Purdue University
Purdue e-Pubs

Open Access Theses | Theses and Dissertations

Fall 2014

Impact Of A Visual Programming Experience On The Attitude Toward Programming Of Introductory Undergraduate Students

Saurabh Godbole
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_theses

Part of the Computer Sciences Commons, Educational Assessment, Evaluation, and Research Commons, and the Higher Education Commons

Recommended Citation
Godbole, Saurabh, "Impact Of A Visual Programming Experience On The Attitude Toward Programming Of Introductory Undergraduate Students" (2014). *Open Access Theses*. 327.
https://docs.lib.purdue.edu/open_access_theses/327

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Saurabh Godbole

Entitled
IMPACT OF A VISUAL PROGRAMMING EXPERIENCE ON THE ATTITUDE TOWARD PROGRAMMING OF INTRODUCTORY UNDERGRADUATE STUDENTS

For the degree of Master of Science

Is approved by the final examining committee:

Alka Harriger

Bradley Harriger

Grant Richards

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Alka Harriger

Approved by: Jeffrey Whitten 12/05/2014

Head of the Department Graduate Program Date

IMPACT OF A VISUAL PROGRAMMING EXPERIENCE ON THE ATTITUDE TOWARD PROGRAMMING OF INTRODUCTORY UNDERGRADUATE STUDENTS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Saurabh Godbole

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

December 2014

Purdue University

West Lafayette, Indiana

To my entire family. Thank you for standing by me through ups and downs.

ACKNOWLEDGEMENTS

I would like to thank my major professor, Alka Harriger, for her invaluable input on each and every step of this research project. From the very first day of my graduate studies, Prof. Harriger created for me opportunities ranging from being a Teaching Assistant to conducting outreach sessions for High School students. I am grateful for her constant support throughout my journey as a graduate student. Also, I would like to thank Professor Bradley Harriger and Dr. Grant Richards for their timely feedback, advice, and support for my research work, especially for creating microcontroller demonstration unit for the experiment conducted during this study.

I also want to thank Prof. Guity Ravai for allowing me to conduct my study with the students in CNIT 15501. Without her consent, this research would have been impossible.

Also, I want to thank Rongrong Zhang from the Statistical Consulting Service in the Department of Statistics, Purdue University. Her tremendous help was critical in analyzing the data by creating a statistical model.

I want to thank my grandparents who have always believed in me and encouraged me throughout these years at Purdue.

I would also like to thank Swaroopa Kanade for her steadfast support, constant encouragement, and incredible patience during discussions on student interests, programming, and statistical analyses.

I want to thank all who have made these years at Purdue unforgettable.

TABLE OF CONTENTS

Page

LIST OF TABLES vi
LIST OF FIGURES vii
LIST OF ABBREVIATIONS viii
GLOSSARY ix
ABSTRACT xiii
CHAPTER 1. INTRODUCTION 1
1.1 Background 1
1.2 Significance 2
1.2.1 Current Techniques for Teaching Programming 3
1.2.2 Improving Student Understanding of Logic Using Flowcharts 4
1.3 Research Question 5
1.4 Assumptions 5
1.5 Limitations 6
1.6 Delimitations 6
1.7 Summary 7
CHAPTER 2. REVIEW OF RELEVANT LITERATURE 8
2.1 Importance of Procedural Programming and Graphical Elements 9
2.2 Research on Programming Tools to Improve Learning 11
2.3 Graphical Programming Tools and Their Impact on Student Learning 18
2.4 Physical Implementation of Graphical Programming Languages 30
2.5 Past Research on Student Attitudes Toward Programming 35
2.6 Summary 38
CHAPTER 3. METHODOLOGY 40
3.1 Experimental Setup 40
3.2 Hypotheses 44
3.3 Participants 45
3.4 Methodology 46
3.4.1 IRB 47
3.4.2 Procedures 47
3.5 Privacy and Confidentiality of the Participant Data 51
3.6 Statistical Analysis 51
3.7 Summary 53

Page

CHAPTER 4. PRESENTATION OF THE DATA AND FINDINGS 54
4.1 Data Preparation and Analysis 54
4.2 Test of Significance for the Dataset 58
4.3 Equivalence Testing 63
4.4 Qualitative Analysis of the data 64
4.5 Summary 67

CHAPTER 5. CONCLUSIONS, DISCUSSION, AND RECOMMENDATIONS ... 68
5.1 Conclusions 68
5.2 Implications of the Study 70
5.3 Challenges of using Graphical Programming Languages and Student Comprehension 71
5.4 Future Work and Recommendations 72

LIST OF REFERENCES 74

APPENDICES
Appendix A: Approval to Use Attitude Survey 81
Appendix B: Instructor Approval 82
Appendix C: IRB Approval for Research 83
Appendix D: Participant Pre- and Post-Instructional Surveys 84
Appendix E: Attitude Category and Related Questions 87

LIST OF TABLES

Table Page

2-1 Visual Programming Language Tools and their Functionality ... 12
3-1 Timetable for Instructional Session ... 49
3-2 Descriptive and inferential Data Collected for Each Participant ... 52
4-1 Demographics of the Participant ... 55
4-2 Coded Values Survey Participant Responses ... 57
4-3 Coded Value for Participant Grade Level ... 59
4-4 Coded Values for Participant Gender ... 59
4-5 P-Values for Attitude Changes Between Surveys for All Participants ... 61
4-6 Attitude Changes Between Survey Differentiated by Gender ... 62
4-7 Average Attitudes By Categories Measured ... 63
4-8 Confidence Interval for all Significance Testing ... 63
4-9 Goodness Probability of Frequent Words in Pre Instructional Survey ... 65
4-10 Goodness Probability of Frequent Words in Post Instructional Survey 1 ... 65
4-11 Goodness Probability of Frequent Words in Post Instructional Survey 2 ... 66
4-12 Goodness Probability of Repeated Words on All Three Surveys ... 66
5-1 Results of Hypothesis Testing ... 67

Appendix Table

D-1 Pre- and Post-Instructional Survey Questions ... 85

LIST OF FIGURES

Figure Page

4-1 Q-Q Plot of the Responses 58
4-2 Change in Attitude Over the Research Period 61
4-3 Attitude Changes Based on Participant Gender 62

LIST OF ABBREVIATIONS

BASIC	−	Beginner's All Purpose Symbolic Instruction Code
COMMS	−	Communication
CPU	−	Central Processing Unit
EEPROM	−	Electronically Programmable Read Only Memory
I/O	−	Input / Output
IDE	−	Integrated Development Environment
IT	−	Information Technology
IRB	−	Institutional Review Board
OOP	−	Object Oriented Programming
RAM	−	Random Access Memory
RAPTOR	−	Rapid Algorithmic Prototyping Tool for Ordered Reasoning
SICAS	−	Interactive System for Algorithm Development and Simulation
WYSIWYC	−	What You See Is What You Code

GLOSSARY

Algorithm – "Any well-defined computational procedure that takes some value, or set of values, as input and, through a series of processes, produces some value, or set of values, as output" (Cormen, Leiserson, Rivest, & Stein, 2009, p. 5)

Alice – "An innovative 3D programming environment that makes it easy to create an animation for telling a story, playing an interactive game, or a video to share on the web" ("What is Alice?," 2014).

Arduino – "An open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board" ("What is Arduino?," 2014).

Central Processing Unit – "The unit of a computer system, which fetches, decodes and executes programmed instructions" ("IEEE Standard Glossary of Computer Hardware Terminology," 1995).

Compiler – "A computer program that translates programs expressed in a high order language into their machine language equivalents" ("IEEE Standard Glossary of Software Engineering Terminology," 1990).

CPU – See Central Processing Unit.

Critical Thinking – "Critical thinking is defined as a process of reflective thinking that goes beyond logical reasoning to evaluate the rationality and justification for actions within context" (Forneris & Peden-McAlpine, 2007).

Debug – "To detect, locate, and correct faults in a computer program" ("IEEE Standard Glossary of Software Engineering Terminology," 1990).

Decisions – An essential program control structure that denotes a branch point in the logic in which the path to follow is predicated by a Boolean condition.

Eclipse Graphical Modeling Framework (GMF) – "Provides a generative component and runtime infrastructure for developing graphical editors" (Golubev, Istria, & Irawan, 2014).

Envigilator – “An assignment level learning analytics system, which captures screenshot of the users every set number of seconds, which can viewed live by a proctor” (Lutes, 2013).

Flowchart – “A formalized graphic representation of a program’s logic process” (Aguilar-Savén, 2004).

Iconic Programmer – “[A]…learning and development tool for introductory programming in flowcharts, Java, Turing, and more…[which] eliminates the overhead of programming – no syntax errors and no text editors or compilers – and allows [one] to focus on algorithm development” (Chen, n.d.).

IDE – See Integrate Development Environment.

Integrated Development Environment – “Applications that present many of the tools required for creating software within a single user interface” (Kenefick, 2011)

Interactive System for Algorithm Development and Simulation – “[A] system [that] allows students to implement algorithms to solve problems, using a flowchart representation” (Santos, Gomes, & Mendes, 2010).

Interpreter – “A computer program that translates and executes each statement or construct of a computer program before translating and executing the next” (“IEEE Standard Glossary of Software Engineering Terminology,” 1990).

LabVIEW – “A graphical programming platform that helps engineers scale from design to test and from small to large systems” (“What is LabVIEW?,” 2014).

Logic – “Reasoning conducted or assessed according to strict principles of validity” (“logic,” 2014).

Loops – An essential program control structure that involves repeating a sequence of one or more program instructions.

Microcontroller – “A CPU plus random access memory (RAM); electrically erasable, programmable, read only memory (EEPROM); inputs/outputs (I/O); and communication (Comms)” (Park, 2003, p.1-2).

NanoNavigator – “…[A] software tool for all setup, programming…the [NanoLine] programmable logic module [using flowchart based programming language]” (“Quick and easy programming,” 2014).

Object-Oriented Programming – “Programming in terms of a collection of discrete objects that incorporate both data and behavior” (Nikishkov & Kanda, 1999).

PORTUGOL – “An integrated development environment (IDE) for structured programming [which] incorporates the ability to generate structured program statements by creating corresponding flowcharts” (de Jesus, 2011).

Procedural Language – “A programming language in which the user states a specific set of instructions that the computer must perform in a given sequence” (“IEEE Standard Glossary of Software Engineering Terminology,” 1990).

Program Code – “In software engineering, computer instructions and data definitions expressed in a programming language or in a form output by an assembler, compiler, or other translator” (“IEEE Standard Glossary of Software Engineering Terminology,” 1990).

Programming – “The transforming of logic and data from design specifications (design descriptions) into computer applications and software” (“IEEE Standard Glossary of Software Engineering Terminology,” 1990).

Programming Language – “Any language used to create a set of instructions for a computer to follow in carrying out a task, a framework to use in solving a problem, when that solution is storable for future use” (DiNitto, S.A., 1988).

Rapid Algorithmic Prototyping Tool for Ordered Reasoning. – “[T]ool [that] allows students to create programs using basic flowcharting symbols” (Carlisle, Wilson, Humphries, & Hadfield, 2005)

RAPTOR – See Rapid Algorithmic Prototyping Tool for Ordered Reasoning.

Robot – “A mechanical device that can be programmed to perform some task of motion under automatic control” (“IEEE Standard Glossary of Computer Hardware Terminology,” 1995).

Scratch – A programming tool designed for young children, which enables the creation of an animation using program-based blocks that snap together like puzzle pieces.

Sentiment analysis – The “computational study of people’s opinions, appraisals, attitudes, and emotions toward entities, individuals, issues, events, topics and their attributes” (Liu & Zhang, 2012).

SICAS – See Interactive System for Algorithm Development and Simulation.

Software – “Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system” (“IEEE Standard Glossary of Software Engineering Terminology,” 1990).

Splish – “Icon-based programming on a PC, compiles the visually-created program into an object code for a stack based virtual computer, transfers the object code to the target Arduino board via the USB interface, and executes the object code by the interpreter located on the Arduino board” (Kato, 2010).

Syntax – “The structural or grammatical rules that define how the symbols in a language are to be combined to form words, phrases, expressions, and other allowable constructs” (“IEEE Standard Glossary of Software Engineering Terminology,” 1990).

Tools – Related to programming, a tool is typically the software development environment in which one writes a computer program.

Visualization – “Visualization is defined as representations of information consisting of spatial, non-arbitrary (i.e. "picture-like" qualities resembling actual objects or events), and continuous (i.e. an "all-in-oneness" quality) characteristics” (Rieber, 1995).

WHILE – “A small imperative programming language whose programs are based on a signature Σ and are made from assignments, sequential composition, conditional statements, and while statements” (Daintith, 2004).

What You See Is What You Code – “A programming tool that allows the programmer to write program instructions using basic code while manipulating visual program objects” (Hundhausen & Brown, 2007)

WYSIWYC – See What You See Is What You Code.

ABSTRACT

Godbole, Saurabh S. M.S., Purdue University, December 2014. Impact of Programming Cyber-physical Systems on the Interest Level of Freshmen College Students. Major Professor: Alka Harriger.

Traditionally, textual tools have been utilized to teach basic programming languages and paradigms. Research has shown that students tend to be visual learners. Using flowcharts, students can quickly understand the logic of their programs and visualize the flow of commands in the algorithm. Moreover, applying programming to physical systems through the use of a microcontroller to facilitate this type of learning can spark an interest in students to advance their programming knowledge to create novel applications. This study examined if freshmen college students' attitudes towards programming changed after completing a graphical programming lesson. Various attributes about students' attitudes were examined including confidence, interest, stereotypes, and their belief in the usefulness of acquiring programming skills. The study found that there were no statistically significant differences in attitudes either immediately following the session or after a period of four weeks.

CHAPTER 1. INTRODUCTION

This chapter gives a basic overview of the research, defining the research question. It also delineates the scope and the significance of the study in addition to the assumptions, limitations, and delimitations that form the basis of this research project.

1.1 Background

Many college freshmen embarking on a computing major may have little to no background in programming (Bevan, Werner, & McDowell, 2002). Students may experience difficulty in grasping many programming concepts stemming from the nebulous and abstract nature of these topics; therefore, solving this problem warrants a new approach.

