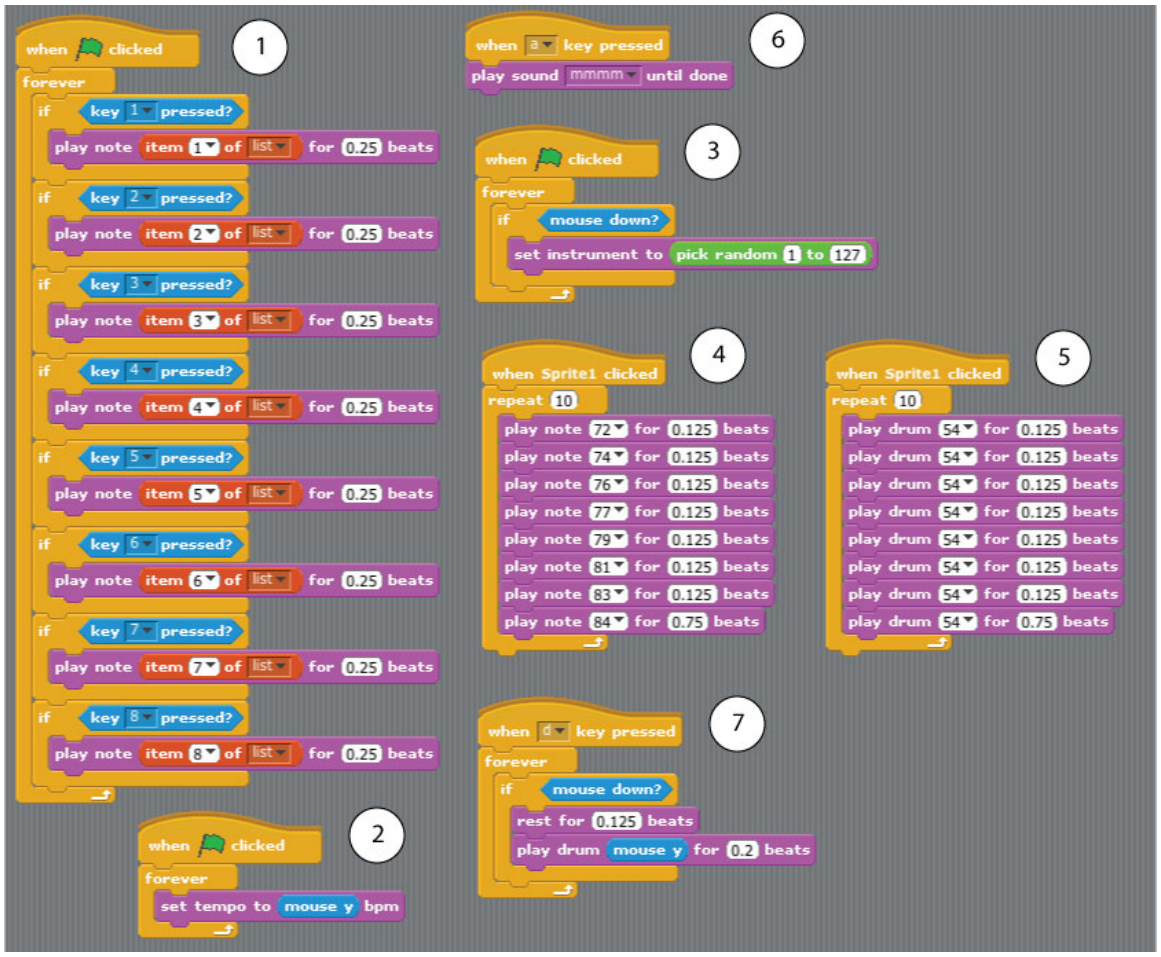


Figure 26: Timeline of Participant G's addition and removal of Scratch objects

Participant H

This project performs a number of different functions. Program 1 creates a musical keyboard using the computer keyboard's numerical keys. The project uses a list to index the notes of a major scale. Clicking the sprite plays a major scale and series of drum hits 10 times (4, 5). Pressing the mouse sets the tempo to the mouse's y value (2) and the instrument selection to a random number (3), these changes dramatically impacting programs 1, 4, and 5. Pressing the a key triggers an audio recording (6) and pressing the d key and mouse simultaneously plays a drum sound equal to the mouse's y value (7). Based on discussion and observation, these final two programs seemed largely extraneous to the participant's final project.



```
1
when ⚑ clicked
forever
  if key 1 pressed?
    play note item 1 of list for 0.25 beats
  if key 2 pressed?
    play note item 2 of list for 0.25 beats
  if key 3 pressed?
    play note item 3 of list for 0.25 beats
  if key 4 pressed?
    play note item 4 of list for 0.25 beats
  if key 5 pressed?
    play note item 5 of list for 0.25 beats
  if key 6 pressed?
    play note item 6 of list for 0.25 beats
  if key 7 pressed?
    play note item 7 of list for 0.25 beats
  if key 8 pressed?
    play note item 8 of list for 0.25 beats

2
when ⚑ clicked
forever
  set tempo to mouse y bpm

3
when ⚑ clicked
forever
  if mouse down?
    set instrument to pick random 1 to 127

4
when Sprite1 clicked
repeat 10
  play note 72 for 0.125 beats
  play note 74 for 0.125 beats
  play note 76 for 0.125 beats
  play note 77 for 0.125 beats
  play note 79 for 0.125 beats
  play note 81 for 0.125 beats
  play note 83 for 0.125 beats
  play note 84 for 0.75 beats

5
when Sprite1 clicked
repeat 10
  play drum 54 for 0.125 beats
  play drum 54 for 0.125 beats
  play drum 54 for 0.125 beats
  play drum 54 for 0.125 beats
  play drum 54 for 0.125 beats
  play drum 54 for 0.125 beats
  play drum 54 for 0.125 beats
  play drum 54 for 0.75 beats

6
when a key pressed
play sound mmmm until done

7
when d key pressed
forever
  if mouse down?
    rest for 0.125 beats
    play drum mouse y for 0.2 beats
```

Figure 27: Participant H's Final Scratch Project

Participant H's computational thinking diagram, seen in Figure 28, shows four *user interaction* elements controlling seven discrete programs. The participant uses a combination of *conditional statements*, *lists*, and *user interaction* elements to create a musical keyboard.

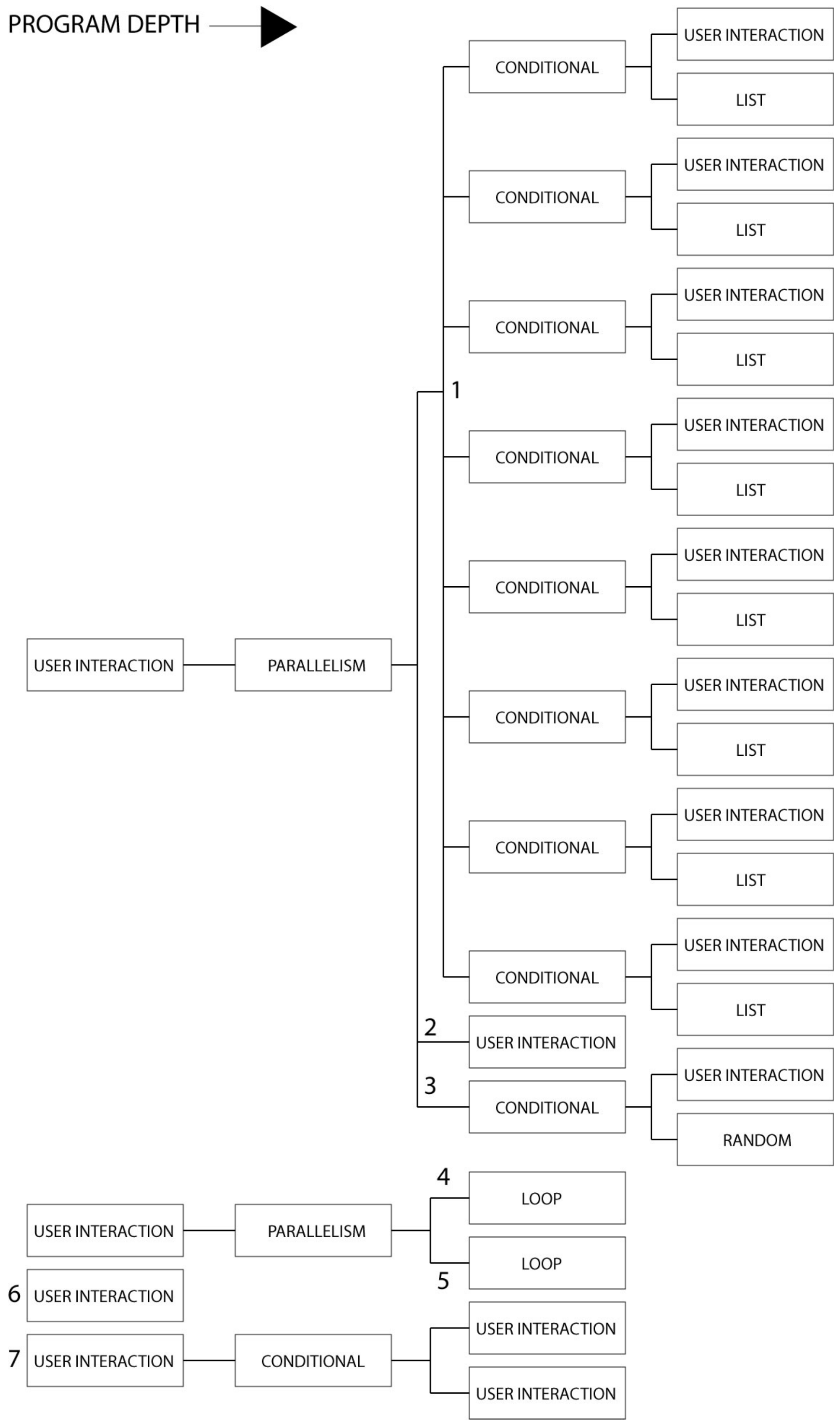


Figure 28: CT Diagram of Participant H’s Final Scratch Project

In Figure 29 we see Participant H steadily adding objects to his Scratch project. Between 30 and 40 minutes, he takes an extended break from testing to add several sensing and sound objects. He introduces several operator objects before removing them all with the exception of a single random number objects.

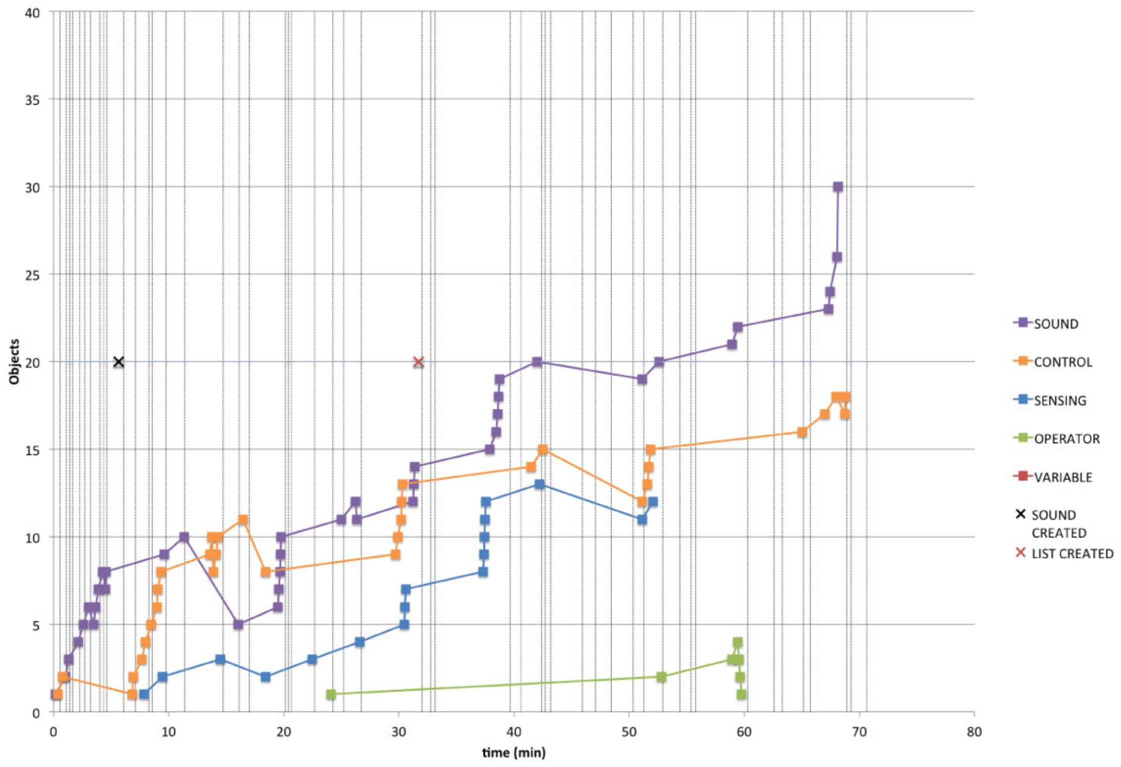
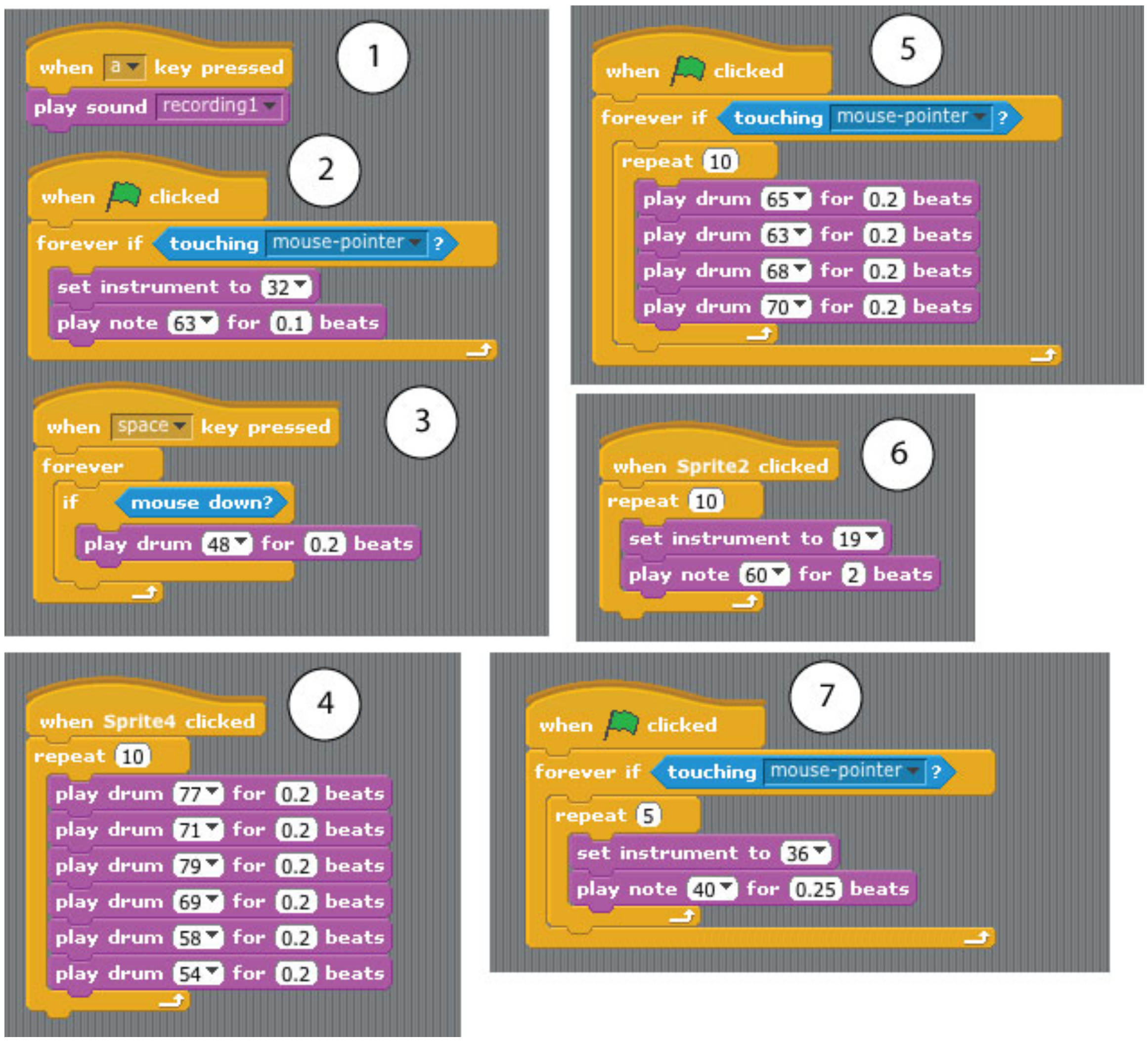


Figure 29: Timeline of Participant H's addition and removal of Scratch objects

Participant I

This program uses seven discrete programs spread across four sprites. Program 1 uses the a key to play an audio recording; program 2, triggered by the green flag, sets an instrument and plays a note whenever the mouse and sprite are touching; program 3, triggered by the space bar, plays a drum whenever the mouse is held down. Program 4 plays a drum sequence 10 times when its sprite is clicked. Program 5, triggered by the green flag, repeats a drum sequence 10 times when the mouse is touching its sprite. Program 6 sets its instrument and play a note, repeated 10 times, when its sprite is clicked. Program 7, like chunks 2 and 5, is triggered by the green flag and repeats a note when the mouse and sprite are touching.



```
1
when a key pressed
play sound recording1
```

```
2
when clicked
forever if touching mouse-pointer ?
    set instrument to 32
    play note 63 for 0.1 beats
```

```
3
when space key pressed
forever
    if mouse down?
        play drum 48 for 0.2 beats
```

```
4
when Sprite4 clicked
repeat 10
    play drum 77 for 0.2 beats
    play drum 71 for 0.2 beats
    play drum 79 for 0.2 beats
    play drum 69 for 0.2 beats
    play drum 58 for 0.2 beats
    play drum 54 for 0.2 beats
```

```
5
when clicked
forever if touching mouse-pointer ?
    repeat 10
        play drum 65 for 0.2 beats
        play drum 63 for 0.2 beats
        play drum 68 for 0.2 beats
        play drum 70 for 0.2 beats
```

```
6
when Sprite2 clicked
repeat 10
    set instrument to 19
    play note 60 for 2 beats
```

```
7
when clicked
forever if touching mouse-pointer ?
    repeat 5
        set instrument to 36
        play note 40 for 0.25 beats
```

Figure 30: Participant I’s Final Scratch Project

Participant I's computational thinking diagram, seen in Figure 31, shows a series of *user interaction* elements triggering *conditional statements* and *loops*. The *conditional statements* contain additional *user interaction* and *loop* elements.

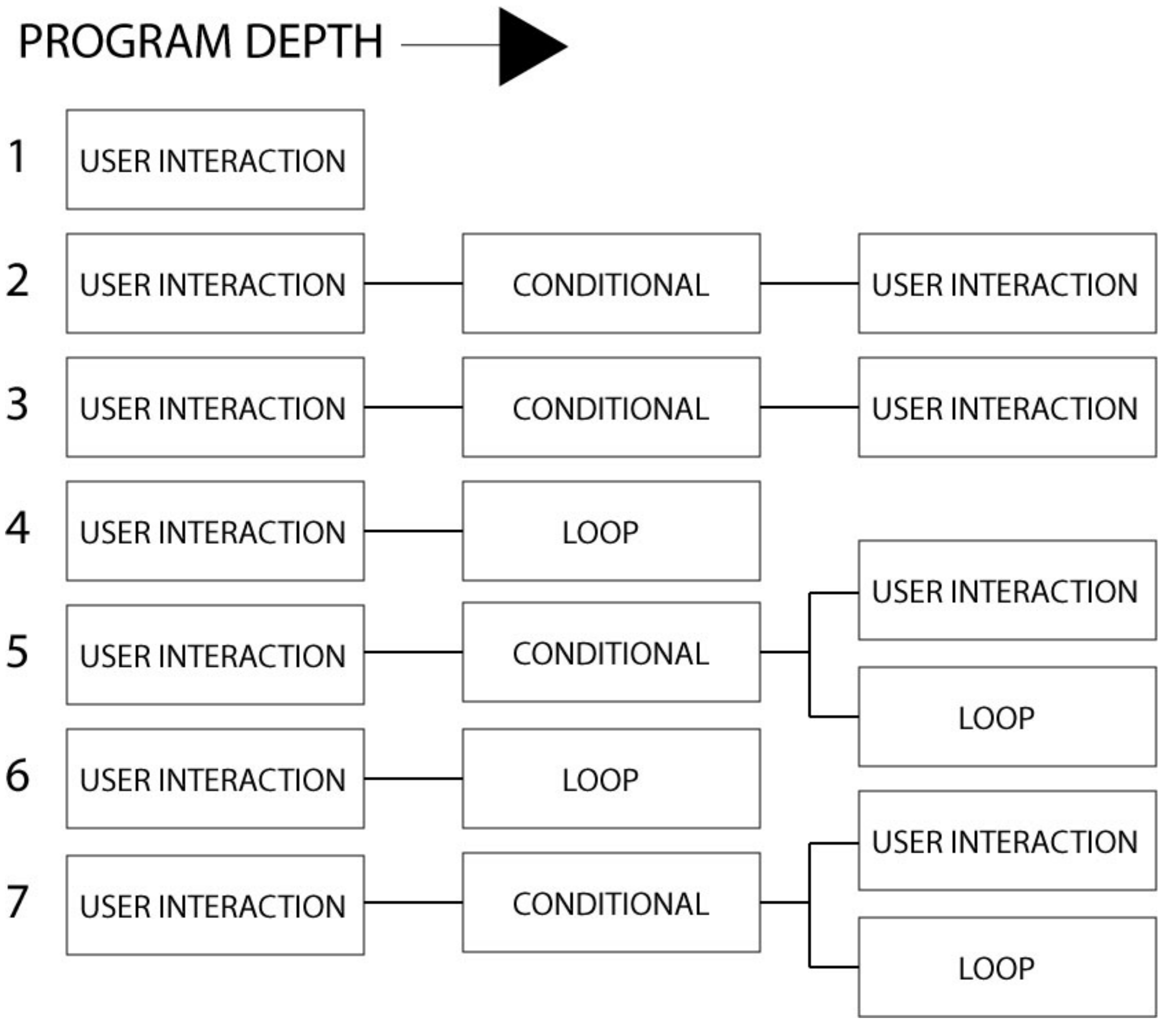


Figure 31: CT Diagram of Participant I's Final Scratch Project

Seen in Figure 32, Participant I worked for a relatively short period of time and added relatively few elements. Most notably, several chunks contain relatively large numbers of sound objects, included many added at the end of the programming time.

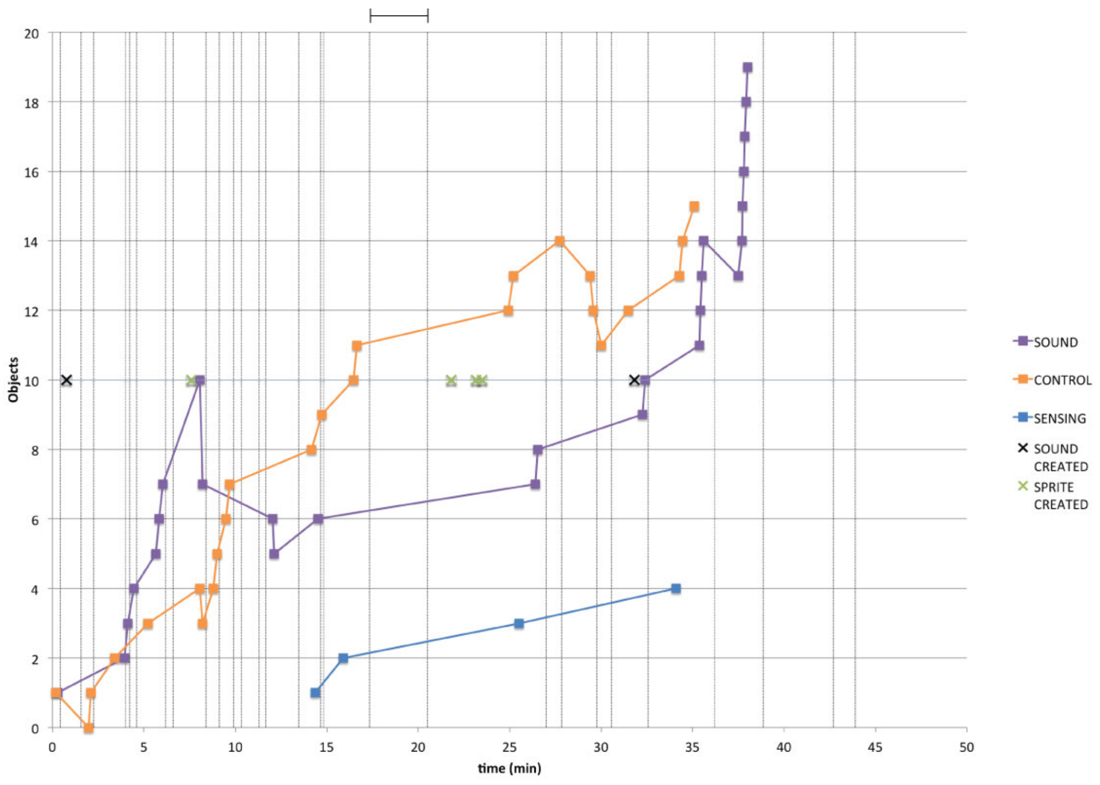


Figure 32: Timeline of Participant I's addition and removal of Scratch objects

Composers

Participant J

This project, seen in Figure 33, contains six (non-unique) discrete programs. The three on the right (4, 5, 6) are all triggered by the green flag. Each one updates a variable on the basic of the mouse position, sets an instrument this variable, and plays a note. Two add random values to the note value. The multiple instances of variable updating and instrument setting are largely redundant. The three programs on the left (1, 2, 3) behave similarly. Each repeats a random number of times, setting a new variable by subtracting a random number from the variable in the right programs. Each then plays a drum corresponding to this second (left) variable. Two introduce random wait times.

