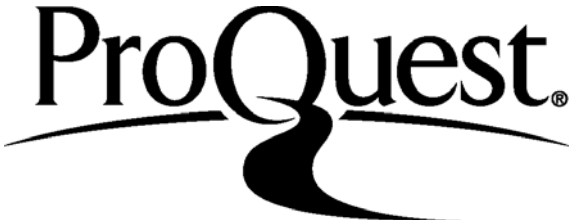
EFFECTS OF COMPUTER PROGRAM VISUALIZATION TOOLS ON STUDENT POPULATIONS

By
Meghan Peterson

A Thesis Submitted in Partial Fulfillment of the Requirements for Masters of Science in Educational Technology In Educational Studies: K-12 and Secondary Programs

Minnesota State University, Mankato
Mankato, MN

November 2016



DATE: November 11, 2016

This thesis is submitted as part of the required work in the Department of Educational Studies, K-12 and Secondary Programs, KSP 610 Scholarly Writing, at Minnesota State University Mankao, and has been supervised, examined, and accepted by the professor.

_____
Kathleen Foord, Ed. D, Associate Professor

_____
Qijie Cai, Ph. D., Assistant Professor

_____
Carrie Chapman, Ph. D., Associate Professor

Abstract

This study examined how program visualization tools affect Advanced Placement Computer Science students' understanding of abstract programming concepts. A literature review was conducted to determine if program visualization is effective and which students benefit from it the most. The findings were used to design a causal comparative study in which students would experience instruction with and without program visualization. The study took place in an AP Computer Science course during the first challenging unit about an abstract concept: loops. Participants (n = 24) were selected using convenience sampling and were assessed before, during, and after the study took place. While it was difficult to reach to any significant conclusions about program visualization's effect on student understanding as a whole, there were several conclusions that could be made about different groups of students. The results suggested that math confidence is a factor in the effectiveness of program visualization and there appeared to be a similar trend with ethnicity. There was also significant evidence that program visualization is most effective for students who are not considered high or low achieving. These results provide insight into how computer science teachers can create lessons using program visualization that are meaningful for all students and for specific groups in particular.

Table of Contents

Table of Tables

CHAPTER ONE

Introduction

In a society where computer programming is becoming a vital skill in every field, it is reasonable to assume that Computer Science will make its way into the core curriculum within the next few years. As it stands now, most high school programming classes are filled with motivated, high achieving students who elected to take the class, but when programming becomes a requirement to graduate, programming teachers will encounter students who are unmotivated and uninterested in computer science. These teachers will be challenged to find a way to help all students learn, not just the students who *want* to program. Unfortunately, programming can be an extremely difficult concept for students who struggle with abstract ideas and processes. Students cannot move past the novice method of programming, which involves spending "little time testing code, and [attempting] small 'local' fixes rather than significantly reformulating programs" (Robins, Rountree, & Rountree, 2003, p. 151). Novice programmers tend to be limited to the surface knowledge of a programming language, lacking the mental models necessary to problem solve abstractly. According to Ma, Ferguson, Roper, and Wood (2011), students who have viable mental models, or "internal explanations of how something works," perform significantly better than those without mental models (p. 58). The challenge for introductory programming teachers is determining what strategies help students build those models.

One tool that computer science teachers sometimes use is program visualizations. These visualizations are either static diagrams or dynamic animations that demonstrate how a program or algorithm works. Many researchers have tried to use program visualization tools to develop novice programmers' mental models, but the results of the studies have been inconclusive. Some

researchers have found that specific visualization tools like ViLLE work well for novice students, but not the students who are more advanced (Rajala, Laakso, Kaila, & Salakoski, 2008). Other studies have found that the influence of program visualization is not significant at all. A meta-study conducted by Hundhausen, Douglas, and Stasko (2002) revealed that the way visualization tools are implemented has more impact than the specific type of visualization. When reading the existing research, it becomes clear that there is no definitive verdict on whether or not program visualization is an effective tool to help students.

One reason the results have been so inconsistent is the variability in software, implementation, and analysis of program visualization tools. Kehoe, Stasko, and Taylor (2001) suggest that researchers have not focused enough on *how* animations help students. Indeed, each study uses different software, teaching techniques, and implementation strategies to examine programming visualizations. This turns the focus of the research to whether or not the unique software, technique, or strategy is effective instead of how the students benefit from the visualization activity in general. To combat this trend, some authors have chosen to focus on comparing novice and expert programmers instead of focusing on a specific software or strategy. For example, a study by Rajala, et al., (2008), found that a specific program visualization tool called ViLLE can "effectively [even] out the differences caused by previous programming experience" (p. 29). This research study and others like it show that program visualization can be beneficial for certain students more than others. Perhaps more research should be done to compare the effects on different student populations instead of studying variations among different program visualization software.

Statement of the Problem

It is clear to computer science teachers and researchers that novice programmers think about code in a drastically different way than more advanced programmers. There are traits of novice programmers that are easy to observe, such as a tendency to change small parts of code without considering the larger structure of a program. But underneath the surface, it is the weak mental models that keep novices from succeeding. According to Wiedenbeck, Fix, and Scholtz (1993), there are five abstract characteristics of mental models that novices lack: hierarchical and multi-layered thinking processes, mappings between those different layers, recognition of programming patterns, understanding of how the parts of a program work together, and comprehension of the program text and structure. The struggle that teachers of introductory programming classes face is how to develop those abstract characteristics.

Program visualization could be a valuable tool for helping beginning programmers build mental models and develop their coding skills. More research is needed, however, in order to understand how visualization benefits different groups of students. This study will add to existing research by identifying how several factors like gender, ethnicity, programming experience, and academic achievement impact the effectiveness of the program visualization. The purpose of this research thesis is to compare Advanced Placement students' understanding of *for loops* and *while loops* after using program visualization tools among several different student populations. Two primary research questions will be investigated:

1. How do program visualization tools relate to understanding of *for loops* and *while loops* in an AP Computer Science course?

2. How do program visualization tools influence mastery of *for loops* and *while loops* for different potential student populations (i.e. gender, ethnicity, programming experience, academic achievement)?

Importance of the Study

Some high school students progress from novice programmer to beginner very quickly. Instead of learning the structure of the basic operations like loops and conditional statements, these students learn *why* their code works and investigate new applications of the concepts learned in class. While these students become more independent and stronger in their programming skills, others struggle to move past the very basics. It becomes increasingly difficult for these novices to succeed in AP Computer Science because they have little prior knowledge on which to build new concepts.

Many researchers have concluded that these novices are lacking strong mental models which is why they tend to struggle with abstract concepts. Wiedenbeck, et al. (1993) have identified five abstract characteristics of mental models that novices lack. The first characteristic, hierarchical and multi-layered representations of a program, refers to the way programmers visualize the way a program runs. The researchers found that "advanced programmers, but not novices, used a strategy of reading a program in the order in which it would be executed" which helped them develop hierarchical representations (Wiedenbeck, et al., 1993, p. 74). It seems natural that some sort of visualization is necessary to help novices understand how to read a program in the order in which it is run instead of from top to bottom.

Visualization in the form of static pictures has been a staple of textbook-based and direct instruction for decades, but recently computing educators have developed software tools to make these visualizations more appealing, interactive, and easier to use. Sorva, Karavirta, and Malmi

(2013) have analyzed over 100 studies on various software tools, concluding that many of them can significantly help novice programmers develop their mental models. Unfortunately, most of the software reviewed in their article "appeared to have been short-lived research prototypes that have been soon discarded once the system had been constructed or an evaluative study carried out" (p. 54). These studies do not provide lasting implications for computer science teachers and researchers, since the products used in the studies are no longer available.

It is imperative, then, that future research is conducted using tools that are still in existence and easy for computer science teachers to access. Several program visualization tools have been maintained and updated after initial research was conducted. "One of the longest-lasting and most-studied program visualization tools" is Jeliot, which was developed to help students learn Java (Sorva, et al., 2013, p. 36). Jeliot is currently free and easy to download online, making it an ideal tool for high school teachers. It has survived for almost 20 years because the authors have frequently updated the software based on new research. There have been many studies conducted to examine Jeliot's effectiveness as a program visualization tool in various situations.