Microcontroller technology has revolutionized the world of information technology (IT). Devices have continued to become more and more complex, and at the same time, their functionality is increasing. These advances create an opportunity to utilize such technology in a pedagogical setting, increasing the instructional effectiveness. These devices can be used to make programming concepts easy to understand, relevant, and still teach the basic theoretical constructs by making the results of programming more tangible to the students. Present-day introductory software development courses that focus only on

teaching concepts and creating logical programs to reinforce lessons learned are ideal candidates for using tools that enable creation of physical computing applications.

1.2 Significance

While working as a teaching assistant, the researcher witnessed students' difficulty in grasping many programming concepts stemming from the ambiguous and theoretical nature of such topics. Therefore, the researcher decided to study the attitudes of students toward programming, as attitude can have a tremendous impact on their performance. A solution to the abstract nature of programming can be provided by using visual learning in programming classes (Robins, Rountree, & Rountree, 2003). Mateas (2005) argued that programming is a fundamental component of "procedural literacy". The researcher noted that "… the ability to read and write processes…" is crucial to "…understand interplay between…human meaning-making and technically-mediated processes" (Mateas, 2005). Therefore, the study of programming can be viewed as an essential building block of logical thinking.

Since the beginning of Information Technology as a discipline, research has been done on how to best teach the fundamentals of programming with greater comprehension and retention of concepts (Burton & Bruhn, 2003). At the advent of procedural languages, flowcharts were regarded as one of the best tools to assist novice programmers to learn and master the methodical thinking required for complex programming tasks (Robins et al., 2003). Due to its inherent property of visualization, this type of aid can assist students in creating the logical

flow of a program without learning a particular programming language. Flowcharts can even be drawn on paper, so they can help students visualize the logical flow of the commands through a computer-based application even without knowledge of any specialized software.

1.2.1 Current Techniques for Teaching Programming

Many typical introductory courses focus on writing an application in a particular programming language. Novice programmers are exposed to a particular syntax with elaborate examples to illustrate intricacies of the specific language. Generally, first year college students become familiarized with a particular programming language, but their problem-solving skills stay undeveloped through this approach. Students in introductory programming courses are often tasked with complicated projects, which require a higher level of understanding with an ability to decompose problems into smaller chunks in systematic fashion. In the past, computer science students were taught procedural languages such as BASIC in the first programming course (DiNitto, S.A., 1988); but, currently, students in programming courses are generally taught an object-oriented (OO) programming language, such as Python (Robins et al., 2003). Although object-oriented and procedural paradigms may seem different, OO still involves considerable procedural coding. If decomposed into smaller pieces, the flow of logic in the methods used in OO languages is sequential in nature (Gosling, Steele, Joy, Bracha, & Buckley, 2013). Therefore, if students learn how to program using flowcharts, they will understand the procedural flow.

1.2.2 Improving Student Understanding of Logic Using Flowcharts

As noted previously, flowcharting, although very basic, can be extremely beneficial for novice programmers to think in a process-oriented manner. One can employ hardware (microcontroller) technologies in order to reinforce the foundational concepts of programming. The literature suggests that this approach seems promising (Carlisle et al., 2005; Dabroom, Refie, & Matmti, 2013; Goadrich, 2014). As students embark on information technology related careers, combining both visual and hands-on approaches to teach programming to college freshmen can lead to innovative solutions to problems (Chun & Ryoo, 2010; Hwang, Su, & Tseng, 2010).

Based on the discussion above, the researcher studied the impact of teaching a flowchart-based software tool to novice programmers. Graphical programming software was used in direct conjunction with a programmable microcontroller. By using a visual language that can enable interaction with a physical medium, students were be able to see the actual results of their flowchart program. As stated in Chapter 3, a quantitative analysis of student feedback was used to determine if their interest had increased in the programming discipline. Examining the results of this study may also help educators review the student attitudes toward programming and create a curriculum that appeals to their interests by revising their approaches in introductory programming classes.

1.3 Research Question

The research question for this study is:

- Can student interest in learning basic programming concepts be increased through the use of microcontroller technology and flowchart programming?

To answer this question, as noted above, visual learning in programming classes can be utilized. Using graphical software programming tools, students can picture their creations to understand the basic programming concepts. Graphical elements such as flowcharts capture the procedural flow of commands through a program. Using such a technique can help students think in a logical manner, improving their understanding of structure and flow of a program (Crews & Butterfield, 2002). This increased understanding may spark interest in college freshmen to further their knowledge of programming in a different paradigm, while improving their learning. Coupling such an approach with microcontroller technology can help students witness their creations, “things” they can touch and feel. More importantly, this approach may expand the boundaries of student innovation.

1.4 Assumptions

An instructional session was provided to participants in which they created a program for a microcontroller using a graphical programming language. After the session, data was collected using Likert Scale based surveys. The following assumptions have been made:

1. Each participant will work individually and not be influenced by other participants.
2. Participants will be able to learn at least two basic concepts, decisions and loops of graphical programming language within the instructional period.
3. The participants lack programming knowledge prior to the study.
4. All participants will be honest while answering survey questions.
5. Because the software chosen for the study runs only in Windows Operating System (version 7 or less), all participants will be able to use a computer with a Windows™ environment.

1.5 Limitations

The limitations of this study are noted below.

1. If participants have prior programming experience, it may impact the study results.
2. The study was carried out over the period of four weeks. The instructional session and pre- and first post-instructional surveys were administered only on the first day. Therefore, the length of study may affect results.

1.6 Delimitations

The delimitations of this study are as follows:

1. To facilitate the feasibility of the study, only students attending Purdue University who are enrolled in CNIT 15501 during Fall 2014 will be used

as participants. This should result in study subjects mainly between the ages of 19-25.

2. Due to the small size of the sample, the generalization of the results may be limited.
3. The research only studies freshmen. This may limit the generalization of the results.
4. The research only presents the interest levels of the participants. It does not claim to predict the future performance of participants in programming classes.
5. The research study is conducted only using the equipment stated in the Methodology chapter.

1.7 Summary

This chapter provided the background for analyzing the student attitudes toward programming. Research suggests that retention of programming concepts can be increased by incorporating physical hardware devices in the coursework (Carlisle et al., 2005; Dabroom, Refie, & Matmti, 2013; Goadrich, 2014). As noted above, if students are provided with novel technology, their interest in programming may change, possibly leading to attitude changes. An analysis of student attitudes was undertaken to test this theory. The overall background of this study is stated in the previous sections in addition to any assumptions by the researchers. The scope of this study is limited by previously stated limitations and delimitations.

CHAPTER 2. REVIEW OF RELEVANT LITERATURE

In the last three decades, the demand for programmers has increased (Robins et al., 2003). Because computing programs require at least one programming course, all new students in this field are required to successfully learn programming (Robins et al., 2003). Those with an initial lack of understanding and background related to programming may encounter difficulties in the course. As a result, programming courses are often cited as difficult and, historically, tended to have high dropout rates (Smith & Delugach, 2010).

Although there are many schools of thought related to how programming should be taught in an introductory course, there is consensus about the importance of programming (Robins et al., 2003). This knowledge is important because it leads to the development of analytical and problem-solving abilities in students. Due to the abstract nature of the topic, it can also promote creative thinking. Therefore, understanding the various approaches for teaching programming may be especially helpful to instructors of first year computing students.

2.1 Importance of Procedural Programming and Graphical Elements

In the early days of programming, procedural programming languages were the norm, but in the last 30 years, object-oriented programming (OOP) has been the leading paradigm used to teach programming to students (White & Sivitanides, 2005). Object-oriented programming languages, for example, Java or C++, have been at the center of this change in teaching technique. Robins, Rountree, and Rountree (2003) note that this OOP approach may be popular because of the real-life like constructs or user-friendliness. Nonetheless, researchers argue that the process of identifying objects is not easy, and further, correlating problem domain and program domain objects is a cumbersome process (Robins et al., 2003). This may explain why learning object-oriented programming is especially challenging for novice programmers.

Robins et al. (2003) cite a study that analyzed the level of comprehension of procedural and object-oriented programs. The participants in the study were second semester college students and were learning different programming languages, either PASCAL or C++. These subjects were then quizzed on the code written in the language they were taught in their respective course. There were no significant differences in the level of understanding when subjects were given smaller programs. On the other hand, when given longer and more intricate programs, students learning the procedural language performed better in all areas studied by researchers. The researchers also noted that novice programmers may develop a good understanding of how a small problem may be solved by the OO paradigm but longer and more complex programs may require

a representation in a procedural flow of instructions (Robins et al., 2003). Therefore, creating a procedural depiction may help new programmers more easily identify and solve complex problems.

Object-oriented programming languages inherently have a low coupling of methods, making them very distributed. A study by Wiedenbeck, Ramalingam, Sarasamma, and Corritore (1999) suggests that the program flow and distributed functions in an object-oriented program may make the program's logic difficult to understand for novice programmers. A corresponding program in a procedural language, though, can make it easier to picture a conceptual depiction of the logic. Some of the literature reviewed does lend support to the claim that the concept of object-oriented programming is an easier way of envisioning and creating solutions to real world problems (Burton & Bruhn, 2003; Wiedenbeck et al., 1999).

Many information technology students find it tough to master the craft of programming because this requires the fundamental knowledge of conceptual thinking, problem solving, and mathematics (Winslow, 1996). In addition to deciphering the unclear nature of the various tasks involved, students must learn the specific semantic conventions of the language. Although numerous approaches to help minimize issues related to learning have been tested and developed over the years, there is no concrete and definite strategy that can easily overcome barriers to comprehension due to the kind of problems programming presents. Therefore, it is important for educators to find ways to minimize issues with lack of learning.

De Jesus (2011) states that one of the ways to improve student understanding of logical flow and procedural thinking is the usage of flowcharts. This strategy can be especially helpful because visual features are often easier to grasp than abstract notions. The researcher states that structural/procedural languages may be used to aid students in understanding the fundamental building blocks of programming.

First year introductory programming courses in information technology are very critical as they lay the groundwork for learning programming throughout the remaining college years. Programming tools that provide visual representation of concepts may help achieve better results in teaching students programming because graphical exemplification, such as flowcharts, can allow students to better understand algorithms (de Jesus, 2011). This way, students can visualize how the actual program runs and even follow the step-by-step execution of the program to understand each and every part of the solution.

2.2 Research on Programming Tools to Improve Learning

Programming is a complex skill to acquire, so educators have created numerous tools to promote learning of programming among novices. There are tools that allow new students in information technology to design and test objects (de Jesus, 2011), manipulate robots through visual programming interfaces (Anderson, McKenzie, Wellman, Brown, & Vrbsky, 2011) or generate and control animated worlds (“What is Alice?,” 2014). Table 2-1 provides a concise summary of features of select graphical programming tools that have been used in other

studies to gauge student interest and/or performance in programming. The following discussion elaborates further on each of these tools.

Table 2-1
Visual Programming Language Tools and their Functionality

No.	Name	Functionality			
		Flowchart-type Interface	Loops	Conditions	Code Visualization
1	Alice	×	✓	✓	✓
2	Iconic Programmer	✓	✓	✓	✓
3	LabVIEW	✓	✓	✓	×
4	PORTUGOL	×	✓	✓	✓
5	RAPTOR	✓	✓	✓	×
6	Scratch	✓	✓	✓	×
7	SICAS	✓	✓	✓	✓
8	vIDE	✓	✓	✓	✓
9	WYSIWYC	×	✓	✓	✓
10	NanoNavigator	✓	✓	✓	✓

Although, there are several tools to encourage learning, the challenge is not to create more tools but to examine current environments to probe if current technology is working as expected. To understand this issue, Gross and Powers (2005) studied the programming tools designed to improve programming skills of new learners.

The researchers studied multiple novice programming environments to assess their impact on learning. They chose these environments due to the unique approaches that each tool uses for teaching the concepts. The

environments chosen were: Alice, BlueJ, Jeliot, Lego Mindstorms, and RAPTOR (Gross & Powers, 2005). Alice is a well-documented environment, which empowers users to create algorithms to operate a multitude of three-dimensional objects through animation. BlueJ is a Java-based IDE used for introducing the object-oriented paradigm to students; it allows users to create and manipulate objects in real-time. Jeliot is a juxtaposition of environments, integrating animation of Java code; this tool animates the entire Java program, enabling users to step through the program execution. Lego Mindstorms is a robotics kit that includes a microcontroller capable of controlling the robot.

Gross and Powers (2005) describe several studies pertaining to all five tools discussed in the article. One of the studies cited by the researchers used Alice as the tool to teach students various programming concepts. In order to determine if employing such tool had made any significant difference on student learning, the researchers tracked student grades for a period of two years. The students in the treatment group exhibited higher GPAs and a greater percentage of them continued to the following course compared with other control groups. The students exhibited positive attitudes toward programming in addition to improved performance. This study clearly highlights the positive influence of visual programming on learning. Similar results were found in a study involving BlueJ. It showed that the comprehension of OO concepts among students improved. A study employing Jeliot found significant improvement in student learning by calculating and evaluating scores from pretest and posttest. Researchers found that the programming classes using Lego Mindstorms

improved student attitude toward programming. These results strongly suggest that students respond positively to visual and/or physical programming environments.

Burton and Bruhn (2003) argue that it is important to teach students procedural programming languages first, even before teaching object-oriented languages. The researchers argue that OO is not a replacement for the aforementioned programming paradigm but is complementary. They note that although OO is a new paradigm, it does not replace old paradigms such as procedural programming. They also argue that the algorithmic paradigm needs to be absorbed first before learning OO because of the “need”…for students “…to know how OOP fits into the bigger picture” (Burton & Bruhn, 2003).