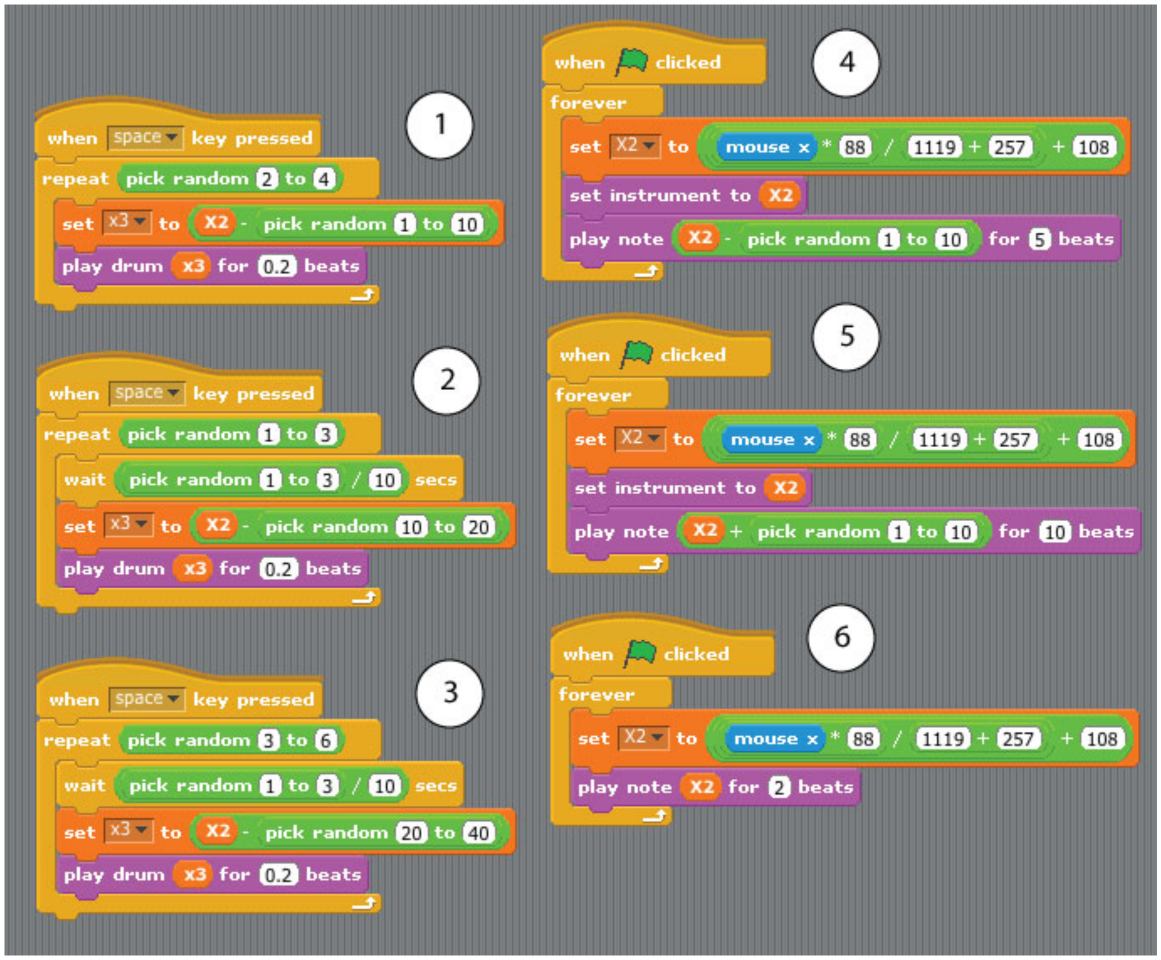

```
1
when [space] key pressed
repeat (pick random 2 to 4)
    set [x3] to (X2) - (pick random 1 to 10)
    play drum (x3) for 0.2 beats

2
when [space] key pressed
repeat (pick random 1 to 3)
    wait (pick random 1 to 3) / 10 secs
    set [x3] to (X2) - (pick random 10 to 20)
    play drum (x3) for 0.2 beats

3
when [space] key pressed
repeat (pick random 3 to 6)
    wait (pick random 1 to 3) / 10 secs
    set [x3] to (X2) - (pick random 20 to 40)
    play drum (x3) for 0.2 beats

4
when clicked
forever
    set [X2] to (mouse x) * 88 / 1119 + 257 + 108
    set instrument to (X2)
    play note (X2) - (pick random 1 to 10) for 5 beats

5
when clicked
forever
    set [X2] to (mouse x) * 88 / 1119 + 257 + 108
    set instrument to (X2)
    play note (X2) + (pick random 1 to 10) for 10 beats

6
when clicked
forever
    set [X2] to (mouse x) * 88 / 1119 + 257 + 108
    play note (X2) for 2 beats
```

Figure 33: Participant J's Final Scratch Project

Participant J's computational thinking diagram, seen in Figure 34, uses *user interaction* and *parallelism* elements to control six programs with two buttons. What follows are a deep combination of *loops*, *operators*, *variables*, and *random numbers*. The *variables* are all based on *user interaction* in the form of mouse position.

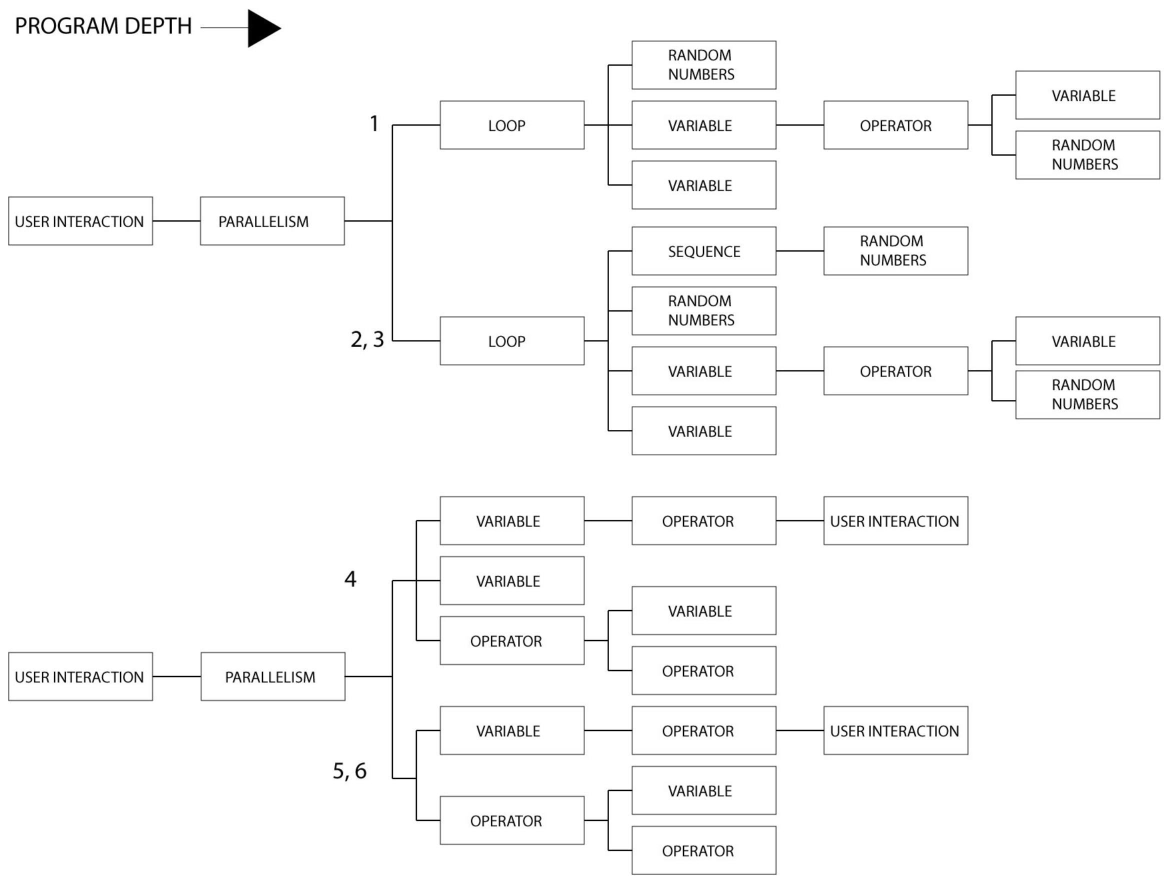


Figure 34: CT Diagram of Participant J's Final Scratch Project

Seen in Figure 35, Participant J spends 50 minutes working on a large project, involving many original sounds, before deleting the entire thing. For the next almost 10 minutes he carefully adds objects of all kinds without testing the program at all. He continues adding objects while testing the project, including letting it run continuously for more than 10 minutes at the conclusion of his work.

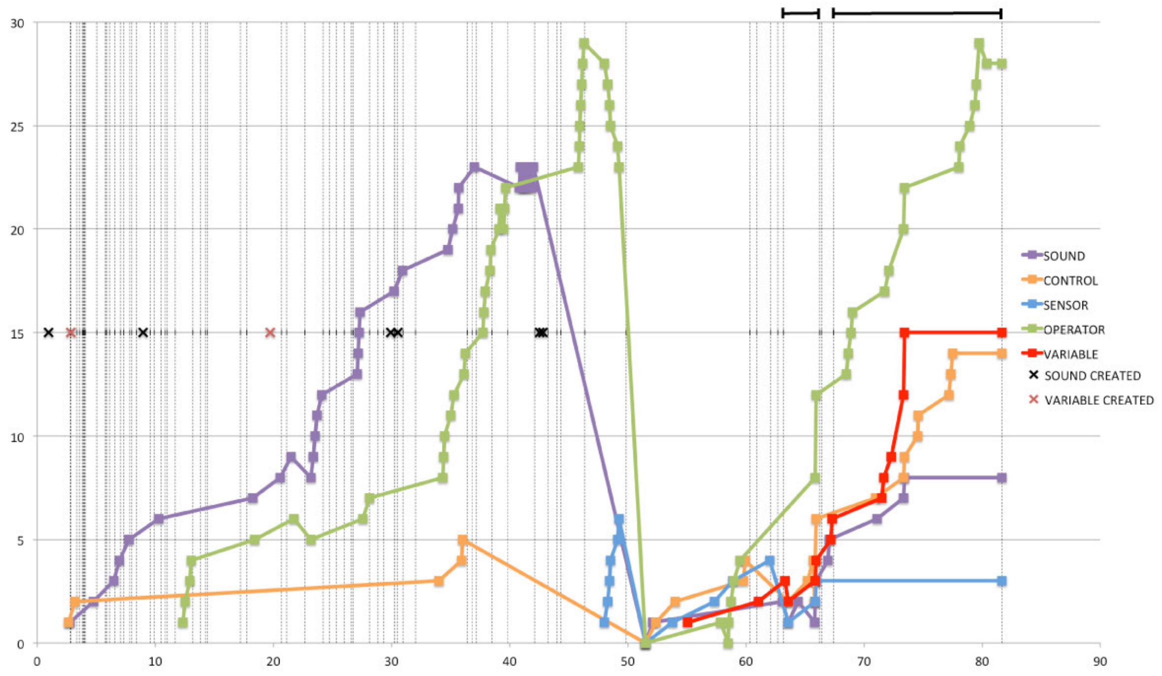
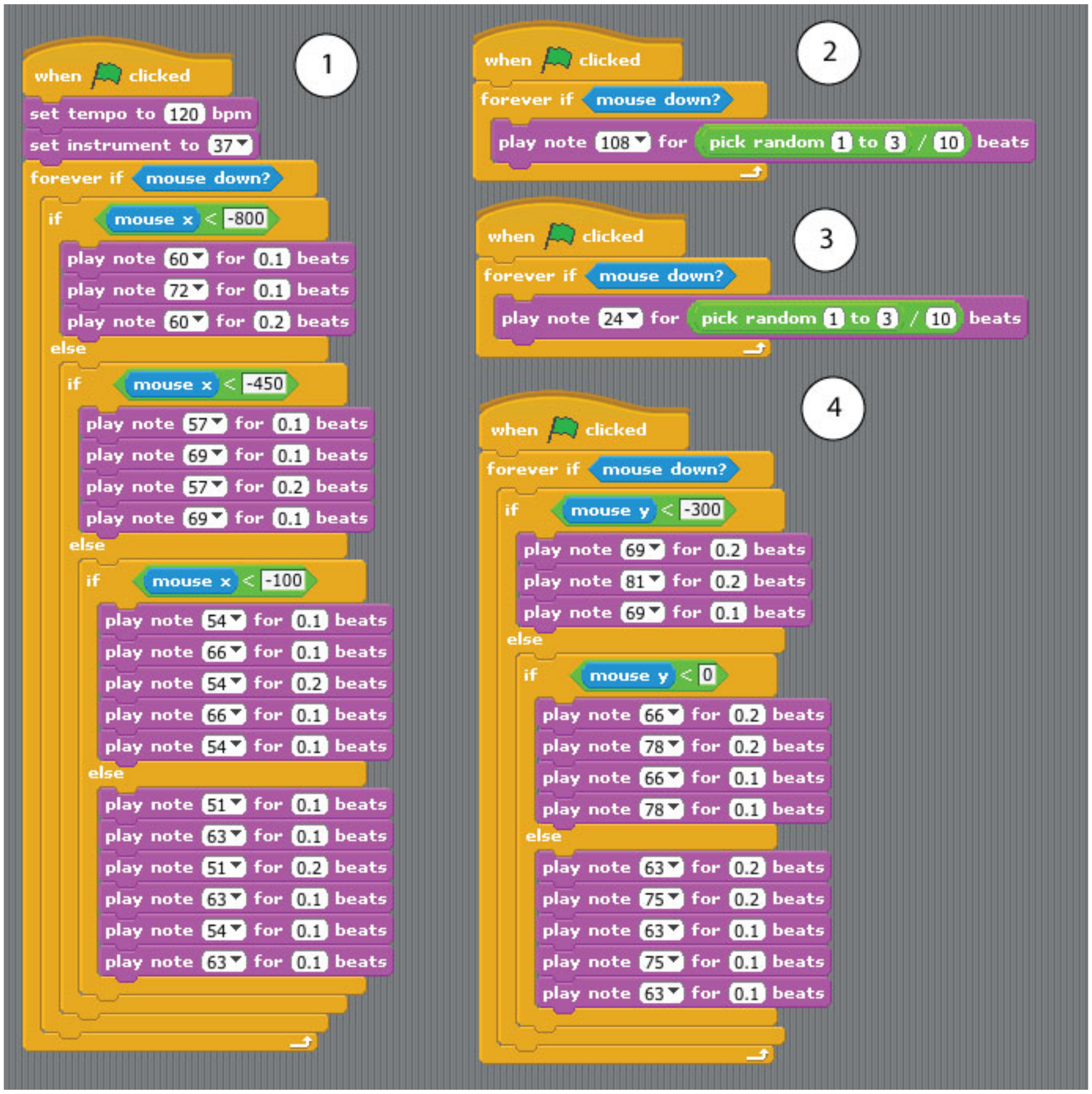


Figure 35: Timeline of Participant J's addition and removal of Scratch objects

Participant K

This project divides the screen into a 3x4 grid. Program 1 divides the screen horizontally and program 4 divides it vertically. When the mouse is pressed, both chunks play a single note (expressed in octaves) depending on the position of the mouse, creating a changing series of harmonies as the mouse moves around the screen. Programs 2 and 3 play related and unchanging notes in uneven rhythms while the mouse is pressed.



```
when [flag] clicked
set tempo to 120 bpm
set instrument to 37
forever if <mouse down?>
    if <(mouse x) < -800>
        play note 60 for 0.1 beats
        play note 72 for 0.1 beats
        play note 60 for 0.2 beats
    else
        if <(mouse x) < -450>
            play note 57 for 0.1 beats
            play note 69 for 0.1 beats
            play note 57 for 0.2 beats
            play note 69 for 0.1 beats
        else
            if <(mouse x) < -100>
                play note 54 for 0.1 beats
                play note 66 for 0.1 beats
                play note 54 for 0.2 beats
                play note 66 for 0.1 beats
                play note 54 for 0.1 beats
            else
                play note 51 for 0.1 beats
                play note 63 for 0.1 beats
                play note 51 for 0.2 beats
                play note 63 for 0.1 beats
                play note 54 for 0.1 beats
                play note 63 for 0.1 beats
```

```
when [flag] clicked
forever if <mouse down?>
    play note 108 for ((pick random 1 to 3) / 10) beats
```

```
when [flag] clicked
forever if <mouse down?>
    play note 24 for ((pick random 1 to 3) / 10) beats
```

```
when [flag] clicked
forever if <mouse down?>
    if <(mouse y) < -300>
        play note 69 for 0.2 beats
        play note 81 for 0.2 beats
        play note 69 for 0.1 beats
    else
        if <(mouse y) < 0>
            play note 66 for 0.2 beats
            play note 78 for 0.2 beats
            play note 66 for 0.1 beats
            play note 78 for 0.1 beats
        else
            play note 63 for 0.2 beats
            play note 75 for 0.2 beats
            play note 63 for 0.1 beats
            play note 75 for 0.1 beats
            play note 63 for 0.1 beats
```

Figure 36: Participant K's Final Scratch Project

Participant K's computational thinking diagram, seen in Figure 37, reveals a project of much greater depth than most prior projects. The entire project is triggered by a single *user interaction* element. Two series of nested *conditional statements*, using *user interaction* and *operator* elements to track the position of the mouse, divide the screen into a grid upon which pitches are based. Another pair of *conditional* and *user interaction* statements use *random numbers* to introduce other sounds into the mix.

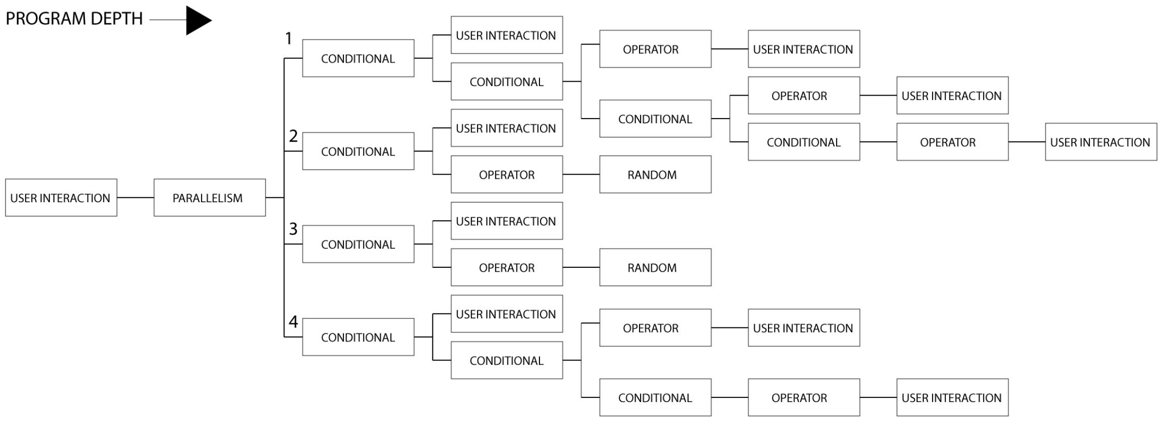


Figure 37: CT Diagram of Participant K's Final Scratch Project

In Figure 38, Participant K initially set up a series of control, sensing, and operator objects. She then primarily added sound objects, with extended breaks from adding Scratch objects to adjust not parameters while testing. At the end of the programming, she adds another series of control, sensing, and operator objects to create her two chunks based on random numbers. She also briefly introduces lists and variables before rejecting them.

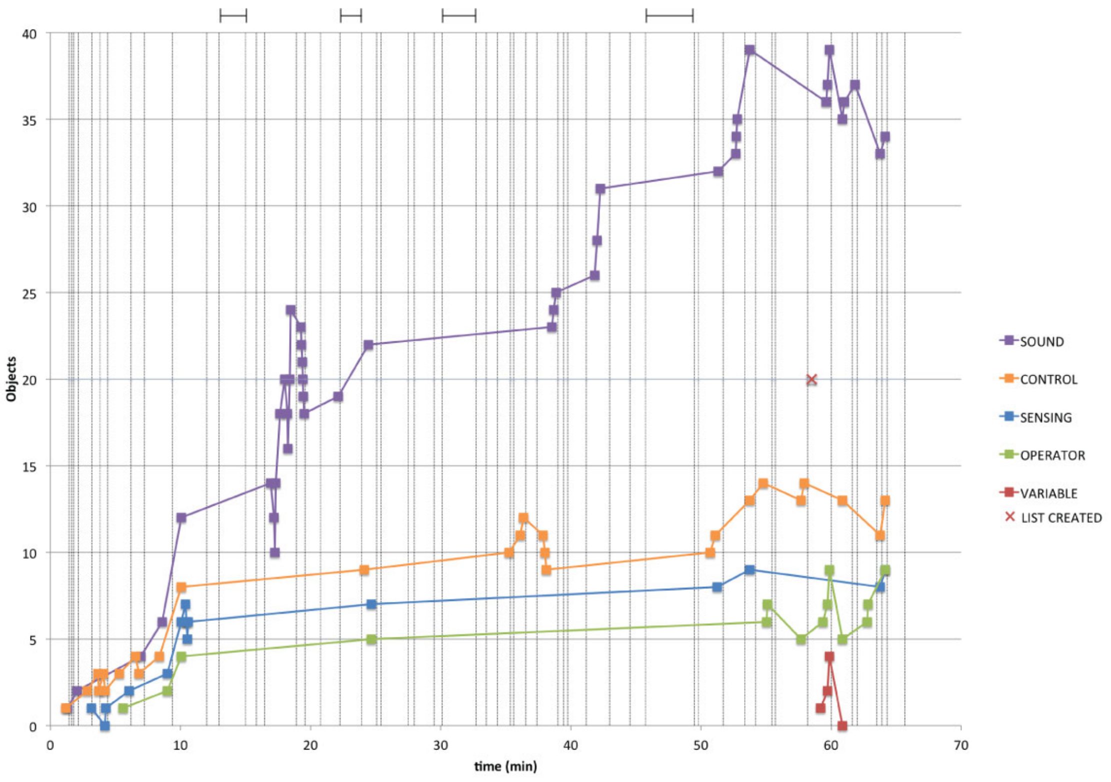
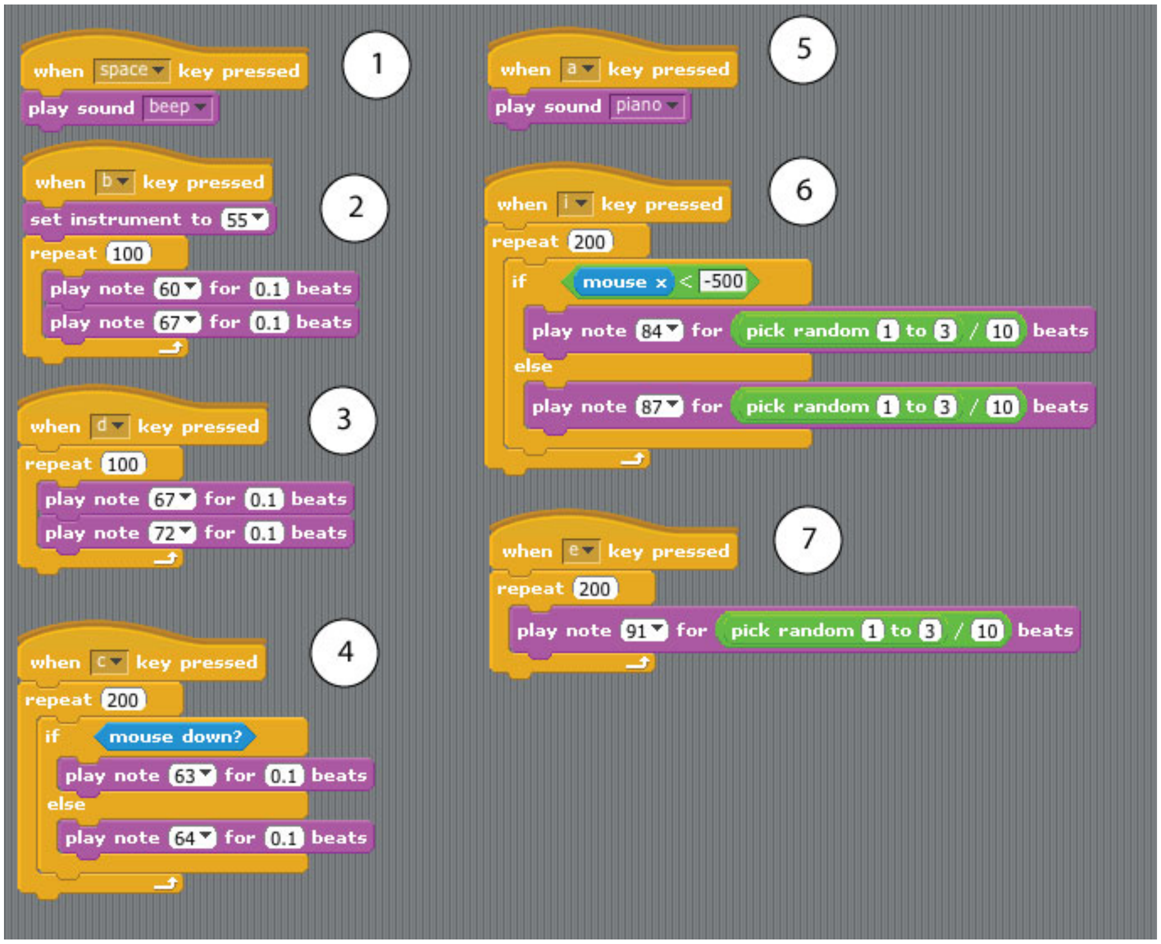


Figure 38: Timeline of Participant K's addition and removal of Scratch objects

Participant L

This program uses audio recordings and notes to create a tapestry of sound. Programs 1 and 5 use key presses to play a pair of audio recordings (of smartphone ringtones). Programs 2-4, 6, and 7 use key pressed to repeat sequences of notes in harmony with the audio recordings. Programs 2 and 3 simply repeat 2-note sequences. Programs 4 repeats a single note, its value changing based on whether or not the mouse is pressed. Program 6 repeats a single note for a random duration, the note's value changing based on the mouse's x position. Program 7 repeats a single note for a random duration.



```
when space key pressed
play sound beep

when b key pressed
set instrument to 55
repeat 100
    play note 60 for 0.1 beats
    play note 67 for 0.1 beats

when d key pressed
repeat 100
    play note 67 for 0.1 beats
    play note 72 for 0.1 beats

when c key pressed
repeat 200
    if mouse down?
        play note 63 for 0.1 beats
    else
        play note 64 for 0.1 beats

when a key pressed
play sound piano

when i key pressed
repeat 200
    if mouse x < -500
        play note 84 for pick random 1 to 3 / 10 beats
    else
        play note 87 for pick random 1 to 3 / 10 beats

when e key pressed
repeat 200
    play note 91 for pick random 1 to 3 / 10 beats
```

Figure 39: Participant L's Final Scratch Project

In Participant L's computational thinking diagram, seen in Figure 40, we find *user interaction* elements primarily triggering *loops*. More than half utilize *conditional statements* depending upon and/or triggering *user interaction*, *operators*, and *random numbers*.