The most recent study about Jeliot was conducted by Moreno, Sutinen, and Joy (2014) who studied a version of Jeliot called Jeliot ConAn that used conflictive animations, which intentionally show errors in the visualization that students must identify. The results, based upon 18 students in an introductory programming course, indicated that students who used the conflictive animations improved their metacognitive skills and conceptual knowledge compared to the control group which used the unaltered Jeliot 3 software. Another recent study of Jeliot by Wang, Bednarik, and Moreno (2008) examined the timing of introducing program visualization to students. The authors concluded that "explanations *after* animations have positive effects on

learning gain while explanations *before* have little to no effect" (p. 107). Their research contributes to the conclusion that *how* programming visualizations are implemented is just as important as which software is used. Finally, one of the largest studies on Jeliot was conducted by Cisar, Radosav, Pinter, and Cisar (2011). In this study, 400 students were divided into three groups: the control group had traditional instruction while two experimental groups received either blended instruction (PowerPoint presentations in addition to Jeliot) or completely electronic instruction (solely Jeliot). The researchers concluded that there was a significant difference between the non-Jeliot and Jeliot groups but there was no difference found between the two experimental groups. This large study indicated that even a small amount of program visualization activities could contribute to increased student understanding. Together, these three studies show that Jeliot can be extremely beneficial for students.

Clearly there has already been extensive research on whether or not Jeliot is effective, but little has been done to compare the effectiveness of the software on different populations. Moreno, et al. (2014) compared students with no programming experience to those with a background in coding and found that visualizations were more helpful for inexperienced programmers. There are many other factors, however, that may influence the effectiveness of program visualization tools. The goal of this study is to examine the impact that factors like gender, ethnicity, programming experience, and mathematics skills may have on Jeliot 3's effectiveness. This study will add to the existing research on Jeliot by clarifying if different groups of students benefit more or less from using program visualization tools.

Methods

Because there is great variety in both program visualization software and implementation strategies, the investigation of the research questions stated above began with a wide search for

any studies about the effectiveness of program visualization tools. The Minnesota State University, Mankato's library databases and collections were used to find these sources. These databases included ERIC on EBSCO and the ACM Digital Library using search terms that included "computer science AND visualization" as well as "computer program visualization". The researcher also examined studies referenced in the articles found within the library databases. All sources were evaluated using Creswell's (2014) checklist for evaluating quantitative and qualitative studies in order to ensure that information was accurate and relevant. Most of the literature was published within the last 15 years, although some older studies on mental models were included due to their in-depth definitions and exploration of terms.

To organize the information, a literature log was used to keep track of methodology, key findings, quotes, and definitions. The studies were organized by research question and later synthesized into themes to conduct a literature review. The literature review guided the development of the experiment and the analysis of the results.

Summary of Experiment

This study was designed to address both research questions, examining whether or not program visualization tools are effective in teaching *for loops* and *while loops*, and if different populations benefit differentially from their use in learning. Students were taught using traditional methods for several days before using Jeliot to help them visualize *for loops* and *while loops*. The students took pre-tests, intermediary tests, and post-tests, which were analyzed to see if student understanding increased. Different populations were also examined to see if Jeliot was more useful for some students than others. Permission for this study was obtained from the university IRB in August 2016.

Setting and Population

This study was conducted in an AP Computer Science class in a large high school in a medium sized city in the upper Mid-West United States in the fall of 2016. The participants were students that chose to enroll in the course. Most of these students were seniors or juniors with an interest in a STEM career path after high school. During the second month of school, students who were enrolled in AP Computer Science were contacted through email in order to obtain permission for participation. Permission to conduct the study at the identified high school was obtained in September of 2016.

Experiment Design and Data Collection

The first month of the course was designed to introduce students to the basic Java programming concepts, including primitive types, Strings, and conditional statements. This study began when students were introduced to *for loops* and *while loops*. The students took a pre-test to gauge their understanding of *for loops* and *while loops* before the unit began. In the first phase of the intervention, the students received three days of direct instruction with time for programming exercises after which the students took an intermediary test designed to assess their current understanding of *for loops* and *while loops*. This test was identical to the pre-test but the questions were in a different order. During the second phase of the intervention, the participants spent two days doing activities with Jeliot 3 while the instructor supervised. The students took a post-test similar to the pre-test to reassess their learning, and also completed a questionnaire on demographic information. The pre, intermediary, and post-test scores were compared to determine how the program visualization intervention affected student understanding of *for loops* and *while loops*. This was done in order to answer the first research question. To address the second research question, different demographic groups of students were compared to see if the

program visualization intervention impacted achievement differentially for certain populations of students.

Before data collection began, the researcher worked with the course instructor for over 10 hours in order to ensure that the content was delivered correctly. This training took place from May to September and included some online sessions in the fall before data collection began. The course instructor administered the pre, intermediary, and post-tests as well as supervised all classroom activities. The researcher was not present for the pre-test, intermediary test, or post-test, administration of the questionnaire, or for instruction.

This quantitative study examined the effect of program visualization on several different student populations. Gender, ethnicity, programming experience, and mathematics skills were some of the variables examined. The researcher used paired t-tests to examine differences in understanding between these groups. Confidentiality validity, and reliability were also addressed in the research design and are explicated in Chapter Three.

Limitations of the Study

The sample population was a convenience sample where the researcher had access to a computer science course. Due to limitations on registration for this course, this study could only be conducted on one section of AP Computer Science students at the selected high school. The sample size was between 24 and 28 students. While the selected high school was a fairly good representation of high schools in mid-sized Midwest cities, the small sample size made it difficult to generalize results to larger populations of students. Additionally, the size of some of the subgroups were very small, restricting the possibility for making statistical inference.

Since the unit on *for loops* and *while loops* was short (five school days and one weekend), different pre, intermediary, and post-tests were needed so that there was no internal

threat to validity due to instrumentation or testing. It was important that the pre, intermediary, and post-tests were very similar, however, so scores could be compared.

The conclusions may only apply to visualization used for *for loops* and *while loops* until further research is conducted on other coding units. The conclusions will also be restricted to the use of one visualization software, Jeliot 3, so generalizations to other visualization software may not be warranted.

Definition of Terms

Mental model. According to Ma et al. (2011), mental models are "internal explanations of how something works" (p. 58).

Program visualization. A tool that "depict[s] the steps taken by programs during execution (Moreno, et al., 2014, p. 630). Some examples include ViLLE and Jeliot.

Program visualization strategy. A method of implementing program visualization tools. For example, Moreno, et al. (2014) implemented a program visualization tool (Jeliot) by intentionally showing students incorrect animations and asking students to correct them. This is a unique strategy for implementation.

Schema. A "chunk" or information that is organized by a learner. According to Kranch (2011), schema is "information organized according to how it will be used" (p. 293).

Overview

This study includes a literature review of the research on program visualization and its effect on increasing student understanding of abstract concepts. Methods, procedures, and results of the study designed to explore the impact on learning *for loops* and *while loops* as well as the impact on differing student populations will also be discussed. Finally, a summary of the

research, conclusions, limitations, and future research topics will be examined.

CHAPTER TWO

Review of the Literature

One of the most well researched areas of computer science education is the difference between novice and expert programmers. In a large literature review of the research on how students learn to program, Robins, et al. (2003) found that novices tend to have fragile programming knowledge and skills, which leads to weak mental models. Without a tool or strategy to develop solid understanding of basic programming knowledge and skills, abstract programming concepts are difficult for novices to master.

Many educators and researchers have tried to find a successful strategy to combat this issue. For example, Kranch (2011) conducted a study to determine if the sequence of programming instruction affects how much novices learn from an introductory computer science course. A sample of 34 college students were randomly assigned to one of three programming units, each based on a different instructional sequence (one that focused on students mastering one concept at a time, another that gave students all the knowledge and skills required to complete a task at once, and another that framed new concepts in terms of solutions to problems that students then broke apart). The results showed that there was no significant difference between the three sequences. Interestingly, novices mastered the simple syntax concepts and struggled with complex material no matter which unit they completed. Clearly other instructional tools and strategies need to be investigated to help novices master abstract and complex programming concepts.