Burton et al. (2003) state that the basic concept of an object in the OO paradigm is quite simple to understand. Writing software using this concept, however, requires the understanding of interaction between objects in the problem domain. Also learning about abstract concepts in the OO domain requires focused efforts in addition to the time overhead. The authors argue that becoming an expert in object-oriented programming requires at least three years of training. Conversely, the procedural programming approach heavily focuses on creating a concrete algorithm to solve a problem. Burton & Bruhn (2003) identify the following main steps for problem solving using the procedural approach:

1. Read and comprehend the question
2. Develop a possible answer to the question

3. Validate and construct the solution as an algorithm
4. Transcribe it into an actual working code
5. Examine and fix any issues in the code
6. Create documentation for the code

Burton et al. (2003) advise that students master thinking in a logical manner before learning about the object-oriented approach. This way, students will learn about the process of solving a problem, which can be extrapolated to deciphering problems in an object-oriented environment. The authors also feel that the ability to scrutinize a problem and create solutions in a proper, sequential manner is especially important for novices. They further note that the OO paradigm undermines the learning of efficient and effective procedural design principles. It is important to teach simpler concepts first when teaching programming to make the overall process well-structured for students to understand in an effective manner. For Burton et al. (2003), the natural order for teaching software design should involve educating students first on procedural principles and then on the object-oriented paradigm. The authors deem that teaching programming concepts in a gradual and systematized way can improve learning.

Although teaching using the right paradigm of programming is essential to improved student understanding, the main premise of the argument is the hypothesis that obstacles to learning lie in the process of creating computer programs. In order to write a well-designed program, Kelleher & Pausch (2005) state that students must know the following:

1. How to convey commands to the computer (syntax),
2. How to organize commands (style), and
3. How the computer actually executes these commands.

Kelleher and Pausch (2005) note that many novice programmers struggle with various aspects of programming. Despite efforts to simplify programming languages, students find it difficult to “[remember] names of the commands, the order of parameters, whether or not they are supposed to use parentheses or braces” (Kelleher & Pausch, 2005). The researchers suggest that, in order to facilitate learning of the fundamental constructs of programming, one can completely circumvent the syntax problems by using graphical elements to symbolize various parts of a computer program, for example, variables, control options, and commands. Because various components can be relocated and joined together to create programs, introductory programmers only need to “recognize the names of commands and the syntax of the statements is encoded in the shapes of the objects, preventing them from creating syntactically incorrect statements” (Kelleher & Pausch, 2005).

Most of the environments created to facilitate learning of programming systems have been created with a focus on novices by employing more convenient procedures for programming and many have removed unnecessary syntax, including some visual elements (Kato, 2010; “What is Alice?,” 2014, “What is Arduino?,” 2014). Using this approach, students have been able to see the results of their creations immediately, providing a substitute to typing program instructions. It is possible, according to Kelleher et al. (2005), to design a

software development environment to be suitable for a wide variety of audiences, especially introductory students. Using such graphical programming methodology, students can concentrate on learning about the structure and flow of the programs rather than focusing on writing syntactically correct programs.

As previously discussed, visual programming languages are one of the ways to improve student cognition of programming basics. Hils (1992) notes that the data flow model is one of the more popular ways on which many visual programming languages are based. This model presents introductory students a view of data flowing through the logic of the program, the transformations that data undergoes, and the final result(s) of the computation(s). The author also notes that visual models, such as the data flow model, provide the ability of "viewing monitors at various points to show the data to the user. Consequently, many recent visual programming languages are based on the data flow model" (Hils, 1992).

The notion of utilizing data flow diagram elements for representing an algorithm is quite popular. The central premise of this approach is that the data flow model and related concepts can be used to portray the flow of logic through a program using nodes that represent functions of the actual program (Hils, 1992). The flow going in and coming out is considered as input and output of the node, respectively. Different philosophies recommend varied data modeling methods to represent data.

Hils (1992) describes multiple examples of how the data flow model can be used to depict programs by creating flowchart-style structures, some of which

are discussed next. According to the author, the “pure” model of the data flowchart does not have the constructs such as loops, but instead relies on imperative execution of commands (statements that change the program’s state). This model uses primitive representation of flow using arrow symbols. Most visual programming languages utilize boxes and other constructs to depict functions and lines to denote the data flow. It is possible to insert steps that allow users to examine data values throughout the execution of the program. Unlike the “pure” model, many graphical programming languages include visual elements that permit iteration. Some languages provide the ability to create different types of loops (e.g. FOR, DO WHILE, etc.). This simplifies the process of building a program and removes the complexities involved in manually creating nodes that imply iteration.

Hils (1992) reports that some visual languages can also involve inclusion of data types in visual programs. The author notes that, generally, the type check is performed throughout the construction of the algorithm. This ensures that users can connect nodes to each other that do not violate the language syntax, diminishing the risk of any errors at run-time due to type discrepancies. This study acknowledges the fact that there are significant variations between visual programming languages, but the more important point is that, overall, they simplify the learning process by using graphical elements.

2.3 Graphical Programming Tools and Their Impact on Student Learning

It is important to ensure that novice programmers learn programming

languages in a way that can solidify their grasp of algorithms in addition to developing critical thinking skills. The Instituto Politécnico de Tomar in Portugal developed ‘Portugol’, a structured programming integrated development environment (IDE) (de Jesus, 2011). This IDE incorporates the ability to generate structured program statements by creating corresponding flowcharts. It also provides an ability to generate a flowchart based on a block of structured programming statements. The researcher states that this tool was created to assist first-year computer science students in learning programming concepts. Such tools have been used in the past to improve comprehension and generate interest in programming paradigms.

Carlisle, Wilson, Humphries, and Hadfield (2005) note that students devote a large amount of time learning and dealing with the syntax of a language in introductory programming courses. Moreover, most courses teach programming concepts through the use of textual, editor-based applications; such environments make it difficult for many students to learn programming. Also, many students struggle in courses that use a textual approach due to their inherent inclination to a visual perspective (Carlisle et al., 2005).

A previous study observed that using a textual programming language in introductory programming classes may “annoy and distract attention from the core issue of algorithmic problem solving” (Carlisle et al., 2005; Shackelford & LeBlanc, R.J., 1997). The authors witnessed that this leads to instructors emphasizing potential problem areas such as syntactical errors, instead of focusing on the actual learning of algorithms and foundational concepts.

A study conducted by Carlisle et al. (2005) found that between 75 percent and 83 percent of the students in the programming course were predominantly visual learners. This finding can explain the difficulties many students face while learning programming. To combat this issue, the researchers created a graphical programming application called RAPTOR or Rapid Algorithmic Prototyping Tool for Ordered Reasoning. This tools uses flowcharting symbols to create programs. The program also allows users to execute their algorithms to test proper functionality. Students can execute their programs in a continuous mode or step through the program to examine values of each and every data element (Carlisle et al., 2005).

Graphical programming environments can significantly benefit visual learners. Fischer, Giaccardi, Ye, Sutcliffe, and Mehandjiev (2004) note the importance of such environments in their article titled, “Meta-Design: A Manifesto for End- User Development” (Carlisle et al., 2005),

> “Text-based languages tend to be more complex because the syntax and lexicon (terminology) must be learned from scratch, as with any human language. Consequently, languages designed specifically for end users represent the programmable world as graphical metaphors containing agents that can be instructed to behave by condition-action rules. The aim is to reduce the cognitive burden of learning by shrinking the conceptual distance between actions in the real world and programming” (Carlisle et al., 2005).

To analyze if RAPTOR has made any improvements in student learning, the researchers devised an experiment. The study spanned three semesters, each semester with 365, 530, 429 students being analyzed, respectively. Carlisle et al. (2005) incorporated three questions on the final exam to examine if the problem-solving ability of students had increased. The researchers compared the results using one-sided, two-sided, and two-sample *t*-tests (Carlisle et al., 2005).

The authors noticed that the students, provided with multiple options, overwhelmingly chose to represent their algorithms using graphical elements. A peculiar result of the study was that although students had learned a third-generation programming language, a whopping 95 percent used flowcharts for represent their solutions to the algorithmic problems. The study concluded that this change in problem-solving ability of the students could be attributed to using RAPTOR as a tool for teaching algorithm development. Researchers also noted that the graphical elements of RAPTOR permitted students to solve problems easily because they could easily follow the flow of logic through the problem. This study underscores the importance of offering graphical tools to students to cement their basic knowledge of programming (Carlisle et al., 2005).

One of the most popular tools of teaching introductory programming concepts is Scratch. This tool was developed by MIT Media Labs in order to “nurture a new generation of creative, systematic thinkers comfortable using programming to express their ideas” (Resnick et al., 2009). The authors sought to provide software to people who had no background in programming and had never realized the potential of this technology to create interesting animations

(Resnick et al., 2009). Generally, this software is used in K-12 to motivate and generate interest about computing majors among students before introducing them to more advanced programming concepts. (Resnick et al., 2009).

In order to introduce to the fundamentals of computing and logical thinking, researchers at Harvard University decided to use Scratch to teach initial programming concepts (Malan & Leitner, 2007). The researchers used two lectures during the first week of classes before teaching Java for rest of the course. The research was conducted "not to improve scores but instead to improve first-time programmers' experiences, we surveyed students throughout the summer for their thoughts on Scratch and its impact on their education" (Malan & Leitner, 2007). As this research aims to improve student interest in programming, enhancing the programming experience of new programmers is central to improving their attitude.

There were a total of 25 survey respondents, 52 percent of which had no prior exposure to programming, while 32 percent had limited programming experience. 16 percent of the respondents had used some programming language for at least one year. Malan and Leitner (2007) asked their students about the impact of using Scratch on their experience with Java, 76 percent reported positive influence, eight percent noted negative influence, while 16 percent stated neither positive or negative impact on learning. This study clearly demonstrates the possibility that graphical programming tools can direct impact student outlook on text-based programming languages. Also, such languages can improve student reasoning.

There have been many tools, as previously noted, that can construct pictorial representations of algorithms to help students understand programming fundamentals. To create an active learning environment, educators have created systems in which students can visualize their algorithms created using graphical elements. Hundhausen and Brown (2007) note that such tools “support a similar development model in which coding an algorithm is temporally distinct from viewing and interacting with the resulting visualization” (Hundhausen & Brown, 2007). Because novice programmers have difficulty using correct syntax for code, they will benefit from being able to view the execution process. According to the researchers, the ability of models to provide live feedback can assist the introductory students in information technology to detect and rectify programming mistakes and eventually develop syntactically correct code.

To study the hypothesis that allowing students to type code and show corresponding results simultaneously would aid comprehension, the researchers created a model called “What You See Is What You Code” or WYSIWYC (Hundhausen & Brown, 2007). The software was designed in a way to develop programs using a combination of writing very basic code while manipulating visual program objects. WYSIWYC evaluates code being typed with every edit for syntax errors, allowing novice programmers to receive immediate feedback on the validity of their code. Novices can edit their code because the programming tool provides suggestions on creating syntactically accurate statements. Students can also view their creations in real-time in an adjoining window.

Most of the data collected to analyze students was gathered through observations and videotaping of the participants. Hundhausen and Brown (2007) observed that many students communicated their irritation with regards to the pseudo code language used in the program created. The authors quoted a participant in the study who remarked that it was difficult to visualize the actual result of the algorithm without getting needed feedback from the software tool. The data collected also revealed that only 30 percent time was spent on actually writing any code (Hundhausen & Brown, 2007). This study revealed that introductory students in programming courses need a medium to visualize their code to actually understand fundamentals and gain confidence to write algorithms. Just providing them suggestions on how to write syntactically correct code does not necessarily improve cognition and lead to improved attitudes toward programming.

Due to the importance of computers in engineering areas, programming is one of required topics taught to engineering students. According to Bucks and Oakes (2010) there is substantial evidence that students in introductory programming courses have difficulty learning and employing concepts by writing code in the relatively short period of a semester. This may be due to the tendency of students to learn programing concepts visually. The researchers decided to research the difficulty with learning programming by using graphical programming languages in introductory courses for the engineering students (Bucks & Oakes, 2010).

Bucks et al. (2010) used two course sections comprised of 120 students per section from multiple introductory programing courses. The course sections were modified to integrate usage of graphical programming. This approach was taught in both lecture as well as the laboratory exercises by the same instructors. The LabVIEW graphical language, created by National Instruments, was selected for the study. This language consists of blocks, which can be connected to form a program. The researchers ensured that sufficient instructions were provided to students to allow them to create well-designed programs. Six lectures and laboratory periods were required to teach fundamentals of LabVIEW. The students were given a project to be completed using the aforementioned language.

The results from the experiment devised by Bucks et al. (2010) were significant. Researchers noted student concerns related to the additional workload of learning and implementing LabVIEW. Nevertheless, as the semester progressed, the students became comfortable with using the programming language and even learned about different functionality of the language not taught in class.

A student attitude survey was conducted to record student attitudes toward programming and their overall experience with the LabVIEW language project. Although, many students had complaints about the projects at the beginning of the semester, student attitudes toward LabVIEW improved during the course of the study. The researchers also compared this feedback with the course sections where LabVIEW was not used as a learning instrument. The

overall course rating for the traditional course was between 2.5–3 out of five points, but ratings for the modified course was between 3.5–4 out of five points (Bucks & Oakes, 2010). The researchers noted that students were able to learn the required material well through the medium of a graphical programming language. Also, students in the modified course sections demonstrated overall improvement in cognition of class topics.

It is widely known and researched that many students in information technology programming courses struggle with issues related to syntax, logic, and control flow of algorithms (Chen & Morris, 2005). This problem also affects students enrolled in high school science courses, since many times there is not enough time to teach all aspects of being an effective and efficient programmer. Researchers in Canada, therefore, decided to create a very simple tool called "Iconic Programmer" that is based on flowcharts and visual programming that uses icons to represent programing constructs (Chen & Morris, 2005). This tool can even translate icons and symbols into Java or Turing. This way, students can view and map various flowchart icons to programming language statements.

Simplifying the process of creating a working program, Chen and Morris (2005) used three primary structures of programming, namely sequences, loops, and branches in code. The researchers utilized flowchart icons for denoting activities, branches, and decisions to enable students to create simple algorithms. This tool was used as a supplementary tool to teach students in CS 101 at York University, Canada. Moreover, Iconic Programmer has also been used in a high school setting. Employing this teaching aid in two different pedagogical settings

allowed researchers to study two different groups of students, one at the college level and the other at the high school level (Chen & Morris, 2005).

The researchers, as anticipated, found that students reacted positively to the functionality provided by Iconic Programmer. Many high school science students viewed creating flowcharts on paper as extraneous to the learning process. Nonetheless, it allowed them to envision the design of algorithms, data flows, and overall control structures. Both high school and university students found the functionality to view flowcharts in Java particularly useful (Chen & Morris, 2005). This research further strengthens the argument that visual aids, especially flowcharts, help students better understand programming concepts.