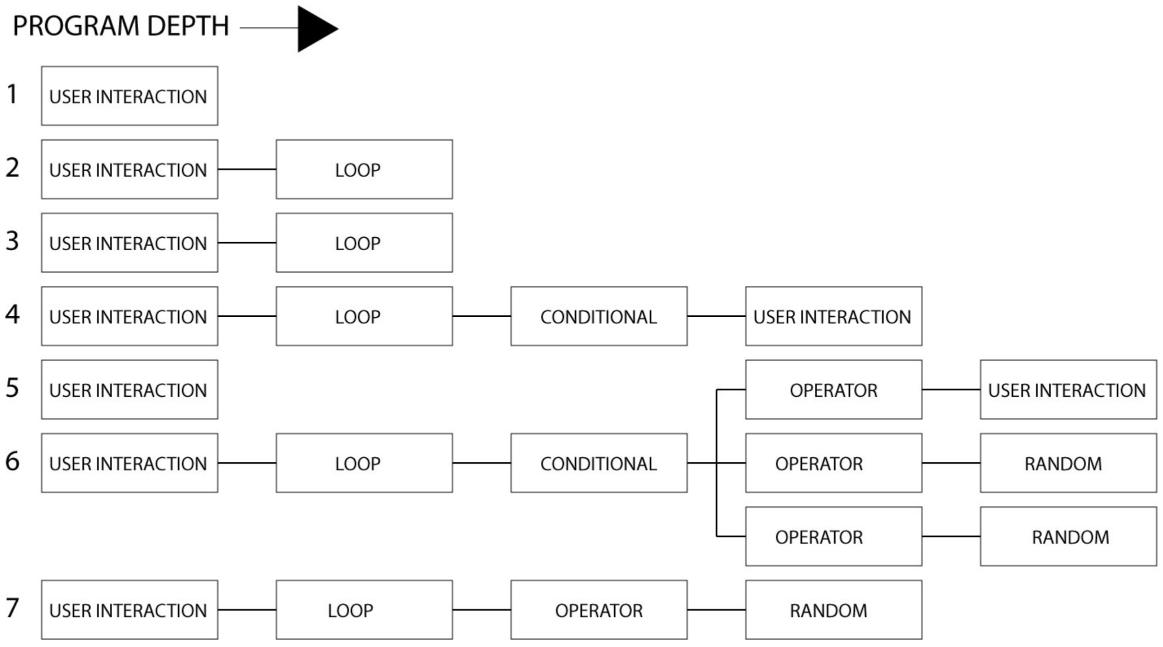


Figure 40: CT Diagram of Participant L's Final Scratch Project

In Figure 41, we see Participant L add a series of control and sound objects before deleting her entire project. She takes a break to create two sounds before beginning again. At this point she adds and removes control and sound objects in tandem with each other, as well as operator and sensor objects.

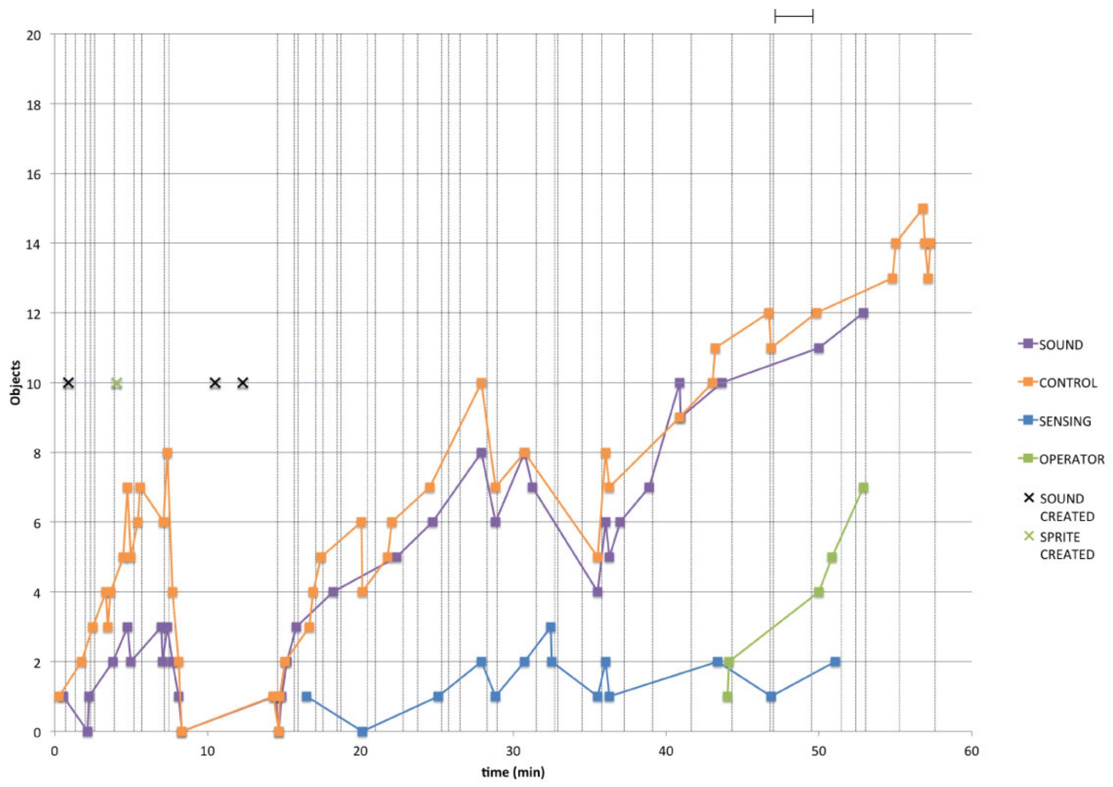


Figure 41: Timeline of Participant L's addition and removal of Scratch objects

Chapter 5: Group Results

My overarching research question explores if different kinds of musical backgrounds play a role in novice programing. I will answer this by first addressing my three smaller research questions, which examine how musical concepts emerge as scaffolds for novice programmers as well as the programming patterns and final projects of learners from a variety of musical backgrounds.

Musical Concepts as Scaffolds

I used transcripts of participant talk to learn how musical concepts emerged as scaffolds during the programming activity. Across the twelve participants, I identified 402 utterances that were coded using the Subject and Type scheme described in Chapter 3; 62 of these were double-coded, resulting in 464 individual codes. The distribution of codes is found in Table 1 (all percentages are out of 464 total codes). Most utterances were categorized as Assertions (51.9%). Questions and Intentions were nearly equal, each representing about 25% of utterances. Utterances coded with the Musical subject represented 18.3% of the utterances, the third-highest out of the subjects. Since my research examines musical concepts as scaffolds, I will next discuss the talk that intersected between Utterance types and Musical subject.

Table 1: Subject and Type Distribution of Participant Utterances

		Subject					
		Operational	Programming	Scratch	Musical	Aesthetic	Total
Type	Question	13 (2.8%)	19 (4.1%)	65 (14%)	16 (3.4%)	0	113 (24.4%)
	Assertion	60 (12.9%)	52 (11.2%)	63 (13.6%)	43 (9.3%)	23 (5%)	241 (51.9%)
	Intention	40 (8.6%)	9 (1.9%)	4 (1%)	26 (5.6%)	31 (6.7%)	110 (23.7%)
	Total	113 (24.4%)	80 (17.2%)	132 (28.4%)	85 (18.3%)	54 (11.6%)	

Musical Questions

Musical-Questions represented the smallest fraction of participant musical talk. Some example questions from Participant E were purely musical, such as, *"So I need to work out a clave somehow so I can do a clave beat. Maybe in 5. Is that possible?"*or *"I should put* Row Row Row Your Boat *in here. Is that diatonic?"*

The majority of musical questions were utterances double-coded as operational or Scratch questions, such as Participant J using musical language in asking a question about a project's operation,*"If the bpm is changing by 10 each time, that's going to be less and less perceptible as it goes?"* or Participant D doing the same to ask about the Scratch environment, *"Does this program have a metronome?"*

Musical Assertions

The majority of musical utterances were assertions, which narrate something the participant was doing or creating. The vast majority of musical assertions involved pitch or tempo (or duration). For example: Participant G said, *"46, 44 … I'm trying to add two octaves below. And then 34 and 32."* Participant K said *"Plus three, plus three, this is all minor thirds, obviously."* (Participants were referring to pitches as integers here.) Participant I said *"I need to figure out this timing. I'm changing this from 0.2 to 0.8 beats. We'll hold that one longer."* Participant E said *"I'm going to turn this to 0.25 beats and it'll give me a quarter note,"* before announcing *"I'd like to do a sixteenth note here, but you can't do decimals. That sucks."* (Scratch timing *does* do decimals, though only with one-tenth resolution which forbids the aforementioned quarter and sixteenth notes.)

Five of the 43 musical assertions were double-coded with other themes. Participant G said, *"Can I access other octaves in Scratch or I can just add or subtract 12 from all of these,"* combining a musical assertion with a Scratch question. Participant E said, *"I want to press different keys that will play short drums things, like 0.2 beats,"* combining a musical assertion with an operational intention.

These Musical-Assertion examples points towards one main way in which musical concepts emerged as scaffolds for these novice programmers. Participants' musical talk demonstrated an understanding of the numeracy involved in programming music in the Scratch environment. In discussing pitch, some participants readily articulated ideas in terms of semitones between traditionally notated pitches and/or explained pitches using Scratch's integer notation. For instance, above Participant K mentions minor thirds (*"Plus three, plus three, this is all minor thirds, obviously"*); a minor third is an interval of three semitones between two pitches (a semitone is the smallest interval in traditional western music and the distance between any two keys on the piano). For instance, C and D# are separated by a minor third (C to C# to D to D#). As Participant K explains, he simply adds three to successive pitch values to create a cascade of minor thirds, resulting in the *"wave of ominous minorness"* he desired. Participants articulated ideas about tempo and note durations similarly, both in familiar musical terms as well as in Scratch's particular duration formulation. These two approaches are highlighted in the examples from Participant E, *"I'm going to turn this to 0.25 beats and it'll give me a quarter note,"* and *"I'd like to do a sixteenth note here, but you can't do decimals. That sucks."* The traditional language of quarter and sixteenth notes is built around fractions of a musical bar, whereas Scratch works in the language of pure duration (in beats) at a particular number of beats per minute. As Participant E's disappointment illustrates, these two approaches do not easily translate back and forth, but participants nevertheless learned to achieve the results they desired in the language of Scratch.

Musical Intentions

Musical intentions primarily related to desires for specific instrumental sounds or used technical musical terminology to describe musical effects. Participant D said, *"What I want it to be able to do is modulate."* Participant K said, *"I want a wave of ominous minorness."* Participant E said, "*I want to make a round sort of thing, where I press a button and something starts and then something else starts a bit later,*" (double-coded as an operational intention). Participant G said, *"We'll make this very small for a glissando effect."* In some cases, these intentions based around music ideas pointed directly towards programming challenges. For instance, Participant G stated a desire for a glissando effect. A glissando is a slide between pitches. Like a piano or guitar, Scratch plays discrete pitches and, as such, a pure glissando in the manner of a trombone or violin is impossible. However, beginning with this simple idea, the participant formulated an approach to approximating it, beginning from a high-level operational perspective and eventually articulating his idea in Scratch. More broadly, while none of the musical terminology used by participants was especially complex or esoteric, these terms, familiar to musicians, may have provided a rich bank of ideas about what a project might actually do. Participant G articulated this approach, focusing specifically on the construction of a piece of music, in explaining:

> Well, what do you need to make music? You need melody, maybe you need a pad, the different elements that could make up a piece of music. Let's make up a couple of those and see how we want to put them together. Just making elements. Being able to control and deploy them as necessary.

Musical Talk by Participant Group

The discussion thus far has focused on describing the musical talk that occurred and its roles, independent of participant groups. Table 2 breaks down the 85 musical utterances by participant group (all percentages are out of 85 total utterances). The non-musician group accounts for less than 10% of musical utterances; the remaining musical utterances are close to evenly distributed between the three musician groups. Musical-questions account for less than one-fifth of all musical utterances. Most are distributed near evenly between the jazz and classical groups, with a small number from the non-musician group. Musical-assertions account for half of all musical utterances and composers account for half of those. The remaining musical-assertion utterances are distributed near evenly between the jazz and classical groups. Musical-intentions were just under one-third of all musical utterances. The jazz group had the most musical-intention utterances, just under half of the total.

In addition to their smaller numbers, musical utterances from the non-musician participants were qualitatively different from the musician groups. All musical intentions and questions except one were focused on instrumental sounds (*"I'd like to have a string* *sound here"*). The other question (*"What does 0.2 beats mean?"*) was similar to questions asked by musician participants, but not leveraged any further than that.

Table 2: Musical Utterances by Participant Group

		Participant Group				
		Non-Musician	Jazz	Classical	Composer	Total
Type	Musical-Question	3 (3.5%)	7 (8.2%)	6 (7.1%)	0	16 (18.8%)
	Musical-Assertion	0	10 (11.8%)	12 (14.1%)	21 (24.7%)	43 (50.6%)
	Musical-Intention	3 (3.5%)	11 (13.0%)	6 (7.1%)	6 (7.1%)	26 (30.6%)
	Total	6 (7.1%)	28 (32.9%)	24 (28.2%)	27 (31.8%)	

Summary

Musical concepts emerged as scaffolds in the form of musical descriptors that provided a structural framework for programming challenges. This kind of talk, exemplified in musical-intention utterances, was most common in the jazz group of participants. Musical talk also helped participants navigate Scratch's musical numeracy. This kind of talk, exemplified in musical-assertion utterances, was most common in the composer group of participants. Musical talk was far more prevalent in musicians than non-musicians.

Patterns in Programming Process by Participants' Musical Backgrounds

My second question asks what kind of patterns learners from a variety of musical backgrounds exhibit in their programming processes. I address this question by examining programming duration, program testing, and the addition and removal of Scratch objects.

Figure 42 shows the duration of each session alongside averages from each participant group. Non-musician sessions, on average, are shorter than any other group of participants. Composer sessions are, on average, longer than any other group of participants.

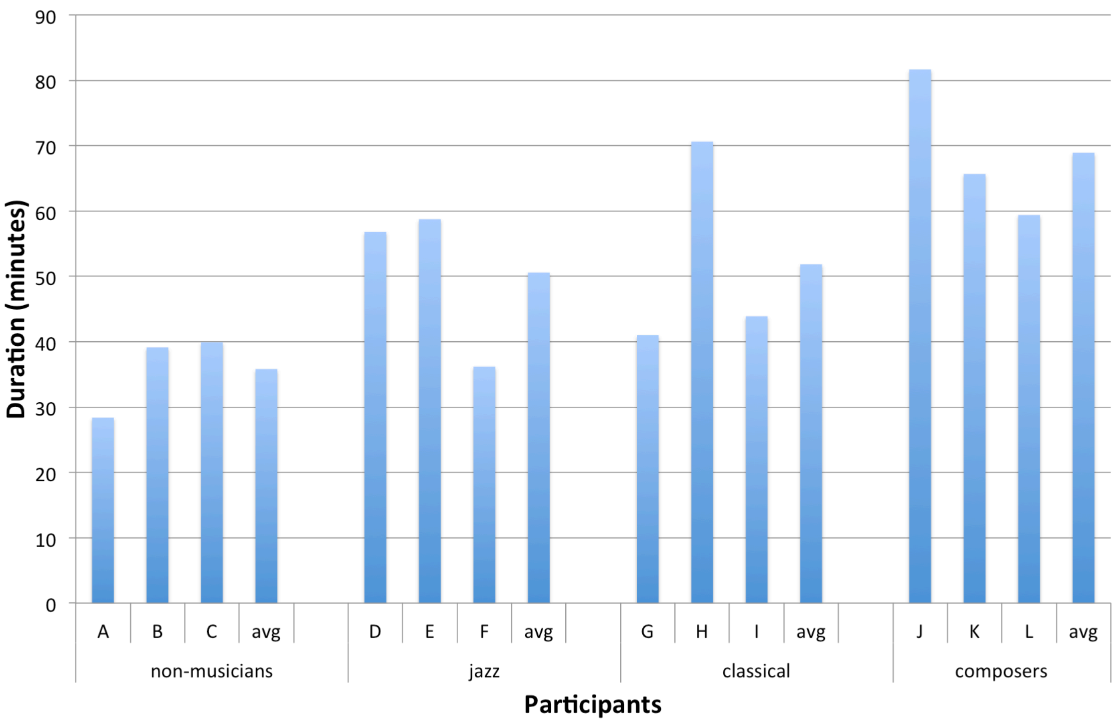


Figure 42: Programming Duration

Figure 43 summarizes program tests per minute. A program test involves simply executing a piece of Scratch code. Once a program was tested I did not record subsequent tests until the participant had changed something in the project. In many cases an early chunk of Scratch code might generate a single tone and a participant might activate it over and over again; this would be recorded as a single test. Averages across groups range from 0.6-0.8 tests per minute, or 1 test every 75-100 seconds. Program tests were not evenly distributed for any participant. Perusing programming diagrams in the appendices (G-R) will show relatively tight clusters of testing and gaps of many time durations during programming or discussion. While the composer group has the highest average tests per minute, there is too much variance within groups to make any group-based inferences.

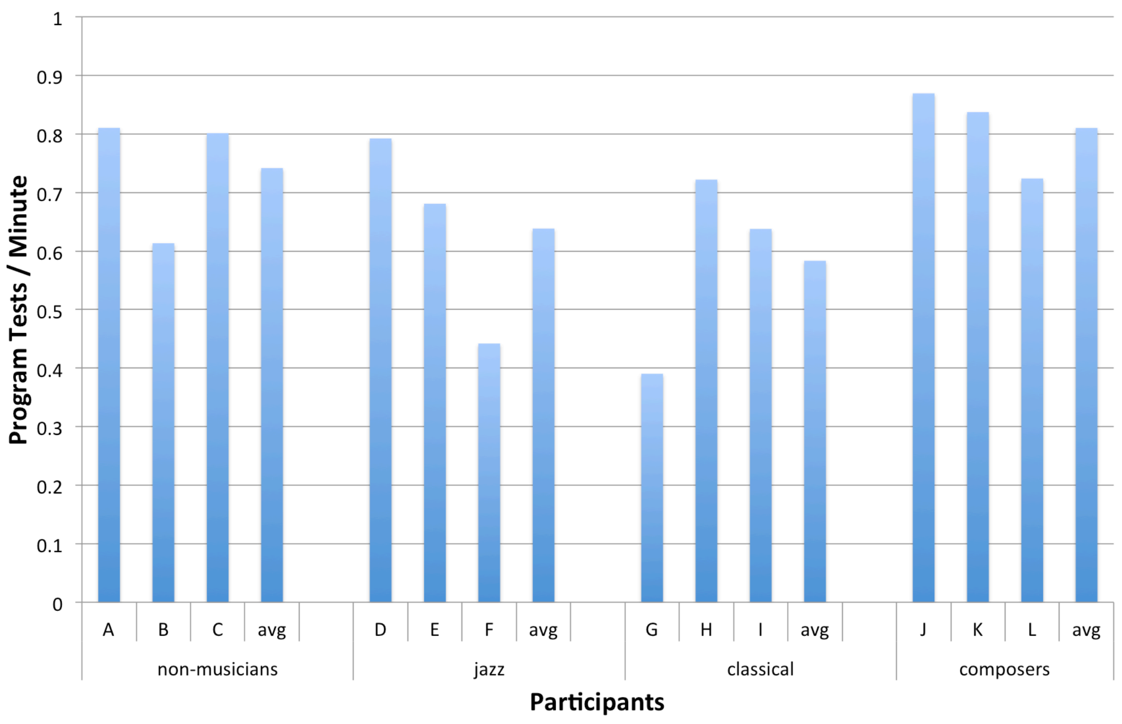


Figure 43: Program Tests Per Minute

Figure 44 shows the net Scratch objects added by each participant. Participant D's many objects are contained in a pair of lengthy chunks of code dense with operators, variables, and random numbers. While the non-musician group's low numbers are in line with the group's shorter programming time, the same does not hold true for the composer group and its overall longer programming duration. Figure 45 shows the composer group adding relatively few objects per minute.

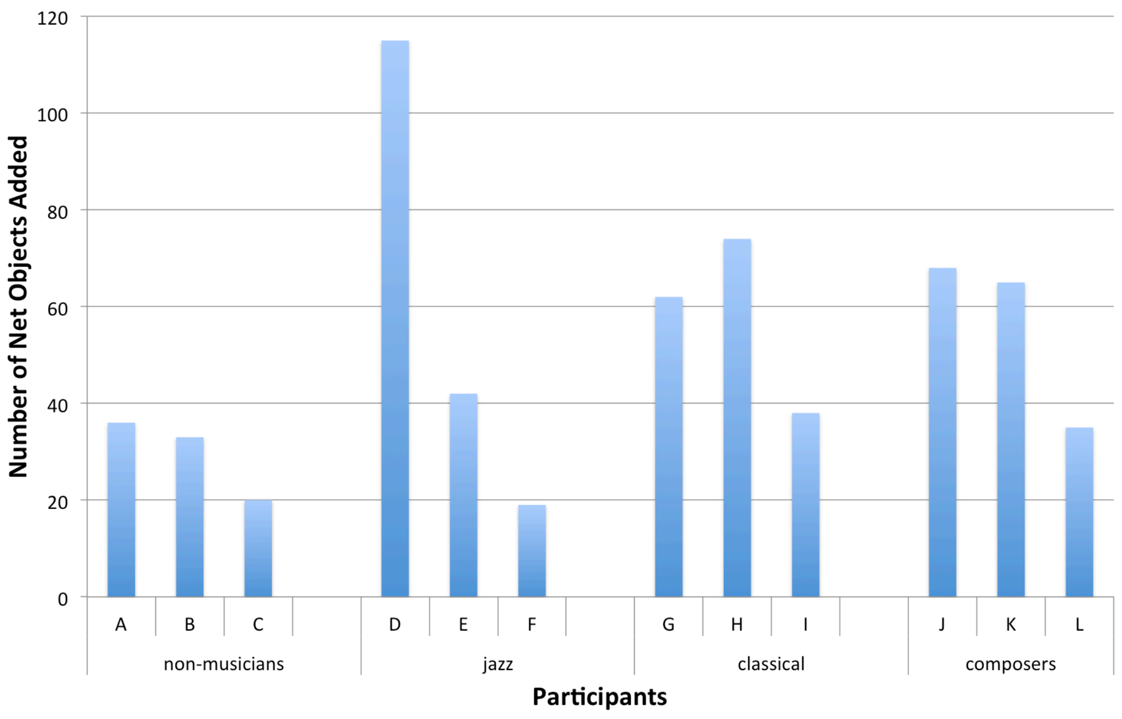


Figure 44: Net Scratch Objects Added

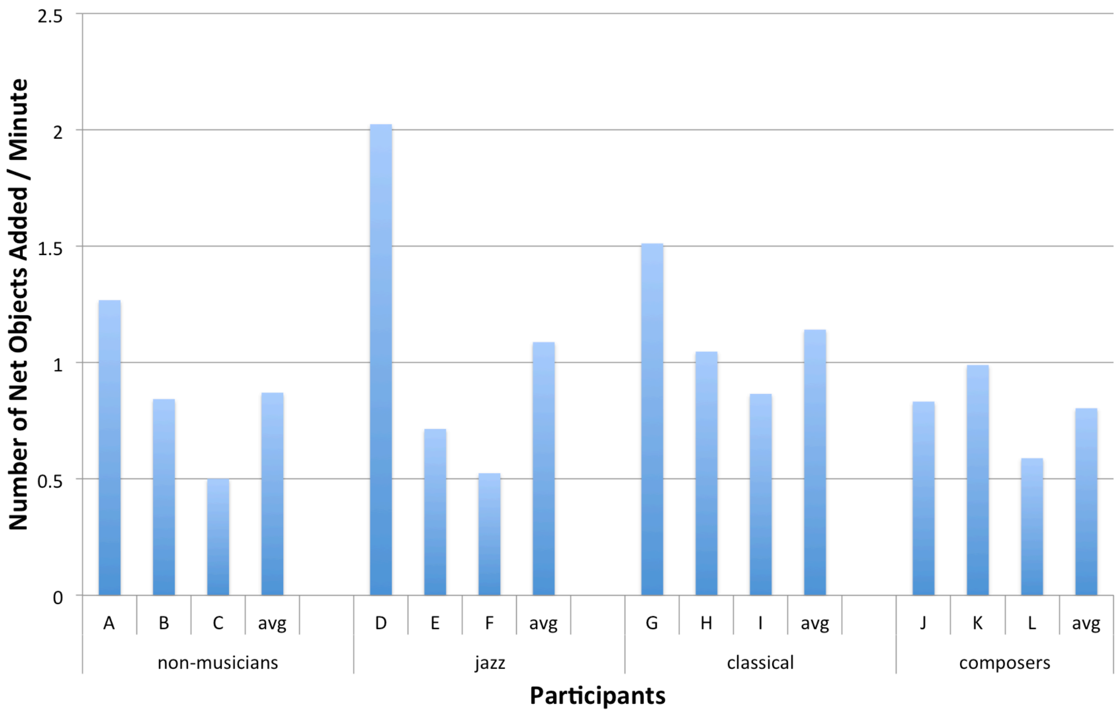


Figure 45: Net Scratch Objects Added Per Minute

Figure 46 breaks net objects added into Scratch object types, and Figure 47 displays object types as a percentage of net objects; Figure 47 more easily allows comparison of the makeup of individual projects between participants and groups. All projects contained Sound and Control objects by definition. Putting aside Motion objects, non-musician projects are dominated by Sound and Control objects and include no Variable objects at all. Operators appear in all composer projects – two of three composer participants took advantage of the screen area in various ways and all used random numbers; Sensor objects appear in all composer and classical projects – the relatively high proportion in the classical group may be related to an impulse to create buttons and an instrument.

