TOWSON UNIVERSITY

COLLEGE OF GRADUATE STUDIES AND RESEARCH

MICROWORLDS TO IMPROVE LEARNING IN INTRODUCTORY COMPUTER SCIENCE COURSES

by

James A. Robertson

A Dissertation

Presented to the faculty of

Towson University

in partial fulfillment

of the requirements for the degree

Doctor of Education

May 2010

Towson University
Towson, Maryland 21252

TOWSON UNIVERSITY
COLLEGE OF GRADUATE STUDIES AND RESEARCH

DISSERTATION APPROVAL PAGE

This is to certify that the dissertation prepared by James A. Robertson

entitled Microworlds to Improve Learning in Introductory Programming Courses

has been approved by his or her committee as satisfactory completion of the dissertation

requirement for the degree Doctor of Education.

Jeffrey Kenton, PhD ______________ Date ________
Chair, Dissertation Committee

William Sadera, PhD ______________ Date ________
Committee Member

Liyan Song, PhD ______________ Date ________
Committee Member

Gabriele Meiselwitz, EdD ______________ Date ________
Committee Member

Marilyn Nicholas, PhD ______________ Date ________
Committee Member

Chao Lu, PhD ______________ Date ________
Dean, College of Graduate Studies and Research

Acknowledgements

There are number of people that contributed significantly to the contents and organization of this study. Without their kind, patient and dedicated support, this research would have not been possible. I am grateful for my dissertation advisor, Dr. Jeff Kenton whose recommendations and excellent support guided this study from its inception to conclusion. His insight and contributions throughout the entire process added value and quality to the research and this dissertation.

I appreciate the technical support, guidance provided by Dr. Scott Mcnary in the areas of research design and statistical analysis techniques. Dr. Mcnary's technical direction helped me align the proper analysis techniques with each instrument used in this research.

Special thanks goes to Dr. Nick Duchon for agreeing to pilot and conduct the full study over two semesters at the university. Dr. Duchon contributed significant insight into the programming environments and the students over several months while still maintaining his other teaching and duties at the university.

Finally, a special thanks to all of my dissertation committee members for patience, leadership and technical guidance throughout this process. Each member provided feedback that shaped and improved this study.

Abstract

MICROWORLDS TO IMPROVE LEARNING IN

INTRODUCTORY COMPUTER SCIENCE COURSES

James A. Robertson

Novice programmers often struggle when attempting to learn how to write code while reducing the number of programming errors. This study investigates tools and techniques that can be used to reduce some of the obstacles many students face when learning to write a computer program. Specifically, this research aims to evaluate if entry-level programming students who use the Alice 2.0 programming environment demonstrate a better understanding of fundamental programming concepts than students who use a traditional C++ programming environment. Approximately 70 students from two face-to-face CS0 sections taught by the same instructor participated in this research. The instruments used in this research included a pre-test, demographic questionnaire, three programming assignments, a post-test, course evaluations, and final course grades. A rubric was used for the instructor to grade each programming assignment. Each assessment activity was carefully aligned with one or more course learning objectives. Results of this study showed students who used the Alice programming environment consistently scored higher in the layout (visual appeal) grading component for all programming assignments. There were no differences found between the two programming environments in regards to code functionality or design, or in the pre- and post-test scores between the two groups. A larger percentage of students from the group that used the Alice programming environment successfully passed the course. However, students in the Alice group rated the instructor and overall course significantly lower than students who used the C++ programming environment.

.

Table of Contents

List of Tables xi

Chapter I. Introduction 1

- Background 1
- Statement of the Problem 9
- Purpose of Research 9
- Significance of Research 10
- Research Overview 12
- Research Questions 12
- Limitations 13
- Definition of Terms 13

Chapter II. Literature Review 16

- Programming within a Computing Curriculum 16
  - CS0 18
  - CS1 19
  - CS2 20
- Constructivist Learning 21
- Adult Learner Theory 23
- Types of programming environments 25
  - Text-based programming environments. 27
  - Visual programming environments. 29
  - IDEs. 32
- Microworlds 33
  - Characteristics of microworlds. 35
  - Teaching with microworlds. 36
  - Issues with microworlds. 38
  - Examples of microworlds for learning how to program. 39
- Why Alice for this study? 42
  - Adoption by universities and industry. 42

Designed for novice programmers. 44
Previous results using Alice. 45
Easy-to-use. 46
Motivating and fun to use. 47
Increased academic performance. 48
Increased retention and success. 49
Increased self-efficacy. 50
Strengths and Limitations of Alice. 51
Strengths. 51
Limitations. 52
Summary 53
Contributions of this Research 55
Chapter III. Methods and Materials 56
Research Questions 56
Participants 58
Instruments 59
Pre-test. 59
Demographic questionnaire. 59
Programming assignments. 60
Grading rubric. 61
Post-test. 62
End-of-course evaluation. 62
Time-on-task survey. 62
Final course grades. 63
Schedule of assessment instrument application. 63
Threats to Validity 64
Internal threats to validity. 65
Instrumentation. 65
Differential selection of participants. 67

Attrition. 67
External threats to validity. 67
Research Design 68
Procedures 68
Analysis 69
Demographics questionnaire. 69
Pre-test. 70
Programming assignments. 70
End-of-course evaluations. 71
Time-on-task survey. 71
Post-test. 72
Final grades. 72
Limitations and Key Assumptions 73
Summary 75
Chapter IV. Results 77
Demographics Questionnaire 77
Pre-test 78
Programming Assignment 1 79
Programming Assignment 2 81
Programming Assignment 3 83
Post-test 84
Pre-post Gain 86
Student End-of-course Evaluations 87
Time-On-Task Survey Results 88
Grade Distributions 92
Research Questions 92
Research question 1. 93
Research question 2 93
Research question 3 94

Research question 4 94
Summary 95
Chapter V. Conclusions 96
Research Overview 96
Research Questions Findings Discussion 98
Research question 1. 98
Research question 2. 100
Research question 3. 101
Research question 4. 102
Other Findings Discussion 103
General code testing deficiency. 103
Weak repetition control structures prerequisite knowledge. 105
Alice interface concerns. 106
Study population. 109
Male learners. 109
Adult learners. 110
Student end-of-course evaluations. 112
Use of rubrics. 113
General Summary 114
Recommendations 115
Appendix A. Pilot Study 119
Appendix B. Questionnaire 120
Appendix C. Pre-test and Post-test Questions 123
Appendix D. Experimental Group Programming Assignments 128
Appendix E. Control Group Programming Assignments 133
Appendix F. Course Evaluation Form (Page 1) 138
Appendix F. Course Evaluation Form (page 2) 139
Appendix G. Time-on-task Collection Form 140
Appendix H. Cross-tabulation Results for Questionnaire 141

Appendix I. Pre-Test Chi-square and *t* test Results 146
Appendix J. Project 1 Statistics 154
Appendix K. Project 2 Cross-tabulation and Chi-square statistics 155
Appendix L. Project 3 Cross-tabulation and Chi-square statistics 156
Appendix M. Post-Test Chi-square and T test results 158
Appendix N. Pre- and Post-test Analysis of Covariance Results 165
Appendix O. Student Evaluation Average Weekly Study Hours 166
Appendix P. IRB Approval 167
Appendix Q. Informed Consent Form 168
Appendix R. Programming Assignment Grading Rubric 169
References 170
Curriculum Vita 184

List of Tables

Table 1 Control and Experimental Group Activities ... 64
Table 2 Programming Assignment 1Rubric Component Counts for Control (C), Experimental (E), and Total (T) Groups ... 80
Table 3 Programming Assignment 2 Rubric Component Counts for Control (C), Experimental (E), and Total (T) Groups ... 82
Table 4 Group Statistics for Programming Assignment 2 Total Points ... 82
Table 5 Programming Assignment 3 Rubric Component Counts for Control (C), Experimental (E), and Total (T) Groups ... 83
Table 6 Group Statistics for Programming Assignment 3 Total Points ... 84
Table 7 Post-test Question 4 Counts for Control (C), Experimental (E), and Total (T) Groups ... 85
Table 8 Post-test Question 4 chi-square Tests ... 85
Table 9 Group Statistics for Pre- Post-test Gain ... 87
Table 10 Group Statistics for Overall Instructor and Course Evaluations ... 88
Table 11 Group Statistics for Programming Assignment Time-on-Task ... 89
Table 12 Programming Assignment 1 Programming Environment Comments from Control and Experimental Groups ... 90
Table 13 Grade Distribution Counts for Control (C), Experimental (E), and Total (T) Groups ... 92

Chapter I. Introduction

The purpose of this chapter is to explore the challenges computer science educators are currently facing and how this research may help mitigate some of these challenges. These challenges include high attrition and failure rates in introductory programming courses where students of all ages find learning to program very difficult. Understanding the fundamental computer programming concepts at the beginning of a degree program help a student establish self-efficacy for future courses. This chapter discusses the background of these challenges followed by subsections discussing the statement of the problem, purpose of the research, significance of the research, research overview, research questions, study limitations, and definitions of key terms.

Background

Network systems and data communications analysts, computer software engineers, computer systems analysts, and database administrators are included in the top 30 fastest-growing occupations for the years 2006 through 2016 (*Bureau of Labor Statistics*, 2008). The President's Information Technology committee report (2005) stressed the need for good software engineering practices and the importance of the development of these best practices through education. These sources suggest a strong immediate and future demand for qualified computer scientists requiring educational institutions to help students learn and prepare for these jobs.

Educational institutions help provide a workforce trained with skills to develop software that is easy-to-use and effective for multiple applications. Understanding how to write computer programs is vital for most computer-related fields but of particular importance to those who pursue careers as software engineers and computer systems analysts as they routinely write code for computer programs and applications.

For those students pursuing an undergraduate degree in computer science or a related field, several programming courses are required in a specific order (*Computer Curricula*, 2001). This sequence of programming courses often includes three courses referred to as CS0, CS1 and CS2. Although each course has specific learning objectives, the overall goal of the sequence is to help students establish a fluency in a programming language that will assist students in obtaining jobs and enhancing their careers in computer programming and software engineering.

Although each of the courses in this sequence is important in helping students become proficient in writing computer programs, CS0 is critical as foundational design concepts and control structures, allowing students to understand how to logically construct a program, are introduced. For example, within a CS0 course, students learn fundamental programming constructs such as *if/else* control structures, *for/while* control structures and functions. These control structures are used in subsequent CS1 and CS2 courses as well as many additional computer programming courses.

Despite the programming sequence present in most educational institutions, learning to program can be very difficult for beginners of all ages (Kelleher & Pausch, 2005). Beginning programmers often become overwhelmed with syntax, logic, nomenclature, and design as they struggle to understand how to build a computer program. In addition, many schools have reported high attrition and failure rates preventing students from completing their degrees in computer science. Seymour and Hewitt (1997) reported that 50 percent of undergraduate students majoring in computer science either dropped out of school or changed to another major during their freshman year. Seymour and Hewitt determined the factors most contributing to this attrition included lack or loss of interest in science, poor teaching by faculty, and feeling overwhelmed by the pace and load of courses. Although all of the factors contributing to this attrition rate are not known, higher than desired attrition rates have prompted researchers to analyze the processes, students, and other factors to better understand and mitigate these issues.

As educators study attrition, failure and learning difficulties in computer science undergraduate students, four contributing factors have been identified: 1) selecting and using control structures, 2) visualizing the steps a computer takes as it executes a program, 3) writing error-free programs, and; 4) motivating students (Areias & Mendes, 2007; Bishop-Clark, Courte & Howard, 2006; Cooper, Dann & Pausch, 2003; Moskal, Lurie & Cooper, 2004; Kelleher & Pausch, 2005; Winslow, 1996).

Cooper et al. (2003), Lahtinen, Ala-Matka and Järvinen (2005), Kelleher and Pausch (2005), and Winslow (1996) found students had difficulty in selecting and using control structures in beginning computer programming classes. The abundance of control structures (including *if/else* and *for/while* structures) with differing functionality and syntax, added to the complexity and depth of materials that computer science students must learn. Lahtinen et al. (2005) surveyed over 550 students and 34 teachers and determined understanding how to design a program to solve a certain task and organizing code into subsets were challenging for students. Soloway and Spohrer (1989) and Winslow (1996) noted students may understand the syntax and semantics of individual statements but still have difficulty composing these components into valid and fully functional programs.

Bishop-Clark et al. (2006) and Cooper et al. (2000) expressed concern that many programming environments lacked the ability for students to visualize the results and execution steps of a computer program. This lack of visualization left students with vague, text-based interfaces without the ability to debug to identify and fix errors within their code. Even though many of these environments provided the ability to report values of variables while the program runs, the results and underlying visualization of those results were often misleading for a beginning programmer (Cooper et al., 2000).

The use of graphics and visual display may assist students to better understand programming concepts and enhance learning by helping them to see results quicker through a more intuitive interface. Conway (1997) found that students preferred a visual tool that allowed selecting, dragging and then dropping control structures into their programs as opposed to having to type in each command or statement. Conway also found a well-designed interface reduced the text-entry related errors many novice programmers made within their programs.

Kelleher and Pausch (2005) suggested that programming languages usually contain confusing rules increasing the number of errors in students' code. Students often had difficulty remembering the names of commands as well as any required command-specific syntax. Students encountered steep learning curves and difficulty finding program errors while using non-visual, text-based programming environments. Bishop-Clark et al. (2006) shared this concern and added students may become frustrated attempting to create their code and eventually leave the course without really understanding the fundamental concepts. For example, many students complained about the length of time text-based environments required before they were able to produce functional code. Novice programmers became frustrated as the number of programming errors grew without seeing real progress in the functionality of the application.