As the graphical user interface technology continues to advance, the methods of creating programs should also become simpler to use in the future. Lucanin and Fabek (2011) note that there are many visual programming languages that can allow programmers to use icons and flowchart-based approaches to create applications rather than focusing on working with specific programming languages. The researchers used the WHILE programming language to demonstrate a new way of generating code. The language was implemented using a system built on the GMF or Eclipse Graphical Modeling Framework. In addition, the authors contend that this method easily allows mapping of a flowchart to the program code (Lucanin & Fabek, 2011).

In order to demonstrate the functionality of a programming tool that can shift the burden of creating the program code from the programmer to the development environment, Lucanin et al. (2011) suggested that the programming

tool should be able to express the algorithm for a program in a certain manner and be capable of translating such logic into machine code for execution by the processor. From these two basic low-level requirements, Lucanin et al. (2011) created four models to implement the aforementioned functionality. First, the graphical elements were defined, and then researchers decided which tools would be used to draw the flowchart. The authors used a mapping model that would dictate how graphical elements would map to the custom WHILE language code, in addition to the Eclipse Modeling Framework (EMF).

The solution created by Lucanin et al. (2011) for graphical programming was able to create flowchart structures and map such structures to syntactical constructs of a programming language. The tool was able to provide users with a novel interface that simplified the program development. Studies similar to the one conducted by Lucanin et al. (2011) are vital to being able to innovate new means of teaching programming to students. If such technology continues to mature, the necessity to learn, remember, and apply textual-based programming languages to compose algorithms will be greatly reduced for introductory programmers.

Santos, Gomes, and Mendes (2010) discuss a similar approach as other researchers noted previously. They point out the fact that efforts have been made to enhance learning activities related to programming. Nonetheless, the success of such activities remains disputed. Difficulties with learning these concepts and subsequent failure rates led to courses with large populations of struggling students. The problem is exacerbated by the intellectual diversity of

the class, comprised of students with dissimilar ability for comprehension and varying degree of knowledge (Santos et al., 2010). The conventional methodology the courses utilize generally fails because teachers are unable to support the needs of students and provide them guidance regarding programming when needed.

It is generally agreed upon that students have various levels of aptitude for programming-related tasks, but empirical research also suggests that all students can succeed in this field, provided they are dedicated and have adequate guidance (Santos et al., 2010). The authors discuss different tools that have been created so as to facilitate education of programming topics. Santos et al. (2010) provide details on a tool developed by researchers called SICAS, which is a Portuguese acronym and translates into English as Interactive System for Algorithm Development and Simulation This tool is fundamentally based on the paradigm of graphical programming to enable students to develop their programming skills.

SICAS enables students to apply algorithms for solving given problems using a flowchart-based illustration. This tool also includes the ability to create and assign variables, perform input/output tasks, and apply iterative and conditional structures. To allow students to create complex programs, SICAS also supports recursive functions. According to the authors, this tool “supports common data types, namely numbers, strings and one-dimensional arrays” for familiarizing students with basic programming concepts (Santos et al., 2010). The program also has the ability to export solutions to external programming

languages such as C, Java, and even pseudo-code. Students can visualize their programs without being bogged down by syntactical issues. The primary goal of SICAS is to enhance algorithm construction skills and improve students' critical thinking skills; this makes the ability to learn a certain programming language is a secondary objective.

The researchers argue that such tools are mere stepping-stones for programming education. Such tools cannot meet all students' needs to improve their understanding of algorithmic concepts. Generally, flowcharts are easier to understand, leading many students to visualize and predict their solutions to challenges in a problem domain.

2.4 Physical Implementation of Graphical Programming Languages

When programming, it is important to continuously test solutions because it may be difficult to visualize and simulate all of the functionality of the algorithm. For robotic applications, it is even more important because such systems involve many hardware components. These components have monetary value and could become damaged if used incorrectly. If physical systems are not tested well, there may be a risk of damage to the hardware and of injury to the person operating such systems.

Graphical programming techniques allow students to create, test, and modify their algorithms quickly. According to Rogers and McVay (2012), this is especially true when the students need to learn and become proficient at a programming language in brief period. Using an environment that will reduce

development time for algorithms is needed and a possible solution can be provided by graphical programming. This method, according to the authors of the study, “allows an engineer to move quickly from theory to proof-of-concept and into prototyping” (Rogers & McVay, 2012). Also, such a programming language may not constrain the ability of students to materialize their ideas into physical robotic movements.

To verify their theory in an engineering environment, Rogers et al. (2012) used students in a mechatronics class who were tasked with creating a robotic algorithm. The students were given ATMega 128 microcontroller and an E-Maxx truck, which could be controlled using a radio. Students could use C to program their algorithms. During the study spanning two semesters, only one team in the first semester and none in the second semester were able to create a functional program. After the failure of students to perform well, the researchers changed their methods, and the following semester gave students a PIC32 microcontroller and used Simulink, which provides a graphical interface for microcontroller programming (Rogers & McVay, 2012). This interface resulted in improved student performance, and enabled them to conceive somewhat more complicated programs than students who only used the C programming language for completing their projects.

Based on observations by the researchers, students were able to accomplish more when they were provided with the graphical programming interface instead of textual C. Students achieved more sophisticated results using Simulink compared with almost absolute failure of using the C language

alone. This study underscores the importance of graphical interfaces when programming for microcontroller environments. Many microcontrollers, due to their basic architecture, only allow native code compilations such C/C++ (Rogers & McVay, 2012). This increases the level of difficulty for students, demotivates them, and makes cyber-physical systems seem too difficult to work with.

Chun and Ryoo (2010) propose a new system to teach physical computing to students in various stages of their academic career, ranging from elementary school to college students. The authors created this new learning method using a graphical programming interface and Light Emitting Diode (LED) display kit. The LED display shows various images or animations created using a flowcharting tool. Educating introductory students in information technology can spark a passion that may go well beyond the standard objectives of courses. Because students spend an enormous amount of time struggling to learn the syntax of a language, providing them a tool to minimize these problems and improve learning is important (Chun & Ryoo, 2010).

Physical computing can be an excellent tool to teach students fundamentals of programming. Students can not only visualize their creations but also touch, feel, and improve corresponding physical devices. LEDs have become increasingly inexpensive, so the researchers decided to utilize a display kit comprised of an 8X8 matrix LED panel. This kit also includes a microprocessor, and a serial communication component. In order to control LEDs, Chun et al. (2010) created a web-based flowchart interface. This flowcharting tool was designed to support basic programming constructs such as variables,

conditional statements, loops, one-dimensional arrays, and simple functions. Because the LED panel contains the 8X8 matrix of LEDs, the students could create 2^{64} different light configurations, allowing them to create multi-colored intricate shapes and patterns.

To observe the effect of these tools on elementary students, Chun et al. (2010) undertook an experiment in which they studied 126 students enrolled in three different elementary schools. This experiment was conducted with only elementary students as subjects. The overall positive attitude demonstrated by the students may suggest that physical computing may have some merits while improving students' attitude toward programming regardless of the educational level. Throughout the experiment, the students demonstrated positive attitude towards programming. Researchers also noticed that students felt much more engaged while creating algorithms using provided tools. The use of LED kits also raised their interest in creating a working application (Chun & Ryoo, 2010). Using such tools in courses improved student collaboration, which, in turn, can lead to improved learning. This study underscores the importance of the visual learning medium. Such techniques, described above, can especially be beneficial when paired with physical devices.

Employing physical systems that complement software development tools can lead to more students understanding basic, even advanced, programming concepts. The Arduino is the one of the multifaceted tools that can be used to accomplish various activities, ranging from educational to recreational. Kato (2010) provides a synopsis of a graphical programming language called Splish,

which enables users to develop applications for the Arduino using visual, icon-oriented, programming interface. The Splish language was developed using the JavaFX framework, which made this language platform independent and allowed for greater overall portability between different operating systems.

The researcher notes that Splish code can be interpreted by a virtual stack machine, after it is compiled and translated into machine code. This allows the user to debug the code without having an Arduino connected to a computer. If an Arduino is connected to a computer, this approach enables students to perform interactive debugging. Graphical programming, employing physical devices, is very different from working with a software-only approach; physical computing can attract students to learn about such techniques and improve their programming abilities (Kato, 2010).

Kato (2010) explains the overall design of the system in detail to provide a complete picture of how the system works in real life. The researcher also provides an example of how a flowchart-like, icon-based program was created. The Splish language actually creates C code behind the scene for Arduino. One of the problems faced by the researcher was related to how big a program can get to run on Arduino. Kato (2010) noted that there are some empirical statistics related to memory allocation of Arduino, which can be used to efficiently manage memory on the Arduino. According to Kato (2010), the Splish language is easy to use for programming and for debugging, and therefore, it can be beneficial for educators to teach and students to learn fairly easily. Due to the graphical nature of this language, Splish can be used to “accelerate the physical computing

experience” of students, in addition to generating interest in programming and reinforcing their programming skills (Kato, 2010).

2.5 Past Research on Student Attitudes Toward Programming

Students’ early attitudes toward programming are critical in understanding various attributes in their academic careers such as satisfaction, future success, and willingness to learn. Understanding their attitudes can help educators tailor the introductory courses in order to build positive attitudes toward programming. A study was done by Garrett and Walker (2008) to examine the overall attitude of students toward programming languages.

The researchers conducted a year-long (two semesters) study in which the participating students were exposed to a variety of programming languages ranging from traditional (C++, Java) and scripting (MATLAB™ script) to graphical programming languages such as RAPTOR and LabVIEW (Garrett & Walker, 2008). Even Alice was taught to help students develop critical thinking skills. The students who participated in this survey were from various majors such as Computer Science and Electrical Engineering. The courses taught students fundamental programming knowledge in multiple languages in a significantly short period. The authors also attempted to find if students demonstrated more positive attitudes toward graphical languages or the traditional programming languages. A survey was used to collect data about student attitudes, which used a five-point Likert Scale. Also, the questions asked were worded positively and

negatively. Reverse coding technique was used to standardize the responses to the Likert Scale questions.

The data analysis found that the majority of the students had negative attitudes toward Alice, but neither negative nor positive attitudes toward traditional (C++, Java) or graphical (RAPTOR, LabVIEW) programming languages. Using two-tailed *t* tests and an alpha of 0.10, the authors concluded that was not enough evidence to suggest either positive or negative attitude toward traditional or graphical programming languages even after the year-long study period (Garrett & Walker, 2008). Further analysis of the data suggested that graphical programming languages might enable students to think in a logical manner in addition to providing them with graphical interface.

As programming is generally considered a difficult topic to understand, studying student attitudes can help instructors introduce technologies, which can improve student learning and overall attitude. A study was conducted by Baser (2013), in Turkey, to gauge differences in attitudes among males and females as well as to understand the impact of student attitude about programming on their success in the computing major.

The participants in the study were 137 sophomore students learning Python in an introductory programming language course. An attitude survey was created by the researcher in the Turkish language to measure various attitude elements – confidence, usefulness, attitude toward success, and motivation – about programming (Baser, 2013). The study used a five-point Likert Scale for gathering attitude data. The instructors conducted a pilot study to ensure that the

survey was a valid and reliable tool to measure attitudes. The surveys also included positively and negatively worded statements, which were reverse-coded for the proper analysis.

The average minimum attitude was 1.66, while the average maximum attitude was 4.94, which led the researcher to conclude that students did not have positive attitude towards programming but their attitude was not very negative either (Baser, 2013). The differences among males and females were significant – males tended to have more positive attitudes toward programming than females. Baser (2013) also found that correlation between student grades and attitude was significant but only accounted for 16.7% of overall attitude. This means that the attitude toward programming is not the only factor affecting student success. The researcher also found that the difference between the genders about confidence, usefulness, and motivation was significant, but the difference between overall attitudes toward success was not significant (Baser, 2013). This study demonstrated that there is a need to improve student attitudes in order to improve their overall outlook on programming, which may lead to increased success.

As previously noted, attitudes toward programming impacts students' performance and related success in computer science and related fields, which require strong programming skillsets. Therefore, it is critical to increase student confidence and their opinion about the usefulness of programming. A study was conducted at the University of Alabama by researchers who wanted create an

environment that could boost student confidence by combining graphical programming language and robotics (Anderson et al., 2011).

The researchers used PREOP or “Providing robotic experiences through object-oriented programming…” which “ is a syntax-free graphical programming tool” (Anderson et al., 2011). This environment is based on Alice and has been shown to amplify student curiosity in Computer Science. The course used iRobot for teaching students programming concepts. For conducting the research, students were taught new concepts each week during a two-hour session for a period of ten weeks. The student data was collected using surveys, which were completed by 71 students but due to the age limitation data for students below the age of 19 years were not considered for the analysis.

The student attitudes after taking the course were considerably more positive, but the results were not statistically significant. Moreover, the overall interest in Computer Science increased slightly as high overall interest was recorded at the beginning of the ten-week long instructional period. The study did not produce statistically significant results and the authors concluded that more research is required in order to study how to change attitudes and increase interest and learning by using graphical programming language in conjunction with robotics (Anderson et al., 2011).

2.6 Summary

As noted in the review, many studies have found that the physically interactive systems can abet student learning of programming concepts. Among

freshmen college students, programming is one the most dreaded topics (Robins et al., 2003). Using the technological approaches mentioned above, the student interest in various aspects of programming can be improved. Leveraging microcontroller technology to teach programming is a relatively new technique. More research is warranted to investigate the impact of using cyber-physical systems in increasing student interest in programming.

CHAPTER 3. METHODOLOGY

As discussed in the previous chapters, being able to write code is an important aspect of information technology curricula. This ability enables students to not only create useful programs but also think and solve problems in a logical manner. Because most students today are visual learners, it is possible to leverage graphical user interfaces for learning and teaching of programming concepts (Carlisle et al., 2005). This study aimed to expose freshmen college students to a graphical programming environment to program a microcontroller and examined if such experience changed their interest in programming. Increased interest may lead to innovative ideas and may even improve their problem-solving and decision-making skills. Attitudinal data was collected using online surveys to determine whether the experiment caused significantly improved attitudes. This chapter explains the design of the experiment for this study. The participants, the procedures for data collection from the participants, the variables, and the methods for data analysis are also described in this chapter.