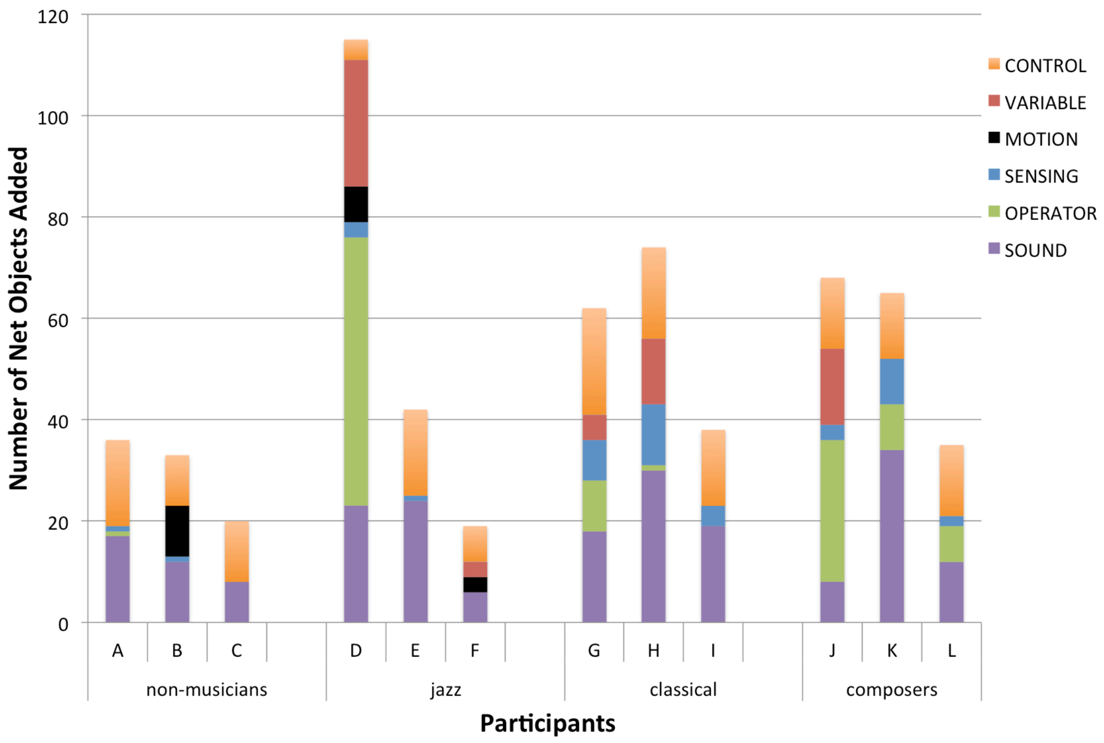


Figure 46: Types of Net Scratch Objects Added

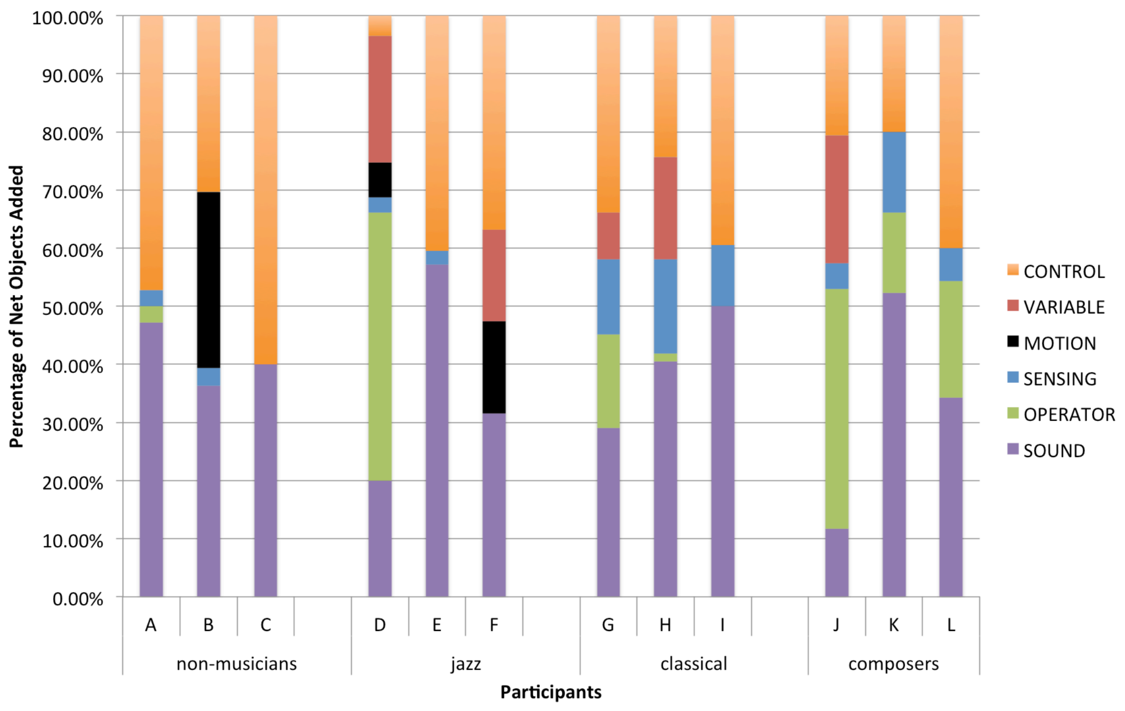


Figure 47: Types of Net Scratch Objects Added as a Percentage of All Net Scratch Objects Added

Figure 48 shows the number of Scratch objects removed by object type. Figure 49 presents these values as percentages of total objects added (that is, net objects added plus objects added but later removed). The preponderance of Sound and Control objects being removed may be related to participants removing entire chunks of code coupled with those objects' prominence overall in the projects.

Participant J deleted his entire project, but created a new project with nearly as many objects, as seen in Figure 49. Participant C, on the other hand, continually added and deleted objects in the eventual creation of a rather simple and straightforward project. While there was a great deal of variance in the non-musician, jazz, and classical groups, the composer group overall removed far more objects, which accounted for a similarly larger fraction of total objects added.

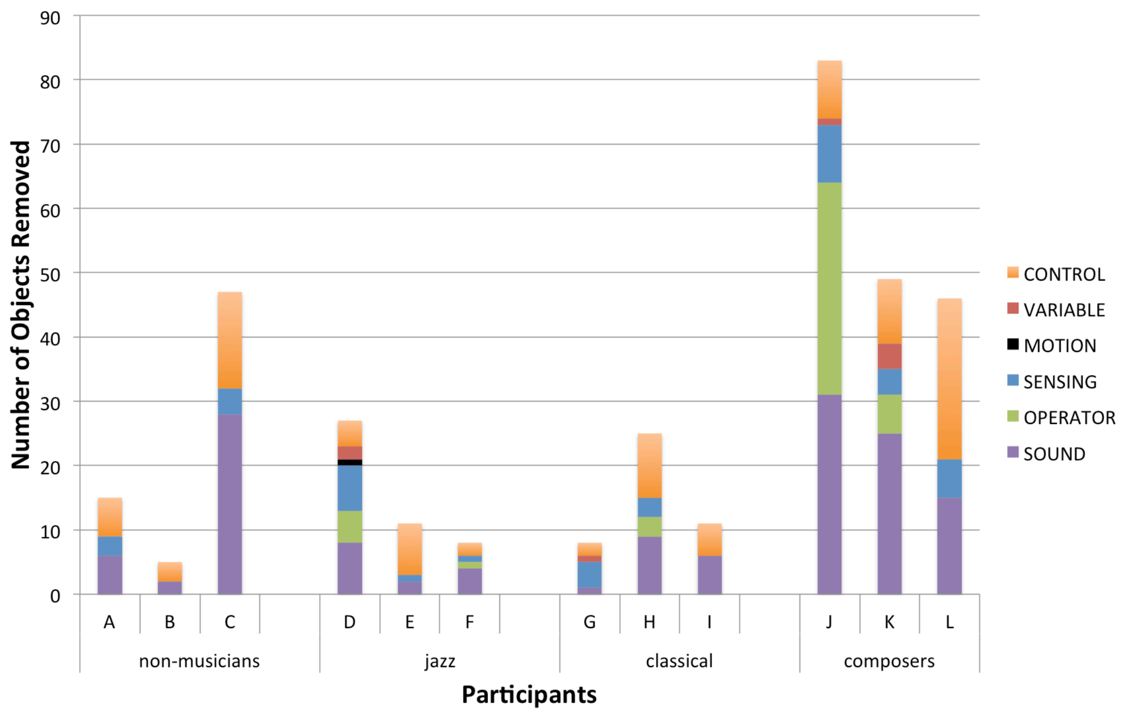


Figure 48: Total Objects Removed Across All Programming Time

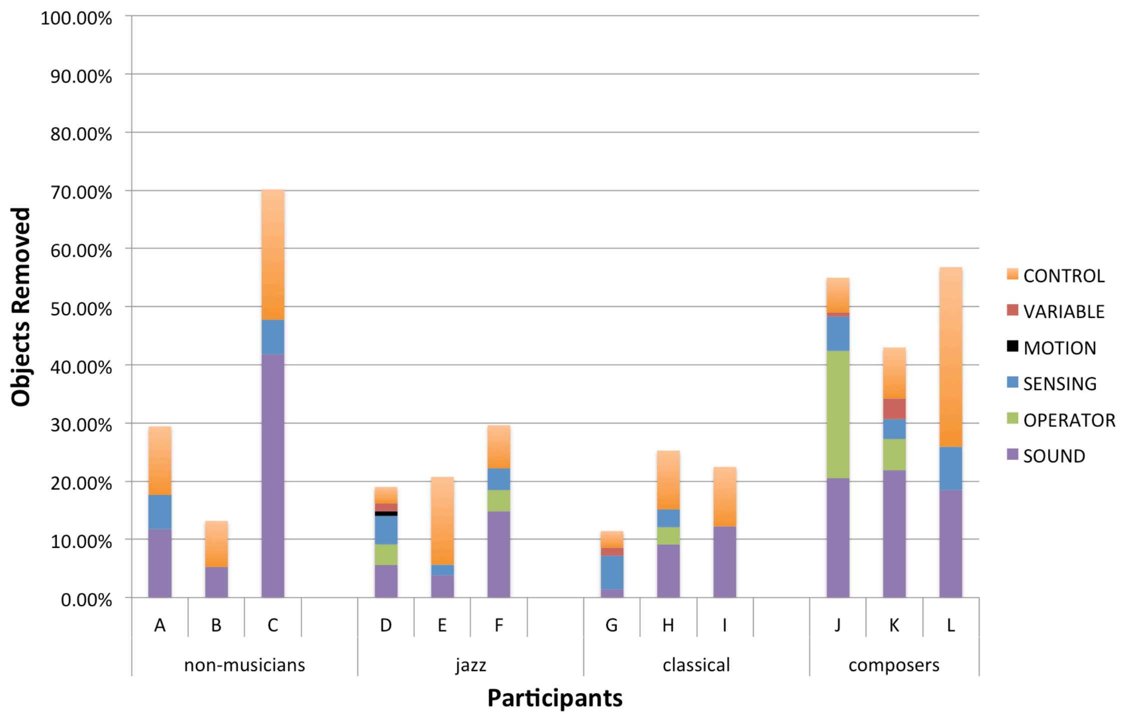


Figure 49: Objects Removed as a Percentage of All Objects Added

Summary

These results show several patterns in programing process across participant groups. Participants in the composer group, on average, worked for the longest periods of time, tested their projects the most frequently, added items the slowest, and removed more objects (by number and percentage of all objects added) than all but one other participant, Participant C. All participants in the composer group used sensor, operator, and random number objects. Participants in the classical group, on average, tested their projects the least often and added items the most quickly of all the groups. All participants in the composer group used sensor objects. Participants in the jazz group were, on average, in the middle of all metrics. The jazz group did include an outlier with respect to the "objects added" metrics, Participant D. Participant D's project included a program featuring a long series of sequential instructions, each influenced by a random number or operator object. He then copied this program, resulting in a very high number of overall objects. Participants in the non-musician group worked the least amount of time and added the fewest Scratch objects.

Differences In Final Projects by Musical Background

My final sub-question asks how the final projects differed by participants' background. I answer this question by examining the final projects' number of distinct programs and depth, as well as participants' use of computational thinking concepts.

Distinct Programs

Figure 50 shows the number of distinct operable programs within each final project. There is a great deal of variance in number of programs created within most groups. Participants E and G are similar in their approaches; that is, they both created many relatively small programs, each triggering individual and often unrelated sounds.

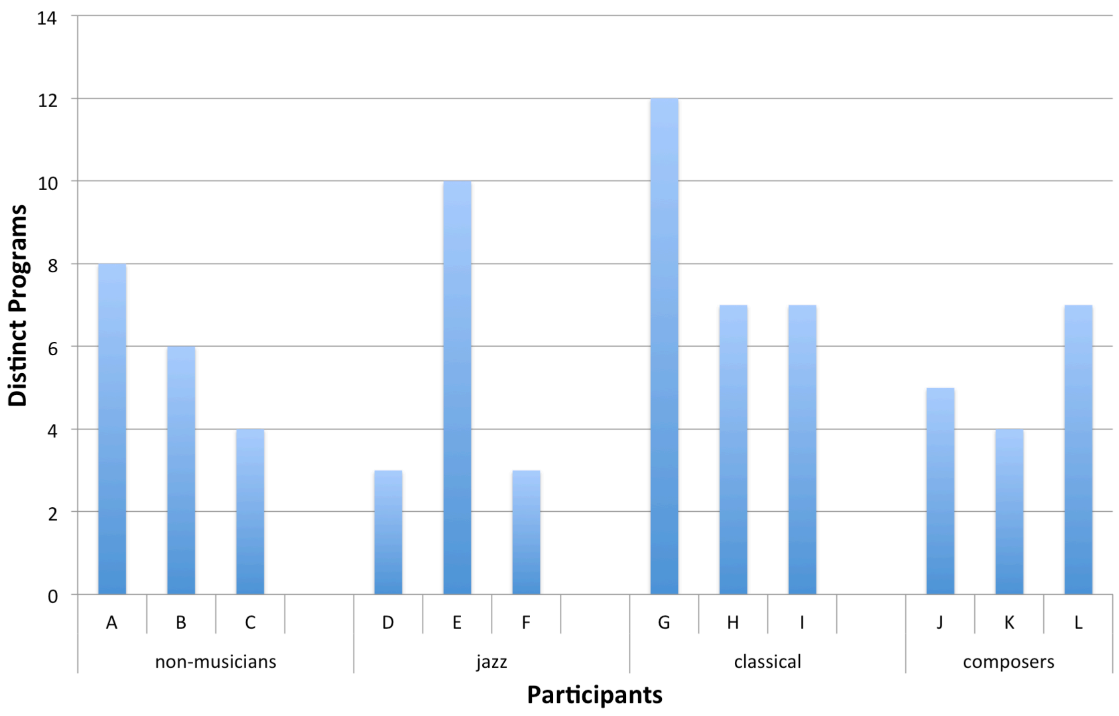


Figure 50: Distinct Programs in Final Projects

Maximum Computational Thinking Depth

Figure 51 shows the maximum depth of each project, based on the computational thinking diagrams found in the appendices (G-R). The composer group is overall greater in maximum depth and the non-musician group lower in depth, with the jazz and classical groups in between the two.

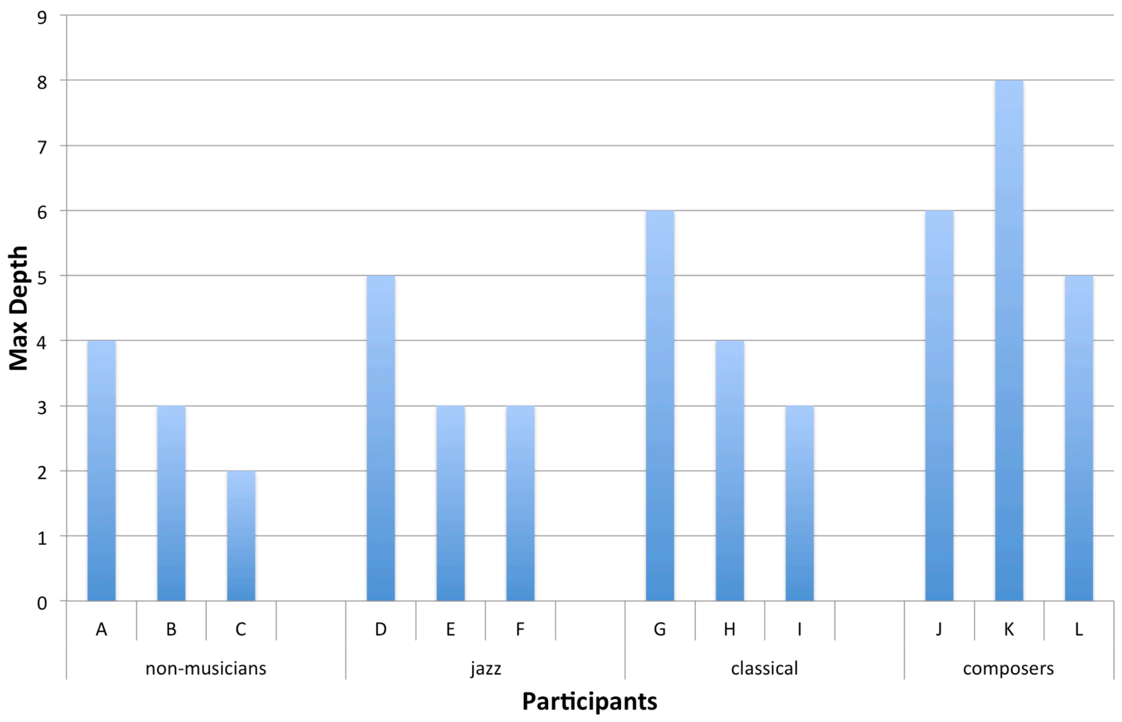


Figure 51: Maximum Depth of Final Scratch Projects

Computational Thinking Concepts

The use of specific computational thinking concepts in participant projects is examined using the Instances and Average Depth metrics explained in Chapter 3. In this section, I will describe the use of each of the 10 computational thinking concepts across the participants' projects.

User Interaction

Instances of *user interaction* by participant appear in Figure 52. A degree of *user interaction* was one of the two project requirements and, as such, appeared in all participant projects. Classical participants used the greatest number of *user interaction* elements, on average.

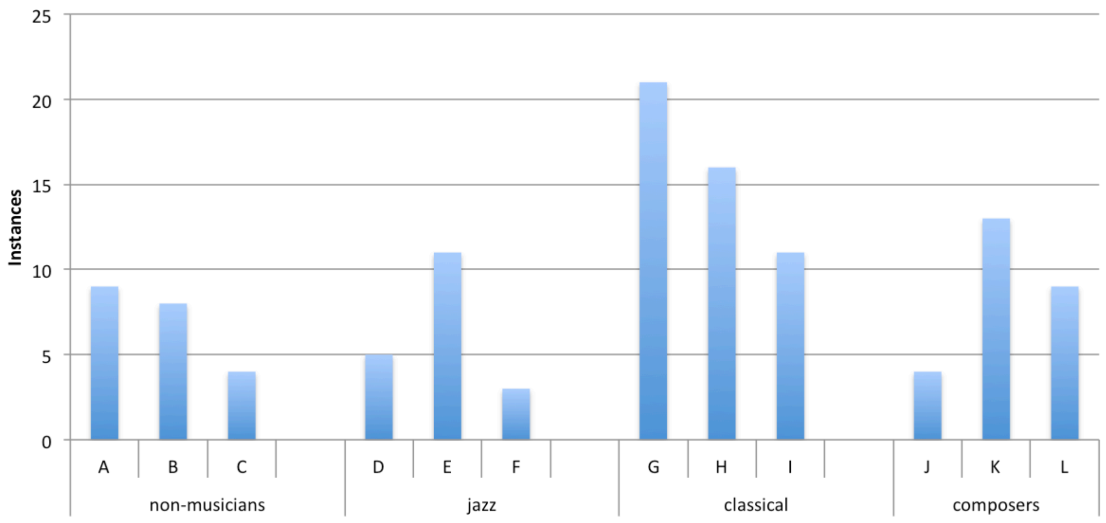


Figure 52: Number of Instances of *User Interaction*, by participant

In Figure 53, I report the average depth for *user interaction* is close to one for five of the 12 participants (all non-musicians and two of three jazz musicians); in these cases *user interaction* is primarily limited mouse clicks or key presses that trigger Scratch actions. That these depth of one triggers are a virtual necessity effectively pulls down the average depth of *user interaction*; the five participants with average *user interaction* depths greater than two all utilize *user interaction* in constructive ways. Three participants based various actions on the mouse button (following an initial activation). Four tracked the mouse's x and/or y position, rescaling the values for the creation of new variables or triggering different actions based on the mouse position on the screen. One used a series of key presses inside *conditional statements* to create a simple piano keyboard.

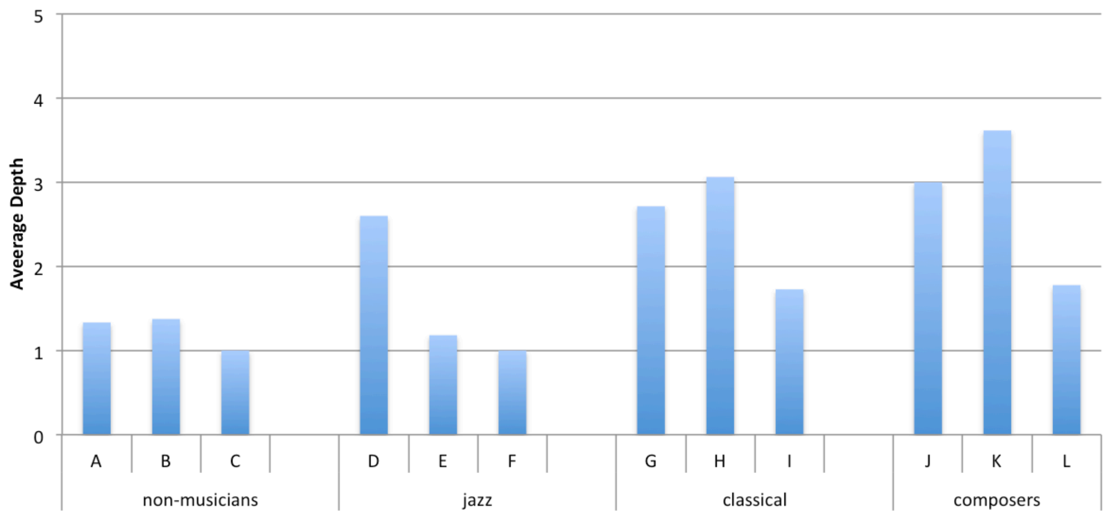


Figure 53: Average Depth of *User Interaction*, by participant

Classical participants stand out for having more instances of *user interaction* on average while also being one of the two groups with higher user interaction depth.

Loops

Loops appeared in a majority, but not all, of the participant projects, as seen in Figure 54. This includes all participants in the classical group. I considered finite repeats in this section, such as “repeat x times” or “repeat until…” rather than inclusion of repeat infinitely. This excludes the Scratch *forever* object, which was presented as a fundamental part of the Scratch tutorial.

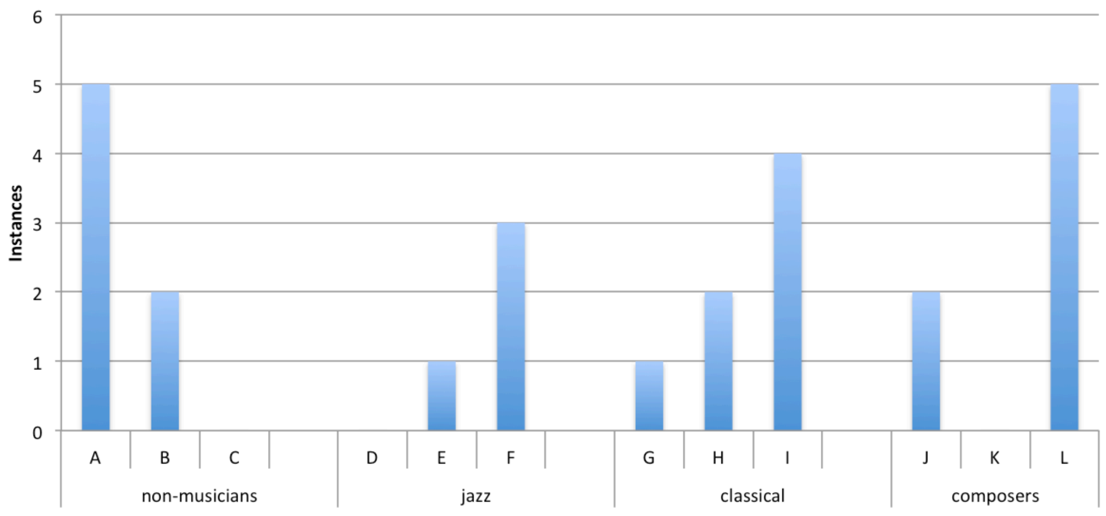


Figure 54: Number of Instances of *Loops*, by participant

From Figure 55, *loops* appeared relatively early, depth-wise, in most projects, immediately following a user interaction trigger or following a *parallelism* or *conditional statement*.

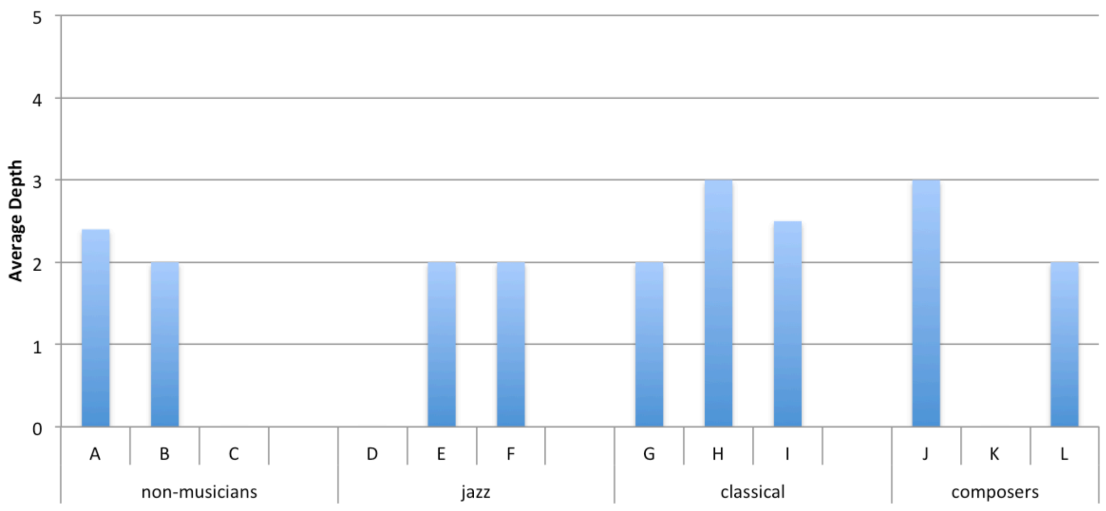


Figure 55: Average Depth of *Loops*, by participant

Conditional Statements

As seen in Figure 56, eight of the twelve participants, including all participants in the classical group, used *conditional statements* in their projects. Most used *conditional statements* to check for mouse presses, key presses, or sprite touches to trigger audio. Two that employed moving sprites used Scratch's *if on edge, bounce* object. Four used *conditional statements* to check the position of the mouse: one rescaled mouse position to create new variables when the mouse was within a certain ranges, while three used if-else statements to play different notes based on mouse position.

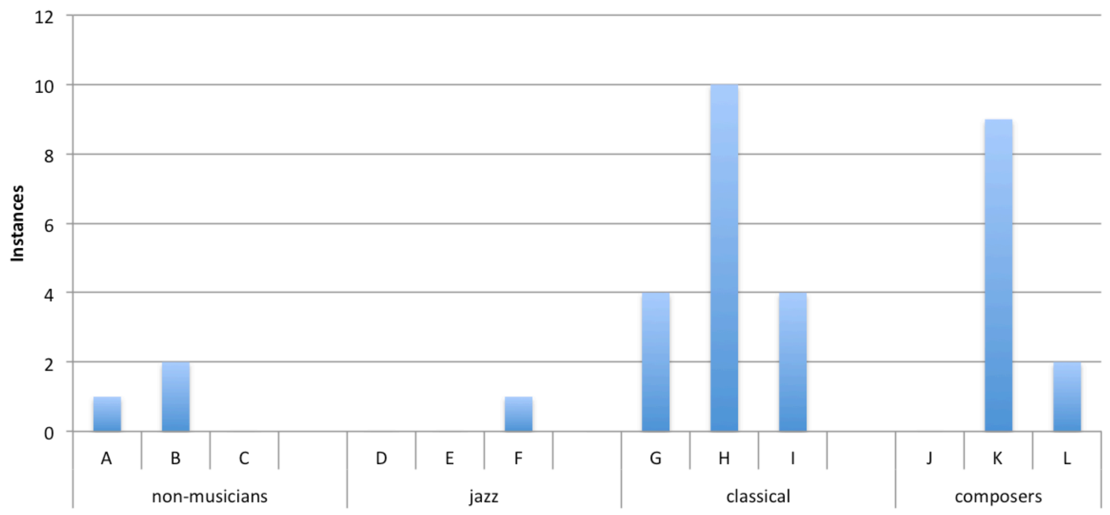


Figure 56: Number of Instances of *Conditional Statements*, by participant

Participant K, with the highest average *conditional statement* depth as seen in Figure 57, uses two series of nested *conditional statements* to demarcate the computer screen into a grid.