Sharing the same concern over the number of programming errors and the frustration associated with those errors, Garner, Haden and Robins (2005) gathered data over a one year period for over 250 CS1 students documenting the types of programming errors they generated. After analyzing over 11,000 errors, the top two were categorized as "basic mechanics" and "stuck on program design". Basic mechanics errors included syntax related errors such as the proper use of braces, brackets, semi-colons and naming conventions. Stuck on program design errors included understanding the task and creating a correct solution to the given task.

Additional complexities associated with programming errors have been attributed to the use of math in graphical display systems and functions or sub-routines designed to help isolate programming functionality into smaller more reusable components.

Conway, Audia, Burnette, Cosgrove, and Christiansen (2000) observed students did not like to use complicated math or coordinate systems to tell how or where an image should move. Instead of using 3-dimensional x-, y-, and z-axis notations and complicated syntax to move an image from point A to point B, students preferred to use familiar terms such as *move forward two steps*.

Functions are introduced in CS0 courses but students are often confused by their use and design. Historical results from one institution showed students earned on average 20 percent fewer points on function assignments compared to assignments on *for/while* control structures (Robertson, 2007). Additional issues have been reported in the literature indicating that learning functions can be challenging. Ruehr (2008) noted how difficult the concept of functions can be for beginning programmers. Students with some math background tried to incorrectly equate math and computer functions only to be confused when attempting to print or compare functions on a computer. Xing (2008) identified several issues with students learning functional programming including properly identifying and naming functions and higher-order functions. Turning to a strongly-typed functional programming language, Xing was able to demonstrate increased understanding of functions through additional hands-on practice using an interactive programming environment.

Cheung, Ngai, Chan and Lau (2009) observed a gap in programming instruction between graphical or iconic programming environments often used in elementary schools, and the conventional, text-based programming environment appropriate for more advanced high school and university students. Cheung et al. combined the use of a graphical-based programming environment that reduced syntax issues and a text-based editor to gently expose students to the programming syntax to better prepare the students for more powerful programming environments that used both visual and text-based approaches.

Student motivation has been recognized as a concern in computer science programs as indicated by increased attrition (Seymour & Hewitt, 1997) and decreased enrollments. Not only are students leaving the program, the number of declared computer science majors has dropped. For example, the number of new computer science majors in fall 2006 was half of what it was in fall 2000 (Vesgo, 2007). Recently, the decline seems to be slowing indicating the numbers are beginning to stabilize (Markoff, 2009). Another indication that educators are concerned about this apparent lack of motivation is the proliferation of programs and activities related to increasing interest in computer related programs in general. Many universities are working to improve their program by adding game courses in order to motivate students and increase the number of students interested in pursuing careers in computer science (Garris, Ahlers, & Driskell, 2002; Kelleher et al., 2007).

Attempting to increase enrollments while filling gaps in diverse groups, educators are also reaching out to underrepresented groups such as females and minorities through summer camps and other recruiting approaches (Anewalt, 2008; Hu, 2008; Kelleher et al., 2007). Continued outreach to strengthen gender and ethnic diversity in computer related programs increases the pool of students who may become interested and motivated to pursue careers in computing.

Seymour Papert (1980) quickly recognized students struggled learning how to program and introduced the concept of a microworld. Papert created a visual environment to assist students in learning how to program. This environment was designed to help students concretize difficult programming concepts, such as learning to use selection and decision control structures, and increase motivation and enjoyment.

Through Papert's inspiration, researchers (e.g. Conway, 1997) began to develop new microworlds taking advantage of the enhanced processing and 3-dimensional graphical and visualization capabilities of modern computers. The goal of Conway's research was to increase the understanding of fundamental programming concepts and increase retention in programming classes. With their rich, visualization features, some microworlds have great potential for increasing student motivation. A beginning programming student is less likely to become frustrated if the environment provides the capability to quickly see progress while creating their programs and reduce the dependence on difficult rules of a programming language (Conway, 1997).

Although many publications exist that explore programming environments designed to help students learn how to program, there is a scarcity of quantitative research that address all of these areas. After a review of publications of several programming environments, Gross and Powers (2005) determined that research questions were frequently answered using insufficient evidence. Many times only anecdotal evidence was provided supporting the position that students preferred using a specific programming environment but details were not provided to validate the approach. Assessments of the environments were often performed by the developers of the programming environment and not from a third party. These findings suggest that additional researchers are needed to further validate hypothesis and add additional insights to the initial research.

In addition to the shortages Gross and Powers (2005) refer, much of the research using microworlds to date is on traditional-age students in the K-16 range. Adult learners have some unique characteristics including: 1) Self-concept 2) experience; 3) readiness to learn; and 4) orientation to learn (Knowles, 1970). More research is needed to determine if a sample consisting of largely adult learners will yield different results than existing studies on more traditional learners.

This research has the potential to provide educators with details on how to learn to program, the strengths and weaknesses of programming environments, and the learning characteristics of a unique, relatively under-studied population. All of these can possibly be used by educators to help meet the demand for employers who are in need of better and larger numbers of computer programmers and software engineers.

Accrediting organizations also recognize the challenges associated with teaching introductory programming and suggest a wide range of strategies is needed to help students learn. The Computer Curricula (2001) final report encourages institutions and individual faculty members to continue experimentation in this area believing that “pedagogical innovation is necessary for continued success” (*Computer Curricula,* 2001, p. 22).

Statement of the Problem

The computer science field has a critical need for qualified and well-trained computer programmers. However, traditional training methods have led to high attrition and failure rates among students in computer science programs. These issues arise, in part, due to programming environments that make it difficult for students to: select and use appropriate control structures; visualize the steps a computer takes as it executes a program; write error-free programs, and; complete the computer science curriculum. Because of these issues and the shortage of quantitative research in this area, there is a need to determine if newer programming environments have an effect on grades, success rates, and motivation to learn within introductory programming courses.

Purpose of Research

The purpose of this research is to evaluate if CS0 students who use a microworld environment demonstrate a better understanding of programming concepts than students who use a traditional C++ programming environment.

The CS0 course provides a foundation for future programming courses by preparing the student to construct simple applications using logical design processes and programming constructs. If these critical concepts are not understood and retained early, a student may have difficulties in future programming courses that build on these fundamental concepts and challenge the student to design and build larger and more complex applications.

Within a microworld, students use trial and error to explore concepts and ideas in a simplified computational environment. Students who find their assignments fun or engaging are far more likely to proceed beyond the basic requirements (Mullins, Whitfield, & Conlon, 2009). When using a traditional C++ environment, students type their code into a text editor and then compile, run and debug as needed. Carlisle, Wilson, Humphries, and Hadfield (2005) found text-based environments distracted attention from the teaching of problem solving as instructors spent valuable class time attempting to resolve syntax issues instead of teaching fundamental program design.

Significance of Research

This research will provide additional quantitative and qualitative findings related to students learning how to develop computer programs using a popular microworld programming environment.   Focusing on novice programmers from a population consisting of a large percentage of ethnic-minority, adult-learners this research will provide details about a student’s ability to use selection and repetition control structures, create and use functions, and the motivational characteristics for a relatively understudied population.

This research will also employ grading rubrics, programming assignments and other assessment activities aligned with course learning objectives designed to reveal details and challenges associated with learning how to program that may have not previously been reported. Grading rubrics provide additional details into crucial aspects of programming such as design, layout, testing, and functionality that may allow educators to better differentiate and identify specific strengths and weaknesses of programming environments. Being able to determine the major contributions from each learning environment, may provide developers of these environments a blueprint to improve existing or build new programming environments. Aligning assignments with specific learning objectives will provide insight into challenges at the course and program levels. This information could be used to assist curriculum designers and instructors focus on problem areas while deemphasizing areas where knowledge and learning appear to be more successful.

Finally, this research has the potential to partially answer the call from computing organizations and government agencies suggesting a need for more qualified computer programmers and software engineers by providing insight into specific issues associated with learning how to program. These insights could help developers improve programming environments and instructors focus on challenging programming areas allowing more students to successfully complete their degree requirements.

Students and teachers agree that even though the theory behind programming is important, students need practical experience to understand the concepts (Lahtinen et al., 2005). Additional research in microworlds as applied to learning how to program may provide instructors reasonable alternatives to today’s traditional programming environments while possibly providing more hands-on experience, motivation and success.

Research Overview

At a high level, this research will compare learning results of two sections of a CS0 course delivered in face-to-face format at a large accredited university on the east coast of the United States. The experiment is designed to determine if significant differences exist in the learning outcomes of these two groups under similar conditions with each group using a different programming environment to build and run code. The control group used a standard C++ Integrated Development Environment to build and run their code. The experimental group used a popular microworld programming environment. Each selected assessment method was aligned with one or more course specific learning objectives. With this approach it should be possible to determine if one group performs better than the other group with respect to specific course learning objectives.

Research Questions

The approach of this research will help answer several questions related to the use of a microworld to improve grades and student motivation compared to a traditional C++ programming environment. Specific questions to be answered through this research include:

1) Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on *if/else* control structure related exercises?

2) Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on *for/while* control structure related exercises?
3) Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on function related exercises?
4) Is there an increase in the time devoted to a CS0 course for students who use the Alice programming environment compared to those who use a C++ IDE?

Limitations

This research has a number of limitations. The sample population is small. Only two sections with approximately 35 students in each section will be used. In addition, the typical withdrawal and failure rate for a CS0 class is 30 percent or more during most semesters. In addition, only one semester of data was gathered. This further reduces the number of samples for analysis reducing the strength and generalization potential of the study.

There is the possibility both the instructor and researcher could introduce some biases into this study. The same instructor taught both sections of the class but he was very aware of the purpose of the study and could potentially have added some biases towards the microworld when grading or reporting results during interviews and discussions with the researcher.

Definition of Terms

This section provides definitions of several key terms related to this research and used often throughout this document.

Control structure. Basic constructs for creating a program or an algorithm including sequence, selection and repetition (Venit, 2009).

CS0. Computer programming 0. An introductory course in computer programming focusing on problem solving methods and algorithm design (Computer Curricula, 2001).

CS1. Computer programming I. The second course in the recommended introductory sequence of programming courses focusing on high-level programming languages and teaching students how to design, code, debug and document programs. (Computer Curricula, 2001).

CS2. Computer programming II. The last course in the recommended introductory sequence of programming courses focusing on data structures and more advanced programming topics. (Computer Curricula, 2001).

Function. A special type of subprogram that can be assigned a value (Venit, 2009).

Integrated Development Environment (IDE). An IDE is a programming environment that provides an easy-to-use interface to integrate the coding and execution environments (Kelleher & Pausch, 2005).

Microworld. A microworld is a graphical, visual computational environment that may help increase motivation to learn (Cooper, Dann, & Pausch, 2000; Papert, 1980). Papert (1980b) defined a microworld as “a subset of reality or a constructed reality whose structure matches that of a given cognitive mechanism so as to provide an environment where the latter can operate effectively “ (p. 204).

Repetition control structure. A loop structure containing a branch to a previous statement in a program module which results in a block of statements that can be executed many times (Venit, 2009).

Selection control structure. A decision structure where there is a branch forward at some point which causes a portion of the program to be skipped (Venit, 2009).

Text-based Programming Environment. Text-based programming environments allow users to type text and commands into a simple editor to create programs (Kelleher & Pausch, 2005).

Visual Programming Environment. A visual programming environment refers to any system that allows a user to specify a program in two or more dimensions (Myers, 1986).

Chapter II. Literature Review

The purpose of this chapter is to review the existing literature related to the use of tools and techniques that help students learn fundamental computer programming and design concepts. After providing curricula standards supporting the inclusion of programming fundamentals into computer science programs, the foundational work by constructivist theorists such as Seymour Papert and Jean Piaget will be discussed. An overview of adult learner theory and teaching practices will be presented to support the non-traditional age students present in this group. Next, a taxonomy of various programming tools will be presented. This taxonomy will define, differentiate and provide examples of simple text-based processors, visual environments, integrated development environments (IDEs), and microworlds. Since microworlds are a major focus of this research, in-depth discussions of this topic will be provided including examples of existing microworlds applied to learning computer programming, general issues associated with using microworlds and a detailed justification of the specific microworld selected as the experimental programming environment for this research. Finally, a summary of this chapter along with contributions of this research will be provided.

Programming within a Computing Curriculum

Standards help define the curricula for schools and the path of students enrolled in these schools. The Computing Curricula 2001 project was a joint undertaking of the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE) and the Association for Computing Machinery (ACM) to develop curricular guidelines and standards for undergraduate programs in computing (*Computer Curricula*, 2001). Within these guidelines, a typical undergraduate computer science student is expected to complete several programming courses in a specific sequence to demonstrate fluency in a programming language. The computer science final report from this project listed a number of knowledge focus groups including programming fundamentals, discrete structures, operating systems, programming languages, and others that are highly recommended for inclusion into computer science curricula.

Knowledge focus groups are used to build a series of courses in an institution's computing curriculum. The programming fundamentals knowledge focus group includes core topics consisting of fundamental programming constructs, algorithms and problem-solving, data structures, recursion and event-driven programming. Each of these core topic areas includes specific learning objectives and goals. For example, fundamental programming constructs include learning basic syntax and semantics of a higher-level language, variables, types, expressions and assignment, simple input and output, conditional and iterative control structures, functions and parameter passing, and structured decomposition. Each of these topics can be used by instructional designers and facilitators to build a course and a set of assessment materials closely aligned with learning objectives.