The purpose of this study is to determine if program visualization tools will help teachers assist novice programmers. Two research questions will be investigated in this literature review. First, the current literature will be examined to determine if program visualization tools are

effective in helping introductory programming students master abstract concepts. The results from researchers have been inconclusive, so this literature review will focus on several different conclusions that researchers have made. Rather than determining if program visualization works for all students, the second research question deals with exploring which group of students benefits the most from using the tool. Many studies have concluded that program visualization does not work for every student, but is effective for certain populations, such as novice programmers or students with high confidence levels. The review will conclude with a summary of the main points in the existing research and an explanation of the need for this study.

Effectiveness of Program Visualization

In most introductory programming courses, students start to struggle when they learn about loops, programming elements that repeat segments of code a certain number of times. In a survey of 173 introductory programming students, Butler and Morgan (2007) concluded that "the elements of the curriculum of a highly conceptual nature proved to be acknowledged as the most challenging" (p. 106). Until students learn about loops, they only need to read a program from top to bottom. Part of the confusion with loops comes from the fact that the flow of control, or order of program execution, has changed. When students struggle with the idea of flow of control, teachers often try to draw pictures to show students what is happening during a loop. Static drawings, however, do little to convey the *movement* of the program because they are static and unable to move.

Many researchers believe that program visualization tools could solve the problem of teaching complex concepts, such as loops, because they animate the program as it is executed, giving students a clearer picture of what is happening when they run their code. One of the "longest-lasting and most-studied program visualization tools" is Jeliot, software that has been

used, edited, and adapted by educators for almost 20 years (Sorva, et al., 2013, p. 36). Several studies have shown that students who use Jeliot score significantly better on post-assessments than students who use traditional resources. Researchers Cisar, et al. (2011) conducted the largest of these studies, assigning 400 introductory programming students into one of three groups: a control group (which consisted of traditional teaching methods without Jeliot), an experimental group (which consisted of some traditional teaching and some activities with Jeliot), and a second experimental group (which consisted of electronic notes and Jeliot activities). They found that there was a significant difference between the control group and the two experimental groups ($p < 0.01$) with the experimental groups outperforming the control group. This massive study demonstrates that program visualization tools can help students learn how to master complex concepts.

Other researchers who have worked with Jeliot have found similar positive results that show program visualizations helping students with complex ideas. For example, Hongwarittorrn and Krairit (2010) conducted an experiment to determine if the visualization tool could help students learn and develop positive attitudes about object oriented programming. In their quasi-experimental study, 54 students were split into a control group (which consisted of no program visualization tools) and an experimental group (which used Jeliot). Throughout the 15-hour course, everything but the visualization tool usage was kept consistent between the two groups. The researchers found that there was a significant difference in scores for the experimental group, suggesting that Jeliot 3 could help introductory students master complex object oriented programming tasks. Program visualization appears to aid mastery of complex concepts, but this success may not be universal.

Ineffectiveness of Program Visualization

There are some studies that have shown that program visualization is not effective at all. In order to determine if Jeliot 3 helped students in an introductory programming course, Moreno and Joy (2007) enlisted six undergraduate computer science students to participate in their research. All students used Jeliot 3 in the course for two hours a week. Additionally, some students did extra Jeliot activities outside of class. The researchers conducted interviews with students when the course concluded and found that the students mostly used Jeliot for debugging. There was no difference in vocabulary or understanding between the two groups, showing that extra time with program visualization tools does not help students learn. Clearly in some instances, program visualization is not effective.

Similar results have been found with other program visualization tools. For example, Rajala, et al. (2008) conducted research with 72 first year college students to determine if ViLLE, a tool that is very similar to Jeliot, was effective. They split the students into two groups: one that used traditional text material to learn concepts and another that used ViLLE, which animates code in any language. The researchers concluded that there was not significant evidence that ViLLE was better than no visualization tool. The wide variety of results between different researchers and studies suggests that there might be something else, perhaps implementation strategies that affects whether or not program visualization tools benefit students.

Implementation Strategies of Program Visualization

Some recent studies have tried to determine if implementation strategy is the hidden factor in whether or not program visualization tools are effective. One of the most studied strategies revolves around cognitive dissonance. Ma, et al. (2011) conducted three studies,

involving sample sizes of 43-66 college students in which some participants worked with unaltered program visualization tools while others were given cognitive conflict activities. Essentially, these activities presented students with incorrect animations and tasked the learners with finding the error. Pre-tests and post-tests were used to determine the effectiveness of both learning methods. The researchers concluded that "visualization alone appeared to help with more straightforward concepts whereas a combination of cognitive conflict and visualization showed more promise for more demanding concepts" (p. 58). Although the tool was not as effective for simple concepts, the fact that it helped learners with more complex tasks is good evidence that program visualization is a worthwhile tool for teachers to use. The essential additional variable for success appears to be the addition of cognitive dissonance.

The way in which program visualization is used seems to determine the impact on learning. Moreno, et al. (2014) also conducted a study to address the connection between cognitive dissonance and program visualizations. While the control group used Jeliot to learn programming concepts, the researchers intentionally gave their experimental group animations that were incorrect with the modified software Jeliot ConAn. These students had to determine what was wrong with the animations while the students in the control group watched correct animations. The researchers found that the conflictive animations helped students learn programming concepts, although the results were not significant due to the small sample size. Still, the authors pointed out that "the content of the visualizations are not as decisive for learning as the intended use of the visualization" (p. 629). This study suggests that the method of program visualization implementation is more important than the choice of software being used or even the content presented in the software.

In order to investigate how implementation practices impacted student learning, a working group of 11 computer science instructors reached out to many programming teachers through an online survey to determine if programing visualization tools help students learn complex programming concepts. In their conclusions, Naps, et al. (2002) stated that technology does little to help students unless it is integrated in the form of active learning. The researchers determined that most often visualizations are used for demonstrations or as optional additional practice, but that a visualization is "of little educational value unless it engages learners in an active learning activity" (p. 131). They categorized the implementation methods according to Bloom's Taxonomy and concluded that the integration methods with active learning were more effective at helping students learn. This study provides more evidence that the way program visualization tools are implemented can influence their effectiveness.

Other studies have identified different factors such as placement and level of integration that influence the effectiveness of program visualization tools. Wang, et al. (2012) chose to examine whether the placement of animation explanations was one of these factors. In their study, 18 participants were split into two groups; one group took a programming course where animations in Jeliot were shown before an explanation while the other group saw explanations before animations. The researchers concluded that there was a significant improvement in the learning of the animation-first group, but no difference for the explanation-first group, showing that the implementation placement could be the key factor to whether or not program visualization is effective. Another factor that has been investigated is the level of program visualization tool integration into existing course materials. Researchers Lahtinen, Ahoniemi, and Salo (2007) conducted research to determine if integrated program visualization could help students learn in an introductory programming course. The 302 participants in their study were

given optional pre-exercises that were integrated into printed material and website resources for the course. The findings suggested that program visualization was more helpful for abstract concepts, but only for some students. The placement and level of integration of program visualization tools is yet another factor that might influence how effective these tools really are.

Clearly the effectiveness of program visualization tools has created a research tension that compels additional clarification. This could be due to the fact that each study implemented the tool using a slightly different strategy. It is still unclear if program visualization is beneficial only in certain conditions or for certain students.

Comparing Student Populations: The "Middle Third" Theory

While some studies focus on whether or not visualization works for all students, several groups of researchers have concluded that it is only effective for certain groups of students. In one of the only studies about how special education students learn to program, researchers Ebel and Ben-Ari (2006) examined students' behaviors while using program visualization tools. They observed 10 special education students during an introductory programming course and found that "bad behaviors" such as being off task were non-existent when program visualization was used in class. The researchers named this the "middle third effect", a theory that states that program visualization is most beneficial for students whose skills are neither strong nor weak (Ebel & Ben-Ari, 2006, p. 4). This theory was originally proposed by Ben-Bassat Levy, et al (2002). One of the authors taught two parallel introductory programming courses to high school students. In one class, students used Jeliot 2000 (a slightly older version of Jeliot 3 with similar functionality) while the control group used no program visualization tools. The researchers found that the group of students who benefited the most was the group that they called "mediocre" (Ben-Bassat Levy, et al., 2002, p. 10). For strong and weak students, the program visualization

did not have an effect, but the students who were struggling yet not at the bottom of their class benefited greatly from the tool. This theory points out the fact that not all students will benefit from the same method of instruction.