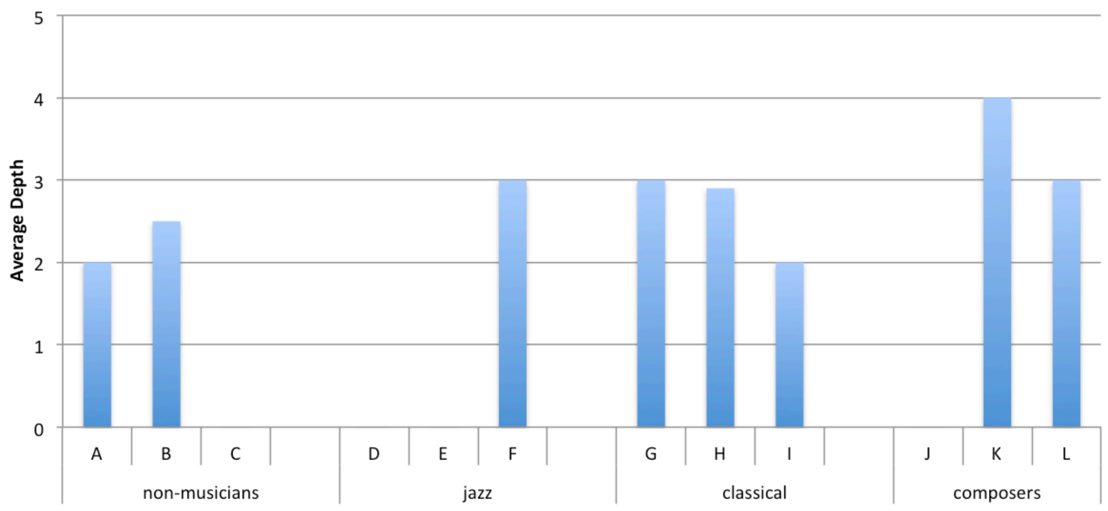


Figure 57: Average Depth of *Conditional Statements*, by participant

Parallelism

As seen in Figure 58, *parallelism* appeared in just over half of the participant projects. Participants used *parallelism* to activate between 2 and 4 different distinct programs with a single trigger. Two participants had a pair of *user interaction* elements that each activated multiple programs.

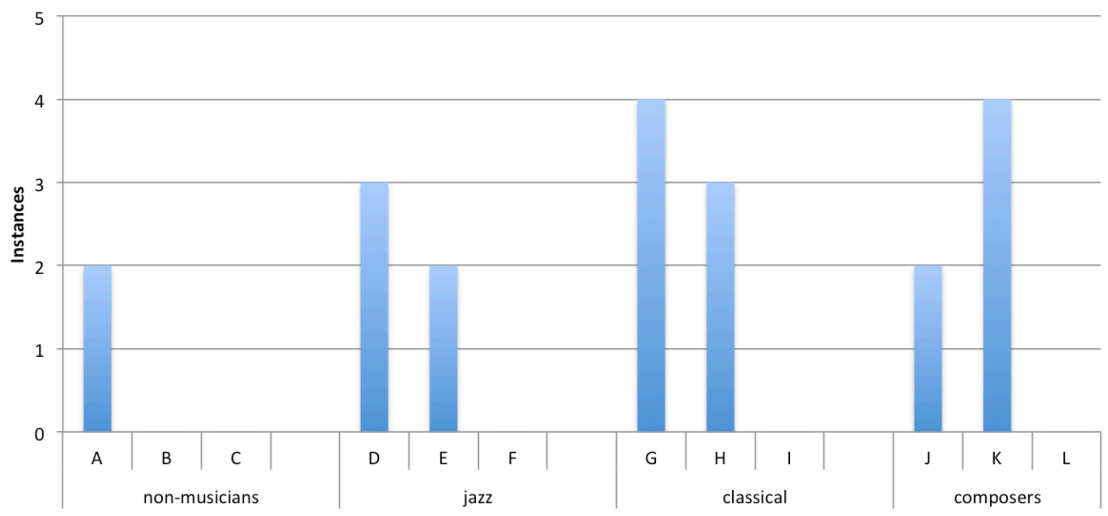


Figure 58: Number of Instances of *Parallelism*, by participant

From Figure 59, all were at a depth of 2 – that is, following a *user interaction* element.

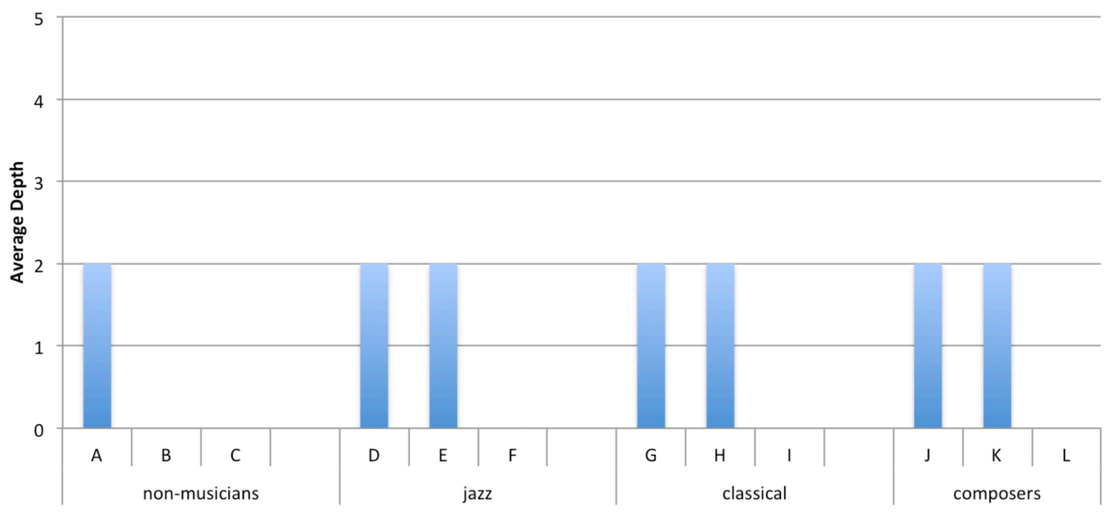


Figure 59: Average Depth of *Parallelism*, by participant

Sequences

Sequences do not correspond to any Scratch object in particular – rather, I define *sequences* as actions triggered in a certain order by means other than simply being coded sequentially. Seen in Figure 60, five participants included *sequences* of this kind in their final projects. Three projects used Scratch's *wait* object to create particular audio outputs. One of these projects did so across three distinct programs that were triggered simultaneously. Two projects used Scratch's *glide* object – this object moves the sprite a given distance or to a particular location over a prescribed amount of time. Use of the sprite was not something I anticipated, but several participants did exactly that, including creating new sprites, for a variety of reasons to be discussed later. While using glide rather than instantaneously relocating a sprite may have been as much or more for visual aesthetics than to introduce time into a programmatic sequence, this accomplished it nevertheless.

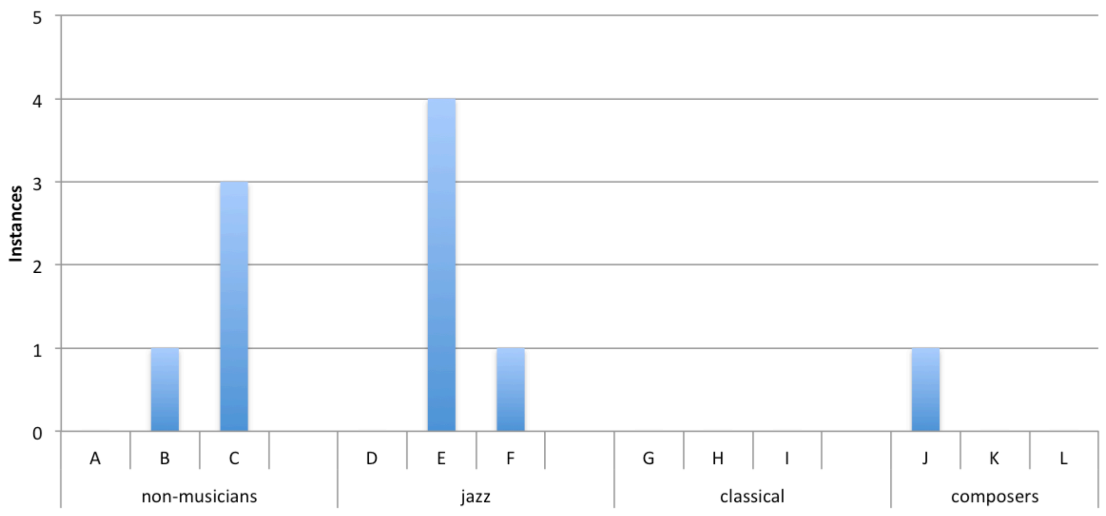


Figure 60: Number of Instances of *Sequences*, by participant

Sequences were employed at a variety of depths across participant projects, as seen in Figure 61.

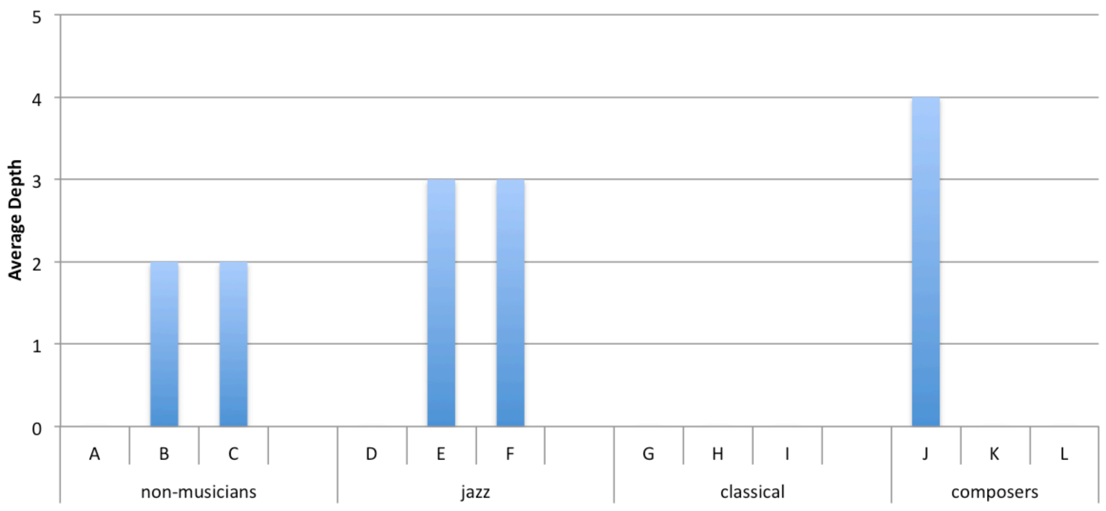


Figure 61: Average Depth of *Sequences*, by participant

Operators

Seen in Figure 62, six participants used *operators* in their projects. This includes all participants in the composer group as well as one dramatic outlier (Participant D). Two participants used greater- or less-than objects to check mouse position. Five used mathematical *operators*: three divided random integers into fractional values for the purpose of beat duration, two added random numbers or variables to prescribed note values (Participant D did so 20 times), and three used addition and division to rescale mouse positions into new variables.

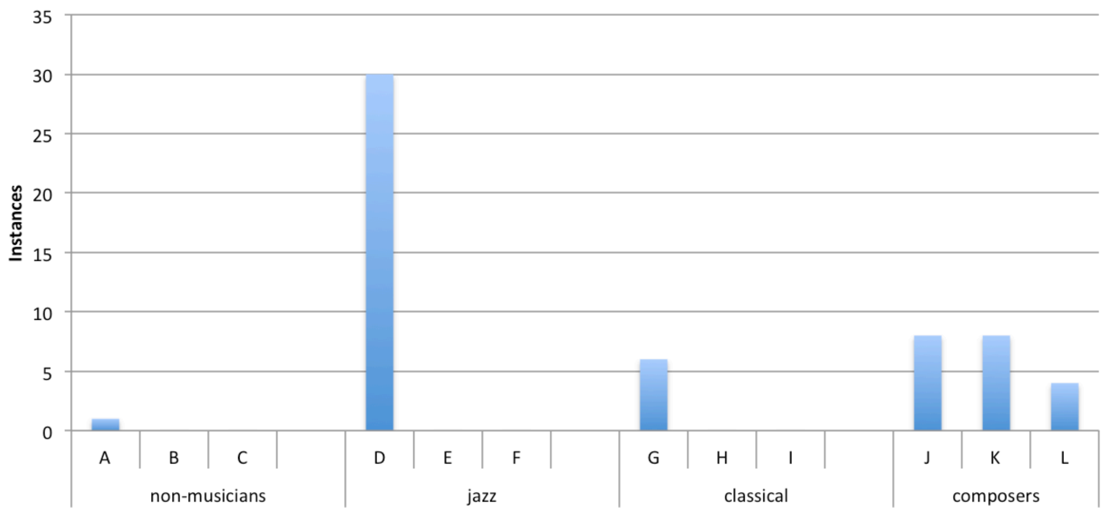


Figure 62: Number of Instances of *Operators*, by participant

The concepts preceding this one are what I refer to as "container" concepts. They encapsulate program outputs (like a *loop*) or control the activation of program outputs (like a *conditional statement). Operators*, and the concepts that follow, tend to interact with these "container" concepts (controlling the number of repeats in a *loop*, for instance) or interact with output directly (controlling a particular integer pitch, for instance). As such, these latter concepts tend to appear more deeply in programs. As seen in Figure 63, no project employs *operators* with an average depth lower then three.

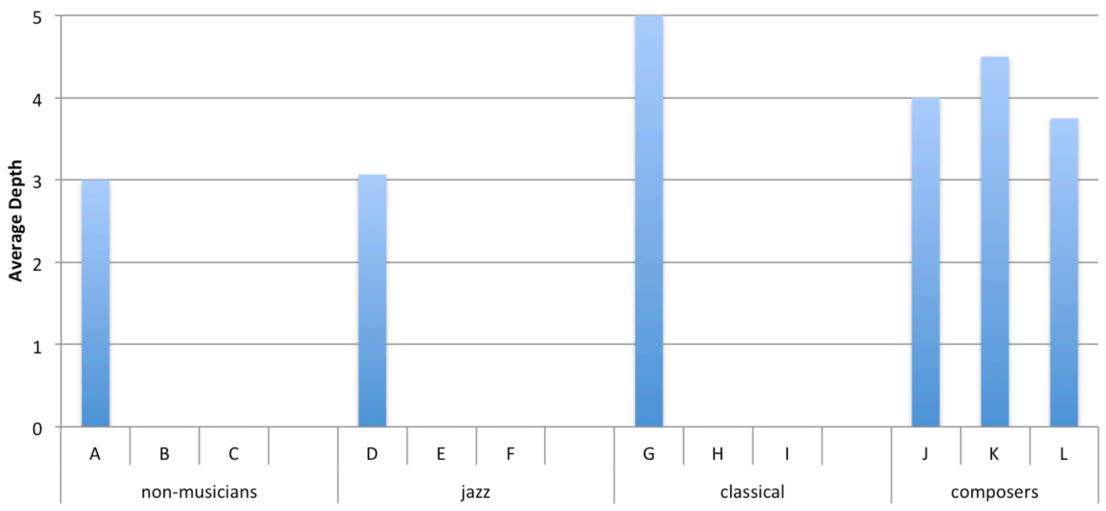


Figure 63: Average Depth of *Operators*, by participant

Boolean Logic

Boolean logic was the least occurring CT concept in participants' programs. Seen in Figure 64, it appeared twice in Participant G's project, both times inside a *conditional statement* and utilizing AND so as to define the left and right (or upper and lower) bounds of the Scratch sprite window, before values within the window were rescaled.

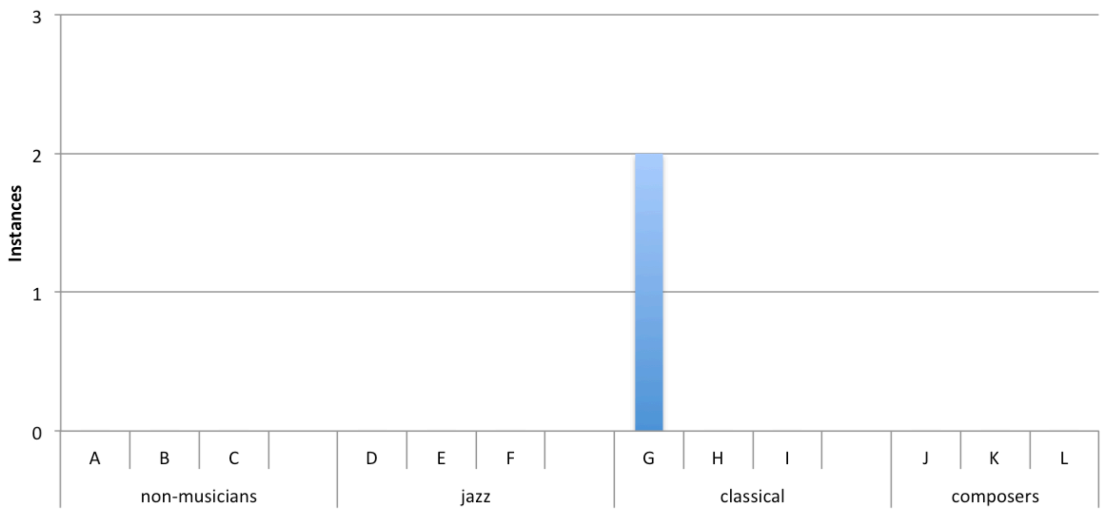


Figure 64: Number of Instances of *Boolean Logic*, by participant

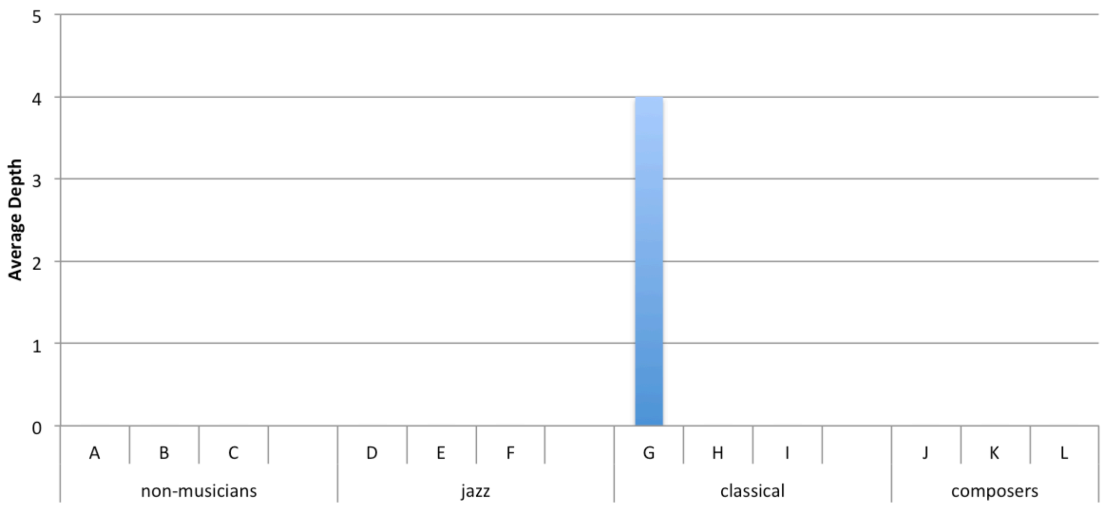


Figure 65: Average Depth of *Boolean Logic*, by participant

Random Numbers

Seen in Figure 66, five participants used *random numbers* in their final projects, including all participants in the composer group. One project used 20 *random number* elements, to turn the sprite a random number of degrees and to play notes for a random number of beats (by dividing a random integer by 10 or more). Others used this random beat tactic as well as random instrument choice. While only using five *random number* elements, Participant D's project was far and away the most randomly determined, using random waits, repeats, adjustments to rescaled variables, and adjustments to prescribed note choices.

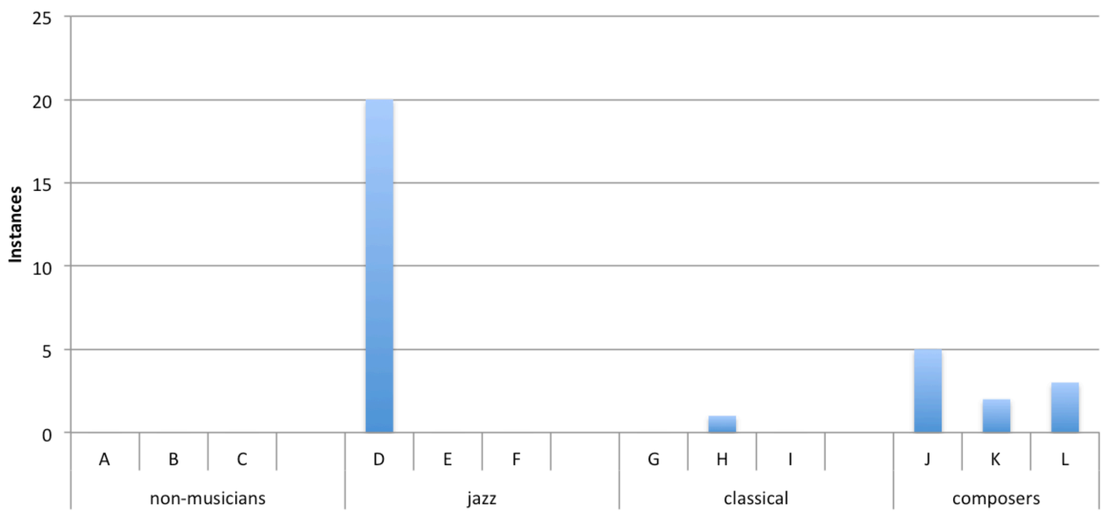


Figure 66: Number of Instances of *Random Numbers*, by participant

As similar to the depth of operators above, average depth for *random numbers* shown in Figure 67 is relatively high compared with other concepts.

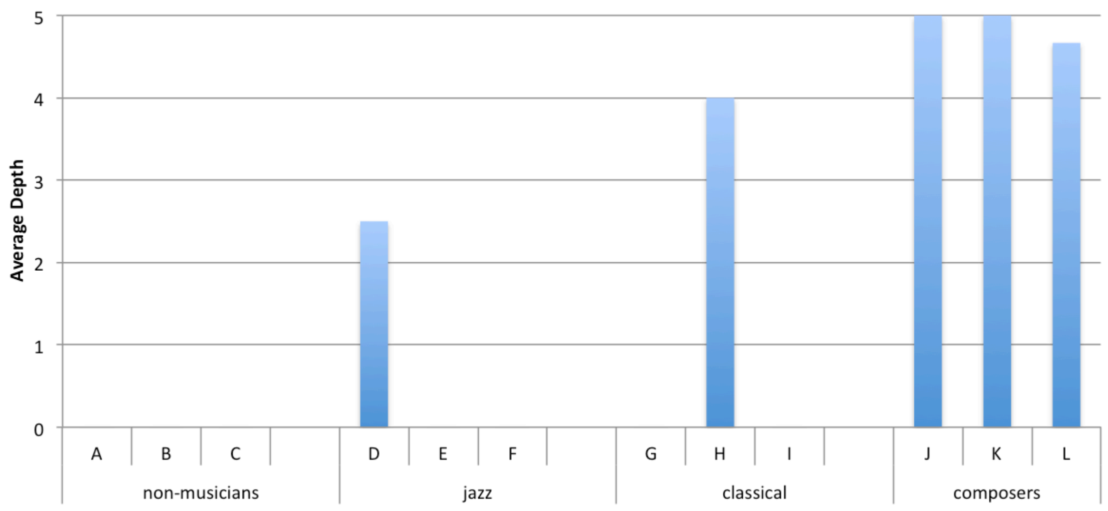


Figure 67: Average Depth of *Random Numbers*, by participant

Variables

Three participants included *variables* in their final projects, shown in Figure 68 below. In all cases, *variables* were used in rescaling mouse position into a variable more useful for Scratch (often from 0-127, the range of available pitches and instruments in Scratch). In two cases, participants created new x and y variables (one using the range of the entire screen, the other the Scratch sprite window). The third worked with x only, using the full screen. These new variables were used to choose notes or instruments. Participant G used *variables* based on the bounds of the Scratch window to modulate prescribed note values.

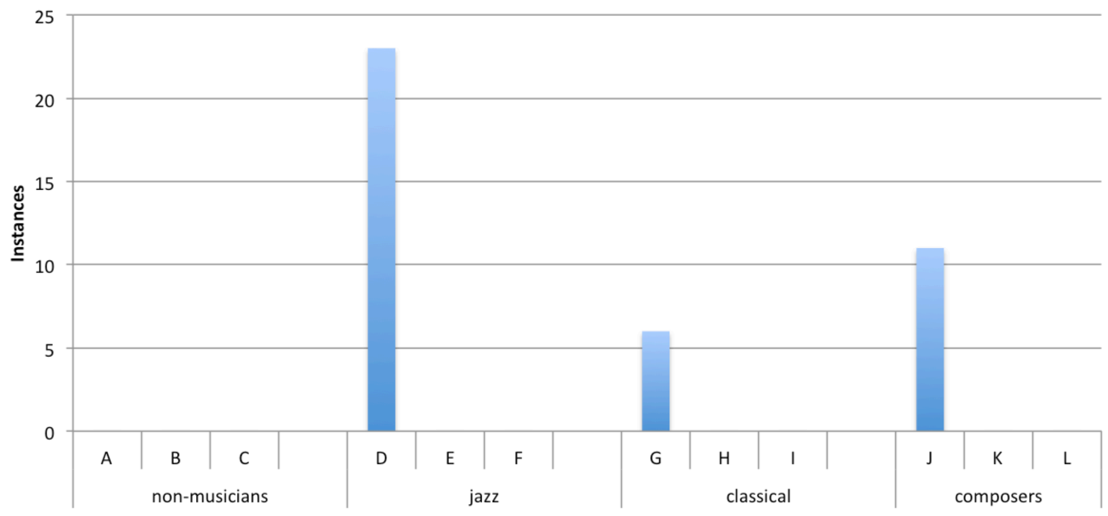


Figure 68: Number of Instances of *Variables*, by participant

Variable depth values, seen in Figure 69, are relatively high and very close for all three participants.