3.1 Experimental Setup

Empirical evidence suggests that programming is one of the most challenging learning aspects of information technology education. Many students

dislike programming, and information technology-related disciplines experience high dropout rates for this reason. This experiment involved examining if usage of microcontroller technology and flowcharting tools improves the interest level of freshmen college students in programming. As part of this experiment, subjects were required to participate in one, two-hour-long session in which the subjects programmed a microcontroller using a flowchart-based language. Researchers in the past have mainly used just visual programming languages to test if student understanding of programming fundamentals changes. The researcher has conducted few outreach sessions for Purdue University's College of Technology in which high school students are taught how to program the microcontroller used in this research. The outreach sessions results have been generally positive and suggested that students' attitude may improve after such session.

This research focused on providing students with a graphical programming experience and its impact on the interest levels of freshmen students. In this study, the following hardware and software were used:

- Hardware
    - Phoenix Contact nanoLine Microcontroller (nLC-055-024D-08I-04QRD-05A)
    - Phoenix Contact Operator Panel (nLC-OP1-LCD-032-4X20)
    - Input Switch Simulator for nanoLine
    - Output Simulator for nanoLine
    - USB Cable for nanoLine
    - Light Bulb

- Software
    - Phoenix Contact nanoNavigator (Version 4.1.0 (617))
    - Envigilator Proctoring Software

Data was collected using three scientific online surveys, which were created using Qualtrics Survey Software. These surveys included: one pre-instructional survey and two post-instructional surveys. The second post instructional survey was administered four weeks after the instructional session. During this period of four weeks, no additional treatment was provided to students. In the past, the researcher has noticed a positive response regarding the session and overall programming immediately following the outreach session. Therefore, it was important to understand if the attitude changes prevail over time. All of three surveys were identical and contained 16 multiple-choice statements about various aspects of programming education in addition to two short answer questions. Students answered these statements using a Likert-scale with four options: Strongly Agree, Agree, Disagree, and Strongly Disagree. This allowed the researcher to examine student opinions about included statements at sufficient granularity. The surveys, which were completed electronically, were based on an attitude survey reported in an article by Munson, Moskal, Harriger, Lauriski-Karriker, & Heersink (2011). The survey measured various attributes related to the field of information technology. These attitude attributes included (Shashaani & Khalili, 2001):

- Confidence
- Interest
- Stereotypes
- Usefulness

The following sample statements illustrate the kind of statements that comprised the attitude survey (Munson et al., 2011):

- Confidence
    - I am comfortable learning programming concepts.
    - I have a lot of self-confidence when it comes to taking programming courses.
- Interest
    - I am able to think in a logical manner to innovatively create new programs.
    - I think programming is boring.
- Stereotypes
    - A student who performs well in programming courses will probably not have a life outside of computers.
    - Men are more likely to excel in programming classes than women.

- Usefulness
  - The challenge of using programming languages to solve problems appeals to me.
  - I am confident that I can find a job as a software engineer/software programmer.

Some of the survey statements above are quoted directly from a study conducted by Munson et al. (2011). The researcher obtained an approval from Dr. Barbara Moskal to use a version of the attitude survey. This approval email can be found in Appendix A.

Ensuring the quality of answers was critical and, therefore, the statements in the survey were both positively and negatively worded. The aforementioned concepts are discussed in further detail in the following sections.

3.2 Hypotheses

Many introductory college students with limited or no background in programming struggle throughout their programming classes in information technology and computer science. As noted previously by Robins et al. (2003), programming courses experience high dropout rates; therefore, making the initial introduction to programming more engaging and personally relevant may lead to improved learning and interest in the programming field in students. Consequently, this study aimed to explore the possibility that using a graphical programming environment within the physical computing realm could increase

interest in programming among freshmen college students. A two-tailed test was carried out in order to test the following hypothesis.

H_0 = There is no statistically significant increase in positive attitudes about programming in students who are exposed to a graphical programming interface for microcontroller programming.

H_a = There is statistically significant increase in positive attitudes about programming in students who are exposed to a graphical programming interface for microcontroller programming.

Statistical tests were performed individually for each of the measurements (confidence, interest, stereotypes, and usefulness).

3.3 Participants

There were criteria that a participant must satisfy in order to take part in this study. The participants were required to be above 18 years of age and enrolled at Purdue University. Also, they had to be comfortable with completing the online survey. The participants were asked to volunteer for this research. The participants were instructed throughout the session on how to use the microcontroller, create programs, and complete an activity at the end of the session. All students in CNIT 15501 were invited to participate in this study. This should have provided a sufficient sample size to perform analyses to spot any statistical significance. The number of subjects depended on the number of students enrolling in this course and accepting the invitation to participate. The researcher obtained approval from the instructor teaching this course (see

Appendix B). Prior to the start of the study, the expected enrollment in this study was 60 participants, but the actual enrollment was 43. Additionally, not all enrolled students completed all three surveys – the number of participants who completed all three surveys was recorded as only 32.

3.4 Methodology

As noted previously, the aim of the study was to see if freshmen college students who engage in a cyber-physical programming session would become more interested in programming in general. Empirical research suggests that programing courses are dreaded by many students due to the difficulty level, which results from the abstract nature of the topic. Many freshmen college students have traditionally learned programming through text-based editors with few to almost no graphical elements. This study examined whether their interest in programming improved or not by employing a system in which they developed their programs using a graphical tool and could also see the physical product of their programs using a microcontroller.

As indicated by the literature review, earlier studies have conducted tests that mainly related to graphical programming languages for comprehension of fundamental information technology concepts. These pedagogical approaches have included teaching procedural programming, tools for visualizing algorithmic development, and flowchart-based development environments that only operate in cyber space. Every graphical programming language development environment is different, so the attributes to be analyzed vary by some degree.

Chun and Ryoo (2010) conducted a similar study on South Korean high school students in which the subjects used a web-based flowchart program to control an LED kit to create novel shapes on the LED display and noted overall positive results and demonstrated that overall problem-solving capability of the subjects increased. The study described here recruited freshmen college students in order to determine if their interest level in overall programming increased by employing graphical programming to create programs for a microcontroller.

3.4.1 IRB

This study was categorized as human subject research and required to receive an approval from the Institutional Review Board (IRB) as the data was collected from students in a programming course. An application for requesting permission to conduct research was submitted to the IRB on 06/13/2014, and approval was received on 06/17/2014, which can be found in Appendix C.

3.4.2 Procedures

As noted previously, all students in CNIT 15501 were given the opportunity to participate in the study. The students were provided a consent form and they could opt out of the study without any penalty. CNIT 15501 had three laboratory sections during the semester of research. The study used these laboratory periods to conduct the instructional session and administer pre- and post-instructional surveys (see Appendix D). The hardware and software mentioned earlier was used for the instructional session.

At the beginning of the session, all participants were given a unique 10-digit identification number (ID), which was used to correlate the survey responses with participant demographics during data analysis. All participants were provided an overview of the experiment, instructions on how to use each tool (hardware and software), and details on how to complete surveys. They were given sufficient time to login to the computers. Once logged in, the participants were asked to start the Envigilator proctoring software. This enabled the researcher to capture screenshots of every participant's computer every two seconds, providing insights into how the flowcharting software was used by the participants. The participants were instructed to complete a pre-instruction survey to capture their initial opinions about computer programming. The post-instruction surveys asked participants exactly the same questions immediately after the intervention and four weeks after the intervention. The pre-instruction survey defined a baseline to compare results of the experiment immediately after the activity session and four weeks after the session. The four-week post survey was used to mitigate concerns regarding short-term, positive feedback immediately following the instructional session. The entire dataset was analyzed after conclusion of the second post-instruction survey.

Table 3-1 provides a timetable for the various phases of the instructional session.

Table 3-1
Timetable for Instructional Session

No.	Name	Description	Total Time
1	Provide unique identification number (ID)	Participants were handed their unique ID	5 minutes
2	Collect consent forms	Participants were given the consent form for participation in the study upon entering laboratory	5 minutes
3	Overview of the research project	The participants provided an overview of the research project, methodology, and the data collection methods	10 minutes
4	Pre-Instructional survey	The participants completed a survey	15 minutes
5	Hands-on activity	This was completed in three steps: 1) Familiarized participants about the hardware and software 2) Walked participants through an activity 3) Asked participants to modify a program to include new functionality	50 minutes
6	Post-Instructional survey	The participants completed a post-instructional survey	15 minutes
7	Post-Instructional survey	This survey was conducted four weeks after the initial instructional session.	15 minutes

During the research session, after the subjects completed the pre-instruction survey, they were given step-by-step instructions on how to create a simple program using Phoenix Contact's nanoNavigator software. This simple program employed the basic foundational elements of a programming language, including loops and decisions. The participants learned how the Phoenix Contact

microcontroller utilizes various programming elements. It was important for participants to visualize their creation, because this was one of the main focuses of the experiment. The subjects were familiarized with the built-in simulator in nanoNavigator, the software for programming the microcontroller. The example program for the hands-on activity was very simple – turning a light on/off when a switch was on/off. They were also shown how to download the created program onto the Phoenix Contact microcontroller. Using this microcontroller, the participants were able to physically observe the functionality of their creation on the actual microcontroller.

In order to challenge the participants to think logically, they were given a simple task that required them to modify the program they created with the researcher. The researcher walked students through an activity during the instructional session. This activity involved creating a simple program, which would turn on a light bulb attached to the microcontroller when a switch was turned on. When the switch was turned off, the light bulb turned off. The activity was intentionally designed to be simple but instructional. The flowchart-based program utilized two programming concepts – variables and loops. After this activity, the students were tasked to add a timer, which would track how long a light had been turned on. When the light turned off, the timer stopped and displayed the total time on the display for 5 seconds. The researcher was available to answer any questions that participants may have, ensuring that the participants understand how to work with nanoNavigator software. They were allowed to work on their assigned task individually for approximately 20 minutes.

The researcher reserved the next 10 minutes to demonstrate some correct modifications made by the participants using the Phoenix Contact microcontroller. In the last 15 minutes of the session, the participants completed the post-instructional survey. Participants' responses were analyzed to gauge the outcome of the experiment, in addition to the quantitative data analysis.

3.5 Privacy and Confidentiality of the Participant Data

It was paramount to protect all data related to the participants. The participants only used a 10-digit unique ID to complete all surveys. A Microsoft Excel file was used to store the names and associated unique IDs of the participants. This file featured password protection with password known only to the researcher. The data gathered using the Envigilator proctoring application was transcribed at the conclusion of the study, after which it was permanently deleted. The participants were also instructed before beginning of the all Qualtrics surveys not to include any personally identifiable information.

3.6 Statistical Analysis

Statistical analysis of the data collected was performed to identify if there was a significant attitude change in the interest level of the participants when they completed a two-hour experimental session designed to expose them to programming a microcontroller using a graphical programming language. The analysis also inspected any feedback provided by participants in the short answer questions so that similar studies in the future can include new techniques

based on this research to measure overall attitudes of the introductory information technology students.

Purdue's Department of Statistics' Statistical Consultation Service aided in the analysis of the data. Based on their advice, the data collected through this study was analyzed using a two-sided significance test on a linear mixed model. The data, including the baseline figures, was entered in RStudio 0.98 for statistical analysis. Table 3-2 identifies the data to be entered in the statistical analysis software:

Table 3-2
Descriptive and Inferential Data Collected for Each Participant

Data Type	Data Collected
Descriptive Data:	College Grade Level Gender
Inferential Data:	Confidence Interest Stereotypes Usefulness

The enrollment in CNIT 15501 was expected to be 60 students, although the only 43 participants were enrolled in the course at the time of the experiment. The researcher used various mixed models to differentiate between the participants based on college grade level and gender. This enabled the researcher to determine if participant attitude had changed significantly after the session compared to the baseline attitude at the beginning of the session.

3.7 Summary

This chapter described the design of the research, the hypothesis, the setup of the experiment, the methodology, and the analysis methods used for scrutinizing the data gathered in this study. It also provided the justification for the experiment design and methodology utilized in this research.

CHAPTER 4. PRESENTATION OF THE DATA AND FINDINGS

This chapter includes an explanation of how the gathered data was prepared for statistical analysis. It introduces and presents the outcomes of both quantitative and qualitative data gathered during the research.

4.1 Data Preparation and Analysis

As stated in the research hypothesis, this analysis gathered survey data from a group of freshmen in CNIT 15501 course by providing them an introductory session of programming a Phoenix Contact NanoLine microcontroller. The instructional sessions were monitored using Envigilator software. Based on the analysis of the Envigilator sessions, all participants were able to successfully use the nanoNavigator software and follow instructions. Although there were instances where students fell behind while following instructions being given, in all such situation, the participants demonstrated sufficient ease of use while working with the software.

For this experiment, the data was gathered in the form of three surveys – pre-instructional session survey and two post-instructional surveys, latter of which was conducted four weeks after the instructional session. The participants created a program for the selected microcontroller using the NanoNavigator software by following the series of instructions by the researcher during the

instructional session. The data collected through surveys provided the participants specific statements about four specific variables that the researcher aimed to study: confidence, interest, stereotypes, and usefulness. The original enrollment in the course was estimated at 60 students, while the actual enrollment was only 43 students. Therefore, the analysis is based on analysis of the students who participated in the instructional session and completed all three required surveys. The breakdown of the students who completed all three surveys follows:

Table 4-1
Demographics of the Participant

Grade Level	No. of Participants	Gender	
		Male	Female
Freshmen	24	18	6
Sophomore	6	4	2
Junior	1	1	0
Senior	1	1	0
Total	32	24	8

The data was methodically organized in a simple way based on the descriptive variables. The categorical data collected through this survey was vast and needed to be divided into multiple subsets. As this research project aims to analyze the change in attitudes toward programming among freshmen students, the data was split into two separate groups who completed all required surveys – one with only freshmen participants, one with all participants who completed the surveys. The organization of the data into two separate groups allowed for an easier analysis. The analysis was performed both with and without taking gender

into consideration. In addition, the data was scrutinized by comparing student grade levels. Although the analysis did not discard the data gathered from non-freshmen students, the conclusions are based on the analysis of freshmen attitudes toward programming. The researcher did not include past programming experience of the participants in any of the surveys because students with a prior course in programming are excused from enrolling in the course. This was considered to have reduced the chances of participants with substantial programming taking part in the experiment.