Confirmation of this "middle third" theory has been found in other research. The "middle third" theory was also supported by the findings of Lahtinen, Jarvinen, and Melakoski-Vistbacka (2007), who analyzed survey results from over 300 introductory programming students around Europe. The researchers chose to ignore different implementation strategies of program visualization tools and instead determine *how* the tools were used to help students learn. They found that program visualization was most helpful for the students that thought programming was challenging but knew they could still learn, noting that "the biggest user group of visualizations are students who find the course challenging but not too difficult to pass" (p. 260). Students who were overwhelmed in their course found no help from program visualization and students who did not struggle to learn had no need for the program visualization tools. A similar result was reached in the Rajala, et al. (2008) study previously described above. Although they did not find any significant evidence that ViLLE was better than no use of a visualization tool, they were able to conclude that ViLLE was significantly more effective for novices than experienced programmers. The researchers noted "the learners' short exposure to the tool makes the result even more remarkable" (p. 29). Although these studies showed that program visualization did not work for *all* students, most teachers would agree that a tool that helps even some students to be more successful could still be considered useful. The use of program visualization may be one factor in assuring greater student success, but are there other factors that could influence student success in using program visualization?

Comparing Student Populations: Other Factors

In addition to examining the difference between novices and experts, some researchers have chosen to examine additional factors that might help predict student success in learning programming. Rountree, Rountree, and Robins (2002) conducted a large study of 472 first year college students to determine if certain factors can predict student success. They found that students who expected to do well in the class were more likely to be successful. They concluded that "a positive attitude is more important than having the right background, and that students are fairly good at estimating their own ability" (p. 124). This means that a positive attitude and accurate estimates of ability can be more beneficial than programming experience or good grades in math classes. In another similar study, Bergin and Reilly (2005) analyzed 15 different factors that could potentially influence a student's ability to learn how to code. They used a test and questionnaire to determine which of the 15 factors was most influential among 96 first year college students. Their results showed that comfort level with the course material as well as math and science scores had the strongest correlation with performance. Clearly there are other student characteristics that can help teachers determine how much scaffolding and support the students will require. These factors, when attended to by teachers, might influence which students find program visualization beneficial.

Summary

While the existing research offers no clear conclusion on whether or not program visualization tools are effective at helping all students master abstract concepts, there has been some evidence that it is helpful for at least some groups of learners. This study aims to add to the existing research by determining if program visualization helps students master *for loops* and *while loops*, the first abstract concept that Advanced Placement Computer Science students face.

While most of the existing research has taken place in international college courses, this study will take place in a high school setting in the United States. This study will also seek to determine if program visualization is more effective for a certain group of students. Results of the study will be used to help AP Computer Science teachers create lessons that help students learn the complex concepts with which students traditionally struggle.

CHAPTER THREE

Methodology

This study was conducted to gain more insight into whether or not program visualization tools are helpful for high school students in an introductory programming course.

There were two research questions that guided the research:

1. How do program visualization tools relate to understanding of *for loops* and *while loops* in an AP Computer Science course?
2. How do program visualization tools influence mastery of *for loops* and *while loops* for different potential student populations (i.e. gender, ethnicity, programming experience, academic achievement)?

This chapter will discuss the context for research, the research design, the sample characteristics, the intervention, and the validity of the study.

Context for Research

Advanced Placement Computer Science is a relatively new course at the high school selected for this research. It was offered for three years prior to the beginning of this study. Enrollment in the course varies from 25 to 50 students each year. Although non-AP computer science classes were offered in the past, AP Computer Science was the only computer science course being offered when the study took place. This meant that the students in the course had a wide variety of programming abilities; some of the students already knew a programming language and others knew nothing about code.

The instructor of AP Computer Science had taught at the high school in this study for 12 years and taught at a different school for two years before that. This was his first year teaching computer science. The researcher acted as his mentor as he transitioned into his new position.

Five months before the study began, the researcher and the instructor met to discuss course material, lesson plans, and instructional strategies. This initial meeting was for eight hours, and the primary focus was to familiarize the instructor with Java. A second meeting was held four months before the study began. During this meeting, the researcher assisted the instructor in completing all of the lab exercises that students would be doing during the first semester. In the weeks leading up to the study, the researcher and instructor exchanged emails nearly every day to discuss course content and lesson plans. The researcher shared the research intervention materials with the course instructor two weeks before the study began and included detailed instructions with daily lesson plans for the nine-day study. These conversations between the instructor and the researcher ensured the validity of the interventions described below.

Research Design

This research was a causal comparative study conducted to determine if Jeliot 3 makes a significant impact on students' understanding of *for loops* and *while loops* and to determine if Jeliot 3 is more effective for certain populations of students. A pre-test, intermediate test, and post-test was used to examine achievement over time. The research design was influenced by the fact that instruction had to be identical for every student. Because of this restriction, it was impossible to design an intervention that consisted of a control group with no program visualization and a treatment group with program visualization. The resulting intervention consisted of students learning without program visualization first and learning with program visualization second. This made it difficult to draw conclusions about whether or not program visualization was the reason for improvement, but it did allow for meaningful analysis of which groups of students benefited from the tool.

Sampling Procedures and Participant Characteristics

This research was conducted using a convenience sample of 24 students of a possible 25 students enrolled in Advanced Placement Computer Science at the selected high school. A convenience sample was chosen because of the researcher's connection to the school and the program; the researcher helped develop the AP Computer Science curriculum, and thus had a thorough understanding of what students would be expected to know and do. As explained above, this class was the only programming course available at the high school selected, so the programming experience and ability varied greatly among the students in the course. Five of the 24 participants were girls. The sample consisted of 14 seniors, 9 juniors, and 1 sophomore. Sixteen students identified as white, 6 identified as Asian, 1 identified as Hispanic, and 1 identified as "other". The students ranged in programming experience with 9 having no experience and 15 having some programming experience.

Intervention

The nine-day study consisted of the following intervention plan. On the first day of the study, which was a Tuesday, the instructor administered a pre-test (Appendix A), which was designed to assess students' initial understanding of *for loops* and *while loops*. The pre-test consisted of three questions:

1. A question that asked students to use a *while loop* to print numbers that increased by six after each iteration of the loop. This question was designed to assess whether or not students could use a loop to iterate through several numbers.
2. A question that asked students to use *nested for loops* to print a pattern with asterisks and commas. This question was designed to assess students' understanding of nested loops (a loop within a loop).

3. A question that asked students to use both a *for loop* and a *while loop* to count the number of appearances of a letter in a word. This question was designed to assess student's understanding of the accumulator algorithm.

After administering the pre-test, the instructor directed students to a website that the researcher created to host all of the materials for the study. The instructor showed a video on the website that was created by the researcher. The video was about *while loops*. This video was projected on the screen in the front of the room. Students watched the video together. The instructor paused the video at certain points so that students could experiment with some code. This style of teaching is a traditional instructional practice that involves no program visualization tools, only a few static pictures and verbal explanations.

On the second day, the instructor played another video on the screen in the front of the room, this time about *for loops*. The researcher also created this video, and it once again included points at which the students stopped and experimented.

On the third and fourth day, the instructor helped students as they worked on lab assignments. These assignments were short programming problems that the students worked on individually or with their classmates. The problems were designed to be very similar to the pre-test questions.

On the fifth day (which was a Monday and meant that students had a gap of two days in between original learning and assessment), the instructor administered the intermediary test. The intermediary test consisted of the same three questions from the pre-test in a different order (Appendix A). After completing the intermediary test, students watched a video about how to use Jeliot 3, a program visualization tool, and then began working on the visualization lab. As with the first lab assignment, the students could work on their own or with their peers. The instructor

helped students as needed. The sixth and seventh day was more work time for this visualization lab.