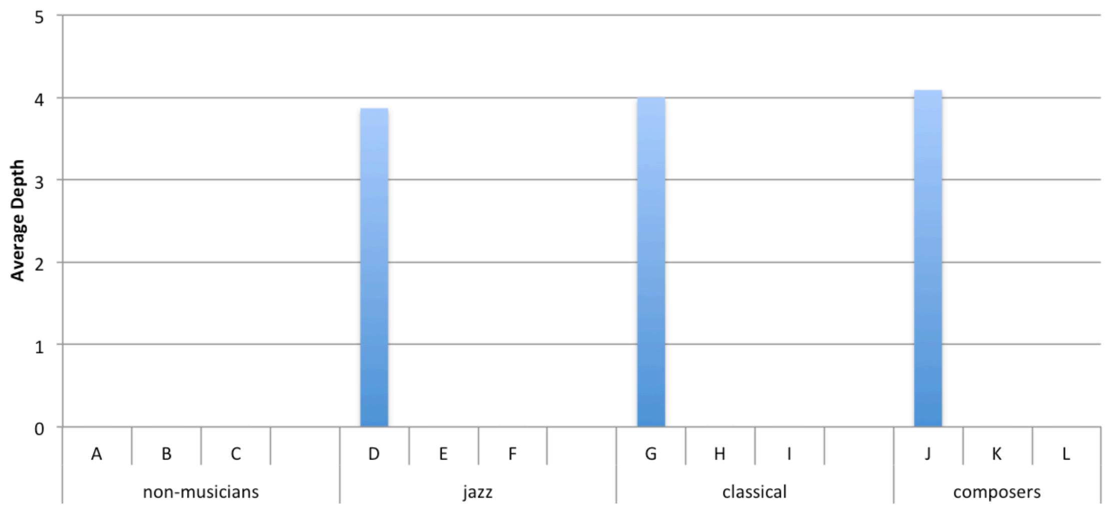


Figure 69: Average Depth of *Variables*, by participant

Lists

Lists appeared in two participant projects, as seen in Figure 70. Despite *lists* arguably being a subcategory of *variables*, use of the two turned out to be mutually exclusive. Participant F's project played three notes from a list, one after the other; Participant H played various notes from a list depending on key presses. As the list values do not change and are only referenced once, these uses of *lists* may be somewhat redundant. That is, participants coded "play note #4 of list A" [with note #4 of list A = 74] in lieu of simply "play 74".

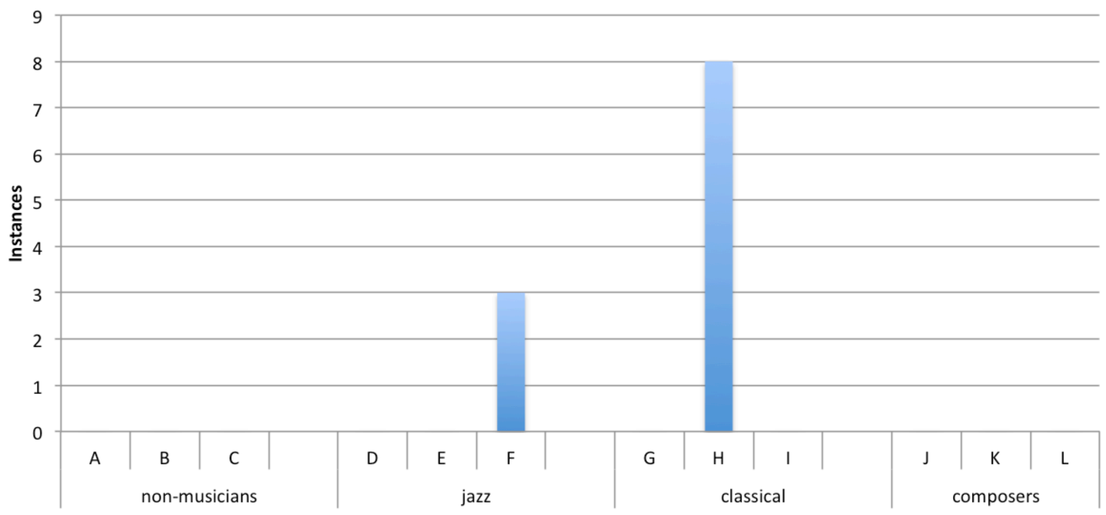


Figure 70: Number of Instances of *Lists*, by participant

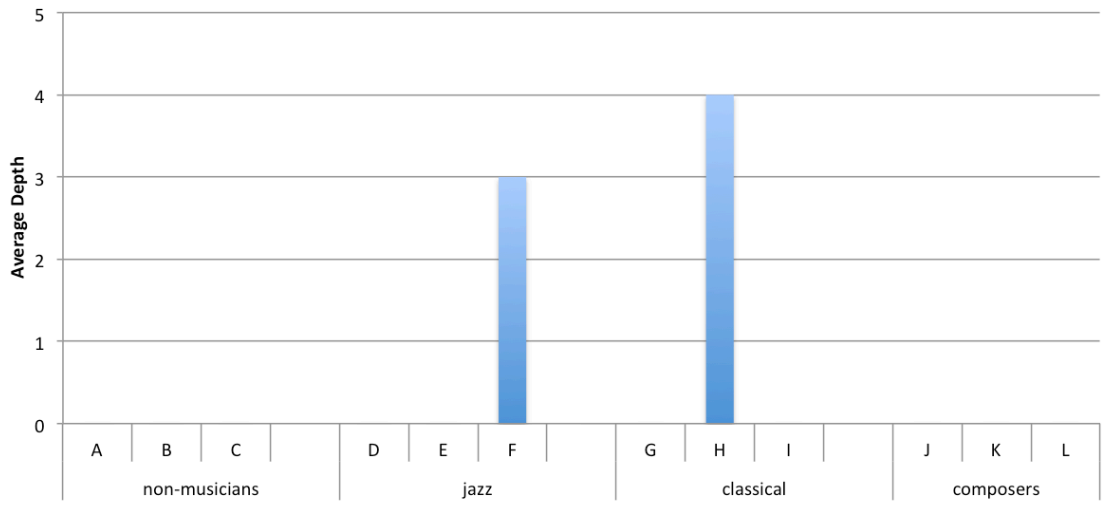


Figure 71: Average Depth of *Lists*, by participant

Overall Use of Computational Thinking Concepts

Figure 72 displays the aggregate frequency of use of CT concepts across all final projects.

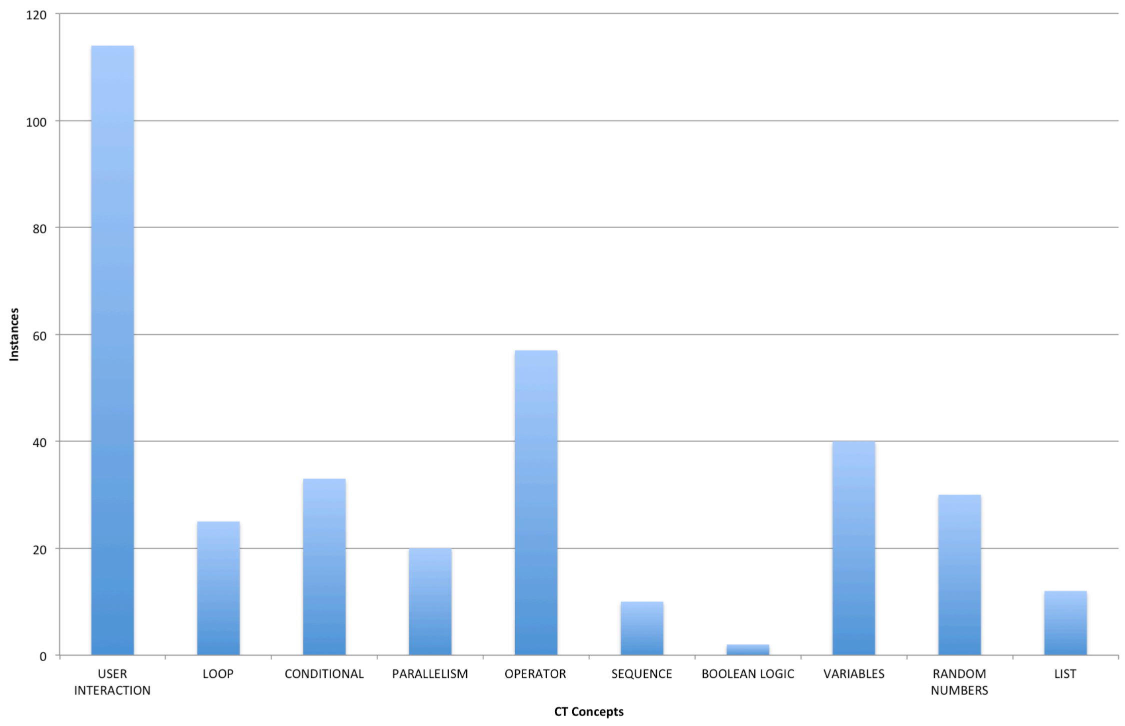
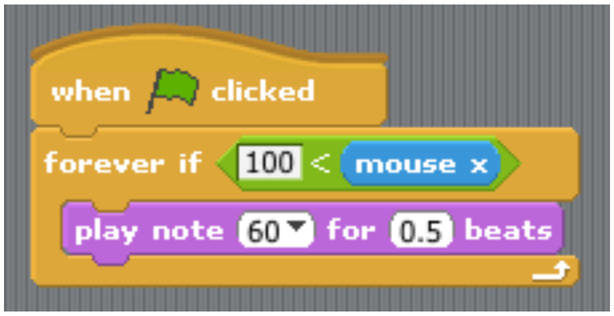


Figure 72: CT Concepts Used, Instance Across All Participants

Given that *user interaction* was one of the two requirements of the programming task, it is no surprise that *user interaction* dominates here. Virtually every distinct program commences with *user interaction* elements and in many cases this initial interaction simply starts a program to listen for further *user interaction*. For instance, the program that follows in Figure 73 is activated using a *user interaction* element; later in the program a *conditional statement* checks the mouse position, an additional *user interaction* element.



```
when [flag] clicked
forever if <(100) < (mouse x)>
    play note (60 v) for (0.5) beats
```

Figure 73: User Interaction Program

The next two highest computational thinking concepts used in programs are *operators* and *variables*, which are often used in similar ways. Participants frequently used *operators* in rescaling mouse x and y values to create *variables* as well as to add or subtract variable or random values from prescribed ones. As discussed earlier, these non-container concepts can be more easily inserted deeply into programs in greater numbers. This is reflected in Figure 74 that follows, as the large number of computational thinking elements in Participant D’s project is dominated by *variables*, *operators*, and *random numbers*. With the exception of Participant D, projects by the classical and composer groups show more instances of computational thinking concepts than the jazz and non-musician groups.

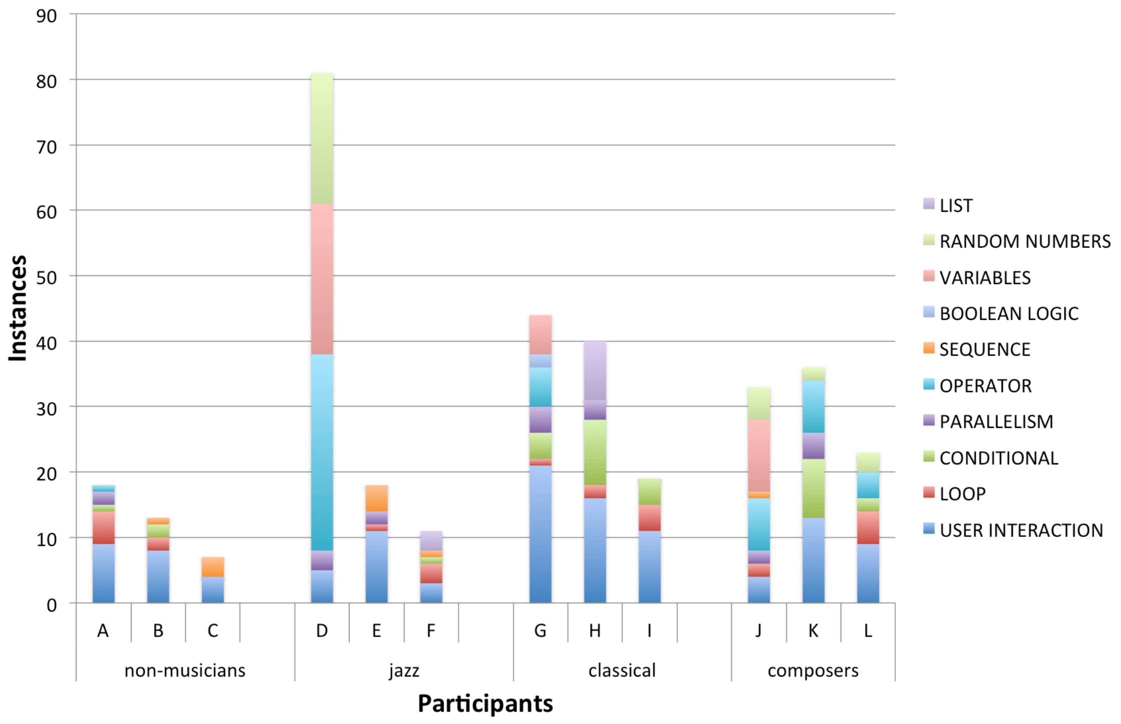


Figure 74: Number of Instances and Type of CT Concepts Used, by participant

Summary

While final projects exhibited a great deal of diversity between and within groups, several results stand out in differentiating the final projects of participants of different backgrounds. On average, participants from the composer group produced projects with the lowest number of distinct programs and the greatest overall maximum depth. All composers also used *operators* and *random numbers*. On average, participants from the non-musician group produced projects with the lowest maximum nested depth. Participants from the classical group used higher numbers of *user interaction* (at high depth) and all classic musicians used *conditional statements*.

MUSICAL BACKGROUND AND NOVICE PROGRAMMING

Different kinds of musical backgrounds did play a role in novice programming. Musical talk demonstrated how musical backgrounds in general provided learners an entry point to the numeracy involved in programming musical projects and a repertoire of musical ideas that could inspire project ideas as well as programming challenges. Participants included musicians trained as jazz and classical musicians and composers. Comparing participants by background in terms of how they produced their projects, the most pronounced differences were apparent among the composer group; their projects could be considered the most carefully crafted, as these participants worked the longest, on average, added Scratch objects relatively slowly, removed the most objects in the course of their work, and produced final projects of the greatest depth.

There were also trends based on musical background in how participants used various Scratch objects and computational thinking concepts, involving the composer and classical groups. All members of the composer group used sensor and operator Scratch objects and employed operator and random number computational thinking concepts. The use of high-depth concepts such as these is in line with the programming process of participants in the composer group. All members of the classical group used sensor Scratch objects and *conditional statement* computational thinking concepts, both of which lend themselves towards interactivity. Participants from the classical group also used *user interaction* computational thinking concepts in greater numbers and at higher depth than most other groups.

Chapter 6: Discussion

In this chapter, I will discuss my findings in relation to chunking, tinkering, and the importance of context. I will also discuss connections between my composer participants and design, and implications for practice and provide suggestions for further research.

CHUNKING

Research has shown how organizing knowledge into meaningful chunks divides novice and expert programmers. This study was motivated in part by curiosity about how musical content knowledge and the organization thereof might influence programming practice. Something as relatively elementary as a D minor 7th chord contains multiple harmonic and intervallic relationships; meanwhile, a jazz musician discussing blues or rhythm changes is referencing a time-based sequence of interrelated chords. A myriad of compositional approaches delve into even deeper layers of complexity.

Participants' use of musical talk demonstrates one potential connection between the chunking of musical knowledge and chunking in the context of programming. In some cases, musical terminology used by the participants could, with some work, be musically expressed in Scratch. For instance, at one point a participant opts to create a *glissando* – that is, a slide from one pitch to another. On an instrument without clearly separated pitches, such as a violin or trombone, this would be a literal slide from one note to another. On a piano, meanwhile, a *glissando* would involve playing intermediate notes as fast as possible. The participant took this latter tact, though given Scratch's capacity for notes of incredibly short duration, the net effect was closer to the trombone's literal glissando. By moving the mouse, the participant could slide from one note to another relatively smoothly. In short, implementing this second-nature musical technique became a non-trivial but solvable Scratch task.

This observation matches with prior research on computer science instruction aimed at trained musicians (Meyers, Cole, Korth, & Pluta, 2009). In that case, instruction was developed to connect specific programming concepts with musical and compositional concepts. These examples were of much greater complexity that this *glissando* example; for instance, one activity involved using arrays to simulate the interconnected rhythms of Steve Reich's *Clapping Music*.

This dissertation was motivated in part by a curiosity about how programming-friendly musical concepts might come to the fore in the context of a free-form programming activity. While the musical concepts that did appear were less sophisticated than those that might be employed in a more premeditated context, they did appear to play some role in helping learners program.

TINKERING

Much of the participants' programming activity aligns with tinkering, a cycle of learning and making frequently involving trial and error (Dorn & Guzdial, 2010), play (Petre & Blackwell, 2007), and a lack of well-defined goals (Petre & Blackwell, 2007). Trial and error was a virtual constant and despite occasional periods of sustained coding, participants tested their projects quite frequently. Many participants did not appear to have larger project goals in mind at any point in the entire session, producing many somewhat disconnected discrete programs; only one began with a concrete plan (make a piano) and executed it. Others made various decisions about directions in which to proceed, but this simply led to a deeper layer of tinkering. One participant summed this approach up in declaring, "I just want to press a button and have it do a bunch of stuff."

Participants would settle on and perhaps refine some mode of interaction, button-pressing or mouse-moving for instance, and then explore the wild possibilities that could be coaxed out of this interaction. Even the participant who threw out an entire project midway through and began again had essentially reached this point, going on to produce a new interface without a succinct plan for the program's output. Many produced ecstatic, often random, waves of sound that veered in delightful and sometimes comical directions with each mouse movement or button press. This is surely encouraged by Scratch's impenetrability to errors. If a program can be assembled, it can be executed. This strength, along with Scratch's ability to execute while editing, are powerful features encouraging tinkering and engaged programming learning.

CONTEXT & COMFORT

The role of musical talk highlights the value of context in a novice programming experience. Musical backgrounds were useful in approaching the numerical work of programming music in Scratch as well as providing a library of musical concepts around which participants could build programming goals and organize programming tasks.

However, the issue of comfort with programming touches on context from the angle of computer experience. Both of the oldest participants advised me (somewhat humorously) to temper my expectations, in light of a self-reported unease with computers in general. Both had no trouble overall, with one taking to the Scratch environment as quickly or more so than any other participants. The other had many thoughts as he became comfortable with the environment:

> If someone put a musical instrument in front of me that I'd never seen before and said 'you blow here or you pluck this or you press this,' and you learn it through muscle memory as well as creating a brain picture and using your short term memory. So now you've showed me all of these lists and menu items, but it's not tactile so I'm trying to remember a mental picture of what you did. Since it's this

> computer 2-dimensional screen thing it's completely floating around. It has nothing I can touch. So I'm trying to recreate it step by step and it's completely trial and error. Like walking through a room and then they turn out the lights and I walk through it again and then I bang into everything. (Participant F)

While other participant created new sprites essentially as a novelty, doing so was a turning point for this participant: chunks of Scratch code seemed to take on more substance as they became associated with clickable and moveable Scratch icons. As he proceeded, his metaphor or working model moved from music in the abstract to theater, with each sprite essentially become a stage performer with a particular role to play:

> This is a stage, so I can actually see the performers and think of leitmotifs and characters associated with sound – that's where I'm going to with it. Since this isn't a musical instrument per se, it's more like a stage and I'm organizing stuff on it. That draws me in. This gives me a visual and then I can make these icons interact with sounds and that stimulates my imagination. This list of things – ugh. My imagination shuts down and my desire to do anything shuts down. (Participant F).

This observation offers a new perspective on the importance and potential of context. Rather than using music as a conceptual framework around which to structure programming tasks, as some participants did, this participant literally used the Scratch stage as a proxy for a real-world performance space. This use of a very concrete mental model is an area I'll consider in my discussions of future research.

COMPOSERS AND DESIGN

Based on analysis of the programming processes and final programming projects of the participants, participants in the composer group stood out in a number of ways. They worked longer while adding Scratch objects more slowly, and removed more objects that participants from the other groups. The projects they produced were of greater depth and utilized more numerical computational thinking concepts. An explanation for the differences may be found in the nature of the professional musician's work. While expertise in jazz or classical music is primarily focused on performance, a composer creates pieces of music, either in a final recorded form or notated in a way that serves as instructions for performance by other musicians. The work of a composer may be viewed as potentially analogous to the work of a designer in certain ways. A designer "translates an idea into a blueprint for something useful" (Design Council, 2014). A designer deals with "a huge number of considerations coming to bear on the design process" (Design Council, 2014). Meanwhile, a composer translates an idea into something listenable or compelling; he or she also deals with all manner of constraining considerations, from available instrumentation to the composition's intended destination and purpose (is this an evening-length piece for the symphony hall or a 30-second jingle for a television commercial?). Furthermore, many design models integrate phases of evaluation and redesign; composer participants demonstrated the most removal of objects of any participant group.

All of the composer participants were at least familiar with various mathematically or structurally rigorous compositional approaches of the 20th century, such as 12-tone serialism (in which all 12 chromatic tones are used equally in a piece of music) or aleatoric music (in which random elements are included in the details of a composition). Whether the composer participants' use of numerical Scratch objects and computational thinking concepts was out of familiarity with these ideas or simply higher mathematical aptitude is an area for further research.

PRACTICAL IMPLICATIONS

The primary practical takeaway of this research is the potential importance of context in programming learning. Presented with a tutorial focused on musical and audio elements of Scratch, non-musicians participants not only worked the least amount of time, but produced arguably less sophisticated projects of lower depth, primarily using Sound and Control objects. (One non-musician participant departed from the musical task and created a project largely focused on moving the sprite around the stage.) In addition to their experience with musical numeracy, many musical participants arrived with musical concepts in hand that could be translated into programming challenges. This is encouraging in the sense of leveraging concepts from other domains with which adults learners might be familiar as frameworks for programming tasks.

Future Research

An initial extension of this study that I would undertake would be to replicate this study with younger learners. My participant groups in this case would likely simply be musicians and non-musicians. Most of the musical concepts that participants discussed were not especially advanced and would not be out of reach for musicians of secondary school age. I would be interested to see if these new participants would leverage concepts in similar ways. Alternately, an extension of this study (with younger or adult learners) might introduce more specific musical concepts in the Scratch tutorial, such as building something based on the *glissando* example discussed earlier. Participants might be inspired to explore these ideas or related ideas further.

A second study that I would propose would look very much like this one, but would draw on Participant F's use of theater as a mental model for structuring his Scratch project (as well as other participants' impulse to move the sprite around the stage). In the study described in this dissertation, I presented the Scratch sprite primarily as a button to be clicked or as a positional marker. I invited to the participants to make something musical; in this new study I would invite participants to make the sprite perform. The Scratch tutorial examples would be built around Motion, Looks, and Pen objects rather than Sound objects. My overarching research question would have less to do with the role of participant background (though the inclusion of theater artists or choreographers would be an interesting consideration), but rather on how the focus on the sprite as a "performer" influenced their programming experiences.

CONCLUSION

This study has examined how various musical backgrounds might play a role in novice programming. I worked with nine musicians of different backgrounds and three non-musicians on projects in Scratch so as to answer this question.

This overarching research question was subdivided into three more actionable research questions. The first asked how musical concepts emerged as a scaffold for novice programmers. To answer this question I logged and coded participant talk while they worked on Scratch projects. My findings were that a musical background helped participants navigate some of the numerical work involved in their Scratch projects and provided them with a library of musical techniques and processes that often lent themselves toward realization in code.

My second research question asked what kind of patterns participants showed in their programming. To answer this question I screen-recorded participants' programming and logged all of their programming activity. Significant observations include musicians working longer than non-musicians and composers working longer overall. Composer deleted far more objects than other groups, both in absolute numbers and percentage.

My final research question asked how the final projects of the participants differed, focusing in particular on computational thinking concepts employed. Composer participant produced programs overall greater programming depth; participants employed a variety of computational thinking concepts in a variety of ways, particularly *user interaction*, *variables*, and *operators*.

This qualitative study served its purpose into examining this very domain-specific entry point into learning computer programming. While the findings above should not be generalized further, they may offer some signposts towards future research. I would like to see future research look at how musical background may scaffold programming learning in a less open-ended scenario, where definite musical goals and content align with particular aspects of programming learning, and how alternative performance metaphors might influence programming learning.

Appendix A: Research Matrix

Research Question	Data Sources	Specific data to answer this question	Analysis required	What will this allow me to say?
Big Q. Do different kinds of musical backgrounds play a role in novice programming?				
1. How do musical concepts emerge as a scaffold for novice programmers?	*1. Participants' think-aloud interviews* *2. Participant pre-survey*	*1. Transcribed interviews* *2. Participant pre-survey responses*	*1. Coding for themes* *2. Coding for themes*	*"For many participants, harmony was an entry point to introducing parallelism into their Scratch programs."*
2. What kind of patterns do participants from a variety of musical backgrounds exhibit in their programming processes?	*1. Screen video capture of participant programming* *2. Participant pre-survey*	*1. Counts of Scratch objects and playbacks* *2. Participant pre-survey responses*	*1. Changes in frequencies of particular Scratch objects and frequency of program playback* *2. Coding for themes*	*"Improvising musicians exhibited less goal-focused programming, dramatically changing their projects as they discovered and explored new features."*
3. How did the final projects of participants from a variety of musical backgrounds differ?	*1.Participant final Scratch projects* *2. Participant pre-survey*	*1. Scratch projects* *2. Participant pre-survey responses*	*1. Frequencies and arrangement of particular computational concepts* *2. Coding for themes*	*"Participants with composition training used randomization elements far more often than any other group."*

Appendix B: Direct Recruitment Letter

Dear [NAME],

As you may or may not know, I am currently working on my dissertation in the Department of Curriculum and Instruction at the University of Texas at Austin. My dissertation study focuses on how musical knowledge and expertise informs the experience of novice programmers.

I would like to ask you to consider being a participant in my study. The study will involve spending no more than two hours working on a (hopefully!) fun musical project in the Scratch programming environment while talking with me about what you're working on.

If you would be interested in participating or have any questions that I could answer, please do not hesitate to get in touch with me.

Thanks,

Tom Benton

Appendix C: Consent Form

Consent for Participation in Research

Title: Musical Expertise as a Scaffold for Novice Programming

Introduction

The purpose of this form is to provide you information that may affect your decision as to whether or not to participate in this research study. The person performing the research will answer any of your questions. Read the information below and ask any questions you might have before deciding whether or not to take part. If you decide to be involved in this study, this form will be used to record your consent.

Purpose of the Study

You have been asked to participate in a research study about musicians and computer programming. The purpose of this study is to examine how musical knowledge informs beginning computer programming.

What will you be asked to do?

If you agree to participate in this study, you will be asked to:

- Complete a survey describing your musical background
- Work on a musical project in the Scratch programming environment while answering general questions about your work.

This study will take approximately 2 hours and will include 12 study participants.

Your participation will be audio recorded and your programming will be screen-recorded.

What are the risks involved in this study?

There are no foreseeable risks to participating in this study.

What are the possible benefits of this study?

You will receive no direct benefit from participating in this study; however, this research may provide insights that benefit programming instruction in the future. You may also find it an engaging introduction to computer programming.

Do you have to participate?

No, your participation is voluntary. You may decide not to participate at all or, if you start the study, you may withdraw at any time. Withdrawal or refusing to participate will not affect your relationship with The University of Texas at Austin (University) in any way.
If you would like to participate please return this form to the researcher at the scheduled study meeting. You will receive a copy of this form.