The participants answered 16, four-point Likert-scale statements that measured their attitudes; this survey was based on work by Munson et al. (2011). These statements were considered as variables for the analysis. The participants were asked exactly the same statements in all three surveys to ensure that the responses were consistent throughout. All statements asked were created in a paired manner in which one of the statements was positively worded, while the other question in the pair was negatively worded. Any responses from students who did not participate in all three surveys were excluded from the analysis. This gave a sample size of 24, which was used for analyzing all three data points.

Specific values were assigned to the responses by students. The four-point Likert Scale statements had the following choices with specific values for analysis.

Table 4-2
Coded Values Survey Participant Responses

Coded Values for Positive Statements	Choices for Response	Coded Values for Negative Statements
1	Strongly Agree	4
2	Agree	3
3	Disagree	2
4	Strongly Disagree	1

This coding meant that for statements that were positively worded, lower values would suggest the participant agreed with the statements, and for negatively worded statements, higher values suggested an agreement. The statements that were used to gather responses from the participants were organized into four separate categories for analysis: confidence, interest, stereotypes, and usefulness. Appendix E notes the specific statements that correspond to each of the four categories.

On the day of the instructional session, data was gathered from a pre-instructional survey and the first post-instructional survey. Four weeks after the instructional session, data was gathered again during the second post-instructional session to measure if the attitudes of the participants had changed further. No treatment was provided during these four weeks. All three datasets were compared to see if the attitudes revealed any changes.

The goal of the study was to see if the attitudes about programming of the freshmen participants changed significantly after completion of a 110-minute instructional session using a visual programming language coupled with microcontroller technology. A linear mixed model was used to analyze the data.

This was due to a small sample size, which did not permit using paired *t*-tests for the analysis. Using this type of model, the researcher was able to include a random-effect variable (participants), in addition to fixed-effect variables such as time and gender. Also, this model enabled the analysis of the data, which was gathered over a period of time on the same participants. Before selecting a particular linear model, a Q-Q Plot of the data was analyzed, which suggested a non-normal distribution. This further solidified the basis for using a mixed linear model. Figure 4-1 shows the Q-Q Plot for the entire dataset.

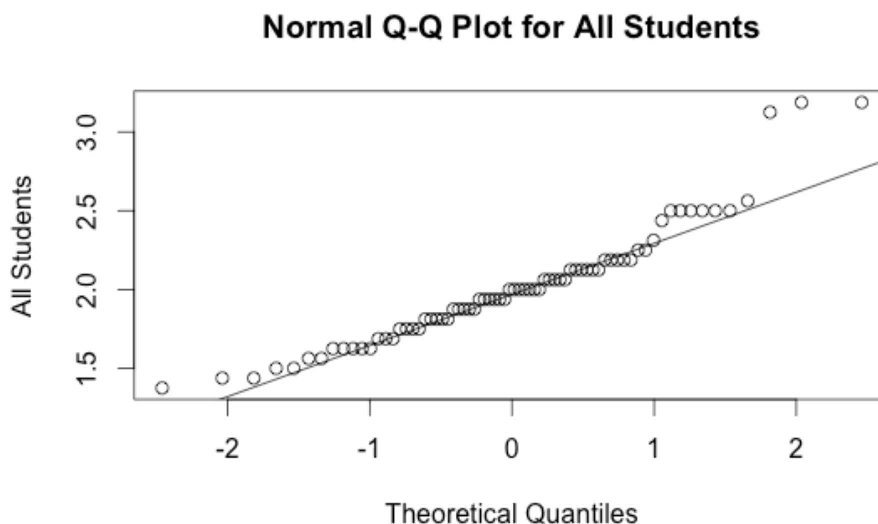

Figure 4-1 Q-Q Plot of the Responses

4.2 Test of Significance for the Dataset

For analysis, the data from all three surveys was combined in a single Microsoft Excel file. This dataset file was loaded into RStudio for further statistical evaluation. An additional column called *Time* was added to the dataset to represent the different points in time when the data was collected, where pre

instructional survey had value of 1, first post-instructional survey was 2 and second post-instructional survey was 3. The data for grade level were coded as follows:

Table 4-3
Coded Value for Participant Grade Level

Grade Level	Coded Value
Freshman	1
Sophomore	2
Junior	3
Senior	4

Table 4-4 notes the coded values for gender data.

Table 4-4
Coded Values for Participant Gender

Gender	Coded Value
Male	1
Female	2

The linear mixed model used for the analysis is as follows (Fox, 2002):

$$y_{ijkl} = \mu + \alpha_i + \beta_j + \gamma_k + \varepsilon_l$$

where

$\alpha_1, \dots, \alpha_i$ are the fixed-effect coefficients, which takes into account the three separate times that data was collected and is represented by $\sum_{i=1}^{3} \alpha_i$.

$\beta_1, \dots, \beta_j$ are the fixed-effect coefficients, which takes into account the gender (male and female) data and is represented by $\sum_{i=2}^{2} \beta_i$.

$\gamma_1, \dots, \gamma_k$ are random effect coefficients, supposed to be normally distributed, represented by $\gamma_k \sim N(0, \sigma_{ID}^2)$.

ε_l is the standard error, presumed to be distributed normally and represented by $\varepsilon_l \sim N(0, \sigma^2)$, in the observations *j* in the group of participants *k*.

After the linear model was constructed, the test for significance was performed for the overall response using RStudio 0.98. The data gathered from the pre instructional survey was used as baseline for the analysis and the level of significance was set at 90% ($\alpha = .1$) primarily due to the small sample size. The test was carried out for the hypothesis noted below:

H_0 = There is no statistically significant increase in positive attitudes about programming in students who are exposed to a graphical programming interface for microcontroller programming.

H_a = There is statistically significant increase in positive attitudes about programming in students who are exposed to a graphical programming interface for microcontroller programming.

This can be stated in the mathematical terms as below:

H_0: $Response_0 = Response_\alpha$

H_α: $Response_0 < Response_\alpha$

The test for significance was carried out by the researcher while taking into consideration time and gender for all freshman participants. The results are noted below.

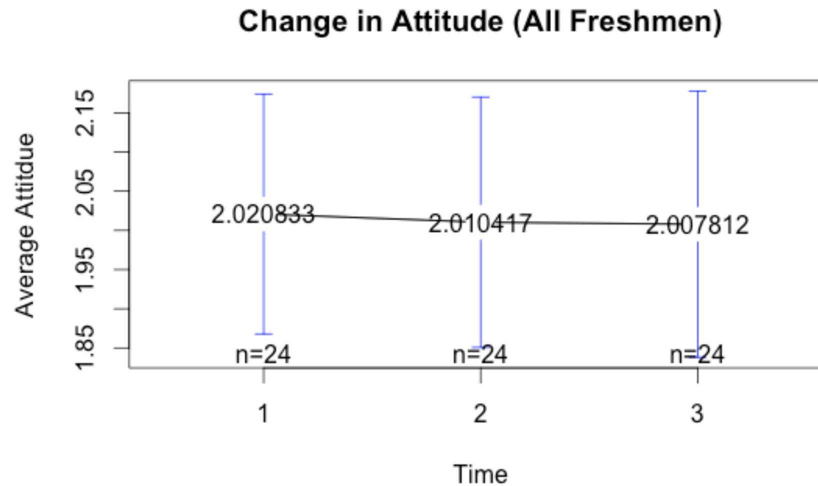

Figure 4-2 Changes in Attitude Over the Research Period

The graph above combines changes in attitudes for all four categories (confidence, interest, stereotype, and usefulness) for all freshmen participants. Based on this, it is clear that, the attitude changes between the pre and first post-instructional survey were marginal. Also, the changes from first post-instructional to the second post-instructional survey were minimal. The findings were corroborated by the results of the significance test.

Table 4-5
P-Values for Attitude Changes Between Surveys for All Participants

Attitude Change Between Surveys	*p*-Value
Pre and Post 1	0.9951
Pre and Post 2	0.9923
Post 1 and Post 2	0.9997

For all freshmen, the changes between first and second post-instructional survey are significant, while the changes from pre and first post-instructional survey were insignificant.

The following graph shows the average changes in attitudes for males and females. In the graph "1" implies males, while "2" denotes females.

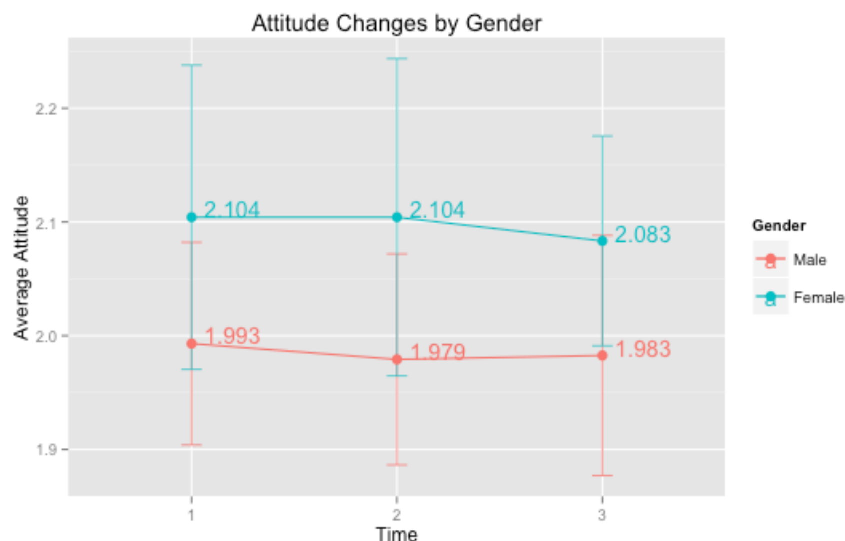

Figure 4-3 – Attitude Changes Based on Participant Gender

It can be seen that after the pre-instructional session, the average score for males increased, while the average score for females decreased substantially. Table 4-6 shows the results for the significance test, in which gender was included as one of the fixed-effect coefficients.

Table 4-6
Attitude Changes Between Survey Differentiated by Gender

Surveys	Average Attitude	
	Male	Female
Pre	1.993	2.104
Post 1	1.979	2.104
Post 2	1.983	2.083

The researcher also measured the attitude changes in all individual categories. Following are the results when the mixed linear model was applied to the categories.

Table 4-7
Average Attitudes By Categories Measured

Attitude Change Between Surveys	Confidence	Interest	Stereotypes	Usefulness
Pre	2.115	1.833	2.052	2.083
Post 1	2.021	1.781	2.135	2.104
Post 2	2.052	1.854	2.083	2.042

4.3 Equivalence Testing

The *p*-value can only provide evidence against the null hypothesis. As the dataset for this research project contains a small sample size, the use of the equivalence test is warranted to ensure proper conclusions are reached. Following is the mathematical representation of the equivalence test for this study:

$$H_0: |\mu_1 - \mu_2| \geq \delta \quad \text{and} \quad H_\alpha: |\mu_1 - \mu_2| < \delta$$

To perform this test, confidence intervals were created for all freshmen participants differentiated by time. The table below notes both the confidence interval and associated *p*-value are noted below.

Table 4-8
Confidence Interval for all Significance Testing

Time	Confidence Interval	*p*-value
Pre	(-0.0792, 0.1000)	0.2032
Post 1	(-0.0948, 0.1000)	0.1958
Post 2	(-0.0740,0.1000)	0.2175

4.4 Qualitative Analysis of the data

In order to understand what the participants in the survey thought of the importance of programming and their views on flowcharting software, two open-ended questions were included in the all three surveys. The questions are:

- Describe, in detail, what you can achieve by learning programming in your academic life.
- Have you learned about flowcharts before? Do you think it can help you think logically?

Based on the visual inspection of the data, most of the participants answered these two questions on all three surveys. All responses to these questions were thoroughly inspected. The researcher found that most of the responses demonstrated a positive attitude toward programming and flowcharting. The researcher attempted to use sentiment analysis and perform test of significance on the data. Sentiment analysis is defined as "the computational study of people's opinions, appraisals, attitudes, and emotions toward entities, individuals, issues, events, topics and their attributes" (Liu & Zhang, 2012).

The researcher analyzed the data using a Sentiment Analysis tool developed by Jain (2014). The tool creates a file that contains the average goodness probability of frequently occurring words also called sentiments. Additionally, standard deviation of a sentiment is also noted for all words appearing more than three times in the text of survey responses. Average

goodness probability of sentiment closer to 1 suggests positive sentiment for a particular word. Tables 4-9, 4-10, and 4-11 show the results of the analyses for the pre- and both post-instructional surveys. Table 4-12 shows responses for the words that appeared on all three surveys to show the progression of sentiment. Responses for both questions were combined for the results.

Table 4-9
Goodness Probability of Frequent Words in Pre Instructional Survey

Word	Count	Average Goodness Probability of Sentiment	Standard Deviation
Programming	230	0.0233	0.0674
Flowcharts	150	0.0065	0.0196
Process	70	0.0019	0.0031
Charts	60	0.0004	0.0007
Flow	60	0.0004	0.0007
Life	60	0.0007	0.0012
Skills	60	0.0453	0.0915
Computer	50	0.0001	0.0001
Courses	50	0.0012	0.0019
Help	50	0.0009	0.0010

The results for the first post-instructional survey are shown in Table 4-10.

Table 4-10
Goodness Probability of Frequent Words in Post Instructional Survey 1

Word	Count	Average Goodness Probability of Sentiment	Standard Deviation
Programming	150	0.0749	0.2083
Flowcharts	130	0.0209	0.0267
Help	80	0.3139	0.3784
Life	70	0.5691	0.4032
Things	60	0.0156	0.0300
Code	50	0.1255	0.2017
Skill(s)	50	0.6865	0.3538
Process	40	0.1782	0.3007
Way	40	0.0958	0.1010
Computer(s)	60	0.0341	0.0662

The results for the second post-instructional survey are noted below for frequently occurring words.