In 1978 most educational institutions used a two-course programming sequence, generally referred to as CS1 and CS2 (*Computer Curricula*, 1978) to fulfill the programming fundamentals knowledge focus group requirements. In 2001, the Computer Curricula final report strongly endorsed moving to a three-course introductory programming sequence to satisfy the programming fundamentals knowledge focus group requirements. This sequence usually includes a course in problem solving and design

(CS0), followed by a course in introductory programming (CS1) and a course in data structures (CS2). Each course in this three-course sequence builds on the materials of the previous courses preparing students for more advanced courses as well as possible employment in the computer field.

CS0.

CS0 students are expected to fulfill a number of requirements covering the topics of fundamental programming constructs and problem solving including being able to: 1) analyze and explain the behavior of simple programs; 2) modify and expand short programs that use standard conditional and iterative control; 3) design, implement, test, and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, and the definition of functions; 4) choose appropriate conditional and iteration constructs for a given programming task; 5) apply the techniques of structured (functional) decomposition to break a program into smaller pieces; 6) describe the mechanics of parameter passing; 7) create algorithms for solving simple problems; and, 8) use pseudocode or a programming language to implement, test, and debug algorithms for solving simple problems (Computer Curricula, 2001).

Programming constructs provide critical functionality supporting logical decisions and iterative processes used in computer applications as they help students understand how to make computers perform important yet basic operations such as sorting a list of words or calculating the results of mathematical equations and operations. Pseudocode remains important for a CS0 class as students learn to use an English-like language to document the important steps in solving the problem. Problem solving remains a critical skill for all computer programmers and is introduced early in a program.

CS1.

The CS1 course builds on the basic design skills provided to students in CS0 while also teaching a language-specific implementation. In a CS1 class, students increase their knowledge of programming by implementing their designs in a specific language. For example, a design used to solve a problem developed in CS0 can now be implemented, or actually coded, in a modern programming language such as Java, C++, or C#. More advanced skills, debugging strategies and complex algorithms help differentiate the CS0 from the CS1 class. Students continue to advance their skills as they are challenged to write larger, more sophisticated programs.

The choice of language to use in CS1 and other programming courses will most likely remain controversial. Java, C++ and C are criticized for being too verbose, enforcing notational overhead that has little to do with learning to write structured programs (Pears et al., 2007). Studies have found that market appeal and industry demand are often the most important factors determining language choice in computer science education (Dingle & Zander, 2000; de Raadt, Watson, & Toleman, 2004).

For CS0, the focus is on design and fundamental constructs with the programming language choice not being critical. CS1 students usually become comfortable using a programming language although additional courses and training are required to obtain proficiency due to the size and complexity of most modern programming languages. In addition to students strengthening their design skills to develop solutions to more challenging problems, CS1 students often write and test applications in a popular and current object-oriented programming language.

CS2.

The CS2 course is the final course in the recommended sequence of courses covering the programming fundamentals knowledge focus group. Students in a CS2 course use a modern programming language to understand and develop more advanced programming structures to efficiently solve challenging scientific and engineering problems. Most CS2 students become confident with a programming language and how to design and implement solutions. A CS2 student is expected to become proficient using data structures and recursion algorithms. Examples of CS2 learning outcomes from the Computer Curricula 2001 include students being able to: 1) describe how data structures are allocated and used in memory; 2) describe common applications for each data structure; 3) implement user-defined data structures in a high-level language. 4) write programs that use arrays, records, strings, linked lists, stacks, queues, and hash tables; 5) choose the appropriate data structure for modeling a given problem; 6) describe the concept of recursion and give examples of its use; 7) implement, test, and debug simple recursive functions and procedures; and, 8) determine when a recursive solution is appropriate for a problem. Meeting these expectations results in CS2 students being able to design fast, efficient solutions that are scalable to larger applications. As students continue to participate in programming courses beyond the CS2 level they acquire additional design and programming skills which help them better prepare for computer related jobs.

CS0, CS1 and CS2 courses designed in accordance with the 1978 and 2001 Computer Curricula documents will help meet the expectations of the fundamental programming knowledge focus group. After successfully completing these courses, students should be proficient in using functional decomposition to design, code, and implement the application in at least one modern programming language. Students will be able to use fundamental programming concepts and basic data structures, and design efficient algorithms as they build and test programs.

Constructivist Learning

Constructivist theory assumes knowledge is constructed as learners attempt to make meaning of their experiences (Driscoll, 1999). According to Jonassen (2002) constructivist learning approaches assist in making meaning through personal or socially constructed knowledge. Key to constructivist approaches is that knowledge is built upon the foundation provided by previous learning. With constructivism, knowledge is constructed by individuals by the combination of sensory input data with existing knowledge the learner already possesses to create new cognitive structures (Ben-Ari, 1998). Another key factor in constructivism is learning is active rather than passive. Learning should be active allowing students to construct knowledge assisted by guidance by facilitators and feedback from other students.

Compared to behaviorist, more traditional teaching approaches where learning is thought to be the result of the passive transmission of information from one individual to another, constructivist teaching philosophy allows students to test the accuracy of their current understanding (Hoover, 1996). In addition, instructors and facilitators using constructivist approaches need to provide learning environments that allow students to discover differences in their current understanding and any new experiences found within these environments. Facilitators emphasize these new experiences and foster group discussions and knowledge comparison with other students. Students within a constructivist teaching environment need time to reflect upon their experiences and how they build and enhance or modify existing knowledge.

Constructivist teachers often adapt and modify their curriculum based on their current students needs (Brooks & Brooks, 1999). In this manner, instructors can use students existing knowledge as a foundation for learning in a semester. Also, constructivist facilitators may assess differently than traditional approaches by looking beyond correct and incorrect answers to better understand their current understanding and knowledge to help students increase their knowledge.

Constructionism can be thought of as subset of Jean Piaget's (1973) constructivism. The simplest definition of constructionism includes the idea of "learning-by-making" (Papert, 1991, p. 6). Papert (1991) differentiated constructionism and constructivism as follows:

> Constructionism – the N word as opposed to the V word – shares constructivism's connotation of learning as "building knowledge structures" irrespective of the circumstances of learning. It then adds the idea that this happens especially felicitously in a context where the learner is consciously engaged in constructing a public entity, whether it's a sand castle on the beach or a theory of the universe. (p. 1)

Both theories share the idea of building knowledge structures through the exploration of an environment. Although the terms constructionism and constructivism are often interchanged, the former can be thought of as a constructivist approach to making meaning through the use of technologies (Bers, 2007). In addition, constructionism approaches often include a social component, suggesting that learning can involve a collective generation of meaning (Crotty, 1998).

Constructionism has the potential to assist designers of tools dedicated to learning computer programming. A learning environment that uses graphical or visual images as opposed to text may help students discover programming bugs and issues quickly, thereby removing some of the learning curve associated with program debugging (Bishop-Clark et al., 2006). Roussou (2004) argued that interactivity, engagement, and learning are strongly connected. A visual learning environment can promote this interactivity and engagement.

Adult Learner Theory

How adults learn is critical for this research as the median age for undergraduate students for this university is 31, with 80 percent identifying themselves as working fulltime (University data). Adults are often working professionals who select an area of study or enter a learning situation when they feel the need to acquire new knowledge or fit a particular need (Kieran-Greenbush, 1991). Taking a closer look at how teachers work with traditional-age students compared to adults will help understand the differences in these groups and assist in the design of a program more suitable for this group.

Pedagogy is defined as the art of teaching children (Scott,1987). This teaching model assumes that the student will learn what they have been told and that the teacher takes responsibility for making decisions about what will be learned and how it will be learned. Andragogy is based on a set of assumptions that describe how adults learn (Koski, Kurhila, & Pasanen, 2008). Unlike pedagogy, andragogy realizes that the lecturer does not possess all the knowledge and students are encouraged to participate in the classroom (McGrath, 2009).

Additional differences between pedagogy and andragogy have been identified in the areas of need to know, self concept, experience, readiness to learn and motivation (Scott, 1987). In terms of need to know, pedagogy learners do not need to know how what they learn will impact their lives whereas adult learners need to know why it is important for them to learn the topic prior to learning it. In terms of self-concept, pedagogy learners are dependent whereas adult learners are independent and self-directed. The role of a learner’s experience is also a differentiator. In pedagogy, learners rarely have topic-related experiences to be able to share with the class. However, adult learners are often older, more mature students that have rich experiences to share. With experiences, learners can contribute significantly to the learning of others in the class. For pedagogy learners, students become ready to learn because they need to pass an exam or an assignment that determines if they successfully complete the class. For adult learners, students become ready to learn because they need to know the concept to cope with or solve a real-world problem. Finally, motivational factors are often different between these two types of learners. Pedagogy learners are motivated by extrinsic factors such as parents, teachers and grades. Adult learners are motivated by intrinsic factors including self-esteem and quality of life.

Baker (2009) provided additional insight into the motivation of adult learners for those in the computing field. Baker suggested motivations of older software engineers to learn and retain new skills have both intrinsic factors such as recognition, feedback and pride and extrinsic factors such as pay and their desire to continue their careers. These motivational factors can help an adult learner persist and successfully complete a course. Ellis (2002) agreed that adult learners are highly motivated and goal-driven. He also suggested hands-on application of material and the incorporation of real-world examples, case studies, and work-related projects in computer education related classes support the motivation and goal-driven needs of adult learners.

Types of programming environments

Students have a number of programming environment options to consider when beginning to program. Kelleher and Pausch (2005) developed an approach to classify systems based on the programming problems the environment was attempting to mitigate. In general, these programming systems have been built for novice programmers and focus on simplifying the mechanics of programming by removing unnecessary syntax and typing, making languages more English-like, and using visible or graphical contexts so students can immediately see results of their code. These are some of the very issues referred to in chapter 1 that need to be mitigated to improve student success in computer programming.

Kelleher and Pausch (2005) defined two categories of programming environments: 1) empowering and; 2) teaching systems. Empowering systems allow students to build applications and solutions specific to their own needs. These systems allow programmers to develop code efficiently without being frustrated with difficult language syntax or interfaces. Empowering systems help to hide many of the details of the underlying language in a user-friendly interface. Empowering systems are subdivided into mechanics of programming and activities enhanced by programming. Mechanics of programming systems attempt to improve programming languages and provide alternative approaches to creating programs. These systems attempt to improve the student's interaction with the language and the environment. Integrated Development Environments (IDEs) fall into this category. An IDE provides an easy-to-use interface to integrate the coding and execution environments. In this manner, these systems allow users to build, test and debug as they experiment.

Teaching systems help students learn to program by focusing on areas that are often considered difficult for novice programmers. Areas that students consider to be difficult include selecting and using programming structures (Cooper et al., 2000; Kelleher & Pausch, 2005), visualizing the steps a computer takes as it executes the program (Bishop-Clark et al., 2006; Cooper et al., 2000; Kelleher & Pausch, 2005), programming syntax (Bishop-Clark et al.,2006; Kelleher & Pausch, 2005), and typing large amounts of code without errors (Bishop-Clark et al.,2006; Conway,1997; Conway et al., 2000). Teaching systems are further grouped by Kelleher and Pausch (2005) into systems that concentrate on the mechanics of programming and systems that provide additional learner support. Mechanics of programming systems help students understand syntax (the rules and semantics of code), how to organize the syntax, and how the computer executes the syntax. For example, a mechanics of programming system may provide a way for students to reduce the amount of code typed by providing an interface allowing the selection of the code from a group of symbols. These environments provide the correct syntax without the student having to type code, possibly reducing the amount of coding errors. Alice is an example of a programming environment that falls into this category.

In addition to Kelleher and Pausch’s (2005) taxonomy to categorize programming environments a number of general types of programming environments exist based on the method students use to enter code including text-based programming environments, visual programming environments, IDEs, and microworlds. The following sections describe each of these programming environments.

Text-based programming environments.

Text-based programming environments allow users to type text and commands into a simple editor to create programs (Kelleher & Pausch, 2005). Text-based programming environments and editors do not fall into any of the categories suggested by Kelleher and Pausch (2005) because most do not provide any features helping the student to learn how to program.

Text-based programming environments allow students to enter their code character by character, save their file, and then use programming language tools to compile and run the resulting program. Within a text editor, programmers are provided a text area where they can type in code and programming constructs. Most editors provide highlighting or color coding of certain types of code but it is usually up to the student to make sure what they type into the editor is correct. Once created in the text editor and saved as an operating system file, the programmer compiles the file to check for errors using the available language specific program tools. If errors exist, the programmer must use a text editor to debug and then compile and test until the program runs as expected. If the program has many errors, the programmer must resolve each error adding time to the project and possibly increasing frustration.

Though students often use text editors to write their programs (Cheung et al., 2009), most text editors are not designed for students to learn how to program. Many text editors provide unrestricted entry of text with minimal syntax checks making it difficult for a novice programmer to write even small applications (Pears et al., 2007). Students, using only a simple text editor, often experience frustration when editing, compiling and running a program. In most cases, students need an environment more sophisticated than a text editor to understand the fundamental constructs and processes associated with programming. However, if the environment is too complicated, students may become frustrated with learning how to use the environment (Pears et. al, 2007).