Will there be any compensation?

You will not receive any type of payment for participating in this study.

How will your privacy and confidentiality be protected if you participate in this research study?

The study is confidential. All data will be assigned a pseudonym rather than your name. Pre-surveys will be stored securely in a locked filing cabinet. All digital data (audio and screen recordings, final Scratch projects, and logged pre-survey data) will be stored on an encrypted external drive. Your contact information obtained during the scheduling process will be destroyed after the experiment is completed. Data will be kept for one year. After this time, pre-surveys will be shredded and digital data will be deleted.

If it becomes necessary for the Institutional Review Board to review the study records, information that can be linked to you will be protected to the extent permitted by law. Your research records will not be released without your consent unless required by law or a court order. The data resulting from your participation may be made available to other researchers in the future for research purposes not detailed within this consent form. In these cases, the data will contain no identifying information that could associate it with you, or with your participation in any study.

If you choose to participate in this study, you will be audio recorded. Any audio recordings will be stored securely and only the research team will have access to the recordings. Recordings will be kept for one year and then erased.

Whom to contact with questions about the study?

Prior, during or after your participation you can contact the researcher Tom Benton at 512-293-4509 or send an email to tombenton@utexas.edu for any questions or if you feel that you have been harmed.

This study has been reviewed and approved by The University Institutional Review Board and the study number is **[STUDY NUMBER].**

Whom to contact with questions concerning your rights as a research participant?

For questions about your rights or any dissatisfaction with any part of this study, you can contact, anonymously if you wish, the Institutional Review Board by phone at (512) 471-8871 or email at orsc@uts.cc.utexas.edu.

Participation

You have been informed about this study's purpose, procedures, possible benefits and risks, and you have received a copy of this form. You have been given the opportunity to ask questions and you have been told that you can ask other questions at any time. You voluntarily agree to participate in this study. By verbally consenting to participate in this study, you are not waiving any of your legal rights.

Appendix D: Pre-Survey

Name:

1. Do you read music? **Y / N**

2. Do you play an instrument or sing? **Y / N**

2a. If yes, please list (if more than 1, please circle a primary, if there is 1):

3. Do you improvise on an instrument? **Y / N**

4. Do you compose music? Y / N

5. Please fill out the following table describing your musical study:

Area of Study	**Did you study this area in college (Y / N)?**	**Please check if you earned a degree in this area:**
Instrumental or Vocal Performance		
Music Theory		
Composition		
Jazz		
Other (please list below):		

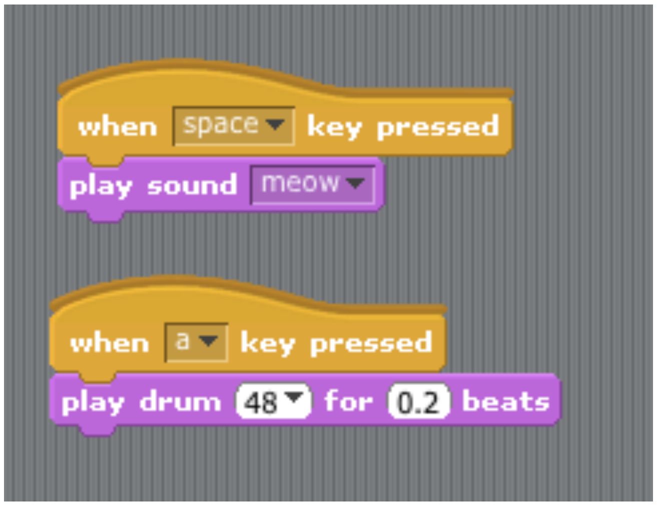
6. Please briefly describe any professional and/or personal musical activities that you participate in currently:

7. Have you done any computer programming? **Y / N**

7a. If yes, please describe:

Appendix E: Scratch Sample Programs & Reference Sheet

1. User Interaction

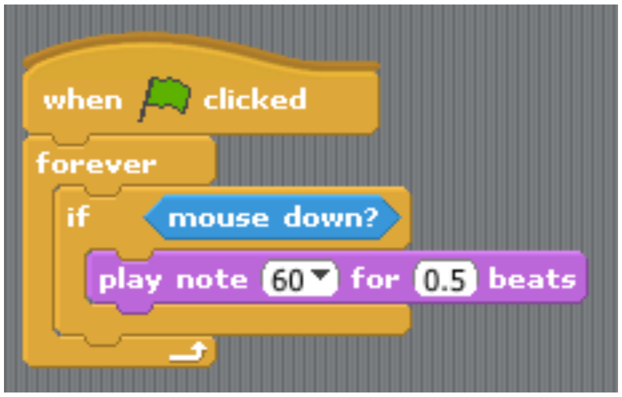


```
when [space] key pressed
play sound [meow]

when [a] key pressed
play drum (48) for (0.2) beats
```

This simple program uses the keyboard to play one audio sample and one drum sound.

It uses objects from the CONTROL and SOUND tabs.

2. Program Structure & Mouse Button



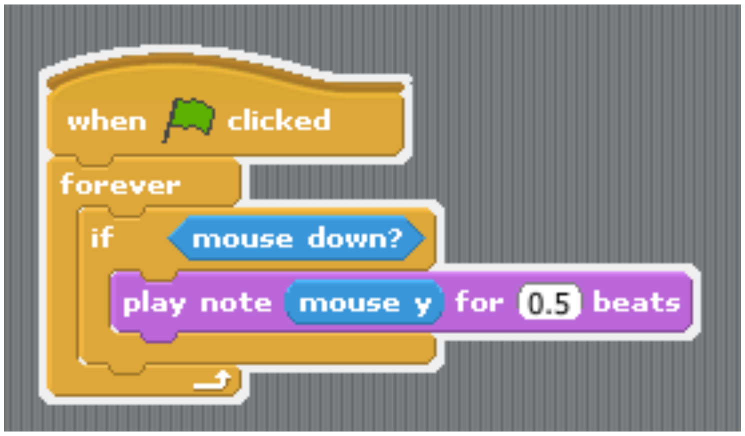
```
when [flag] clicked
forever
    if <mouse down?>
        play note (60) for (0.5) beats
```

Using the Flag click and ‘forever’ object will keep your program repeatedly executing your code.

This program uses the mouse button to play a single note.

It uses objects from the CONTROL, SOUND, and SENSING tabs.

3. Mouse Position

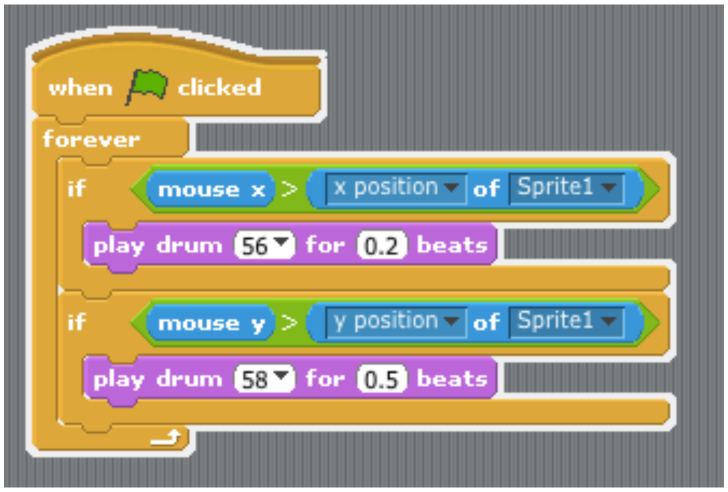


```
when [flag] clicked
forever
  if mouse down?
    play note mouse y for 0.5 beats
```

This program uses the mouse button to play a note based on the mouse position. (Mouse position appears at the bottom of the Scratch window.)

It uses objects from the CONTROL, SOUND, and SENSING tabs.

4. The Sprite & Operations

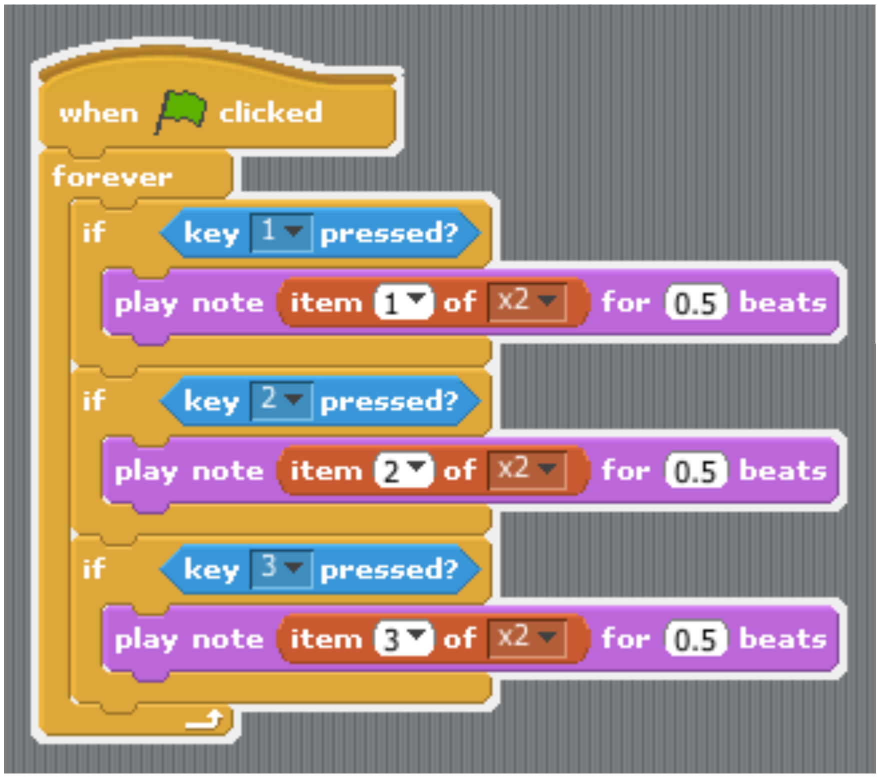


```
when [flag] clicked
forever
  if mouse x > x position of Sprite1
    play drum 56 for 0.2 beats
  if mouse y > y position of Sprite1
    play drum 58 for 0.5 beats
```

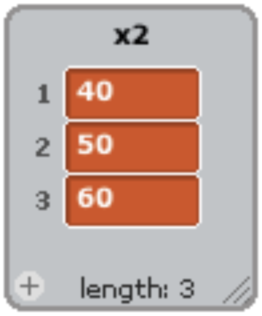
This program evaluates the position of the mouse relative to the Sprite and plays notes based on the result.

It uses objects from the CONTROL, SOUND, SENSING, and OPERATORS tabs.

5. Lists



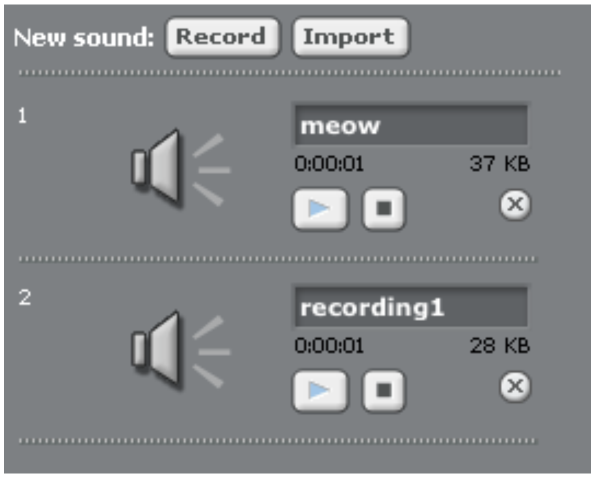
```
when green flag clicked
forever
    if key 1 pressed?
        play note item 1 of x2 for 0.5 beats
    if key 2 pressed?
        play note item 2 of x2 for 0.5 beats
    if key 3 pressed?
        play note item 3 of x2 for 0.5 beats
```



x2	
1	40
2	50
3	60
length: 3	

This program uses the keyboard to play notes pulled from a list of integers. It uses objects from the CONTROL, SOUND, SENSING, and VARIABLES tabs.

6. New Sounds



Record new sounds in the **Sound** tab and play them as in Example 1.

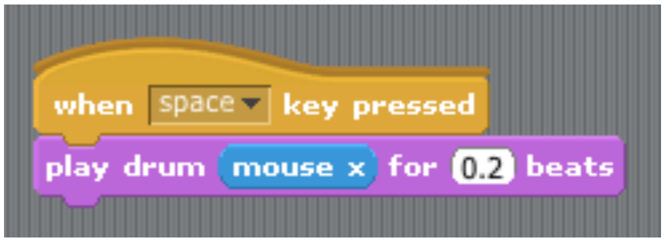
Appendix F: Computational Thinking Coding

In analyzing participant programming project, I coded for 10 computational thinking concepts. The examples below both illustrate and discuss how these concepts might be instantiated in Scratch.

Most of the concepts below could be considered present based on the successful implementation of a single Scratch object: *user interaction*, *operators*, *loops*, *conditional statements*, *Boolean logic*, and *random numbers*. *Parallelism* and *sequences* are relatively easily identified but completely undefined in their scopes. Finally, *variables* must be actively ushered through a program, and *lists* much be actively accessed.

To help illustrate how these differences will be practically dealt with in coding, each of the examples below is a fully executable Scratch program. Some are simple and others are relatively complex by comparison. Many contain computational thinking concepts from prior examples in addition to the concept in question.

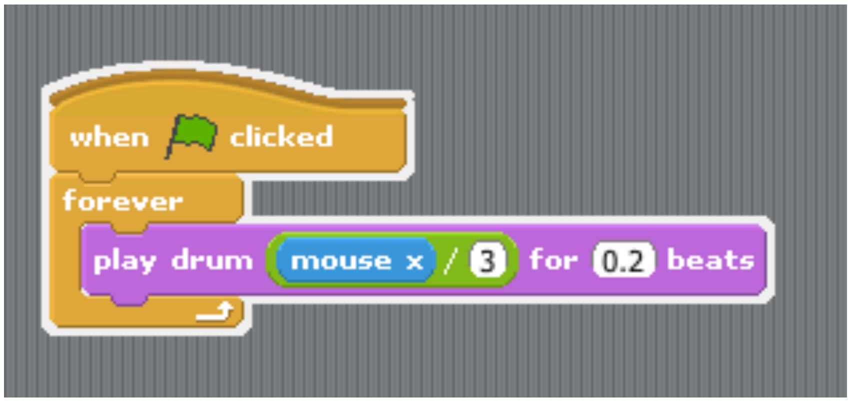
1. *User Interaction*



```
when [space] key pressed
play drum (mouse x) for (0.2) beats
```

This simple program includes two instances of *user interaction*. A user can use the keyboard to play a note, while the pitch is controlled by the position of the mouse.

2. *Operators*

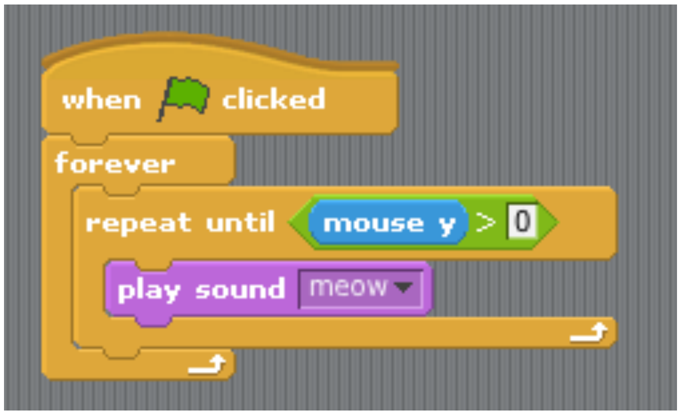


```
when [flag] clicked
forever
    play drum (mouse x / 3) for 0.2 beats
```

Performing basic numerical operations is the heart of computation. Two of the computational concepts further down this list (*random numbers*, *Boolean logic*) are technically *operators* themselves, but for this study I will consider them independently and limit *operators* to arithmetic operations and numerical comparisons. In the program above, mouse position is used to control the note's pitch, but is first scaled down to one-third to bring it more in line with playable pitch values.

Additional concept present: user interaction

3. ***Loops***

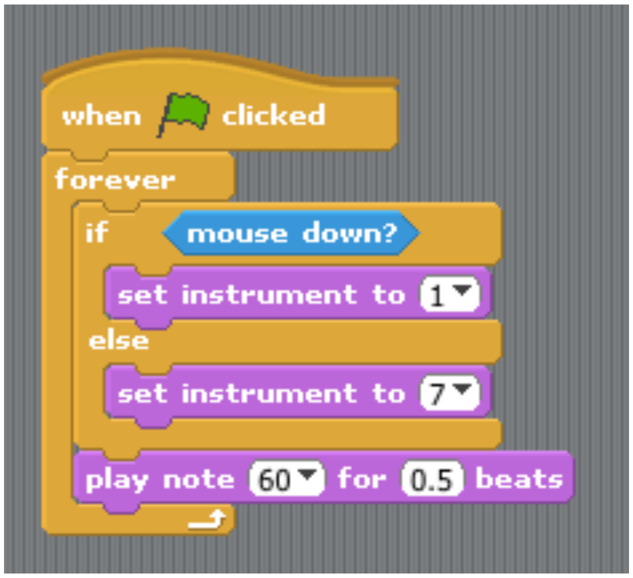


```
when [flag] clicked
forever
    repeat until <mouse y > 0>
        play sound meow
```

A *loop* simply repeats a command or series of commands a prescribed number of times or until some other condition has been met. In the program above, an audio clip is repeated until the mouse has crossed the midpoint of the Scratch window.

Additional concepts present: user interaction, operator

4. ***Conditional Statements***

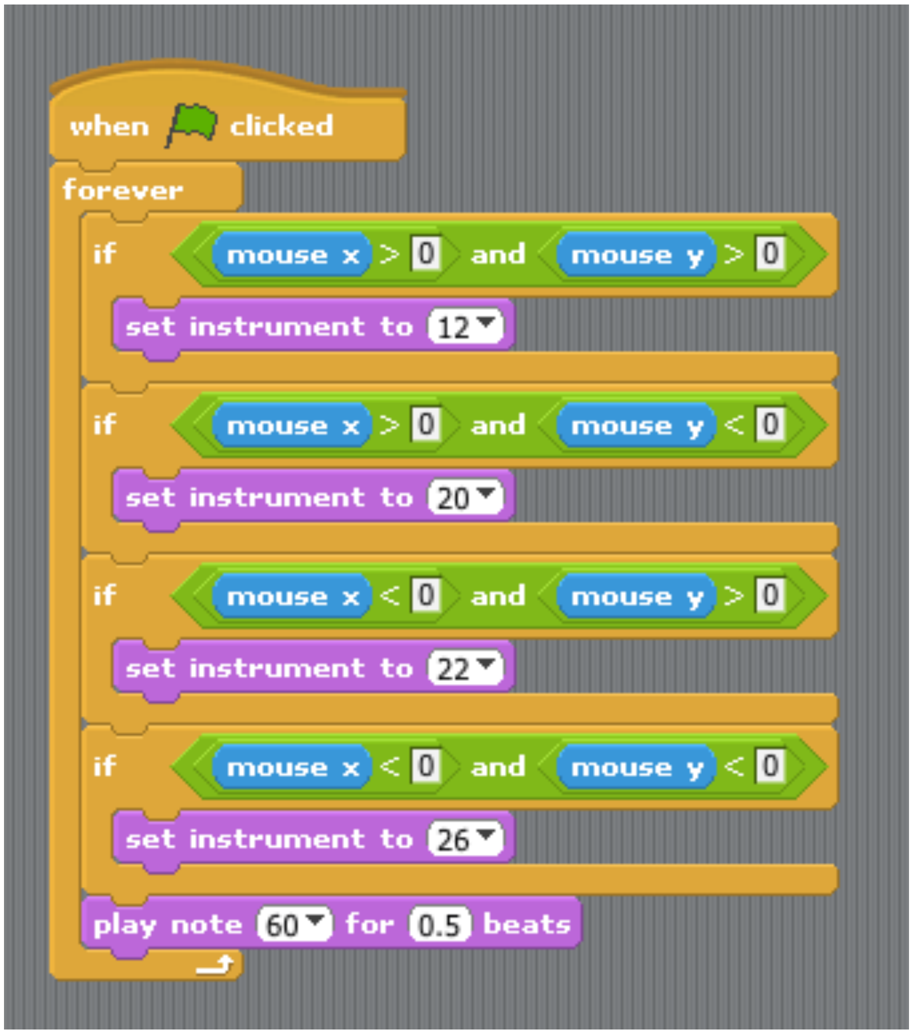


```
when [flag] clicked
forever
  if <mouse down?>
    set instrument to (1 v)
  else
    set instrument to (7 v)
  play note (60 v) for (0.5) beats
```

Conditional statements control program flow by testing whether or not a given condition has been met. In the example above, the program continually checks to see if the mouse has been pressed and sets an instrument based on the result.

Additional concept present: user interaction

5. ***Boolean Logic***

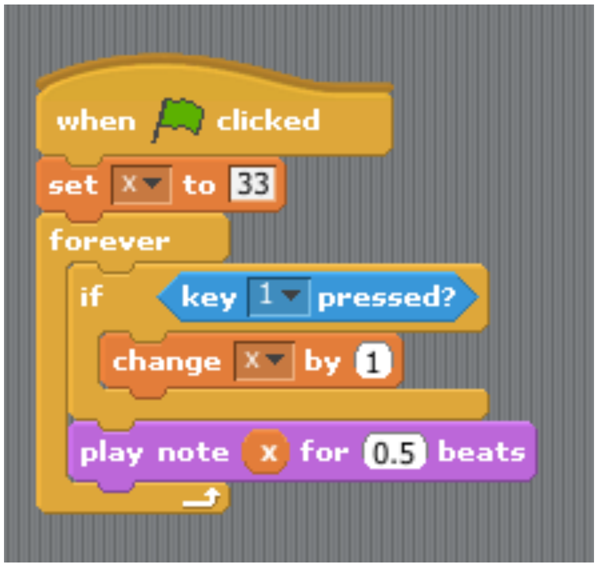


```
when [flag] clicked
forever
    if <<(mouse x) > (0)> and <(mouse y) > (0)>>
        set instrument to (12 v)
    if <<(mouse x) > (0)> and <(mouse y) < (0)>>
        set instrument to (20 v)
    if <<(mouse x) < (0)> and <(mouse y) > (0)>>
        set instrument to (22 v)
    if <<(mouse x) < (0)> and <(mouse y) < (0)>>
        set instrument to (26 v)
    play note (60 v) for (0.5) beats
```

Boolean logic involves the use of *and*, *or*, and *not*, typically in the context of a conditional statement. The program above uses a series of *and* conditionals to divide the screen into four quadrants, with each assigning a different instrument tone.

Additional concepts present: user interaction, operators

6. *Variables*

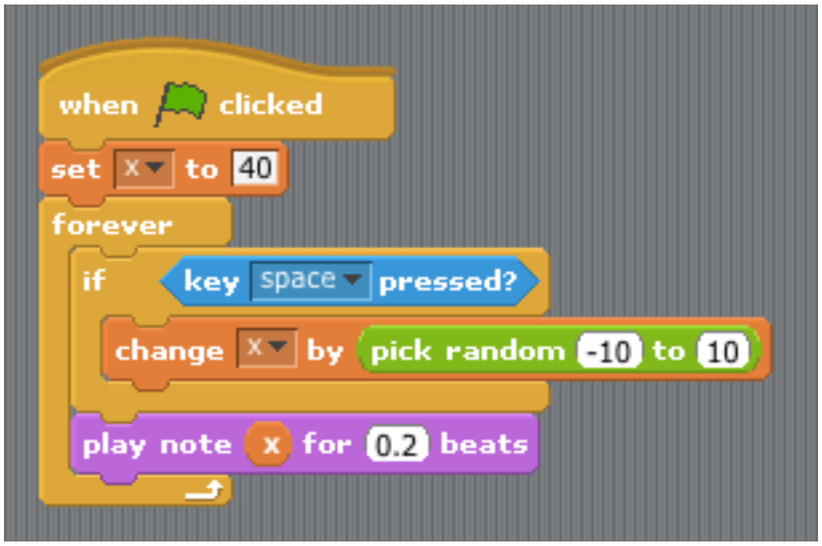


```
when [flag] clicked
set x to 33
forever
    if key 1 pressed?
        change x by 1
    play note x for 0.5 beats
```

A *variable* is treated like a number in Scratch but its value can be changed while the program executes. In the program above, a variable *x* is first set to an initial value and then used to define the pitch of an output note. Each key press increases the value of the variable and the pitch.

Additional concepts present: user interaction, conditional statement

7. *Random Numbers*

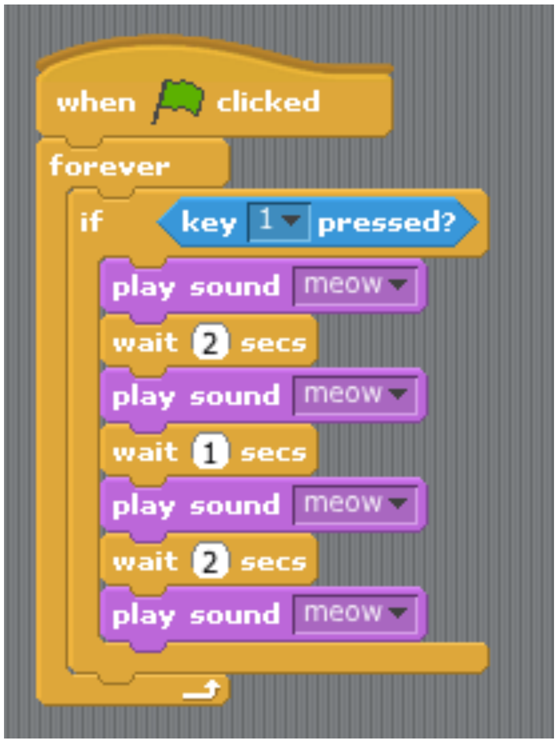


```
when [green flag] clicked
set x to 40
forever
    if key space pressed?
        change x by pick random -10 to 10
    play note x for 0.2 beats
```

Scratch can generate random integers in a range defined by the user. In the example above, a variable is changed by a random amount and played as a musical note each time the space bar is pressed.