Table 4-11
Goodness Probability of Frequent Words in Post Instructional Survey 2

Word	Count	Average Goodness Probability of Sentiment	Standard Deviation
Programming	170	0.1432	0.3042
Flowcharts	90	0.0157	0.0221
Skills	90	0.5052	0.3503
Life	70	0.6440	0.3764
Problem	70	0.4354	0.3658
Problems	60	0.0470	0.0801
Job	50	0.1454	0.2070
Knowledge	50	0.4225	0.4515
Skill	50	0.6421	0.3330
Code	40	0.0003	0.0004
Computer(s)	30	0.1432	0.3042

The analysis of the three tables 4-9, 4-10, and 4-11, it is clear that few words occurred frequently in all three surveys.

Table 4-12
Goodness Probability of Repeated Words on All Three Surveys

Word	Average Goodness	Standard Deviation	Average Goodness	Standard Deviation	Average Goodness	Standard Deviation
	Pre-session		Post-session 1		Post-session 2	
Programming	0.0233	0.0674	0.0749	0.2083	0.1432	0.3042
Flowcharts	0.0065	0.0196	0.0209	0.0267	0.0157	0.0221
Skill(s)	0.0453	0.0915	0.6865	0.3538	0.5052	0.3503
Life	0.0007	0.0012	0.5691	0.4032	0.6440	0.3764
Computer(s)	0.0001	0.0001	0.0341	0.0662	0.0225	0.0321

The five words – computer(s), flowcharts, life, programming, and skill(s) – that appeared on all three lists, show an overall improved goodness sentiment after the pre instructional survey. The overall positive sentiment in the common words can be observed as increasing.

4.5 Summary

This chapter provided detailed information about how the data was conditioned and the type of analysis performed on both quantitative and qualitative data, which was collected during the experiment.

During the quantitative analysis of the Liker Scale data, no significance was found. Analysis of answers to the descriptive questions at the end of surveys pointed to overall positive attitude among participants about programming.

CHAPTER 5. CONCLUSIONS, DISCUSSION, AND RECOMMENDATIONS

This thesis described the process used to measure changes in participant attitudes, in terms of confidence, interest, stereotypes, and usefulness. The participants were given an instructional session, which utilized microcontroller programming using flowchart-based visual programming tool. This chapter presents relevant conclusions and recommendations based on the work described in the previous chapters.

5.1 Conclusions

Historically, textual-based programming languages have been used to teach introductory programming courses. As students tend to be visual learners, these programming languages do not enable them visualize the flow of logic throughout the program. Graphical programming languages provide a new approach to teaching students programming, in addition to critical thinking skills. Flowcharting is one the most basic techniques used to map the logic of a program. When flowcharting technology is paired with a microcontroller, students can create a novel application, which they can touch and feel. This research project augmented the current approach of using visual programming languages with a microcontroller technology to study if it was possible to improve the attitude of freshmen college students in programming. The study also

hypothesized that students' interest level would improve after providing them a demonstration of visual programming language and microcontroller technology.

The data analysis measured changes in attitude based on the pre-instructional survey, post-instructional survey 1, and post-instructional survey 2. Table 5-1 shows results of the hypothesis testing.

The data does not provide enough evidence to show a significant different between attitudes throughout the experiment, and, therefore, the results of the experiment are inclusive.

Table 5-1
Results of Hypothesis Testing

Sessions	Hypotheses Testing Results
Pre- and Post-Instructional Survey 1	H_0 Not Rejected
Pre- and Post-Instructional Survey 2	H_0 Not Rejected
Post- 1 and Post-Instructional Survey 2	H_0 Not Rejected

The results show that there was not a significant difference between the participant attitudes after the instructional session, which included a demonstration of Phoenix Contact NanoLine microcontroller. Also, no positive change in student attitudes between the post instructional survey 1 and post instructional survey 2 was recorded. As the *p*-values were quite large to be not significant, further investigation of the impact of graphical programming languages with microcontroller is warranted. The hypotheses for the study were tested for changes between all three surveys.

The participants in the experiment were students in CNIT 15501. One of the assumptions for the study states that participants have little to no prior programing experience. The lack of deep knowledge about programming may explain the insignificant results from all three surveys. The participants continued to learn about different programming techniques throughout the course of experiment during their regular course lectures and labs; they may continue to form opinions regarding programming throughout the course of the semester. It can be theorized that the insignificance found between surveys may be due to changing attitude toward programming concepts.

Also, the results gathered for freshman and non-freshman participants were very similar; inconclusive. The analysis also pointed to the conclusion that the gender did not impact overall attitudes of the participants. No statistical significance was found across all three surveys for the four categories stated previously in Chapter 3 and Chapter 4.

5.2 Implications of the Study

The research project investigated the idea of using cyber-physical systems for creating a positive attitude about programming and showed that overall attitude toward programming may be improved by providing a subjects a prolonged exposure to graphical programming technology. This is the very first study done at Purdue University, in which the participants utilized a flowchart-based programming language to program a Phoenix Contact NanoLine microcontroller. The study proposed a different way to introduce programming in

a simple, easy to understand but methodical way to students before teaching them textual-based programming languages. This project also offered a notion, which can be further enhanced into curriculum courses to promote enhanced student learning of programming languages throughout their early college education.

A study like this, which can find significance difference in student attitude, may have wide-ranging consequences for researchers in technology-education or engineering-education who are tasked with improving learning outcomes of programming courses. The project and methodology used to accomplish this shows a promise to improve student attitudes in long term.

5.3 Challenges of using Graphical Programming Languages and Student Comprehension

Although using graphical programing languages may, theoretically, improve student attitude toward programming, it is important to keep in mind challenges related to such study.

1. This approach assumes that most of the students are visual learners. If the students do not fall in this category, using graphical programming languages in conjunction with microcontroller may not change their interest in programming and, in turn, attitudes.
2. If students are taught graphical programming language in an introductory course, they may struggle to transition to textual-based object oriented

programming languages, which, today, are exclusively used for application development.

These issues can be potential threats to the efficacy of the advocated approach to improve student attitudes about programming. Nonetheless, the method proposed in this research may be further investigated to mitigate or minimize the challenges.

5.4 Future Work and Recommendations

This was a study designed to study changes in attitude toward programming by providing participants an instructional session, which incorporated flowchart-based programming language and microcontroller technology. There are multiple ways to further this study to investigate attitude changes.

1. Consideration must be given to the fact that the experiment was conducted on students who were already in a programming class. It is possible to students were interested in programming even before the experiment. Therefore, to improve their interest and measure such change, a future study, similar to one described in this thesis, may use participants from General Studies or undecided majors. These students can be taught the fundamentals of critical thinking and programing through a newly designed course, which uses microcontroller technology coupled with flowcharting software.

2. The sample size for this study was quite small. Increasing sample size may yield more significant data related to how students perceive the instructional session.
3. It is possible to study two groups of students, in which one group of students are taught graphical programming languages, while the other group are taught traditional textual programming languages. Their attitudes can be measured after completing the programming curricula.
4. A new course can be designed, which teaches students graphical programming language. A follow-on course can be taught using a textual programming language. The overall attitude of the students can be assessed at the conclusion of the introductory graphical language course and at the end of the textual language course.
5. Only one instructional session was delivered to the students in this study. In future, multiple sessions may be delivered to students and student attitudes can be measured after each session.
6. When measuring attitudes of the students, more statements may be included in the Likert Scale-based survey. More survey questions may allow researchers to gather more data points about each category, providing greater insight into student attitude.
7. A five or seven point Likert scale survey may be used to capture attitude data. This may allow for increased granular information about specific attitude characteristics of the participants.

LIST OF REFERENCES

LIST OF REFERENCES

Aguilar-Savén, R. S. (2004). Business process modelling: Review and framework. *International Journal of Production Economics*, *90*(2), 129–149. doi:10.1016/S0925-5273(03)00102-6

Anderson, M., McKenzie, A., Wellman, B., Brown, M., & Vrbsky, S. (2011). Affecting attitudes in first-year computer science using syntaxfree robotics programming. *ACM Inroads*, *2*(3), 51–57. doi:10.1145/2003616.2003635

Baser, M. (2013). Attitude, gender and achievement in computer programming. *Online Submission*, *14*(2), 248–255.

Bevan, J., Werner, L., & McDowell, C. (2002). Guidelines for the use of pair programming in a freshman programming class. In *15th Conference on Software Engineering Education and Training, 2002. (CSEE T 2002). Proceedings* (pp. 100–107). doi:10.1109/CSEE.2002.995202

Bucks, G., & Oakes, W. (2010). Integration of graphical programming into a first-year engineering course. In *American Society for Engineering Education*. American Society for Engineering Education.

Burton, P. J., & Bruhn, R. E. (2003). Teaching programming in the OOP era. *SIGCSE Bull.*, *35*(2), 111–114. doi:10.1145/782941.782993

Carlisle, M. C., Wilson, T. A., Humphries, J. W., & Hadfield, S. M. (2005). RAPTOR: A visual programming environment for teaching algorithmic problem solving. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education* (pp. 176–180). New York, NY, USA: ACM. doi:10.1145/1047344.1047411

Chen, S. (n.d.). Classroom tools for computer science and mathematics. Retrieved November 20, 2014, from http://www.edutoolresearch.com/index.html

Chen, S., & Morris, S. (2005). Iconic programming for flowcharts, Java, Turing, etc. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (pp. 104–107). New York, NY, USA: ACM. doi:10.1145/1067445.1067477

Chun, S.-J., & Ryoo, J. (2010). Development and application of a web-based programming learning system with LED display kits. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 310–314). New York, NY, USA: ACM. doi:10.1145/1734263.1734369

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). Cambridge, Mass: MIT Press.

Crews, T., & Butterfield, J. (2002). Using technology to bring abstract concepts into focus: A programming case study. *Journal of Computing in Higher Education*, *13*(2), 25–50. doi:10.1007/BF02940964

Dabroom, A., Refie, W. M., & Matmti, R. (2013). Microcontroller-based learning kit: Course design using constructive alignment principles. In *2013 21st Mediterranean Conference on Control Automation (MED)* (pp. 558–566). doi:10.1109/MED.2013.6608777

Daintith, J. (2004). While programming language. *A Dictionary of Computing* (2004th ed.). Encyclopedia.com. Retrieved from http://www.encyclopedia.com/doc/1O11-whileprogramminglanguage.html

De Jesus, E. (2011). Teaching computer programming with structured programming language and flowcharts. In *Proceedings of the 2011 Workshop on Open Source and Design of Communication* (pp. 45–48). New York, NY, USA: ACM. doi:10.1145/2016716.2016729

DiNitto, S.A., J. (1988). Future directions in programming languages. In *, International Conference on Computer Languages, 1988. Proceedings* (pp. 169–176). doi:10.1109/ICCL.1988.13061

Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A. G., & Mehandjiev, N. (2004). Meta-design: A manifesto for end-user development. *Commun. ACM*, *47*(9), 33–37. doi:10.1145/1015864.1015884

Forneris, S. G., & Peden-McAlpine, C. (2007). Evaluation of a reflective learning intervention to improve critical thinking in novice nurses. *Journal of Advanced Nursing*, *57*(4), 410–421. doi:10.1111/j.1365-2648.2007.04120.x

Fox, J. (2002). Linear mixed models appendix to an R and S-PLUS companion to applied regression. Retrieved from http://cran.r-project.org/doc/contrib/Fox-Companion/appendix-mixed-models.pdf

Garrett, J., & Walker, T. (2008). Student attitudes towards the use of graphical programming languages. In *Proceedings of 2008 ASEE Southeastern Section*. Memphis, Tennessee.

Goadrich, M. (2014). Incorporating tangible computing devices into CS1. *J. Comput. Sci. Coll.*, *29*(5), 23–31.

Golubev, M., Istria, M., & Irawan, H. (2014, November 17). What is GMF ? Retrieved from https://wiki.eclipse.org/Graphical_Modeling_Framework

Gosling, J., Steele, G., Joy, B., Bracha, G., & Buckley, A. (2013, February 28). Chapter 12. Execution. Retrieved April 6, 2014, from http://docs.oracle.com/javase/specs/jls/se7/html/jls-12.html

Gross, P., & Powers, K. (2005). Evaluating assessments of novice programming environments (pp. 99–110). ACM Press. doi:10.1145/1089786.1089796

Hils, D. D. (1992). Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, *3*(1), 69–101. doi:10.1016/1045-926X(92)90034-J

Hundhausen, C. D., & Brown, J. L. (2007). What You See Is What You Code: A “live” algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing*, *18*(1), 22–47. doi:10.1016/j.jvlc.2006.03.002

Hwang, C.-S., Su, Y.-C., & Tseng, K.-C. (2010). Effects of computer game-based instruction on students' programming achievement in Taiwan. In *2010 International Conference on Computational Aspects of Social Networks (CASoN)* (pp. 233–236). doi:10.1109/CASoN.2010.60

IEEE Standard Glossary of Computer Hardware Terminology. (1995). *IEEE Std 610.10-1994*, i–. doi:10.1109/IEEESTD.1995.79522

IEEE Standard Glossary of Software Engineering Terminology. (1990). *IEEE Std 610.12-1990*, 1–84. doi:10.1109/IEEESTD.1990.101064

Jain, A. (2014, May). *Analyzing responses to open ended questions for SPIRIT using aspect oriented sentiment analysis*. Purdue University, West Lafayette.

Kato, Y. (2010). Splish: A visual programming environment for Arduino to accelerate physical computing experiences (pp. 3–10). IEEE. doi:10.1109/C5.2010.20

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, *37*(2), 83–137. doi:10.1145/1089733.1089734

Kenefick, S. (2011, December 19). Integrated development environment. Retrieved April 6, 2014, from https://www.gartner.com/doc/1879714/integrated-development-environment

Liu, B., & Zhang, L. (2012). A survey of opinion mining and sentiment analysis. In C. C. Aggarwal & C. Zhai (Eds.), *Mining Text Data* (pp. 415–463). Springer US. Retrieved from http://link.springer.com/chapter/10.1007/978-1-4614-3223-4_13

logic. (2014). *Oxford Dictionaries*. Oxford University Press. Retrieved from http://www.oxforddictionaries.com/us/definition/american_english/logic

Lucanin, D., & Fabek, I. (2011). A visual programming language for drawing and executing flowcharts. In *2011 Proceedings of the 34th International Convention MIPRO* (pp. 1679–1684).