Cheung et al. (2009) found that in text-based environments, the logical thinking behind programming is introduced simultaneously with the constraints of the language syntax. This, combined with learning a specific language such as C, C++, or Java, resulted in the course focusing on the programming language syntax as opposed to learning how to solve problems and use fundamental programming constructs. Carlisle et al. (2005) ran into similar issues where learning a language within a text-based environment distracted attention from teaching problem solving skills and approaches. Instructors spent valuable class time devoted to syntax issues instead of on design and programming fundamentals. Masterson and Meyer (2001) suggested that visual programming systems can hide some of the confusing details in the background allowing the student to better understand the programming fundamentals.

Although text-based programming environments are popular and readily available as most operating systems come bundled with a tool for editing text, researchers look for additional solutions and more optimal environments in an attempt to remove some of the obstacles novice programmers face.

Visual programming environments.

A visual programming environment refers to any system that allows a user to specify a program in two or more dimensions (Myers, 1986). A dimension is a geometric measure in one direction. A visual programming environment would use objects that at a minimum have length and width measurements. These environments often have interfaces that resemble traditional programming flow charts where a set of symbols are connected to represent an application or simple program (Kahn, 1996). Unlike text editors where students enter commands to represent loops, variables and other programming structures, visual programming environments use graphics and images to symbolize these programming structures. Programmers create code by connecting these images together (Masterson & Meyer, 2001). Visual programming environments are often included in Kelleher and Pausch's (2005) taxonomy under teaching systems in the sub-category of mechanics of programming.

Visualization environments attempt to take advantage of the human visual processing potential (Gomes & Mendez, 2007). The human visual system and human visual information processing is optimized for multi-dimensional data (Myers, 1986). Visual environments enable programmers to use color to support learning (Meyer & Masterson, 2000). For example, blue could be used to represent control logic and green to represent variables. The graphical symbols and icons within a visual environment are used as building blocks in an algorithm without the constraints of a specific programming language. This allows students to focus on design and not on syntax. This higher-level description of the programming actions makes the task of programming easier even for professionals (Myers, 1986). Many visualization tools also have an execution feature allowing the algorithm to run so students may observe the results and debug, and modify the program as needed.

Compared to text-based programming environments, researchers have determined visual programming environments increase productivity, engagement and learning. McKinney (2003) concluded user-friendly interfaces that include icons, menus, and drag-and-drop features within a visual programming environment often required less instruction than their text-based predecessors and increased productivity. Fischer, Giaccardi, Ye, Sutcliffe, and Mehandjiev (2004) suggested that, as with human languages, programming language syntax must be learned from scratch, making text-based languages more challenging to learn than visual programming languages. Visual environments helped students by allowing them to become actively involved and interact with the environment. The more an environment makes a student feel they are participating in and doing actual computer programming the better the student will be prepared for more complex and challenging real-world computer programming projects. Hundhausen and Brown (2005) found activities involving visualization technology improved performance and active engagement in the course.

Masterson and Meyer (2001) suggested since some students are able to grasp programming concepts and methodologies using traditional text-based environments the fault may not be with the code itself as much as the way in which the code is presented on the computer screen. This suggests visual programming should be used in conjunction with other teaching environments including textual and IDEs to help novices learn programming.

There are many examples of visual programming environments. RAPTOR allows students to create algorithms by combining basic graphical symbols while using a flowchart based presentation of the program's control flow. RAPTOR was developed to improve student problem solving skills and reduce syntax issues. User-defined functions are supported and made visible in the flowchart representation on the computer screen. Students can run their algorithms in the environment using step-by-step or continuous play mode (Carlisle et al., 2005).

Carlisle et al., (2005) provided some preliminary results using RAPTOR and compared the results to other programming environments. Final exam analysis showed students who used RAPTOR had significantly higher scores than those that used other programming language environments for two out of the three sections. When surveyed, students preferred to express their algorithms visually as opposed to within a text-based programming environment.

BACCII++ is another example of a visual programming environment that emphasizes graphical icons. The environment was developed at Texas Tech University for teaching procedural and object-oriented programming concepts and languages (Calloni & Bagert, 1997). Experimental results comparing BACCII++ to C++ text-based programming environments showed students performed significantly better on final exams and overall course grades when they used BACCII++.

Some weaknesses and issues of visual programming environments have been identified. Visual programming environments do not lend themselves to displaying all of the possible connections and programming structures required for large complex programs (Fry, 1997; Myers, 1986). Therefore, these environments are often limited to smaller, simple programs. Petre (1995) and Myers (1986) listed additional weaknesses of visual, graphical programming including: slowness of response compared to text-based, graphics are not always intuitive or clear, and the inability to add comments to a program. Although not perfect, visual programming environments can be effective for helping novice programmers write simple programs.

IDEs.

An IDE provides an interface to integrate the coding, debugging and execution environments (Kelleher & Pausch, 2005). Programmers create code through a combination of typing code and selecting, dragging and then dropping graphical icons into a sophisticated text-editing area. Differing from other visual programming tools discussed earlier, IDEs provide access to advanced capabilities including the ability to organize multiple files into a project, automatically complete code segments, debug, and automatically generate documentation. Most popular programming languages have one or more commercially available IDEs designed for creating applications. Experienced programmers and professionals use an IDE to enhance productivity (Pears et al., 2007). Novice programmers often struggle using these more advanced tools as in addition to the programmer having to learn programming structures and a language specific syntax, they must also learn how to use a tool that provides features that may be difficult to use and understand for beginners(Cheung et al., 2009).

IDEs support visual programming languages like Microsoft’s Visual Basic, and Visual C++. Visual programming languages are really just an IDE wrapped around a specific programming language that uses visual design elements to assist programmers in efficiently creating and testing complex code. Meyer and Masterson (2000) classified these languages as a specific type of visual programming languages that use a graphical user interface (GUI) to help programmers to design and test applications.

The term IDE has also been used in reference to microworlds. Pears et al. (2007) defined two broad categories of IDEs specific for novice programmers that included programming support tools and microworlds. Programming support tools assist learners in the creation of programs within an environment that supports running of the application or code. Microworlds provide environments based on physical metaphors with program execution reflected by the visual state of objects within the environment. Microworlds may help with the gap between the purely graphical programming environments appropriate for elementary school students and the conventional, textual programming languages appropriate for more advanced high school and university students (Cheung et al., 2009).

Microworlds

Microworlds may help students learn how to program by providing graphical interfaces, interactivity and the ability for students to construct meaningful knowledge. Papert (1980b) defined a microworld as “a subset of reality or a constructed reality whose structure matches that of a given cognitive mechanism so as to provide an environment where the latter can operate effectively “ (p. 204). In computer science, this is analogous to a simplified computational environment allowing students to experiment with programming constructs. Designers of microworlds use key components of constructivism as students actively construct their knowledge through meaningful hands-on experiences. Within a microworld, students have the ability to play and experiment. Constructivist learning theorist Jean Piaget (1973) believed that playing supports learning. As students discover and play within the microworld, existing cognitive structures can be modified based on new knowledge obtained within the environment.

Researchers have provided multiple, yet similar definitions for a microworld within the construct of computer technology. Clements (1989) defined a microworld as a “playground of the mind” (p. 86). Similarly, Hogle (1995) defined microworlds as “interactive environments allowing learners to explore and manipulate the logic, rules and relationships of the modeled concept” (p. 5). diSessa (2000) defined a microworld as:

> A genre of a computational document aimed at embedding important ideas in a form that students can readily explore. The best microworlds have an easy to understand set of operations that can be used to engage tasks of value to them, and in doing so, they come to understand powerful underlying principles. (p. 47)

Microworlds help motivate students to learn programming in a fun environment rich in visual information that facilitates learning better than text-based (non-visual) systems (Dougherty, 2007). Through trial and error a student explores concepts and ideas in a simplified computational environment that may not be possible in a classroom setting. Students who find their assignments fun or engaging are more likely to proceed beyond the basic requirements (Mullins, Whitfield, & Conlon, 2009). Programming environments that excite and engage may allow students to make more meaning, build more knowledge, and actually learn more as they experiment.

Although many similar definitions of a microworld exist, for the purposes of this research, a microworld is defined as an interactive, computer-based environment allowing students to explore fundamental programming constructs using an engaging interface. With this definition, together with characteristics of microworlds, available microworlds suitable for this research can be identified.

Characteristics of microworlds.

Edwards (1995), Hogle (1995) and Reiber (2004) provided several characteristics of microworlds including: 1) supports discovery and making of meaning through interesting and motivating activities; 2) introduces domain specific knowledge to novices; 3) assists users in learning challenging difficult concepts; 4) supports movement to new ideas and concepts based on current understanding; and 5) provides feedback allowing students to learn from their mistakes.

Microworlds support Jonassen’s (2002) constructivist learning as activity theory where students make meaning through personal or socially constructed knowledge as they use and participate in activities within the environment. Similarly, a microworld supports the concept that interactivity, engagement, and learning are strongly connected (Roussou, 2004).

Microworlds help students understand concepts which on the surface may appear too difficult or abstract. Papert used microworlds to allow students to connect and understand through concrete, qualitative means (Rieber, 2004). In reference to the challenge of students learning Newton’s laws, Papert suggested microworlds can serve as “genetic stepping stones” allowing learners to move from their current understanding to new ideas and concepts (Papert, 1980b, p. 206).

Edwards (1995) suggested a microworld provided an entry to a domain for a person in a way that captures the person’s interest. Students should be motivated to enter and stay to learn within a microworld. Students are expected to manipulate the objects and features of a microworld “with the purpose of inducing or discovering their properties and the functioning of the system as a whole” (Edwards, 1995, p.144). Microworlds help students learn from their mistakes. Students should also be able to correctly interpret the feedback from the microworld. This process, known as debugging, is fundamental in computer programming. Edwards (1995) also noted students should be able to “use the objects and operations in the microworld either to create new entities or to solve problems or challenges” (p. 144).

Edwards (1995) provided a set of functional characteristics of a microworld including discovering properties of objects and overall functioning of the microworld through the manipulation of objects, interpreting feedback and debugging the system to achieve specific goals, and creating new objects to solve problems. A critical characteristic in microworlds is being able to connect new and old ideas and experiences in the spirit of constructivist learning (Riebel, 2004). This characteristic leads to the power of microworlds as learning tools (Hogle, 1995). Microworlds seek for learners to find equilibrium when confronted with discrepancies between the environment and their understanding. However, a microworld is not the only resource used in a successful learning model. An instructor and other resources and tools may serve to support student learning and modeling. (Reiber, 2004).

Teaching with microworlds.

Rieber (2004) differentiated microworlds from other educational software by the instructional model used by each. According to Reiber, microworlds are based on the underlying principles of invention, play and discovery whereas other educational software is usually based on the paradigm of explain, practice and test. Jonassen (1991) explained how the epistemology underlying microworlds is constructivism. This alignment of microworlds with constructivism is critical to understanding how it may be possible to increase motivation and learning with the use of tools that allow students to become actively engaged as they learn how to write computer programs.

Computer science education does have some documented use of constructivist approaches to teaching (Lui, Kwan, Poon, & Cheung, 2000; Hadjerrouit, 1998) but overall many institutions use traditional approaches with an occasional use of a learning environment such as a microworld (Ben-Ari, 1998). Often computer science textbooks and materials follow a serial and pre-defined approach to teaching topics such as variables and functions, flow control, lists and arrays, and objects. However, microworlds have been used within courses for students to play and discover knowledge in addition and in support of traditional learning environments (Conway, 1997; Seidman, 2009; Sykes, 2007).

To fully utilize a constructivist learning model, the instructor would include more reflection, group and socially influences, self-assessment, and more authentic tasks with specific goals driven by the student (Open World Learning; Hoover, 1996). However, microworlds can and have been applied to more traditional structured learning approaches (Open World Learning). With the incorporation of microworlds into a classroom, some constructivism is introduced allowing the discovery of knowledge and the possibility for interaction and reflection among other students.

Issues with microworlds.

Researchers have noted several issues with microworlds including restriction to simple scenarios, ill-defined learning goals, time consuming, and portability to different computer platforms and systems.

Henriksen and Kölling (2004) expressed concern that microworlds are restricted to simple scenarios. For novices who are just beginning to learn how to program, this is probably not a major issue. For students who would like to develop more complex models and programs, some frustration from this limitation may result.

Rigas, Carling and Brehmer (2002) noted students are not always aware of the learning goals associated with a microworld. Since many different solutions are possible to develop students may not be sure if their solution was correct or if they fulfilled a specific learning objective. Brouwer, Muller, and Rietdijk (2007) supported this concern when they determined only 23 percent of the faculty felt their students succeeded in creating visible results and learning using a microworld.

Kato (2006) determined there were disadvantages in applying microworlds to XML modeling. Specifically he determined several semesters and significant effort would be required to develop a complex model within a microworld environment.

Finally, Muhlhauser and Gecsei (1996) suggested that microworlds were not portable to different computing platforms thereby limiting and restricting their usefulness to a smaller set of users. The ability for a microworld to effectively run on a variety of operating systems and machines is vital for technology adoption. Strong concerns were expressed in terms of implementing applications over heterogeneous distributed platforms. Distributed platforms are designed to improve scalability by distributing the computational demands over multiple computers supporting complex applications.

Examples of microworlds for learning how to program.

Many microworlds exist that offer some potential to enhance learning in the area of computer programming. Logo, Karel the Robot, and Alice are often referenced as microworlds and used by educators. Each of these environments introduces basic programming constructs through an easy-to-use environment, where simple commands are used to control movements and other behaviors of an object. The objects used for visualization often include simple shapes like a turtle, robot, ice-skater, or a sheep.