On the eighth day (which was a Monday after a four day weekend), the students had another day to work with Jeliot. On the ninth day, they took a post-test (Appendix B). This test consisted of three questions that were very similar to the pre-test and intermediary test. The students also completed a questionnaire about their demographics (Appendix C).

Throughout the nine-day study, the instructor and the researcher emailed every day to assess and discuss the students' progress. The original plan was to complete the study in six days, but changes were made to the schedule to accommodate some unforeseen timing and scheduling issues. The researcher did not factor in the time it would take experienced programmers to finish the pre-test. Additionally, many students were gone during a workday for the PSAT. Both of these issues resulted in the need for a more days.

Validity

Careful consideration was given to the approval process, making sure that all district and research requirements were met. After the student investigator obtained permission from the university Institutional Review Board and the school board of the school selected for the research study, the instructor of the course (not the researcher) sent an email to the parents and guardians of all 25 students in the AP Computer Science course explaining the research study. The email highlighted the fact that all students would be participating in the class activities whether they participated in the research or not. Choosing to participate in the research only meant submitting their assessments and questionnaire answers. The email from the instructor assured parents and guardians that the researcher would never know the names of the students participating. Instead, each student would be given a number for identification. The instructor then emailed the

parent/guardian consent forms and sent paper copies home with the students. He also distributed paper copies of the student assent form in class. The instructor only explained what was on the consent and assent forms to the students; he did not give out any other information about the study. All of these steps ensured that students understood the necessary information about the study without knowing intricate details that could affect their performance in the experiment.

The researcher maintained construct validity by basing the assessments and questionnaires on past AP Computer Science assessments, which have been examined for both validity and reliability in scoring, as well as the current literature. The pre-test, intermediary test, and post-test were designed to represent three common applications of *for loops* and *while loops* (See appendix A and B for the test questions and appendix D for the scoring rubric). The pre-test and intermediary test featured identical questions in a different order. The post-test consisted of similar questions with slight differences that did not affect the nature of the questions. The researcher created the questionnaire to determine what populations were represented in the sample. This document changed as the current literature was reviewed in order to include populations that have already been studied by other researchers.

Confidentiality

The instructor created a list of student names matched with random ID numbers. After administering the pre-test, intermediary test, post-test, and questionnaire, the instructor replaced names on documents with ID numbers. The list matching ID numbers with student names was never shown to the researcher, thus student identity remained completely anonymous throughout the entire study.

CHAPTER FOUR

Results

This study was conducted to gain more insight into whether or not program visualization tools are helpful for high school students in an introductory programming course. There were two research questions that guided this study:

1. How do program visualization tools relate to understanding of *for loops* and *while loops* in an AP Computer Science course?
2. How do program visualization tools influence mastery of *for loops* and *while loops* for different potential student populations (i.e. gender, ethnicity, programming experience, academic achievement)?

The results presented in this chapter will begin to answer both of these questions.

Data Collection

The researcher created a rubric (Appendix D) to assess each problem of the pre-test, intermediary test, and post-test (Appendix A and B). The rubric was the same for each version of the assessment, although the problems were in different orders. The rubric was created so that students could get 50% by knowing the basic idea of a *for loop* or *while loop* without being able to execute it correctly. For example, if a student attempted to write the three parts of a *for loop* (initialization, condition, and increment) but these led to incorrect answers, the student would receive 50%.

The researcher wanted the scores to reflect students' knowledge of *for loops* and *while loops*, so points were not deducted for syntax errors such as misplaced brackets, combined lines of code, and other minor mistakes. The researcher also did not differentiate between the equals

method and the equals operator (==) even though the difference would affect the outcome of the code.

Some of the categories in the questionnaire to explore demographic differences (Appendix C) required a protocol for interpretation. When collecting the data on "programming experience", for example, the researcher gave the students one point for each item they checked off on the list. For example, a student who checked none of the boxes was given a programming experience score of 0. A student who checked four boxes received a programming experience score of 4. For "years accelerated" the students received a 0 if they were on the standard track (Intermediate Algebra in 9th grade, Geometry in 10th grade, and Algebra 2 in 11th grade). The student received a 1 if they were one year advanced (Geometry in 9th grade and Algebra 2 in 11th grade). The student received a 2 if they were two years advanced and so on. Classes after Algebra 2 are electives, and thus did not affect the student's "years accelerated" rating.

Whole Group Comparison

To answer the first research question, the pre-test and post-test scores from the entire sample were compared. The mean score increased from 3.083 on the pre-test (out of 38 points) to 20.083 on the post-test. A dependent t-test was conducted with the alternative hypothesis that the mean pre-test score was less than the mean post-test score. This produced a t score of -6.013 and a p value of less than 0.0001, demonstrating that the mean pre-test score was significantly less than the mean post-test score. Next, the mean intermediary test score of 16.208 was examined by conducting another t-test. This test, which compared the mean intermediary test score and the mean post-test score, revealed that the mean intermediary score was not significantly less than the post-test score ($p = 0.15$). There was, however, improvement overall from one assessment to the next, as evidenced by the mean scores displayed in Table 1.

Table 1

Whole Group Comparison

	Whole Group
Mean Pre-Test Score	3.083
Mean Intermediary Score	16.208
Mean Post-Test Score	20.083

Ethnicity Comparison

To answer the second research question, different groups of students were compared to see if the program visualization tools were more effective for certain populations. First, the researcher investigated ethnicity. There were eight students that identified as something other than "white" (six Asian students, one Hispanic student, and one that identified as "other"). Because there were only eight "non-white" students, the researcher drew a random sample of eight students who identified as "white" to compare to the "non-white" group. The random sample was created using a random number generator. The pre-test, intermediary test, and post-test scores are displayed in Table 2.

Table 2

Ethnicity Comparison: Pre-Test, Intermediary Test, and Post-Test Scores

	"White" Group	"Non-White Group"
Mean Pre-Test Score	4.125	1
Mean Intermediary Score	12.125	11.5
Mean Post-Test Score	18.375	16.25

Before analysis could take place, a t-test was conducted to confirm that there was no statistical difference between the mean pre-test scores of the two groups. This ensured that the groups were comparable. The alternative hypothesis was that the mean pre-test score of the "non-white" students was not equal to the mean pre-test score of the "white" students. This test resulted in a t score of -0.4903 and a p value of 0.63, which is not significant. This means that there was no statistical difference between the groups at the start of the study.

Next, the researcher conducted a t-test using the alternative hypothesis that the mean intermediary test score of the "non-white" students was not equal to the mean intermediary test score of the "white" students. The null hypothesis could not be rejected ($p = 0.92$). There was no statistical difference between the two groups on the intermediary test. This suggests that the instruction that involved no program-visualization was equally effective for the two groups.

Finally, another t-test was conducted to compare the post-test scores of the two groups. This also resulted in failing to reject the null hypothesis; there was no statistical difference between the two groups on the post-test. The p value, however, was much smaller than the p value from the previous test ($p = 0.081$). So although it cannot be concluded that program visualization was more effective for white students, these results do suggest that it might have had more influence on this group than the non-white group.

Math Confidence Comparison

The next factor examined was math confidence, which was measured by self-selected scores on the questionnaire (Appendix C). The question to assess math confidence read, "On a scale of 1-5, how would you rate your confidence level in math classes?" The choices were 1 ("Never confident"), 2 ("Confident less than half the time"), 3 ("Confident about half the time"), 4 ("Confident more than half the time"), or 5 ("Always confident"). For this comparison, "high

confidence" was a score of 4 or 5. In other words, students who said they are confident in math class more than half the time. The "low confidence" was students with scores of 1, 2, or 3. In other words, students who said they are confident in math class half the time or less.

There were 10 students who identified their math confidence as low (a score of 1, 2, or 3). A random sample of students who identified their math confidence as high (a score of 4 or 5) was drawn to achieve the same sample size for testing. The random sample was created using a random number generator. The pre-test, intermediary test, and post-test scores are displayed in Table 3.