Additional concepts present: user interaction, conditional statement, variable

8. ***Sequences***

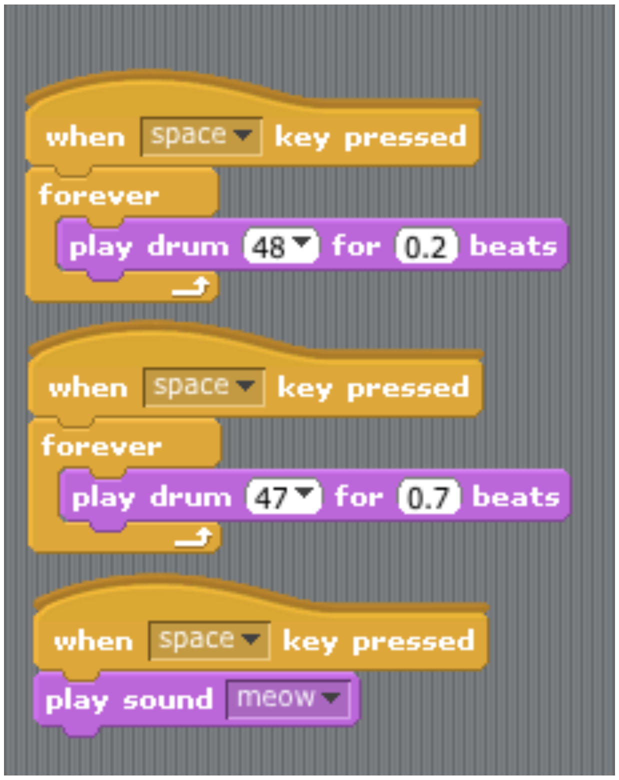


```
when [flag] clicked
forever
  if <key 1 pressed?>
    play sound meow
    wait 2 secs
    play sound meow
    wait 1 secs
    play sound meow
    wait 2 secs
    play sound meow
```

Generally, a *sequence* is a series of operations that can be executed by a program; as such, any Scratch program or fragment of a Scratch program could reasonably be considered a sequence. However, for the sake of this study, I will define a sequence as code controlling the execution of a series of audio output commands beyond simply executing them one after another. For instance, in the program above, a sequence of sounds and pauses are triggered by a key press.

Additional concepts present: user interaction, conditional statement

9. *Parallelism*

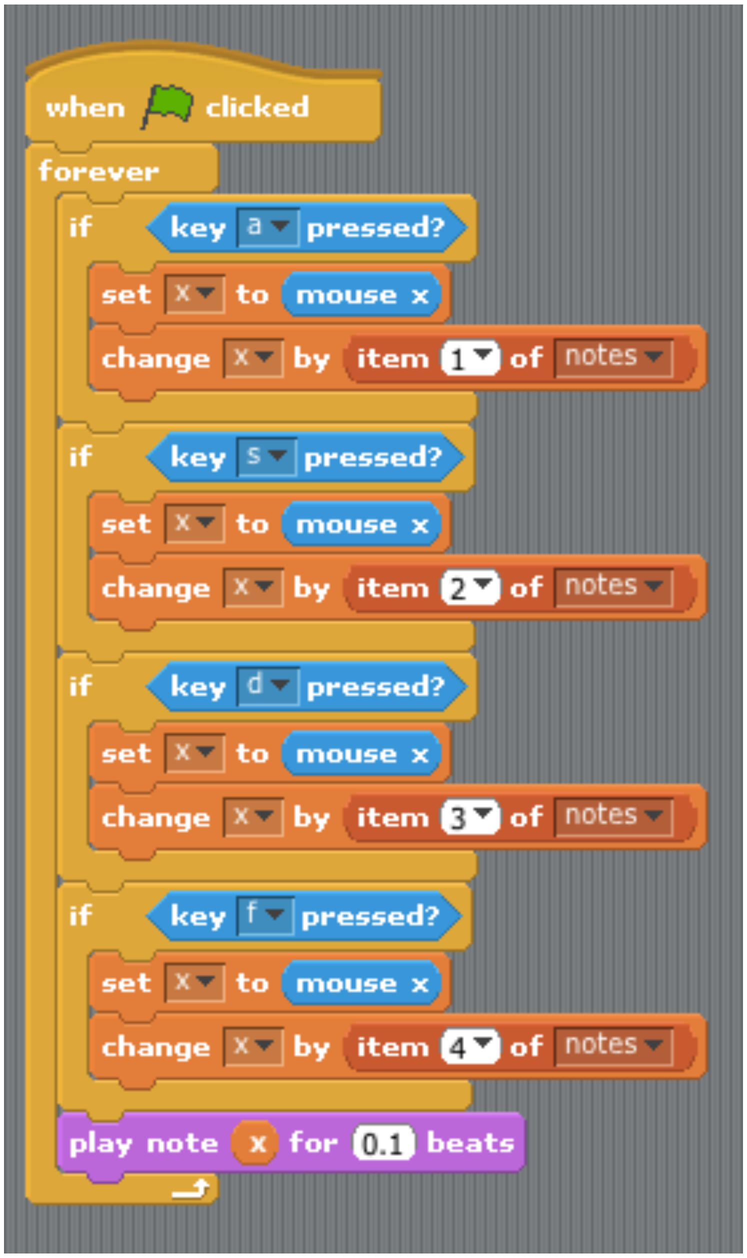


```
when [space] key pressed
forever
    play drum (48) for (0.2) beats

when [space] key pressed
forever
    play drum (47) for (0.7) beats

when [space] key pressed
play sound [meow]
```

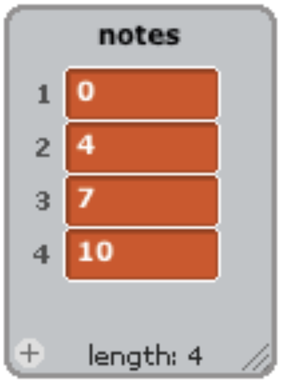
Parallelism describes sets of instructions that are executed at the same time. Because the programming window may contain multiple discreet assemblages of objects, *parallelism* can be implemented easily in Scratch. In the program above, a single key press triggers three different sound operations at once.

Additional concept present: user interaction

10. ***Lists***



```
when [flag] clicked
forever
    if <key [a] pressed?>
        set [x] to (mouse x)
        change [x] by (item (1) of [notes])
    if <key [s] pressed?>
        set [x] to (mouse x)
        change [x] by (item (2) of [notes])
    if <key [d] pressed?>
        set [x] to (mouse x)
        change [x] by (item (3) of [notes])
    if <key [f] pressed?>
        set [x] to (mouse x)
        change [x] by (item (4) of [notes])
    play note (x) for (0.1) beats
```



notes

1	0
2	4
3	7
4	10

length: 4

A *list* is a prescribed set of values that can be accessed and manipulated by a program. In the program above, the variable (or "base note") is set based on the mouse position. A series of conditional statements check for key presses that will adjust the variable by a value from the predefined table, allowing the user to move from any base note through the pitches of a dominant chord.

Additional concepts present: user interaction, conditional statement, variable

References

Amitani, S., & Hori, K. (2002). Supporting musical composition by externalizing the composer's mental space. *Journal of Information Processing Society of Japan, 42*(10), 2369-2378.

Askar, P., & Davenport, D. (2009). An investigation of factors related to self-efficacy for Java programming among engineering students. *The Turkish Online Journal of Educational Technology, 8*(1).

Astrachan, O., Barnes, T., Garcia, D.D., Paul, J., Simon, B., & Snyder, L. (2011). CS principles: Piloting a new course at national scale. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education,* 397-398.

Barfield, W., LeBold, W., Salvendy, G., & Shodja, S. (1983). Cognitive factors related to computer programming and software productivity. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting, 27,* 647-652.

Basawapatna, A., Koh, K.H., Repenning, A., Webb, D.C., & Marshall, K.S. (2011). Recognizing computational thinking patterns. *Proceedings of the 2011 ACM Conference on Computer Science Education*, 245-250.

Bayliss, J.D., & Strout, S. (2006). Games as a "flavor" of CS1. *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, 38*(1), 500-504.

Ben-Ari, M. (2001). Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching, 20*(1), 45-73.

Ben-Ari, M. (2004). Situated learning in computer science education. *Computer Science Education, 14*(2), 85-100.

Bennedsen, J. & Caspersen, M.E. (2006). Abstraction ability as an indicator of success for learning object-oriented programming? *ACM SIGCSE Bulletin, 38*(2), 39-43.

Berg, B.L. (1989). *Qualitative Research Methods for the Social Sciences*. New York: Allyn and Bacon.

Bergin, S., & Reilly, R. (2005). Programming: Factors that influence success. *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, 37*(1), 411-415.

Berland, M., Martin, T., Benton, T., Smith, C.P., & Davis, D. (2013). Using learning analytics to understand the learning pathways of novice programmers. *Journal of the Learning Sciences, 22*(4), 564-599.

Big Think. (2011, June 13). *Larry Wall: Computer Programming in Five Minutes*. Retrieved from http://www.youtube.com/watch?v=UScm9avQM1Y

Blackwell, A. & Collins, N. (2005). The programming language as a musical instrument. *Proceedings of PPIG05*.

Bown, O., Eldridge, A., & McCormack, J. (2009). Understanding interaction in contemporary digital music: from instruments to behavioral objects. *Organized Sound, 14*(2), 188-196.

Brandt, J., Guo, P., Lewenstein, J., Dontcheva, M., & Klemmer, S. (2009). Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. *Proceedings of the 2009 ACM Conference on Computer-Human Interaction,* 1589-1598.

Burton, C. (2011, July 26). In depth: How Bjork's 'Biophilia' album fuses music with iPad apps. *Wired.co.uk*. Retrieved March 12, 2014 from http://www.wired.co.uk/magazine/archive/2011/08/features/music-nature-science.

Bransford, J., Brown, A., & Cocking, R. (2000). *How People Learn: Body, Mind, Experience and School*. Washington DC: National Academies Press.

Brennan, K., & Resnick, M. (2012). Using artifact-based interviews to study the development of computational thinking in interactive media design. Paper presented at annual American Educational Research Association meeting, Vancouver, BC, Canada.

Brown, A.R. (2007). Code jamming. *M/C Journal, 9*(6), 1-4.

Brown, A.R. (2007). Software development as music education research. *International Journal of Education and the Art, 8*(6), 1-13.

Brown, R. (1981). How improvised is jazz improvisation? *Proceedings of N.A.J.E. Research, 1,* 22-32.

Brown, A.R. & Sorensen, A.C. (2009). Interacting with generative music through live coding. *Contemporary Music Review, 28*(1), 17-29.

Buechley, L., Eisenberg, M., Catchen, J., & Crockett, A. (2008). The Lilypad Arduino: Using computational textiles to investigate engagement, aesthetics, and diversity in computer science education. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,* 423-432.

Burnard, P. (1991). A method of analyzing interview transcripts in qualitative research. *Nurse Education Today, 11,* 461-466.

Butler, M., & Morgan, M. (2007). *Proceedings of ascilite Singapore 2007,* 99-107.

Byrne, P., & Lyons, G. (2001). The effect of student attributes on success in programming. *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education, 33*(3), 49-52.

Caspersen, M.E., & Bennedsen, J. (2007). Instructional design of a programming course. *Proceedings of the 3rd International Workshop on Computing Education Research*, 111-122.

Chaffin, R., & Imreh, G. (2002). Practicing perfection: Piano performance as expert memory. *Psychological Science, 13*(4), 342-349.

Charmaz, K. (2000). Constructivist and objectivist grounded theory. In N. K. Denzin & Y. Lincoln (Eds.), *Handbook of qualitative research* (pp. 509–535). Thousand Oaks, CA: SAGE Publications.

Charmaz, K. (2006). *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. London: SAGE Publications.

Chen, J.C.W. (2012). A pilot study mapping students' composing strategies: Implications for teaching computer-assisted composition. *Research Studies in Music Education, 34*(2), 157-171.

Chong, E.K.M. (2012). Contemplating composition pedagogy in the iPad era. *Proceedings of the 19th International Seminar of the Commission for the Education of the Professional Musician,* 42-48.

Coleman, R., Krembs, M., Labouser, A., & Weir, J. (2005). Game design & programming concentration within the computer science curriculum. *ACM SIGCSE Bulletin, 37*(1), 545-550.

Cooper, S., & Cunningham, S. (2010). Teaching computer science in context. *ACM Inroads, 1*(1), 5-8.

Costa-Giomi, E. (1999). The effects of three years of piano instruction on children's cognitive development. *Journal of Research in Music Education, 47*(3), 198-212.

Costa-Giomi, E. (2014). The long-term effects of childhood musical instruction on intelligence and general cognitive abilities. *Update: Applications of Research in Music Education, 2014*, 107.

Crotty, M. (1998). *The Foundations of Social Research: Meaning and Perspective in the Research Process*. Newbury Park, CA: SAGE Publications.

de Guinea, A.O., & Webster, J. (2011). Are we talking about the task or the computer? An examination of the associated domains of task-specific and computer self-efficacies. *Computers in Human Behavior, 27,* 978-987.

Design Council. (2014). *What is design?* Retrieved from http://www.mech.hku.hk/bse/interdisciplinary/what_is_design.pdf

Dillenbourg, P., & Jermann, P. (2010). Technology for classroom orchestration. In M.S. Khine & I.M. Saleh (Eds.), *New Science of Learning* (pp. 525-552). New York: Springer.

diSessa, A. (2000). *Changing minds: Computers, language and literacy*. Cambridge: MIT Press.

Dorn, B., & Guzdial, M. (2010). Learning on the job: Characterizing the programming knowledge and learning strategies of web designers. *Proceedings of the 2010 ACM Conference on Computer-Human Interaction*, 703-712.

Dym, C. (2007). Engineering design: So much to learn. *International Journal of Engineering Education, 22*(3), 422.

Edwards, M. (2011). Algorithmic composition: Computational thinking in music. *Communications of the ACM, 54*(7), 58-67.

Ellis, P. (1995). Designing sound: Developing computer support for creativity in music education. *Research Studies in Music Education, 5,* 11-23.

Ericsson, K.A., & Simon, H.A. (1980). *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: The MIT Press.

Folkestad, G., Lindström, B., & Hargreaves, D.J. (1997). Young people's music in the digital age: A study of computer based creative music making. *Research Studies in Music Education, 9,* 1-12.

Fox, E.J. (2008). Contextualistic Perspectives. In J.M. Spector, M.D. Merrill, J. Van Merrienboer, & M. P. Driscoll (Eds.), *Handbook of Research on Educational Communication and Technology* (pp. 55-66). Englewood Cliffs, NJ: Lawrence Eribaum Associates.

Freeman, J., Magerko, B., McKlin, T., Reilly, M., Permar, J., Summers, C., & Fruchter, E. (2014). Engaging underrepresented groups in high school introductory computing through computational remixing in EarSketch. *Proceedings of the 43rd SIGCSE Technical Symposium on Computer Science Education*, 85-90.

Gartland-Jones, A. & Copley, P. (2003). The sustainability of genetic algorithms for musical computation. *Contemporary Music Review, 22*(3), 43-55.

Glesne, C. (2011). *Becoming Qualitative Researchers: An Introduction*. Boston, MA: Pearson.

Graci, C. (1992). Children, chunking, and computing. *Journal of Computing in Childhood Education, 3*(3-4), 247-258.

Goodling, L., & Standley, J.M. (2011). Musical development and learning characteristics of students: A compilation of key points from the research literature organized by age. *Update: Application of Research in Music Education, 2011*, 32-45.

Guzdial, M. (2008). Paving the way for computational thinking. *Communications of the ACM, 51*(9), 25-27.

Hancock, C.M. (2003). Real-Time Programming and the Big Ideas of Computational Literacy (unpublished doctoral dissertation). Massachusetts Institute of Technology, Boston, MA.

Harel, I. & Papert, S. (1991). *Constructionism*. New York: Ablex Publishing.

Hasan, B. (2003). The influence of specific computer experiences on computer self-efficacy beliefs. *Computers in Human Behavior, 19,* 443-450.

Hewitt-Taylor, J. (2001). Use of constant comparative analysis in qualitative research. *Nursing Standard, 15*(42), 39-42.

Hickey, M. (1997). The computer as a tool in creative music making. *Research Studies in Music Education, 8,* 56-70.

House, D.H., Malloy, B.A., & Buckley, C. (2010). The craft of computer programming: Lifting the veil. *Proceedings of FuturePlay 2010: The International Academic Conference on the Future of Game Design and Technology,* 74-81.

Jacob, B.L. (1996). Algorithmic composition as a model of creativity. *Organized Sound, 1*(3), 157-165.

Jacobson, E. (2010). Music remix in the classroom. In M. Knobel & C. Lankshear (Eds.), *DIY Media: Creating, Sharing, and Learning with New Technologies* (pp. 27-49), New York: Peter Lang International Academic Publishers.

Jenkins, T. (2001). *The motivation of students of programming*. Paper presented at the Annual Joint Conference Integrating Technology into Computer Science Education, Canterbury, UK.

Johnson-Laird, P.N. (2002). How jazz musicians improvise. *Music Perception, 19*(3), 415-442.

Kelleher, C. & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys, 37*(2), 83-137.

Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling Alice motivates middle school girls to learn computer programming. *Proceedings of the 2007 Conference on Human Factors in Computing Systems,* 1455-1464.

Kirkman, P. (2009). Embedding digital technologies in the music classroom: An approach for the new national music curriculum. Retrieved February 10, 2014, from http://www.jsavage.org.uk/jsorg/wp-content/uploads/2013/09/phil_kirkman.pdf

Knobelsdorf, M., & Romeike, R. (2008). Creativity as a pathway to computer science. *Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education,* 286-290.

Knobelsdorf,M., & Schulte, C. (2008). Computer science in context – Pathways to computer science. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research*, 65-76.

Knuth, D.E. (1975). Computer science and its relation to mathematics. *The American Mathematical Monthly, 81,* 323-343.

Knuth, D.E. (1992). Computer programming as an art. *Communications of the ACM, 17*(12), 667-673.

Kora-Ljungberg, M., Yendol-Hoppey, D., Smith, J.J., & Hayes, S.B. (2009). (E)pistemological awareness, instantiation of methods, and uninformed methodological ambiguity in qualitative research projects. *Educational Researcher, 38*(9), 687-699.

Kranch, D.A. (2012). Teaching the novice programmers: A study of instructional sequences and perception. *Education and Information Technologies, 17*, 291-313.

Leutenegger, S., & Edgington, J. (2007). A games first approach to teaching introductory programming. *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education, 39*(1), 115-118.

Levi-Strauss, C. (1968). *The Savage Mind*. Chicago: University of Chicago Press.

Lincoln, Y.S., & Guba, E.G. (1985). *Naturalistic Inquiry*. Newbury Park, CA: SAGE Publications.

Linn, M.C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher, 14*(5), 14-29.

Linn, M.C., & Dalbey J. (1985). Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist, 20*(4), 191-206.

Luther, K., Caine, K., Ziegler, K., & Bruckman, A. (2010). Why it works (when it works): success factors in online creative collaboration. *Proceedings of the 16th ACM International Conference on Supporting Group Work*, 1-10.

Ma, L. Ferguson, J., Roper, M., & Wood, M. (2011). Investigating and improving the models of programming concepts held by novice programmers. *Computer Science Education, 21*(1), 57-80.

Maloney, J., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: urban youth programming with Scratch. *Proceedings of the 2008 ACM Conference on Computer Science Education*, 367-371.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch programming language and environment. *ACM Transactions on Computing Education, 10*(4), 2010.

Margolis, J., Estrella, R., Goode, J., Holme, J.J., & Nao, K. (2008). *Stuck in the Shallow End: Education, Race, and Computing*. Boston: The MIT Press.

Martin, T., Rivale, S., & Diller, K. (2007). Comparison of student learning in challenge-based and traditional instruction in biomedical engineering. *Annals of Biomedical Engineering, 35*(8), 1312-1323.

McCormack, J. (1996). Grammar based music composition. In R. Stocker (Ed.), *Complex Systems 96: From Local Interactions to Global Phenomena* (pp. 320-336). Amsterdam: ISO Press.

McKeithen, K.B., Reitman, J.S., Rueter, H.H., & Hirtle, S.C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology, 13*, 307-325.

Meyers, A., Cole, M., Korth, E., & Pluta, S. (2009). Musicomputation: teaching computer science to teenage musicians. *Proceedings of the 2009 ACM Conference on Creativity and Cognition,* 29-38.

Miendlarzewska E. A., & Trost W. J. (2014). How musical training affects cognitive development: rhythm, reward, and other modulating variables. *Frontiers in Neuroscience, 7*, 1-18.

Miller, K. (2009). Schizophonic performance: Guitar Hero, Rock Band, and virtual virtuosity. *Journal of the Society for American Music, 3*(4), 395-429.

Miranda, E.R. (2003). On the evolution of music in a society of self-taught digital creatures. *Digital Creativity, 14*(1), 29-42.

Miranda, E.R. (2004). At the crossroads of evolutionary computation and music: Self-programming synthesizers, swarm orchestras, and the origins of melody. *Evolutionary Computation, 12*(2), 137-158.

National Research Council. (2011). *ICT Fluency and High School Graduation Outcomes*. Washington DC: The National Academies Press.

Palmer, C., & Pfordresher, P. Q. (2003). Incremental planning in sequence production. *Psychological Review, 110*(4), 683-712.

Papert, S. (1991). *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books.

Patton, M. (1990). *Qualitative evaluation and research methods*. Newbury Park, CA: SAGE Publications.

Pea, R.D., & Kurland, D.M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology, 2*(2), 137-168.

Peppler, K., & Kafai, Y. (2005). Creative coding: Programming for personal expression. Retrieved March 10, 2012, from http://scratch.mit.edu/files/CreativeCoding.pdf

Peterson, J. & Hickman, C. (2008). Algorithmic composition as an introduction to computing. *Journal of Computing in Small Colleges, 24*(1), 212-218.

Petre, M., & Blackwell, A. (2007). Children as unwitting end-user programmers. *Proceedings of the 2007 Symposium on Visual Languages and Human-Centric Computing*, 239-242.

Pillay, N., & Jugoo, V.R. (2005). An investigation in to student characteristics affecting novice programming performance. *ACM SIGCSE Bulletin, 37*(4), 107-110.

Pirolli, P., & Recker, M. (1994). Learning strategies and transferrin the domain of programming. *Cognition and Instruction, 12*(3), 235-275.

Prayaga, L. (2011). Want to become a music composer? Try with intermediate programming skills. *Journal of Computing Sciences in Colleges, 27*(2), 108-113.

Ramalingam, V., LaBella, D. & Wiedenbeck, S. (2004). Self-efficacy and mental models in learning to program. *Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education,* 171-175.

Reich, S. (2002). *Writings on Music (1965-2000)*. New York: Oxford University Press, USA.

Repenning, A., Webb, D., & Ioannidou, A. (2010). Scalable game design and the development of a checklist for getting computational thinking into public schools. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education,* 265-269.

Resnick, M. (1997). Turtles, termites, and traffic jams: Explorations in massively parallel microworlds. New York: Bradford Books.

Roden, T.E., & LeGrand, R. (2013). Growing a computer science program with a focus on game development. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education,* 555-560.

Ruthman, A., Heines, J., Greher, G., Laidler, P., & Saulters, C. (2010). Teaching computational thinking through musical live coding in Scratch. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education,* 351-355.

Savage, J. (2005). Working towards a theory for music technologies in the classroom: how pupils engage with and organize sounds with new technologies. *British Journal of Music Education, 22*(2), 167-180.

Savery, J.R., & Duffy, T.M. (1995). Problem-Based Learning: An Instructional Model and Its Constructivist Framework. In B.G. Wilson (Ed.), *Learning Environments: Case Studies in Instructional Design* (pp. 135-150). Englewood Cliffs, NJ: Educational Technology Publications.

Sawyer, B., Forsyth, J., O'Connor, T., Bortz, B., Finn, T., Baum, L., Bukvic, I.I., Knapp, B., & Webster, D. (2013). Form, function, and performances in a musical instrument MAKErs camp. *Proceedings of the 42nd SIGCSE Technical Symposium on Computer Science Education*, 669-674.

Slade, N. (2011). *How to Make Music in Your Bedroom*. New York: Virgin Digital.

Sleeman, D., Putnam, R.T., Baxter, J., & Kuspa, L. (1987). An introductory PASCAL class: A case study of students' errors. In R.E. Mayer (Ed.), *Teaching and*

Learning Computer Programming: Multiple Research Perspectives (pp. 237-258). Hillsdale, NJ: Erlbaum.

Smith, D.C., Cypher, A., & Tesler, L. (2000). Novice programming comes of age. *Communications of the ACM, 43*(3), 75-80.

Soloway, E., Erlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming? In A. Badre & B. Shneiderman (Eds.), *Directions in human-computer interaction* (pp. 27-54). Norwood: Ablex.

Sorensen, A. & Gardner, H. (2010). Programming with time: Cyber-physical programming with impromptu. *Proceedings of OOP-SLA10: ACM International Conference on Object Oriented Programming*, 822-834.

Steele, C.J., Bailey, J.A., Zatorre, R.J., & Penhune, V.B. (2013). Early musical training and white-matter plasticity in the corpus collosum: evidence for a sensitive period. *The Journal of Neuroscience, 33*(3), 1282-1290.

Swanson, K. (2013). A case study on Spotify: Exploring perceptions of the music streaming service. *MEIEA Journal, 13*(1), 207-230.

Taub, R., Ben-Air, M., & Armoni, M. (2009). The effect of CS unplugged on middle-school students' views of CS. *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education, 41*(3), 99-103.

Turkle, S., & Papert, S. (1990). Epistemological pluralism: Styles and voices within the computer culture. *Signs, 16*(1), 128-157.

Van-Roy, P., & Haridi, S. (2004). Concepts, Techniques, and Models of Computer Programming. Cambridge: MIT Press.

Verwey, W., & Dronkert, Y. (1996). Practicing a structured continuous key- pressing task: Motor chunking or rhythm consolidation? *Journal of Motor Behavior, 28*(1), 71-79.

Walker, M. & Whitaker, S. (1990). Mixed initiative in dialogue: An investigation into discourse segmentation. *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, 70-78.

Wang, G., Trueman, D., Smallwood, S., & Cook, P.R. (2008). The laptop orchestra as classroom. *Computer Music Journal, 32*(1), 26-37.

Webster, P.R. (1998). Young children and music technology. *Research Studies in Music Education, 11,* 61-76.

Webster, P. R. (2009). Children as creative thinkers in music: Focus on composition. In S. Hallam, I. Cross & M. Thaut (Eds.), The Oxford Handbook of Music Psychology (pp. 421-428). Oxford: Oxford University Press.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C.C. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers, 11,* 255-282.

Wilensky, U. (1999) [computer software]. NetLogo. Evanston, IL: Center for Connected Learning and Computer-Based Modeling (http://ccl.northwestern.edu/netlogo).

Wilensky, U. & Resnick, M. (1999). Thinking in levels: A dynamic systems perspective to making sense of the world. *Journal of Science Education and Technology, 8*(1).

Wilson, B.C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors. *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education, 33*(1), 184-188.

Wing, J. (2006). Computational thinking. *Communications of the ACM, 49*(3), 33-35.

Wu, T., Chan, P., & Hallett, M. (2008). Modifications of the interactions in the motor networks when a movement becomes automatic. *Journal of Physiology-London, 586*(17), 4295-4304.

Xu, D., Blank, D., & Kumar, D. (2008). Games, robots, and robot games: Complementary contexts for introductory computing education. *Proceedings of the 3rd International Conference on Game Development in Computer Science Education,* 66-70.

Ye, N. (1996). Expert-novice knowledge of computer programming at different levels of abstraction. *Ergonomics, 39*(3), 461-481.

Zuk, J., Benjamin, C., Kenyon, A., & Gaab, N. (2014). Behavioral and neural correlates of executive functioning in musicians and non-musicians. *PLuS One, 9*(6).