Seymour Papert's Logo (1980) remains the inspiration for most microworlds designed and developed by researchers. The Logo environment allows students to enter commands that manipulate a turtle displayed on the computer screen. Papert used the concept of Piaget's transitional objects to help to connect what a student already knows to the new domain of geometry or math (Reiber, 2004). Logo helps students learn geometry as they issue commands to move the turtle about the screen. Students can immediately see the results of their commands while forming geometric shapes or exploring and solving other problems. Students do not have to worry about understanding a multi-dimensional display axis using complicated x- and y-axis coordinate systems. Instead, the turtle can be directed around the screen to create geometric shapes using simple, easy to understand commands such as forward, backward, left and right. Students can learn from the turtle movements and debug as needed.

Pattis (1981) developed a microworld called Karel the Robot for introducing computer programming to students. The robot simulator provides similar functionality as the Logo turtle. Karel the Robot's 2D microworld is composed of horizontal streets and vertical avenues. Students use simple commands such as move, turn right, and turn left to navigate through the world. Visualization of the graphics is limited to a 2-dimensional top-down view showing the robots path in relation to bounding walls and other obstacles within the environment.

Conway (1997) designed a 3-dimensional graphical programming environment, known as Alice, to help students learn introductory programming concepts in computer science. The goal of Alice was to make it easy for novices to develop and explore 3-dimensional computer programming environments. In Alice, 3-dimensional models of objects inhabit a virtual world where users can control their appearance and behavior through a visual interactive interface. Students are immediately able to see and interact with their animated programs. Similar to Papert's (1980) Logo, Alice users do not have to worry about complicated coordinate system. Instead of x-,y-, and z-axis coordinates, students move their objects about the computer screen using command such as forward, backward, left, right, up, and down. Alice provides an editor allowing students to write code by using a mouse to select and then drag-and-drop commands which significantly reduces the required keyboard typing and associated text-entry errors. Templates are available within Alice to assist in program construction by allowing students to generate code segments (Pears et al., 2007). Alice is an environment in which algorithmic thinking, design and basic object-oriented constructs are introduced (Lorenzen & Sattar, 2008).

Many other microworlds have been developed and used in computer science education. Henriksen and Kölling (2004) created an environment called Greenfoot for introducing object-oriented programming to beginners. Object-oriented programming has become more popular with the introduction of languages such as C++, Java, and C#. With object-oriented programming, developers define operations associated with a specific data structure. For example, a programmer may create a car object that has an operation called drive that changes the position of the car based on certain criteria. Greenfoot provides interaction, visualization, and experimentation to enhance student engagement and learning related to object-oriented programming. Greenfoot includes the ability for students to visualize the relationships between objects and their associated methods. This feature helps students understand how objects interact with one another. Included within Greenfoot is the ability to use a robot and a turtle with similar functionality as provided by Karel the Robot (Pattis, 1981) and Logo (Papert, 1980). Greenfoot also provides the ability to integrate into other learning environments and IDEs for future development.

Anderson and McLoughlin (2007) developed a 3-dimensional game-like world for designing and executing programs called C-Sheep. In this world, a programmer can modify the position and behavior of a sheep in a simulated meadow. C-Sheep supports computer visualization and animation courses. A basic set of control structures and functions are available for students to experiment and discover. Similar to Papert’s Logo (1980) C-Sheep includes commands for turning left and right, forward and backward, and up and down, removing some of the mathematical complexity associated with 3-dimensional coordinate systems. Differing from Alice, C-sheep commands are usually text-based instead of graphical or visual. However, once an object is within the world, mouse clicks and short-cut commands can be used to move it about the world.

Why Alice for this study?

As noted in the previous sections, many programming environments exist that have been used for students and professionals for developing computer programs. These range from simple text-editors to microworlds and advanced IDEs. From all of the possibilities, Alice was selected as the experimental programming environment for this research. This section discusses the process and justification for this decision.

Factors used in determining the selection of the programming environment included adoption by universities and industry, application to novice programmers to build fundamental programming skills transferrable to other programming languages, previous scientific research and findings, and strengths and weakness documented in the literature.

Adoption by universities and industry.

To determine the adoption of programming environments by universities and industry several factors and resources were used including the Tiobe community programming index, the adoption of Alice by universities, corporate and agency sponsors of Alice, and funding organizations supporting the programming environment .

The Tiobe community programming index (Tiobe, 2009) provides a monthly report of programming language adoption, popularity and use. Derived using world-wide-web search engines, the Tiobe community programming index represents a composite of programming language use by skilled engineers, educational courses and vendors. Although programming languages such as Java, C and C++ are listed as being the most popular in the April 2009 index, Alice and Logo are also listed in the top 50. In fact, besides Logo, Alice is the only microworld listed. The fact that Alice and Logo are on the list of widely used environments is a good indication that adoption has occurred beyond the initial researchers who developed the environment.

Researchers have expressed concerned that novices do not have the background to learn popular languages such as C++, C# and Java while learning about problem solving and design (Hadjerrouit, 1998; Biddle & Tempero, 1998; Close, Kopec, & Aman, 2000). In addition, unlike microworlds such as Logo and Alice, popular languages such as C++, C# and Java have not been designed for educational purposes (Pears, et al., 2007).

Literature searches also revealed many universities are using Alice in their courses. Mullins et al. (2009) noted that programming using the Alice microworld has become widely adopted for CS0 and CS1 in particular for courses that are shared in departments offering both computer science and information technology programs. Several colleges and universities have used Alice in summer camps for middle school students, including Carnegie Mellon, the Colorado School of Mines, Georgia Tech, and Westminister (Hu, 2008). This level of adoption of Alice indicates a general consensus the programming environment is stable and considered useful for novice programmers. Other environments considered for this research were rarely adopted beyond the developer’s home institution (Powers, Ecott, & Hirschfield, 2007).

Other indicators of Alice’s adoption include the number of grants, sponsors, and publications and presentations at popular computer science education conferences such as ACM’s Special Interest Group on Computer Science Education (SIGCSE) and Innovation and Technology in Computer Science Education (ITiCSE). Alice has had the support and funding to develop and maintain a user-friendly environment. Carnegie Mellon, the developer’s of Alice, has received multiple grants from the National Science

Foundation and Defense Advanced Research Projects Agency in support of Alice. In addition, Alice is sponsored by companies including Microsoft, Sun Microsystems, Electronic Arts, Hearst Foundation, Google, Disney, and Hyperion (Alice, 2009). With so many sponsors from so many organizations, many of which who are actively involved in information technology, the credibility, user-base, and future support of Alice appears promising making Alice a viable learning platform for researchers.

Alice has also advanced to the point where a number of textbooks have been written and adopted by universities for their curricula. Alice publications are current and have specific titles associated with learning how to program.

Designed for novice programmers.

Alice was developed for introductory students with no previous programming experience. This makes it useful for university students who may not be computer science majors who are taking CS0 or an introductory programming course to satisfy another major (Cooper et al., 2003). The demographics for the target audience are discussed in chapter 3, however; it should be noted that many of the students in this research are true beginners with little or no programming experience that can potentially benefit from the use of Alice. Other environments may not have been developed for the same target audience. For example, Greenfoot was developed to introduce students to objects and their behavior early in their studies. Although no Java or other object-oriented language is required in Greenfoot, the interface assumes some knowledge of objects which may cause issues for many novice programmers. It assumes baseline knowledge of classes, methods and other object-oriented terminology that many novices may not have been exposed.

Commercially available IDEs may be too complex for beginning students. They are packed with features and sophisticated debugging capabilities. For introductory courses, the advantages of professional IDEs may be outweighed by their complexity. With an IDE, students have to spend large amounts of time learning the tool (Pears et al., 2007).

Alice also provides a reasonable transition to other programming languages as it introduces students to the fundamental programming constructs and provides a gentle introduction to the use of objects. Cooper et al. (2003) observed students that used Alice developed not only a strong sense of design and incremental program construction approach using programming constructs but also a contextualization of objects, classes and object-oriented programming. These skills transfer over to programming languages such as C++, Java and C# that are popular languages for industry and universities (Tiobe, 2009).

Previous results using Alice.

With the popularity of the Alice, many researchers have published results of their studies in computer science and education related publications. Several common challenges associated with learning how to program were noted as justification for using Alice including 1) selecting and using programming structures (Cooper et al., 2000; Kelleher & Pausch,2005); 2) visualizing the steps a computer takes as it executes the program (Bishop-Clark et al. ,2006; Cooper et al., 2000; Kelleher & Pausch, 2005); 3) programming syntax (Bishop-Clark et al., 2006; Kelleher & Pausch, 2005) and; 4) aversion to typing (Bishop-Clark et al., 2006; Conway,1997; Conway et al., 2000). Each of these issues may be mitigated by the use of Alice.

Researchers who have used Alice have noted several attractive features of the programming environment including 1) easy-to-use; 2) motivating and fun to use; 3) increases academic performance; 4) increases retention and success; and 5) increases self-efficacy. The following sub-sections group related research that aligns closely to each of these features.

Easy-to-use.

The 3D graphical programming environment of Alice allows students to select, drag and then drop programming structures for use in their programs providing visualization of the steps the computer takes in executing a program. This significantly reduces syntax errors and the amount of typing required by students because syntax is automatically created as the selected programming structure is dropped to the work area. For example, to make an object named Bird print the word "Hello" to the screen, the Bird's "*say method*" is selected and dropped into the work area. This results in the appropriate code being added without the student having to type or introduce any syntax errors. In Alice, the following text would be generated for this scenario:

```
Bird say Hello
```

The Alice environment reduces some of the math complexity associated with visual programming environments such as Microsoft's Visual C++ IDE that require 3-D math coordinates using x, y and z coordinates to provide object positions. Alice uses simple Logo-style commands including forward, backward, right, left, up and down to provide object positions. Students that have weaker Math skills are more likely to be successful using simple commands as opposed to programming using 3-D coordinate systems (Cooper et al., 2000; Moskal et al., 2004).

Motivating and fun to use.

According to Papert (1980), microworlds helped motivate students to take up programming by providing an enjoyable experience. Doughtery (2007) suggested visual information rich environments facilitated learning better than text-based systems. Researchers who have worked with Alice discovered a high level of student interest and involvement (Cooper et al., 2000; Dougherty,2007; Rodger, 2002). Visual programming environments introduced programming constructs in an appealing way which aided student understanding (Anewalt, 2008).

Sykes (2007) found students who used the Alice programming environment self-reported spending up to four times as much time as control groups. Although increased time could be related to increased frustration, more time-on-task could be an indication of a motivating programming environment and may be beneficial for learning. Students who used Alice as their programming environment were found to enjoy programming (Bishop-Clark et al., 2006; Bishop-Clark, et al., 2007). When comparing Alice to text-based environments for design and problem solving in a CS0 course, Dougherty (2007) found students enjoyed creating and putting significant time into their worlds. Students were reported to be more engaged and motivated to design and inclined to continue into the next programming course (CS1).

Rodger (2002) compared different microworlds to determine which environment was the most appealing for students who worked in pairs while learning to program. Using a class evaluation questionnaire, comparisons of Alice with Karel++ (an updated version of Karel the robot that uses C++ as the programming language), and Starlogo (a variant of Logo) showed Alice was the clear class favorite with 100 percent reporting they liked the unit. Students indicated it was the easiest to use with the most interactive programming interface and included the clearest tutorial.

Anewalt (2008) showed students in a CS0 class had a positive reaction to Alice and the activity-based style of the course. A survey provided five weeks into the term revealed the favorite element of the course so far was programming with Alice. All of the students (n=21) stated they would recommended the course to a friend.

Rodger et al. (2009) and Adams (2007) used Alice to help motivate middle school students to become interested in computer science and technology fields. During summer camps for middle school students were very engaged with Alice and were always asking for more free time to work on their own worlds.

Increased academic performance.

Cooper et al. (2003) found students who used the Alice environment often outperformed control groups who used a text-based environments. Although the number of students in the study was small (n=21), students of similar background who used the Alice environment in CS0 performed better in CS1 than those who were not exposed to Alice.

Microworlds, in particular Alice, offer features and outcomes relevant to this research study. Cooper et al. (2003) demonstrated how students who built their programs using Alice were better designers resulting in higher quality computer programs. Students using the Alice environment favored design. Those students who created good designs prior to coding usually produced higher quality programs. Students who used Alice also adopted an incremental, step-wise approach to constructing their programs. The Alice environment promoted building programs in small steps while students visually observed the results. They also found students performed better in subsequent classes such as CS1 and persisted further into more advanced classes. Moskal et al. (2004) found similar results in terms of better performance for students who used Alice.

Alice also introduces students to object-oriented programming through the use of internal objects allowing for student's interpretation, analysis and construction of knowledge (Mullins et al., 2009). This becomes important as follow-on courses such as CS1 and CS2 usually use an object-oriented approached. Alice provides a gentle introduction as it is considered object-based rather than object-oriented. Alice uses objects, properties, methods, but does not include more challenging object-oriented concepts such as inheritance and polymorphism.

Studying students enrolled in CS1 classes, Sykes (2007) showed those in an experimental group that used the Alice environment outperformed students in control groups by scoring ten to twenty percent higher on post-tests. These results were found to be statistically significant.

Increased retention and success.

Students using Alice as the programming environment concurrently within a CS1 course or in a CS0 course had higher retention rates than the control group that did not use Alice (Moskal et al., 2004). Students who were identified as being high-risk with low math and computer science skills and used the Alice environment showed 70 percent higher retention. 91 percent of students who used Alice in CS0 went on to CS2 class compared to only 10 percent in the control group.