Table 3

Math Confidence Comparison: Pre-Test, Intermediary Test, and Post-Test Scores

	High Confidence	Low Confidence
Mean Pre-Test Score	2.8	3.8
Mean Intermediary Score	20.6	13.5
Mean Post-Test Score	26.7	14.4

Once again, a t-test was conducted to confirm that there was no statistical difference between the mean pre-test scores of the two groups. The alternative hypothesis was that the mean pre-test score of the high confidence group was not equal to the mean pre-test score of the low confidence group. This test resulted in a t score of -0.1754 and a p value of 0.86. This means that there was no statistical difference between the groups at the start of the study.

Two t-tests were conducted to compare intermediary tests and post-tests between the two groups. There was no statistical difference between mean intermediary scores for the high

confidence group and low confidence group ($p = 0.23$). There was, however, a statistical difference between the mean post-test scores ($p = 0.047$), which suggests that there was a difference in how effective program visualization was for students with high confidence and students with low confidence. As seen in Table 3, it appears as though students with more confidence in math classes benefited from Jeliot more than the students with low confidence.

Middle Third Comparison

The current literature references a "middle third" theory that states the middle third of students benefit the most from program visualization tools (Ebel & Ben-Ari, 2006, p. 4). Other researchers have noticed that high achieving students do not need program visualization to succeed and low achieving students are so far behind that the program visualization is not helpful. To test this theory in this research, the participants were divided into thirds based on intermediary test scores (because there was so little difference in scores on the pre-test). The means scores of each group can be seen in Table 4. Using the intermediary test scores provided a clear delineation of the sample into three equal groups of eight participants. A t-test was conducted to confirm that there were no significant differences before instruction ($p = 0.94$). This allowed for more tests to be conducted.

To determine if the middle third of students benefited more from program visualization, a t-test was conducted to compare the intermediary test scores. The alternative hypothesis was that the mean intermediary test score for the middle third of students was greater than the mean intermediary test score for the lower third of students. This test produced a t score of 1.569 and a p value of 0.071, meaning that the mean score for middle third students was not significantly greater than the lower third students. A second t-test was conduced to determine if the mean post-test score for the middle third of students was greater than the mean post-test score for the

lower third of students. This test resulted in a t score of 2.079 and a p value of 0.03, meaning that mean post-test score for the middle third of students was significantly greater than the mean post-test score for the lower third of students. These results appear to confirm the middle third theory revealed in the literature review, and as discussed further in Chapter Five.

Table 4

Middle Third Comparison: Pre-Test, Intermediary Test, and Post-Test Scores

	"Upper Third" Group	"Middle Third" Group	"Lower Third" Group
Mean Pre-Test Score	7.875	0.875	0.5
Mean Intermediary Score	32.625	12.5	3.5
Mean Post-Test Score	33	20.25	7

Other Trends and Observations

There were some trends in the data that were noted without formal statistical analysis due to the small sample size. For example, males appeared to improve more than females during the non-program visualization instruction (represented by the intermediary test) while females appeared to improve more than males during the program-visualization instruction (represented by the post-test). This can be seen in Table 5. This trend suggests that program visualization was more effective for female students, although a statistical analysis cannot be conducted due to the small number of female students in the study (n = 5).

Table 5

Gender Comparison: Changes in Scores

	Female	Male
Mean Intermediary Score – Mean Pre-Test Score	11.2	13.63
Mean Post-Test Score – Mean Intermediary Score	6	3.32

Other trends found in the data appear to support the "middle third" theory discussed above. For example, Table 6 shows that the non-program visualization instruction (represented by the intermediary test) helped most students improve their understanding. The program visualization instruction (represented by the post-test), however, helped the un-accelerated students more than the accelerated students. This suggests the students who are not usually considered to be in the "upper third" benefited more from the program visualization than the students who normally excel.

Table 6

Acceleration Comparison: Changes in Scores

	0 Years Accelerated	1 Year Accelerated	2 Years Accelerated	3 Years Accelerated
Mean Intermediary Score – Mean Pre-Test Score	17.6	10.42	10.25	18
Mean Post-Test Score – Mean Intermediary Score	8	4.29	1.63	2.5

Although no formal qualitative research was conducted, there were some relevant observations gathered throughout the study. One participant noted on the post-test that the program visualization was not as helpful as the normal Java compiler used in the non-program visualization instruction, saying "I wrote most of my code in Dr. Java and learned from compile errors, not Jeliot." This student received a nearly perfect score (35/38) on the intermediary test, leaving little room for improvement. His comment further demonstrates the "middle third" theory; the high-achieving students are able to grasp abstract concepts through traditional instruction and therefore show no improvement when given a tool like Jeliot.

Summary

The quantitative results of this case study suggest that the program visualization part of the instruction did not significantly affect the whole group's understanding of *for loops* and *while loops*. The results do point to several differences in mean scores that suggest that program visualization might have a different effect on certain groups of students. The analysis suggests that white students might benefit differently by demonstrating increased achievement from tools like Jeliot than students who are not white. Additionally, students who feel confident in math class more than half the time might also find benefit from Jeliot differently than students with low confidence. This research added evidence to the middle third theory, showing that program visualization is most helpful for students who are neither high nor low achievers. All of this analysis could be strengthened by larger sample sizes, but overall, these results suggest that program visualization tools could be helpful for certain groups of students.

CHAPTER FIVE

Conclusions

This study was conducted to gain more insight into whether or not program visualization tools are helpful for high school students in an introductory programming course. There were two research questions that guided this study:

1. How do program visualization tools relate to understanding of *for loops* and *while loops* in an AP Computer Science course?
2. How do program visualization tools influence mastery of *for loops* and *while loops* for different potential student populations (i.e. gender, ethnicity, programming experience, academic achievement)?

This chapter will examine the conclusions that can be made based on the literature review and the results outlined in Chapter Four. Limitations of this study and suggestions for future research will also be discussed.

Research Question One

The nature of this research design made the first question difficult to answer, since it is hard to tell if students' scores improved because of exposure to Jeliot or simply from more time working with *for loops* and *while loops*. Despite this, some conclusions can be made from the results. The mean scores clearly increased after each form of instruction, although the mean post-test scores were not significantly higher than the intermediary test scores. This is, in part, due to the number of students who scored nearly all of the points on the intermediary test; these students had very little room for improvement, resulting in a small difference between the intermediary tests and post-tests. Overall, the results do not provide convincing evidence that program visualization is beneficial for *all* students, as it is clear that some student did not need it

and some students did not learn from it. This research study, then, will better address the second research question about which groups of students benefit from program visualization.

Research Question Two

Even though Jeliot was not effective for *all* students, there are still many meaningful conclusions to be made when comparing different student populations. Although the t-test comparing white and non-white students did not show a significant difference between the groups, there was a dramatic decrease in p values between the comparisons of the intermediary tests (0.92) and the post-tests (0.081). This is clearly approaching significance, suggesting that ethnicity may play a role in whether or not program visualization is effective. This is extremely important for computer science teachers to consider as they look for ways to accommodate all students in a field that is typically predominantly white. Clearly differentiation is needed to help teachers reach all students, whether students benefit from program visualization or not.

The comparison between math confidence groups also provides insight into how teachers can better accommodate students of all ability levels and backgrounds. There was no significant difference between the high confidence and low confidence groups after the non-programming visualization instruction, suggesting that this form of instruction had comparable impact regardless of how confident the students felt about math. The post-tests, however, were significantly different, meaning the program visualization had more of an impact on the students with high math confidence than the students with low math confidence. This might mean that teachers need to differentiate when implementing technology like Jeliot so that students will succeed no matter their confidence level in math. Extra scaffolding, for example, might be needed to help those students who are not confident to be able to benefit from program visualization tools.

There are other trends in the results that could impact the way that teachers implement program visualization tools in the classroom. For example, it appears that females improved their scores more than males when using Jeliot, but males improved more than females when using non-program visualization tools. Computer science teachers may want to consider this as they plan activities for students, perhaps offering a choice of Jeliot or non-Jeliot programming exercises so that students can learn with the method they prefer. The trends in the data also suggest that students who are not accelerated benefit the most from program visualization. Differentiating lessons for students based on previous math experience could make program visualization more meaningful for some students. Clearly it is important that students are given options as they work on abstract concepts, as some will find program visualization extremely helpful while others will find other methods of instruction more meaningful.