Lutes, K. (2013). *Envigilator*. West Lafayette: DelMar Software Development, LLC. Retrieved from http://www.envigilator.com/

Malan, D. J., & Leitner, H. H. (2007). Scratch for budding computer scientists. *ACM SIGCSE Bulletin*, *39*(1), 223–227.

Mateas, M. (2005). Procedural literacy: Educating the new media practitioner. *On the Horizon*, *13*(2), 101–111. doi:10.1108/10748120510608133

Munson, A., Moskal, B., Harriger, A., Lauriski-Karriker, T., & Heersink, D. (2011). Computing at the high school level: Changing what teachers and students know and believe. *Computers & Education*, *57*(2), 1836–1849. doi:10.1016/j.compedu.2011.03.005

Nikishkov, G. P., & Kanda, H. (1999). The development of a Java engineering application for higher-order asymptotic analysis of crack-tip fields. *Advances in Engineering Software*, *30*(7), 469–477. doi:10.1016/S0965-9978(98)00131-8

Park, J. (2003). *Practical embedded controllers: Design and troubleshooting with the Motorolla [i.e. Motorola] 68HC11*. Oxford ; Burlington, MA: Newnes.

Quick and easy programming. (2014). Retrieved from https://www.phoenixcontact.com/online/portal/us?1dmy&urile=wcm:path:/usen/web/main/products/subcategory_pages/programming_p-19-05/af3bf5fe-db28-41f2-bfe8-f7f9e8ee93f1/af3bf5fe-db28-41f2-bfe8-f7f9e8ee93f1

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. (2009). Scratch: Programming for all. *Commun. ACM*, *52*(11), 60–67. doi:10.1145/1592761.1592779

Rieber, L. P. (1995). A historical review of visualization in human cognition. *Educational Technology Research and Development*, *43*(1), 45–56.

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

Rogers, J. R., & McVay, R. C. (2012). Graphical microcontroller programming. In *2012 IEEE International Conference on Technologies for Practical Robot Applications (TePRA)* (pp. 48–52). doi:10.1109/TePRA.2012.6215653

Santos, Á., Gomes, A., & Mendes, A. J. (2010). Integrating new technologies and existing tools to promote programming learning. *Algorithms*, *3*(2), 183–196. doi:10.3390/a3020183

Shackelford, R. L., & LeBlanc, R.J., J. (1997). Introducing computer science fundamentals before programming. In *Frontiers in Education Conference, 1997. 27th Annual Conference. Teaching and Learning in an Era of Change. Proceedings.* (Vol. 1, pp. 285–289 vol.1). doi:10.1109/FIE.1997.644858

Shashaani, L., & Khalili, A. (2001). Gender and computers: Similarities and differences in Iranian college students' attitudes toward computers. *Computers & Education*, *37*(3–4), 363–375. doi:10.1016/S0360-1315(01)00059-8

Smith, B. J., & Delugach, H. S. (2010). Work in progress - Using a visual programming language to bridge the cognitive gap between a novice's mental model and program code. In *2010 IEEE Frontiers in Education Conference (FIE)* (pp. F3G–1–F3G–3). doi:10.1109/FIE.2010.5673502

What is Alice? (2014). Retrieved from http://www.alice.org/index.php?page=what_is_alice/what_is_alice

What is Arduino? (2014). Retrieved from http://arduino.cc/en/Guide/Introduction

What is LabVIEW? (2014). Retrieved from http://www.ni.com/labview/

White, G., & Sivitanides, M. (2005). Cognitive differences between procedural programming and object oriented programming. *Information Technology and Management*, *6*(4), 333–350. doi:10.1007/s10799-005-3899-2

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., & Corritore, C. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, *11*(3), 255–282.

Winslow, L. E. (1996). Programming pedagogy - Psychological overview. *SIGCSE Bull.*, *28*(3), 17–22. doi:10.1145/234867.234872

APPENDICES

Appendix A: Approval to Use Attitude Survey

From: **Barbara Moskal** bmoskal@mines.edu
Subject: Re: Using Attitude Survey for Graduate Thesis
Date: April 2, 2014 at 12:45 PM
To: Godbole, Saurabh S sgodbole@purdue.edu
Cc: Alka Herring harrigea@purdue.edu

Saurabh:

Yes-- please feel free to use the survey and properly cite it.

Thank you for asking,
Dr. Moskal

On 4/1/14 9:15 PM, "Godbole, Saurabh S" <sgodbole@purdue.edu> wrote:

> Dr. Moskal,
>
> I am Saurabh Godbole and I am working on my graduate thesis under Prof. Alka Harriger of Purdue University. The primary focus of this research is to evaluate attitudes of college freshmen toward programming.
>
> During the literature review, I came across an article written by you, Prof. Harriger, and others in 2011 titled ³Computing at the high school level: Changing what teachers and students know and believe². This article contained an attitude survey, which is very similar to what I was planning to create for the research. After a discussion with Prof. Harriger, she suggested that I should ask you to include this survey in my thesis research.
>
> As note previously, I would like to use this survey to understand and evaluate attitudes of college freshmen after a session of teaching them basic programming concepts. Before I proceed, I would like to formally ask for permission to utilize this same attitude survey in my research.
>
> Thank you and I look forward to your reply.
>
> Best Regards,
> Saurabh Godbole
> Graduate Student
> College of Technology
> Purdue University, West Lafayette

Appendix B: Instructor Approval

From: **Ravai, Guity** guity@purdue.edu
Subject: RE: Participants in Experiment for Thesis
Date: April 4, 2014 at 7:39 PM
To: Godbole, Saurabh S sgodbole@purdue.edu

You are welcome Saurabh!
It is a good project; am also curious to see how it impacts my students. Best,

Guity Ravai
C&IT Continuing Lecturer
Knoy 239
Purdue University
guity@purdue.edu
765-496-6005

From: Godbole, Saurabh S
Sent: Friday, April 04, 2014 1:59 PM
To: Ravai, Guity
Cc: Harriger, Alka R
Subject: Re: Participants in Experiment for Thesis

Prof. Ravai,

I enjoyed our conversation today. It has helped me better formulate my research methodology and improve the overall quality of my research project. I will include the details of our discussion in my thesis proposal and keep you in the loop throughout the planning of this study, which will be conducted in Fall 2014.

I really appreciate your help and participation in this research. Thank you!

Best Regards,
Saurabh Godbole

On Apr 4, 2014, at 1:43 PM, Ravai, Guity <guity@purdue.edu> wrote:

> Dear Saurabh,
> It was nice talking to you about your research project this afternoon.
> I agree with participating in this study next Fall.
> I believe that my students in CNIT 155 will benefit from exposure to programming microcontrollers.
> Please go ahead with your proposal and keep me posted.
>
> Guity Ravai
>
> Guity Ravai
> CIT, Continuing Lecturer
> KNOY Hall 239
> Purdue University
> guity@purdue.edu
>
> From: Godbole, Saurabh S
> Sent: Friday, April 04, 2014 10:02 AM
> To: Ravai, Guity
> Subject: Re: Participants in Experiment for Thesis
>
> I will be at your office at 12:30 then. Thank you!
>
> Best Regards,
> Saurabh
>
> On Apr 4, 2014, at 10:00 AM, Ravai, Guity <guity@purdue.edu> wrote:
>
>> Hi Saurabh,
>>
>> I will be in my office today from 12:30 - 2:30 PM, stop by if you can.
>>
>> Guity Ravai
>> C&IT Continuing Lecturer
>> Knoy 239
>> Purdue University
>> guity@purdue.edu
>> 765-496-6005
>>
>> From: Godbole, Saurabh S
>> Sent: Wednesday, April 02, 2014 5:17 PM
>> To: Ravai, Guity
>> Subject: Participants in Experiment for Thesis
>>
>> Hello Ms. Ravai,
>>
>> I am Saurabh Godbole and I working on my thesis under Prof. Alka Harriger. The primary focus of this research is to evaluate attitudes of college freshmen toward programming.
>>
>> In this research, I will be studying freshmen in CNIT. After having a discussion with Prof. Harriger, she suggested that I may want to study freshmen in CNIT 155. I believe that this research may improve student attitude toward programming.
>>
>> I would like to schedule an appointment with you this week, if possible. I am available tomorrow (4/3) after 1:30pm and all day on Friday (4/4). I would like to discuss my research with you regarding the possibility of using students in CNIT 155 as subjects next semester (Fall 2014).
>>
>> Thanks for you time. I look forward to your reply.
>>
>> Best Regards,
>> Saurabh Godbole
>> Graduate Student
>> College of Technology
>> Purdue University, West Lafayette

Appendix C: IRB Approval for Research

HUMAN RESEARCH PROTECTION PROGRAM
INSTITUTIONAL REVIEW BOARDS

To:	ALKA HARRIGER KNOY 243
From:	JEANNIE DICLEMENTI, Chair Social Science IRB
Date:	06/17/2014
Committee Action:	Approval
IRB Action Date	06/17/2014
IRB Protocol #	1404014778
Study Title	Impact of Programming Cyber-Physical Systems on the Interest Level of Freshmen College Students
Expiration Date	06/16/2015

Following review by the Institutional Review Board (IRB), the above-referenced protocol has been approved. This approval permits you to recruit subjects up to the number indicated on the application form and to conduct the research as it is approved. The IRB-stamped and dated consent, assent, and/or information form(s) approved for this protocol are enclosed. Please make copies from these document(s) both for subjects to sign should they choose to enroll in your study and for subjects to keep for their records. Information forms should not be signed. Researchers should keep all consent/assent forms for a period no less than three (3) years following closure of the protocol.

Revisions/Amendments: If you wish to change any aspect of this study, please submit the requested changes to the IRB using the appropriate form. IRB approval must be obtained before implementing any changes unless the change is to remove an immediate hazard to subjects in which case the IRB should be immediately informed following the change.

Continuing Review: It is the Principal Investigator's responsibility to obtain continuing review and approval for this protocol prior to the expiration date noted above. Please allow sufficient time for continued review and approval. No research activity of any sort may continue beyond the expiration date. Failure to receive approval for continuation before the expiration date will result in the approval's expiration on the expiration date. Data collected following the expiration date is unapproved research and cannot be used for research purposes including reporting or publishing as research data.

Unanticipated Problems/Adverse Events: Researchers must report unanticipated problems and/or adverse events to the IRB. If the problem/adverse event is serious, or is expected but occurs with unexpected severity or frequency, or the problem/even is unanticipated, it must be reported to the IRB within 48 hours of learning of the event and a written report submitted within five (5) business days. All other problems/events should be reported at the time of Continuing Review.

We wish you good luck with your work. Please retain copy of this letter for your records.

Ernest C. Young Hall, 10th Floor - 155 S. Grant St. - West Lafayette, IN 47907-2114 - (765) 494-5942 - Fax: (765) 494-9911

Appendix D: Participant Pre- and Post-Instructional Surveys

Participant Attitude Survey

***Note**: This survey will be administered online using Qualtrics Survey Software. The following questions will be used to create this online survey. The two questions regarding "Demographics" and one question regarding "Prior Programming Experience" will not appear on post-instruction surveys.*

Instructions:

1. Type in your assigned 10-digit unique ID in the box labeled Participant Identification Number.
2. There are 16 multiple-choice survey questions and 2 short-answer questions. For each multiple-choice question, please select the one ***best*** alternative in your opinion.
3. This survey is simply asking your opinion about a number of things related to programming both before and after the instructional session. There are no wrong or right answers.
4. For **questions 1-16**, please select from the choices below:
    - Strongly Agree
    - Agree
    - Disagree
    - Strongly Disagree
5. There are 2 short-answer questions at the end of the survey. Use the boxes provided to type your answers. You can write answers in your own words in the box given for the open-ended questions. While answering these questions, ***do not*** include your name or PUID, or any other personally identifiable information.
6. When you are done, click on ***Submit*** to finish your survey.

Demographics:

- What is your college grade level?
    - Freshman
    - Sophomore
    - Junior
    - Senior

- Are you a male or female?
    - Male
    - Female

Survey Questions:

Table D-1

Pre- and Post-Instructional Survey Questions

No.	Question	Strongly Agree	Agree	Disagree	Strongly Disagree
1.	I am confident with learning programming concepts.				
2.	I think programming is interesting.				
3.	A student who performs well in programming courses will probably not have a life outside of computers.				
4.	I hope that my future career will require the use of programming concepts.				
5.	I do not think that I will take additional programming courses.				
6.	I am not interested in learning programming concepts.				
7.	To do well in programming, a student must spend most of his/her time at a computer.				
8.	Knowledge of programming will allow me to secure a good job.				
9.	I would not take additional programming courses if I were given the opportunity.				

Table D-1 Continued

10.	A student who performs well in programming courses is likely to have a life outside of computers.
11.	I think programming is boring.
12.	I hope that I can find a career that does not require the use of programming concepts.
13.	I have little self-confidence when it comes to programming courses/activities.
14.	I want to learn programming concepts.
15.	Doing well in programming does not require a student to spend most of his/her time at a computer.
16.	Knowledge of programming skills will not help me secure a good job.

Short Answer Questions:

1. Describe, in detail, what you can achieve by learning programming in your academic life.
2. Have you learned about flowcharts before? Do you think it can help you think logically?

Appendix E: Attitude Category and Related Questions

Confidence

1. I am confident with learning programming concepts.
2. I have little self-confidence when it comes to programming courses/activities.
3. I do not think that I will take additional programming courses.
4. I would not take additional programming courses if I were given the opportunity.

Interest

5. I think programming is interesting.
6. I am not interested in learning programming concepts.
7. I think programming is boring.
8. I want to learn programming concepts.

Stereotypes

9. A student who performs well in programming courses will probably not have a life outside of computers.
10. To do well in programming, a student must spend most of his/her time at a computer.
11. A student who performs well in programming courses is likely to have a life outside of computers.
12. Doing well in programming does not require a student to spend most of his/her time at a computer.

Usefulness

13. I hope that my future career will require the use of programming concepts.
14. Knowledge of programming will allow me to secure a good job.
15. I hope that I can find a career that does not require the use of programming concepts.
16. Knowledge of programming skills will not help me secure a good job