Mullins et al. (2009) compared student success rates of students who used Alice (n=414) environment and those that used C++(n=735). Alice resulted in a four percent increase in the number of students that passed and a four percent decrease in withdrawals. It was also noted that Alice increased the popularity of the course for students who were not majoring in computer-related programs. Those majoring in computing showed a five percent increase in successfully passing the class and an eleven percent drop in withdrawal. These results suggest Alice may mitigate the learning curve and issues related to student retention.

Increased self-efficacy.

Another indicator supporting Alice as a good selection for this study is the increase in self-efficacy and confidence in programming following a course that included Alice as the programming environment (Adams, 2007; Bishop-Clark et al., 2006; Bishop-Clark et al.,2007; Moskal et al., 2004). Undergraduate students enrolled in an introductory computer programming course (CS1), had more confidence in programming after participating in a one-week session of Alice programming (Bishop-Clark et al., 2006). Bishop-Clark et al. (2007) showed students were more confident in programming after their experience using Alice during a short 2.5 week introductory computer programming course (CS1). In this study, a five question survey was completed by students before and after the course to measure confidence in programming. The questions measured attitude associated with their programming confidence and included questions such as “I have a lot of self-confidence when it comes to programming” and “I am no good at programming”. In reflective essays, 90 percent of students reported an increased knowledge and understanding of the programming process.

Powers et al. (2007) and Adams (2007) found the impact of Alice on student perceptions and attitudes towards programming seemed largely positive. Alice was successful at increasing their weaker students' self confidence in their programming abilities. However, Powers et al., found the confidence applied primarily to programming within the Alice environment and this confidence did not continue when the student moved from Alice to a textual programming language.

Cooper et al. (2003) observed students who used Alice demonstrated an ability to collaborate. Alice students were observed working on their own objects and characters individually and later having the confidence and ability to combine them to build virtual worlds and animations in group projects.

Strengths and Limitations of Alice.

Strengths.

A number of researchers have documented strengths of using the Alice programming environment for their introductory programming courses. One of the most attractive strengths of Alice is that introductory students can achieve immediate success (Goldweber, Bergin, Lister, & McNally, 2006). This strength is often associated with the built-in features of Alice such as the drag-and-drop interface that essentially eliminates syntax errors (Brown, 2008). Brown also found through informal midterm and final course evaluation questionnaires students recognized and appreciated the ability of the Alice programming environment to allow students to develop code quickly with minimal errors.

Alice's interface also supports graphical programs that manipulate 3-D objects in a 3-D virtual world (Powers et al., 2007). The ability to add interesting animations and visualization of programs often leads to increased student motivation and effort which in turn may lead to increased student success and self-efficacy. Visualization of common programming techniques like looping makes the concepts easier to master (Brown, 2008).

Alice has been mentioned as being very effective in a CS0 environment where students need the time to understand fundamental programming constructs that support future programming courses. (Adams, 2007; Brown, 2008; Dougherty, 2007; Kelleher et al., 2007).

Limitations.

Alice is not without its limitations. Data structures are limited to one-dimensional arrays (Goldmeyer, Bergin, Lister, & McNally, 2006). This is probably fine for most novice students, but not being able to demonstrate or experiment with more complex data structures may limit some students. Many instructors are concerned that some object-oriented techniques are not available such as inheritance and polymorphism. For programmers who go on to CS1 and CS2 object-oriented techniques become quite important. Powers et al. (2007) suggested improvement is needed in Alice to increase confidence during the transition from Alice to object-oriented programming IDEs. They observed the initial shift to object-oriented programming was easier, but as students began exploring more advanced object-oriented concepts difficulty arose.

Another concern about Alice is its lack of real-world applications (Mullins et al., 2009). When creating assignments instructors must be careful that the task can be accomplished with the Alice environment. Powers et al. (2007) also observed students who learned to use Alice did not pay attention to syntax. This could be problematic when students move onto professional IDEs and other possible non-graphical programming environments.

Finally, Klassen (2006) suggested adult learners did not like the Alice programming environment as much as a traditional C++ environment. Klassen concluded adult learners did not need the motivational capabilities of the Alice environment as they were already motivated to learn. The adult learners in Klassen's study also expressed concern about the user interface as it seemed more appropriate for much younger students.

Summary

The Computer Curriculum 2001 ACM report established the need for computing courses to help students build a fluency in computer programming. CS0 is the first and foundational course in a sequence of programming related courses designed to provide students with fundamental programming constructs and problem solving skills. Because of the importance of this course for students who decide to major in computer science or a related field, the focus of this research study will take place within the CS0 course.

As students transition from typing in code in simple text-editors to more advanced integrated development environments (IDEs), programming constructs remain important. In addition, what programming environments students use to learn these programming constructs may also be key (Kelleher & Pausch, 2005) and is a focus of this research study. Programming environments that are easy to use and contain features that motivate and engage students to learn may provide some benefit to novice programmers.

Theoretical foundations supporting this study include constructivist learning and adult learner theories. Teaching models based on constructivist learning principles focus on approaches allowing students to construct knowledge by combining new input data with existing knowledge to form new meaning and understanding. In constructivist learning the student takes a very active role in their learning with the teacher facilitating this process. Adult learner characteristics become crucial for this study because of the large number of adult learners enrolled at the university participating in this study. This implies students may already be ready to learn, focused on specific learning needs and motivated. Knowing the characteristics of adult learners and the underlying definitions and approaches to constructivist learning environments will help in the design and interpretation of the results of this study.

The statement of the problem summarized in chapter 1stated there was a critical need for qualified and well-trained computer programmers and current approaches to teaching computer programming have led to high attrition and failure rates. Although many factors can attribute these issues, researchers have shown significant contributors to the problems include difficulty in selecting and using control structures, visualizing the steps a computer takes as it executes a program, and writing error-free programs. In addition, instructors and facilitators have been challenged motivating students to complete the curriculum. The program environment and approach to this research were selected to help mitigate these problems. Alice was selected as the programming environment due to its popularity as indicated by its high rate of adoption and use by universities and its potential to mitigate issues associated with learning to program. Alice is designed on constructivist principles allowing students to construct new knowledge as they experiment and play in the microworld. Research suggests that Alice is easy-to-use, increases motivation, academic performance, retention and self-efficacy. All of these features and possible benefits have the possibility of increasing the number of students successfully completing the program which may begin to reduce the shortage of computer programmers needed in today's job market.

Contributions of this Research

Although an impressive set of research is available that supports Alice to enhance skills and capabilities for novice programmers, more work is needed to determine how learning is affected in the areas of control structures, functions, and time-on-task. This research delves deeper into the learning potential Alice offers by using assignments carefully aligned with course learning objectives and assessment tools such as grading rubrics to determine specific programming skills that may be enhanced using Alice as the programming environment.

The focus for this research will be on the students enrolled in CS0 classes as this course provides the foundation for all additional programming courses. Students within a CS0 course learn fundamental problem solving skills and how to apply specific programming constructs such as *if/else* control structures, *for/while* control structures and functions. These constructs continue to be used as students move on to CS1, CS2 and other related computer science courses. This study will also focus on novice programmers from a relatively understudied population consisting of a large percentage of ethnic-minority, adult learners.

Chapter III. Methods and Materials

This chapter describes the research questions, participants, instruments, and threats to the validity of this study. The overall research design, procedures and analysis techniques used to answer the research questions are also discussed. Finally, the limitations and key assumptions associated with this study are described followed by a chapter summary.

Although the literature review supports using the Alice programming environment to improve academic performance, motivation, and retention in programming courses, more work remains to confirm its impact on specific learning objectives for a CS0 class and to measure differences between a traditional C++ learning environment and Alice. Quantitative analysis providing additional statistical validation is needed (Gross & Powers, 2005).

A CS0 course provides a foundation in programming fundamentals supporting future programming classes. Students in a CS0 course are provided opportunities to design programs using step-wise refinement and build problem-solving skills. Determining which code components to use can be challenging for beginning programmers (Cooper et al., 2000; Kelleher & Pausch, 2005). These skills must be taught early in the curriculum to support more complex programming tasks in the future to improve success rates in computer-related disciplines to support the growing need for programmers by industry and the government (*Bureau of Labor Statistics*, 2008). Because of these critical needs, a number of research questions have been formed.

Research Questions

This study was designed to answer the following four research questions:

1) Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on *if/else* control structure related exercises?
2) Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on *for/while* control structure related exercises?
3) Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on function-related exercises?
4) Is there an increase in the time devoted to a CS0 course for students who use the Alice programming environment compared to those who use a C++ IDE?

The first three research questions will provide detailed comparisons of Alice and a traditional C++ IDE in terms of specific CS0 learning objectives. Although an IDE does have some advanced editing and debugging features, most novice programming students use an IDE in a non-graphical manner (Pears et al., 2007). For these students, code is often entered directly into an editor. Previous research has shown that tools and approaches that help a student better understand how to build a program will help the student be more confident in their current programming courses and more prepared for future programming courses (Adams, 2007; Bishop-Clark et al., 2006; Moskal et al., 2004). Focusing on the course learning objectives may provide some insight into strengths and weaknesses of programming environments for specific programming constructs and tasks allowing designers and developers of programming environments to improve their products.

The final research question addresses the potential motivation factors associated with the use of an interactive visual environment for a CS0 class. Sykes (2007), Dougherty (2007) and Rodger (2002) indicated students who used Alice demonstrated an increased motivation and ease-of-use resulting in more dedicated study and time-on-task compared to traditional text-based approaches. A student who enjoys and is satisfied with a course is more likely to stay in the class and be more motivated to learn. This research question will help determine if microworlds inspire students to spend more time learning the materials which in turn may impact student success.

Participants

The students participating in this research were enrolled in one of two face-to-face sections of an undergraduate CS0 class taught by the same instructor. Approximately 35 students were enrolled in each section. The instructor was an experienced collegiate professor who has taught CS0 for more than ten years at this university making him very familiar with the learning objectives, goals, materials and assessment methods of the course.

The university is a State university on the east coast of the United States accredited by the Middle States Accreditation Agency. The median age for undergraduate students for this university is 31, with 80 percent identifying themselves as working fulltime (University data). A typical student at this university is an adult learner taking more than one course and working fulltime (CITE Demographic Data, 2009). Historical retention data over the period from 2003 to 2008 for the CS0 face-to-face classes showed 61 percent of the students successfully completed this course with a D or higher, 17 percent failed the course and 22 percent withdrew (CITE Grade Distribution Data, 2008).

Instruments

The instruments used for this research included a pre-test, demographic questionnaire, three programming assignments, a post-test, end-of-course evaluations, and final course grades.

Pre-test.

The pre-test was used to detect pre-existing differences between the two groups that could impact the experimental results. It contained multiple choice and short answer technical questions aligned with course learning objectives in the specific areas of *if/else* control structures, *for/while* control structures and functions. The pre-test consisted of thirteen questions pulled from an exam database developed and reviewed by a team of faculty peers who had taught CS0 in previous semesters and by external program reviewers. The test included four *if/else* questions (questions 1, 3, 6 and 7), three *for* loop questions (questions 2, 9 and 11), two *while* loop questions (questions 8 and 10) and four function questions (questions 4, 5, 12 and 13).

Students were provided approximately 30 minutes to complete the pre-test and the demographics questionnaire on the first day of class. Appendix C shows the pre-test for this research study.

Demographic questionnaire.

The demographic questionnaire included survey questions gathering age, gender, and ethnicity information. Additional questions on the survey included the student’s current major and questions related to the number of previous math and computer programming courses students had taken previously. These last questions were included based on the findings of Moskal et al. (2004) who discovered success rates differed for students who were identified as having weak math and problem solving skills.

The questionnaire was also given to the students on the first day of class. Appendix B shows the questionnaire used for this research.

Programming assignments.

The programming assignments were designed to assess comprehension of control structures and functions. In the first programming assignment, students created a program that returned the letter grade based on the course final numerical value. For example, if a user entered a numeric value of 95, the program should return a letter grade of an “A”. The assignment allows the students to demonstrate their understanding of *if/else* control structures and properly create and test code representative of the assignment requirements.

The second programming assignment challenged the students to create an application using *for/while* control structures to calculate the greatest common divisor for two positive numbers. The steps to solve the problem were provided to the students in the form of pseudocode. If successfully implemented, when a user entered the numbers 18 and 21, the number 3 would be returned. Additional test cases were provided to the students.

The third programming assignment required students to create two simple functions for calculating the area of a trapezoid and the perimeter of a rectangle. Students were provided the mathematical formulas for these functions and they had to write and test the code to properly implement the formulas.

Appendix D shows each of the programming assignments for the experimental (Alice environment) group. Appendix E shows each of the programming assignments for the control (C++ environment) group. The only difference in the assignments between the control and experimental groups was the programming environment where the students were asked to develop and test the code.

Grading rubric.

A rubric was used for the instructor to grade each programming assignment. A rubric provides fair and consistent grading for each student. Grading rubrics have traditionally been applied to many disciplines (Andrade, 2005). Becker (2003) was one of the first to apply a rubric to grade computer programs students submitted. She used specific grading areas including design, style and functionality. Each area had three levels of rating including unacceptable, meets standards and exceeds standards. The institution conducting this research established grading rubrics for each of its CS1 and CS2 programming courses over five years ago. This rubric was refined by faculty peers and adopted for use in the CS0 for this research. Appendix R shows this rubric as it was applied for assessment of each of the programming assignments.

Post-test.

The post-test consisted of a course comprehensive exam that included the same 13 questions found on the pre-test as well as other questions related to CS0 learning objectives that were not part of this specific research project.