The results of this study provide more evidence for the "middle third" theory from the literature which states that the students who find program visualization most meaningful are the students who are neither low nor high achievers. The middle third and lower third of students started out statistically similar before the study began, and remained statistically similar after non-program visualization instruction. After using Jeliot, however, the middle third scored significantly better than the lower third of students. This uncovers several important considerations for computer science teachers who are trying to reach students who are falling behind. First of all, these results serve as a reminder of the importance of differentiation to help all students succeed; a tool that makes a significant impact on one group of students could have no impact on another group. It is imperative, then, that computer science teachers are using multiple representations of abstract concepts so that every student can succeed. Second, the middle third theory suggests that a different tool or strategy is needed to reach students who fall

behind when the programming concepts become more abstract. For students who are already struggling with basic syntax, program visualization will be more confusing than clarifying. The middle third theory is an important reminder that differentiation and scaffolding are absolutely necessary to help *all* students succeed.

Limitations

The biggest limitation of this study was the small sample size. Unfortunately enrollment in AP Computer Science was low when this study took place, resulting in a sample of only 24 students with even smaller subgroups. For example, a comparison of male and female students was impossible to conduct because only five female students participated in the study. The small sample size also prevents any conclusions from being generalized to the population of all AP Computer Science students.

The method of instruction also created some limitations. The researcher provided instruction through videos, which allowed for no interaction with the students. The instructor was able to answer questions on the practice assignments, but could not adapt the video recorded instruction to meet the needs of students like a live teacher would do. The instructor noted that the students were engaged, but it was difficult to tell if they were confused or had questions because the instruction was recorded. Ideally, the researcher would have been present in the class to present the lessons instead of presenting via video.

Another limitation of the study was the design of the study. Students took the post-test after using a program visualization tool, but it is difficult to tell if an improvement in their scores is due to the tool or the extra time spent on *for loops* and *while loops*. It was impossible, however, to design the experiment with a control group, as all students had to learn the same

content in the same manner. This research also only used one kind of program visualization tool, which limits the results to conclusions about Jeliot, not program visualization as a whole.

Suggestions For Future Research

This research design combined with qualitative data would make the results much richer and more helpful for teachers. Because the researcher was not present for the instruction, there were no direct observations made about the students' engagement with or opinion of Jeliot. Comments from the instructor were noted, but more qualitative evidence could help answer more specific questions about the students' experiences using Jeliot. Future studies could include direct observations or feedback from students to create a mixed methods design.

A different research design might lead to results that could more accurately answer the first research question. Future research should be designed to include a control group of students that learn abstract concepts without program visualization tools. An experimental group would learn the same material while using program visualization tools. Comparing the learning between these two groups would produce better conclusions about whether or not program visualization helps students learn because the variable of extra time would not be a factor. This type of research design was implemented in much of the existing literature, but rarely at the high school level.

Replication with larger sample sizes could add more evidence to the effectiveness of program visualization on different groups of students. The small sample size of this study made analysis difficult. Future research could examine the difference in understanding of females vs. males and experienced programmers vs. non-experienced programmers, as well as investigate other factors such as career interest and post-high school plans. This information could help computer science teachers create lessons that are meaningful for every group of students.

The evidence this study provides of the middle third theory exposes the importance of finding an intervention for the "lower third" of students who are so far behind that program visualization does not help them. Future research should investigate other tools, teaching strategies, and interventions that could help this population of students who struggle with the basics of programming. By differentiating instruction with different tools and interventions, teachers can reach more students and help them succeed regardless of their previous experience in the class.

Summary

This study examined the effectiveness of program visualization on students' understanding of *for loops* and *while loops* and determined if certain groups of students experienced varying effects of the program visualization tool being used. The results for the whole group of students were not significant, leading to no conclusion on whether or not program visualization helps students understand *for loops* and *while loops*. There were, however, some significant differences between some populations of students. It can be concluded that math confidence is a factor that influences the effectiveness of program visualization, and it appears as though ethnicity may be a factor as well. Additionally, there was significant evidence that students the middle third of students benefit the most from program visualization. All of these conclusions stress the importance of differentiation so that all students can understand abstract concepts and suggest that other interventions need to be investigated to help the students who do not benefit from program visualization.

References

Ben-Bassat Levy, R., Ben-Ari, M., Uronen, P. A. (2003). The jeliot 2000 program animation system. *Computers & Education (40)*, 1-15.

Bergin, S. & Reilly, R. (2005). Programming: Factors that influence success. *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 4111-415. Doi: 10.1145/1047344.1047480

Butler, M. & Morgan, M. (2007). Learning challenges faced by novice programming students studying high level and low feedback concepts. *Proceedings of ascilite Singapore*, 99-107. Retrieved from http://www.ascilite.org/conferences/singapore07/procs/butler.pdf

Cisar, S. M., Radosav, D., Pinter, R., & Cisar, P. (2011). Effectiveness of program visualization in learning java: A case study with jeliot 3. *International Journal of Computers, Communications & Control, (6)*, 668 – 680.

Creswell, J. W. (2014). *Educational research: Planning, conducting, and evaluating qualitative and quantitative research* (5th ed.). Los Angeles: Sage.

Ebel, G. & Ben-Ari, M. (2006) Affective effects of program visualization. *Proceedings of the Second International Workshop on Computing Education Research*, 1-5. Doi: 10.1145/1151588.1151590

Hongwarittorrn, N. & Krairit, D. (2010). Effects of program visualization (Jeliot3) on students' performance and attitudes towards Java programming. Paper presented at the spring 8th International conference on Computing, Communication and Control Technologies, Orlando, Florida, USA. Retrieved from http://www.iiis.org/CDs2010/CD2010IMC/CCCT_2010/PapersPdf/TA750PM.pdf

Hundhausen, C. D., Douglas, S. A., & Stasko, J. T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages and Computing, (13)*, 259-290. Doi: 10.1006/S1045-926X(09)00028-9

Kehoe, C., Stasko, J., & Taylor, A. (2001). Rethinking the evaluation of algorithm animations as learning aids: An observational study. *International Journal of Human-Computer Studies, 54*, 265 - 284. Doi: 10.1006/ijhc.2000.0409

Kranch, D. A. (2011). Teaching the novice programmer: A study of instructional sequences and perception. *Education and Information Technologies*, *17*, 291-313. Doi: 10.1007/x10639-011-9158-8

Lahtinen, E., Jarvinen, H., & Melakoski-Vistbacka, S. (2007) Targeting program visualizations. *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 256-260. Doi: 10.1145/1268784.1268858

Lahtinen, E., Ahoniemi, T., & Salo, A. (2007). Effectiveness of integrating program visualizations to a programming course. *Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (88)*, 195-198.

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. Doi: 10.1080/08993408.2011.554722

Moreno, A., & Joy, M. S. (2007). Jeliot 3 in a demanding educational setting. *Electronic Notes in Theoretical Computer Science (178)*, 51-59.

Moreno, A.., Sutinen, E., & Joy, M. (2014). Defining and evaluating conflictive animations for programming education: The case of Jeliot ConAn. *Proceedings of the 45th ACM*

Technical Symposium on Computer Science Education, 629-634. Doi: 10.1145/2538862.2538888

Naps, T.L., Rossling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S., & Valezquez-Iturbide, J. A. (2002) Exploring the role of visualization and engagement in computer science education. *Working Group Reports from ITiCSE on Innovation and Technology in Science Education*, 131-152. Doi: 10.1145/960568.782998.

Rajala, T., Laakso, M., Kaila, E., & Salakoski, T. (2008). Effectiveness of program visualization: A case study with the ViLLE tool. *Journal of Information Technology Education*, *7*, 15-32.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13* (2), 137-172. Doi: 10.1076/csed.13.2.137.14200

Rountree, N., Rountree, J., & Robins, A. (2002). Predictors of success and failure in a CS1 course. *ACM SIGCSE Bulletin*, 121-124. Doi: 10.1145/820127.820182.