The post-test was administered on the last day of class. Students were provided approximately 180 minutes to complete the entire post-test. The questions on the post-test related to this research are shown in Appendix C.

End-of-course evaluation.

The course evaluations used were the standard end-of-course student evaluations many universities use to gather anonymous feedback about the course and instructor. The evaluations included information concerning overall course and instructor satisfaction, and the self-reported hours per week of study. Appendix F shows the end-of-course evaluation form used for this research.

Time-on-task survey.

In conjunction with research question four, each student was asked to estimate the number of hours they used during the development of each programming assignment. In addition, students were asked to answer questions regarding the use of the programming environment and if any additional resources were used in support of completing the assignment. This data may provide a more accurate estimate for time-on-task related to each programming assignment as opposed to the weekly estimates on the end-of-course evaluation form. The additional questions helped to capture issues and strengths of the programming environment along with other tools that might help further explain success in the course. Appendix G shows the time-on-task collection forms used for this research.

Final course grades.

The final course grades were used to assist in answering all of the research questions at a high level. Student performance in terms of their final grade may indicate a trend and is worth evaluating to identify additional differences in the programming environments from an overall course perspective.

Schedule of assessment instrument application.

Table 1 shows the weekly schedule of events for both the control and experimental groups that occurred during the semester. Activities show the delineation of events between the two sections and when specific instruments were applied. For example, in week 6, the control group downloaded and installed the C++ IDE environment and created their first C++ program whereas the experimental group downloaded and installed the Alice environment and created their first Alice program. Beginning in week 7, each group had identical readings in the textbook and course modules but read documents specific to their programming environment. These documents provided each group how to use their programming environment to implement control and selection statements in C++ or Alice. The preparation documents were similar in nature in that they included code samples and explanations for students to produce a functional program.

The programming assignments were identical for each group. The only difference between the groups was the programming environment that was required. For the control group, C++ was utilized whereas the experimental group used Alice. Each group was allowed the same amount of time for preparing the programming assignments.

Table 1

Control and Experimental Group Activities

Week	Group Activities	
	Control	Experimental
1	Pre-test and Questionnaire	Pre-test and Questionnaire
6	Download and install C++	Download and install Alice
	create your first C++ program	create your first Alice program
7	Read C++ structure and selection documents	Read Alice structure and selection documents
	Submit Programming assignment 1	Submit Programming assignment 1
8	Read C++ repetition loop document	Read Alice repetition loop document
	Submit Programming assignment 2	Submit Programming assignment 2
11	Read C++ functions document	Read Alice functions document
12	Submit Programming assignment 3	Submit Programming assignment 3
15	Post-test and course evaluation	Post-test and course evaluation

Threats to Validity

Gay and Airasian (2003) determine an experiment to be valid if the results obtained are due to manipulation of the independent variables and if they are generalizable beyond the current participants. Specifically, internal threats to validity are threats other than those that can be attributed to the independent variables whereas external threats to validity are threats that prevent the results from being applied to other groups and settings.

It is challenging to control for both internal and external validity as a highly controlled experiment with maximum internal validity may not generalize to other settings (Gay & Airasian, 2003). The following sub-sections discuss possible threats to internal and external validity for this experiment and how each threat has been mitigated.

Internal threats to validity.

The threats to internal validity believed to be present in this study fall into the general categories defined by Gay and Airasian (2003) of instrumentation, differential selection of participants, and attrition. Instrumentation threats occur due to possibly unreliable or inconsistent measuring device. Differential selection of participants threats occur when predetermined groups are part of the study that could potentially have very different characteristics from one another. Attrition threats are caused by students who drop out of the study and do not participate in many of the activities and events of the research study.

Instrumentation.

There were a number of assessment instruments used in this research study including the pre- and post-tests, and three programming assignments. To reduce the threats to validity for the pre- and post-tests a number of actions took place. To ensure the questions were valid and appropriate for the course learning objective the exam questions were extracted from a common exam database. This database has been in place for several years for the CS0 course at this institution. To be included as a question in the database, each question must be aligned with one or more course learning objectives. The exam database is reviewed each semester by faculty peers, the CS0 course chair, and the academic program directors to ensure the quality of questions available is acceptable and that each question is properly aligned with at least one course learning objectives. The exam database is also placed on our secure administrative website for review and comments by overseas divisions of the university, and external program reviewers. Overseas faculty contributed to the database by providing additional questions and using questions for their own courses in Europe and Asia. External program reviewers are given visitor access to the administrative site and may comment and discuss the process and the quality of the exam questions.

The programming assignments were developed with a similar process to the pre- and post-test questions. A group of faculty peers, who have previously taught the course, designed the assignments based on specific course learning objectives. Each project is refined over time to make them current and in-line with any changing program and course objectives. Contribution and comments from overseas divisions and external academic program reviewers are also used to shape the content and improve validity of the programming assignments.

The review process includes faculty meetings dedicated to discussing statistical results for each question and programming assignment. At the end of each meeting recommendations are made for modifying the exam database, the programming assignments and the course study materials to help best prepare students for successful completion of the course.

Cronbach’s alpha statistical analysis was also performed for the pre- and post-test results to ensure test reliability.

The grading rubric used to provide consistent grading for each of the programming assignments had been refined from use in several semesters at the university and was based on the rubric by Becker (2003). The rubric was reviewed internally by faculty who had taught the CS0 course and other programming courses. The rubric was reviewed externally by members of the Java educator’s group formed out of the JavaOne developer’s conference.

Differential selection of participants.

Participants registered for a specific section of the CS0 based on their personal decision and their availability. The researcher had no control of placement of students into specific sections. To ensure the two classes were equivalent and to reduce this threat to validity, a pre-test and demographics questionnaire was administered to each group. These instruments will be used to document any significant differences between the groups.

Attrition.

Historical retention results for this course showed approximately showed 61 percent of the students successfully completed this course with a D or higher, 17 percent failed the course and 22 percent withdrew (CITE Grade Distribution Data, 2008). This implies approximately 40 percent of the students will likely not complete all of the assignments for this course. Although the threat to external validity will increase, to reduce this internal threat to validity, only participants who completed all of the assessment instruments will be included in the data analysis.

External threats to validity.

The predominant external threat to validity for this study was the small sample size and the unrepresentative nature of the sample. The sample size was less than 70. After attrition has been taken into account the sample size was less than 50. In addition, the demographic composition of this small sample was composed primarily of adult-learners from ethnic minority origins. These factors limit the result generalization to a small subset of adult, minority learners. Future study replication at other universities would help mitigate this threat to validity.

Research Design

The study is based on a quasi-experimental design as participants were not truly randomly selected and assigned to a section. Instead, participants registered for a section and a coin was tossed to randomly assign which section would receive the experimental treatment.

The independent variables included the use of the C++ or Alice programming environment for the control and experimental groups respectively. Dependent variables measured included the question and grade results from the pre- and post-tests, demographics questionnaire, programming assignments, time-on-task survey, final course grades, and the end-of-course evaluations.

Procedures

All students in one of the face-to-face sections were assigned to use the Alice programming environment. The students in the remaining face-to-face section served as the control group where all of their programming assignments were to be implemented using a traditional IDE programming environment (Microsoft Visual C++). The institution where this research was conducted has used the Microsoft Visual C++ IDE programming environment for over six years.

Both sections used the same instructor and textbook and met once a week for approximately three hours during a 15-week semester. Programming materials, examples, and instructions were provided to students on how to use C++ or Alice environments based on the section in which they were enrolled. In this manner, students were provided similar preparation materials to reduce the learning curve for each environment allowing students to concentrate on their assignments.

Analysis

Each of the instruments was analyzed using statistical techniques to assist in answering one or more of the research questions. Chi-square analysis and *t* tests were used on the data from most of the instruments to look for relationships between variables and the experimental or control groups. Chi-square analysis was used when categorical variables were evaluated whereas *t* test analysis was selected when continuous variables needed to be evaluated.

Demographics questionnaire.

Data from the questionnaire were entered into an Excel spreadsheet for each of the sections. Data fields included a unique student identification number, student group (control or experimental), and categorical results from each of the seven questions. The data were entered once and then double checked for any possible entry errors. Variable names were included at the top of the spreadsheet. The final spreadsheet was saved as a comma-delimited file and imported into SPSS for analysis. Once imported into SPSS, labels and categorical values were created for each of the seven questions and the grouping variable. Analysis performed was a cross-tabulation on each of seven questions by the group variable with the chi-square statistics and percentages reported for the categorical outcomes.

Similarly, data were entered into spreadsheets, double-checked for accuracy, imported into SPSS and labeled as appropriate for each of the remaining data collection instruments.

Pre-test.

The pre-test data consisted of thirteen technical programming questions aligned with the research question. The thirteen questions were also aligned with specific learning objectives of the course including learning how to use *if/else* control structures, *for/while* control structures and functions. Summing each of the related questions provided a total score, and subscales corresponding to three of the course learning objectives (*if/else*, *for/while*, and functions).

The analysis for the pre-test data included cross-tabulations for each of the thirteen questions against the student group variable with chi-square statistics and column percentages calculated. A *t* test analysis was performed on the total score and the subscales.

Programming assignments.

Each of the programming assignments used the same rubric to assign points. Layout, design, test, and functionality were included as major components of the rubric with possible scores of 1, 2 or 3 representing “does not meet expectations”,” meets expectations” and “exceed expectations”, respectively. The maximum rubric score for each programming assignment was 12 points. The spreadsheet created for each programming assignment included a unique student number, group (control or experimental) and values from the rubric for the layout, design, test, functionality and total points. After the spreadsheet was prepared and verified for accuracy, data were imported into SPSS for analysis. Similar to the pre-test, all programming assignment analysis included cross-tabulations for each of the four individual rubric components questions with chi-square statistics and column percentages calculated with *t* test analysis performed on the total score.

End-of-course evaluations.

The student end-of-course evaluation data were used to determine how much time students were studying in the course. Data from the student end-of-course evaluation surveys were provided in spreadsheet form from the institutions research group for each of the two classes. Data extracted were used to determine if there were differences in reported time spent on class studies, and overall course and instructor satisfaction. Overall instructor satisfaction was determined by calculating the mean of instructor satisfaction related items [items 5-11].Similarly, overall course satisfaction was determined by calculating the mean of the course satisfaction related items [items 12-31].

Data extracted from the student end-of-course evaluation forms were entered into a spreadsheet containing the student group (control or experimental), self-reported weekly study time, average instructor satisfaction, and average course satisfaction. After importing the file into SPSS, a cross-tabulation with chi-square statistics and column percentages was performed on the categorical study time variable. A *t* test was performed on the average instructor and course satisfaction data.

Time-on-task survey.

Students were also asked to submit a time-on-task survey associated with each project assignment throughout the semester. These data were entered into a spreadsheet and included student identification, group, and time-on-task. The spreadsheet was then imported into SPSS and a *t* test was performed. The time- on-task surveys also contained some responses that were more qualitative in nature including use of additional tools or resources and comments about the process. These results were reviewed for trends and patterns using visual inspection.

Post-test.

The post-test data included the same thirteen questions as the pre-test. Therefore, analysis was identical to that performed on the pre-test including cross-tabulations for each of the thirteen questions against the student group variable with chi-square statistics and column percentages calculated. A *t* test analysis was performed on the total scores and subscale variables.

In order to compare the programming environments for learning gain, a pre-post test analysis was performed. To determine the gain for each group, the post-test scores total points and subscales were subtracted from the corresponding pre-test scores. The results of these differences were imported into SPSS and a *t* test was performed. For completeness, a univariate analysis was performed to compare post-test results for each group while accounting for the pre-test results. Both of these approaches test the same hypothesis but from a different perspective.

Final grades.

The final course grades were entered into a spreadsheet containing student identification number, group and final grade. Once imported into SPSS, cross-tabulations for the final grade variable with chi-square statistics and column percentages were calculated.

Finally, to determine overall pre- and post-test reliability, Cronbach's alpha was calculated on each of the thirteen test questions.

Limitations and Key Assumptions

Some initial limitations to this study were listed in the introduction chapter and included a small population, instructor and researcher biases, and the duration of the study. The total number of students at the beginning of this study was 71. At the end of the semester, approximately 30 percent of the students had withdrawn or stopped participating in the class. The total number of students participating in the study was further limited by restricting the analysis to only those students who completed and submitted each of the assessment instruments. The final number of students participating was 40.

The instructor and researchers are always possible sources of biases and therefore limitations within any research study. These potential biases were reduced by the introduction of programming assignment grading rubrics and a dissertation oversight committee. Rubrics helped to reduce variations of grading between the instructor and grading practices. A second or even third reviewer/grader for each assignment and assessment activity would have eliminated most of this possible bias; however, these additional resources were unavailable. The dissertation committee and advisor provided overarching guidance and served as an additional deterrent from possible researcher bias.

Extending into multiple semesters would have added more students and strength to the study. However; this would also have added more variables such as additional instructors, term lengths and student demographics for consideration that might have been more challenging to control.

Some additional limitations became apparent during the analysis of the data. Even though the questions on the pre- and post-test along with the programming assignments were prepared by a team of instructors, the questions may not fully represent each learning objective. The alignment of each question or programming assignment with specific course learning objectives was used to better identify specific problem areas in learning how write computer programs. However, additional questions and a different variety of questions could have been used to better represent each course learning objective. Selecting the appropriate number of assessment activities to demonstrate mastery or even understanding of a course learning objective may need more consideration.