Sorva, J., Karavirta, V., & Malmi, L. (2013). A review of generic program visualization systems for introductory programming education. *ACM Transactions on Computing Education, 13* (4), 1-64. Doi: 10.1145/2490822

Wang, P., Bednarik, R., & Moreno, A. (2008). During automatic program animation, explanations after animations have greater impact than before animations. *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, 100 – 109. Doi: 10.1145/2401796.2401808

Wiedenbeck, S., Fix, V., & Scholtz, J. (1993). Mental representations of programs by novices and experts. *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, 74-79. Doi: 10.1145/169059.169088

Appendix A

Pre-Test and Intermediary Test

1) Write a code fragment that prints multiples of 6 from 12 up to 60 (including both 12 and 60). Each number should appear on a separate line. Do this using a while loop.

2) Write a code fragment that prints the following:

```
* , , , , ,
, * , , , ,
, , * , , ,
, , , * , ,
, , , , * ,
, , , , , *
```

Do this using nested for loops.

3) Assume that the following code has already been executed:

String letter = "t";

Write a code fragment that determines and prints the number of times that letter appears in a String object called word. Assume that word has already been declared and assigned a String value. Do this twice; once with a for loop and once with a while loop.

Use a for loop and a while loop.

Appendix B

Post-Test

1) Write a code fragment that prints multiples of 7 backwards from 77 to 14 (including both 77 and 14). Each number should appear on a separate line. Do this using a while loop.

2) Write a code fragment that prints the following:

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
```

Do this using nested for loops.

3) Assume that the following code has already been executed:

String smWord = "as";

Write a code fragment that determines and prints the number of times that smWord appears in a String object called word. Assume that word has already been declared and assigned a String value. Do this twice; once with a for loop and once with a while loop.

Use a for loop and a while loop

Appendix C

Questionnaire

What grade level are you currently in? (circle one)

9th grade
10th grade
11th grade
12th grade

How old are you? _____

What is your gender? _____

What is your race? (circle one)

American Indian or Alaskan Native | Asian | Black or African American | Native Hawaiian or Other Pacific Islander | White | Other

Are you of Hispanic, Latino, or Spanish origin? (circle one)

Yes
No

Which of the following programming-related activities had you done before this semester? (check all that apply)

- Started a programming tutorial on Code.org but did not finish
- Completed one or more programming tutorials on Code.org
- Started a programming tutorial on Codecademy but did not finish
- Completed one or more programming tutorials on Codecademy
- Started a programming tutorial on a different website
  - write it here: _____
- Completed one or more programming tutorials on a different website
  - write it here: _____
- Taken a class on programming
- Read a book about programming
- Watched YouTube videos about programming
- Learned a programming language well enough to write code
  - which language or languages? _____
- Other:

How many classes are you taking this semester that are honors or AP (including AP Computer Science)? Include any honors-options you intend to complete. _____

How many classes are you taking this semester that are non-honors? _____
Next to each grade, write the math class (or classes) you took that year. If you didn't take a math class that year or if you haven't reached that grade yet, write an "x" in the blank.

Grade	Math Class(es) Taken That Year
9th Grade	
10th Grade	
11th Grade	
12th Grade	

On a scale of 1-5, how would you rate your confidence level in math classes?

1	2	3	4	5
Never confident	Confident less than half the time	Confident about half the time	Confident more than half the time	Always confident

Appendix D

Rubric

Question 1: Printing Multiples

	2	1	0
Initialization *Ex: int i = 12;*	A variable is initialized that keeps track of the number of times through the loop. The initialization leads to the correct answer.	A variable is initialized that keeps track of the number of times through the loop. The initialization leads to an incorrect answer.	There is no initialization of a variable to keep track of the number of times through the loop.
Condition *Ex: while (i <=60)*	The while loop includes a condition for the initialized variable. The condition leads to the correct answer.	The while loop includes a condition for the initialized variable. The condition leads to an incorrect answer.	There is not condition for the initialized variable.
Increment *Ex: i++;*	The initialized variable is incremented before the loop restarts. The increment is in the correct place that leads to the correct answer.	The initialized variable is incremented before the loop restarts. The increment is in the wrong place that leads to the correct answer.	There is no increment for the initialized variable.
While Loop Content	The content of the while loop is correct and leads to the correct answer.	The content of the while loop is incorrect and leads to an incorrect answer.	There is no content of the while loop.

Question 2: Nested For Loops

	2	1	0
Outer Loop *Ex: for (int r=1; r <= 6; r++)*	The outer loop consists of all three parts of the for loop. The outer loop leads to the correct answer.	The outer loop consists of all three parts of the for loop. The outer loop leads to an incorrect answer.	There is no outer loop. Or the outer loop does not consist of all three parts of the for loop.
Inner Loop *Ex: for (int c=1; c<=6; c++)*	The inner loop consists of all three parts of the for loop. The inner loop leads to the correct answer.	The inner loop consists of all three parts of the for loop. The inner loop leads to an incorrect answer.	There is no inner loop. Or the inner loop does not consist of all three parts of the for loop.
Nested For Loop Content	The content of the nested for loops is correct and leads to the correct answer.	The content of the nested for loops is incorrect and leads to an incorrect answer.	There is no content of the nested for loops.

Question 3: The accumulator algorithm

For Loop:	2	1	0
Initialization	A variable is initialized that keeps track of the number of times through the loop. The initialization leads to the correct answer.	A variable is initialized that keeps track of the number of times through the loop. The initialization leads to an incorrect answer.	There is no initialization of a variable to keep track of the number of times through the loop.
Counter *Ex: int total = 0;*	A counter is initialized to keep track of the number of letters. The counter initialization leads to the correct answer.	A counter is initialized to keep track of the number of letters. The counter initialization leads to an incorrect answer.	There is no counter initialized.
Condition	The while loop includes a condition for the initialized variable. The condition leads to the correct answer.	The while loop includes a condition for the initialized variable. The condition leads to an incorrect answer.	There is not condition for the initialized variable.
Counter Increment *Ex: total = total + 1;*	The counter is incremented whenever a letter is found. This counter increment leads to the correct answer.	The counter is incremented whenever a letter is found. This counter increment leads to an incorrect answer.	There is no increment of the counter.
Increment	The initialized variable is incremented before the loop restarts. The increment is in the correct place that leads to the correct answer.	The initialized variable is incremented before the loop restarts. The increment is in the wrong place that leads to the correct answer.	There is no increment for the initialized variable.
For Loop Content	The content of the for loop is correct and leads to the correct answer.	The content of the for loop is incorrect and leads to an incorrect answer.	There is no content of the for loop.

While Loop:	2	1	0
Initialization *Ex: int i = 0;*	A variable is initialized that keeps track of the number of times through the loop. The initialization leads to the correct answer.	A variable is initialized that keeps track of the number of times through the loop. The initialization leads to an incorrect answer.	There is no initialization of a variable to keep track of the number of times through the loop.
Counter *Ex: int total = 0;*	A counter is initialized to keep track of the number of letters. The counter initialization leads to the correct answer.	A counter is initialized to keep track of the number of letters. The counter initialization leads to an incorrect answer.	There is no counter initialized.
Condition *Ex: while (i<word.length)*	The while loop includes a condition for the initialized variable. The condition leads to the correct answer.	The while loop includes a condition for the initialized variable. The condition leads to an incorrect answer.	There is not condition for the initialized variable.
Counter Increment *Ex: total = total + 1;*	The counter is incremented whenever a letter is found. This counter increment leads to the correct answer.	The counter is incremented whenever a letter is found. This counter increment leads to an incorrect answer.	There is no increment of the counter.
Increment *Ex: i++;*	The initialized variable is incremented before the loop restarts. The increment is in the correct place that leads to the correct answer.	The initialized variable is incremented before the loop restarts. The increment is in the wrong place that leads to the correct answer.	There is no increment for the initialized variable.
While Loop Content	The content of the while loop is correct and leads to the correct answer.	The content of the while loop is incorrect and leads to an incorrect answer.	There is no content of the while loop.