With the exception of the end-of-course evaluations, all assessment instruments had a student identifier allowing tracking trends, comments and results throughout the semester for each student. However, since the end-of-course evaluation data were anonymous it was not possible to align these results with other findings. In addition, students who dropped the course prior to the administration of the course evaluation, or who opted to not participate in the course evaluation survey were missing from this dataset. The use of the time-on-task survey mitigated some of this limitation but many students opted to submit detailed comments or information on their time-on-task surveys.

One final limitation is related to the teaching style used for this research. The traditional approach used by the instructor for teaching the control class was also used for the experimental class. Both sections were taught in a traditional, behaviorist style. The advantage to this approach is that both sections did use the same teaching style and paradigm. There was no additional training required and this approach is used by most computer science programs (Ben-Ari, 1998). The disadvantage to this approach is there might have been a mismatch between the selection of the constructivist learning environment of Alice and the teaching style used within the classroom. Another study is needed that would attempt to use a constructivist teaching style in addition to Alice may have yielded different results and revealed additional strengths of the Alice programming environment.

It was assumed that both sections were similar in demographic and technical background. This is a key assumption and will be shown to be true in the next chapter.

Summary

Traditional approaches to teaching introductory programming concepts have suffered high attrition rates with students struggling in specific areas such as selecting and using control structures, visualizing programming steps, producing error-free code and motivation. Several research questions were developed to address these problems centered on comparing traditional and microworld programming environments to determine if increased grades and motivation resulted from the use of microworlds.

The approach to this research compared two face-to-face CS0 sections, taught by the same experienced instructor using Alice as the programming environment for the experimental group and Microsoft Visual C++ IDE for the control group. Programming materials, examples, and instructions were provided to students on how to use the programming environments based on the section in which they were enrolled. The instruments used for this research included a pre-test, demographic questionnaire, three programming assignments, a post-test, end-of-course evaluations, and final course grades.

Each of the instruments was analyzed using different statistical techniques to assist in answering one or more of the research questions. Chi-square and *t* test analysis were used on the data from most of the instruments to look for relationships between variables and the groups. Chi-square tests were used when categorical variables were evaluated whereas *t* tests were used when continuous variables needed to be evaluated.

Although a number of limitations have been identified including sample size, use of only one grader, and teaching style, the research design and processes helped mitigate these limitations and potential threats to validity by employing grading rubrics, aligning programming assignments and other assessment activities with course learning objectives and working closely with the instructor and outside reviewers in an attempt to reveal details and challenges associated with learning how to program that may have not been previously reported on a population of adult learners.

Chapter IV. Results

This chapter provides results for each of the analysis techniques listed in the Methods and Materials chapter. The first sub-sections of this chapter provide results for each of the statistical analysis conducted on the demographics questionnaire, pre-test, three programming assignments, end-of-course student evaluations, time-on-task survey, post-test and grade distribution data. The final sub-section summarizes the results as they relate to the four research questions.

With the exception of the anonymous end-of-course evaluations, where it was not possible to align student identifiers with other activities, and the time-on-task surveys, where many students did not complete the surveys, a total of 40 students participated in all remaining assessment activities. This resulted in N=40 for most of the analysis conducted on the instruments for this research study.

Demographics Questionnaire

The demographics questionnaire consisted of seven questions providing age, gender, major area of study, ethnicity, semester course workload, number of previous math courses and number of previous computer programming courses for each student. This data provided a baseline comparison between the groups in terms of demographic make-up and previous experience in math and computer science to determine if any significant differences existed between the groups that could possibly account for disparities in the study results.

There were no significant differences found between the control and experimental groups in terms of gender, age, ethnicity, declared major, number of courses currently enrolled, number of previous mathematic courses and number of previous computer courses.

The cross-tabulation results for each of the questionnaire variables are shown in Appendix H. The chi-square statistics for all these questions had values greater than .05 indicating no association between these variables and the groups. The chi-square results for each questionnaire item are also listed in Appendix H.

Pre-test

The pre-test contained questions assessing the prerequisite knowledge of *if/else* control statements, *for/while* control structures and functions. Data were collected including the total score for all of the questions on the pre-test and sub-scales for each of the questions grouped by *if/else* control structures, *for/while* control structures, and functions. The pre-test established a baseline of prerequisite programming knowledge for each group so that any differences discovered on the post-test could be attributed to activities occurring within the semester.

The chi-square statistics for each of the 13 pre-test questions indicated there were no significant differences between the control and experimental groups. There were no significant differences found between the control and experimental groups in the subscale scores for the *if/else* control structures, *for/while* control structures, functions, or total points earned as calculated from the *t* test results. The reliability for all of the pre-test questions was calculated using Cronbach’s alpha (α= 0.862).

Appendix I lists the chi-square and *t* test results from the pre-test analysis.

Programming Assignment 1

Programming Assignment 1 assessed a student’s ability to create a simple program that used *if/else* control statements in their code. The instructor used a rubric (see table 1) to grade each assignment and reported results based on four area including layout, design, test data, and functionality and the total points. For each of these areas, grading criteria was based on the student fulfilling certain expectations with descriptors including “does not meet expectations”, “meets expectations”, and “exceeds expectations”.

As shown in Table 2, a larger percentage of the experimental group exceeded expectations for the layout component compared to the control group. However, for the design and functionality components, the control group had a larger percentage of students who exceeded expectations. For students in the experimental group, Alice allowed most students to exceed the layout expectation as the environment properly formats the code without any additional effort by the student.

Table 2

Programming Assignment 1Rubric Component Counts for Control (C), Experimental (E), and Total (T) Groups

Grading	Count			% within Group		
	C	E	T	C	E	T
			Layout Component			
Does Not	0	0	0	0.0%	0.0%	0.0%
Meets	7	0	7	41.2%	0.0%	17.5%
Exceeds	10	23	33	58.8%	100.0%	82.5%
Total	17	23	40	100.0%	100.0%	100.0%
			Design Component			
Does Not	0	4	4	0.0%	17.4%	10.0%
Meets	0	5	5	0.0%	21.7%	12.5%
Exceeds	17	14	31	100.0%	60.9%	77.5%
Total	17	23	40	100.0%	100.0%	100.0%
			Functionality Component			
Does Not	0	1	1	0.0%	4.3%	2.5%
Meets	0	7	7	0.0%	30.4%	17.5%
Exceeds	17	15	32	100.0%	65.2%	80.0%
Total	17	23	40	100.0%	100.0%	100.0%

Control and experimental groups were found to be different on the distribution of outcomes for the layout (χ^2 (1) = 11.48 ρ <.05), design (χ^2 (2) = 8.58 ρ <.05) and functionality (χ^2 (2) = 7.39 ρ <.05) grading components. The experimental group performed better on the *if/else* control structures assignment than the control group for the layout grading component whereas, the control group performed better on both the design and experimental grading component than the experimental group.

There were no significant differences found between the control and experimental groups for the test data grading component or the total points earned for this assignment. Most students in both the experimental and the control group did not meet the expectations for the test data grading component.

Appendix J shows the chi-square results for the test data component for programming assignment 1 and the *t* test results for the total points earned for the control and experimental groups.

Programming Assignment 2

Programming assignment 2 assessed a student's ability to create a simple program that used repetition control structures such as *for* or *while* loop statements. This assignment also built on the knowledge programming assignment 1 provided in that some use of *if/else* control statements was required.

For this assignment the instructor also used a rubric (see Appendix R) to grade each assignment reporting results based on four areas including layout, design, test data, and functionality. For each of these areas, grading criteria was based on the student fulfilling certain expectations with descriptors including "does not meet expectations", "meets expectations", and "exceeds expectations".

As shown in Table 3 a larger percentage of the experimental group exceeded expectations for the layout and testing components of the rubric than the control group.

Table 3

Programming Assignment 2 Rubric Component Counts for Control (C), Experimental (E), and Total (T) Groups

Grading	Count			% within Group		
	C	E	T	C	E	T
			Layout Component			
Does Not	1	0	1	5.9%	0.0%	2.5%
Meets	12	0	12	70.6%	0.0%	30.0%
Exceeds	4	23	27	23.5%	100.0%	67.5%
Total	17	23	40	100.0%	100.0%	100.0%
			Test Component			
Does Not	17	17	34	100.0%	73.9%	85.0%
Meets	0	0	0	0.0%	0.0%	0.0%
Exceeds	0	6	6	0.0%	26.1%	15.0%
Total	17	23	40	100.0%	100.0%	100.0%

Control and experimental groups were found to be different on the distribution of outcomes for layout (χ^2 (2) = 26.06 ρ <.05), test data (χ^2 (1) = 5.22 ρ <.05) and the total points (t (38) = -3.55, ρ=.001) grading components. The experimental group performed better on the *for/while* control structures assignment than the control group for the layout, test data, and total points grading components.

Table 4 shows the group statistics for total points earned for programming assignment 2.

Table 4

Group Statistics for Programming Assignment 2 Total Points

Variable	M	SD	SE
	Control Group (n=17)		
Total Points	8.94	.966	.234
	Experimental Group (n=23)		
Total Points	10.26	1.287	.268

There were no significant differences found between the control and experimental groups for the design or the functionality grading components for programming assignment 2.

Appendix K shows the results from the design and functionality grading components that did not yield significant differences between the control and experimental groups for programming assignment 2.

Programming Assignment 3

Programming assignment 3 assessed a student's ability to create a simple program that used functions. As in programming assignment 1 and programming assignment 2, the instructor also used a rubric (see Appendix R) to grade each assignment and reported results based on four areas including layout, design, test data, and functionality. For each of these areas, grading criteria was based on the student fulfilling certain expectations with descriptors including "does not meet expectations", "meets expectations", and "exceeds expectations".

As shown in Table 5 a larger percentage of the experimental group exceeded expectations for the layout component of the rubric than the control group.

Table 5

Programming Assignment 3 Rubric Component Counts for Control (C), Experimental (E), and Total (T) Groups

Grading	Count			% within Group		
	C	E	T	C	E	T
			Layout Component			
Does Not	3	0	3	17.6%	0.0%	7.5%
Meets	10	0	10	58.8%	0.0%	25.0%
Exceeds	4	23	27	23.5%	100.0%	67.5%
Total	17	23	40	100.0%	100.0%	100.0%

Control and experimental groups were found to be different on the distribution of outcomes for the layout (χ^2 (2) = 26.06, ρ <.05) and the total points (t (38) = -2.41, ρ=.021) grading components. The experimental group performed better on the functions assignment than the control group for the layout grading component and the total points earned.

Table 6 shows the group statistics for total points earned for programming assignment 3.

Table 6

Group Statistics for Programming Assignment 3 Total Points

Variable	M	SD	SE
	Control Group (n=17)		
Total Points	9.12	1.495	.363
	Experimental Group (n=23)		
Total Points	10.00	.798	.166

There were no significant differences found between the control and experimental groups for the design, test, or the functionality grading components for programming assignment 3.

Appendix L shows the analysis results including the chi-square statistics for the remaining design, test and functionality grading components for programming assignment 3 for the control and experimental groups.

Post-test

The post-test contained the same questions as the pre-test for the *if/else* control structures, *for/while* control structures and function related questions. Data were collected that included a total score for all of the questions on the post-test and sub-scales for each of the questions grouped by *if/else* control structures, *for/while* control structures, and functions. The post-test was used to determine learning gain at the end of the semester compared to the prerequisite programming knowledge established at the beginning of the course using the pre-test.

With the exception of one of the function-related questions (question 4), no significant differences were found between the control and experimental groups for the individual post-test questions. For question 4, a larger percentage of students in the experimental group earned zero points compared to the experimental group. Table 7 shows the cross-tabulation results for this question followed by the chi-square statistics in Table 8.

Table 7

Post-test Question 4 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	2	16	18	11.8%	69.6%	45.0%
1	13	2	15	76.5%	8.7%	37.5%
2	0	0	0	0.0%	0.0%	0.0%
3	2	5	7	11.8%	21.7%	17.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table 8

Post-test Question 4 chi-square Tests

Method	Value	*df*	*p*
Pearson chi-square	19.786*	2	.000
Likelihood Ratio	21.834*	2	.000

Note: **p* < .05

Control and experimental groups were found to be different on the distribution of outcomes for question 4 (χ^2 (2) = 19.79, ρ <.05) grading results. The control group performed better on this question than the experimental group.

The *t* test results for the post-test summation and total points variables showed no significant differences between the control and experimental groups.

The reliability for all of the post-test questions was calculated using Cronbach's alpha (α= 0.839). Appendix M lists the remaining chi-square and *t* test results from the post-test analysis.

Pre-post Gain

To determine the gain in learning for each group, the post-test scores total points and subscales were subtracted from the corresponding pre-test scores. The subscales included the sum for each of the related questions on the topics of *if/else*, *for*, and *while* control structures and functions. The *for* and *while* control structure questions were summed to derive another variable that represented all repetition control structure related questions. Measuring gain in this manner provided a detailed look at learning that took place within the course for each major learning objective in a CS0 course and a comparison between the two groups to determine if one group demonstrated better performance in the course.

The gains calculated for each summation variable between the post- and pre-test are shown in Table 9. With the exception the *if/else*-related questions, students in the control group had a slightly larger gain than the experimental group. However, none of these differences were found to be statistically significant.

Table 9

Group Statistics for Pre- Post-test Gain

Variable	M	SD	SE
Control Group (n=17)			
if/else sum	7.47	5.625	1.364
for sum	3.41	2.320	.563
while sum	2.76	1.985	.481
rep sum	6.18	3.302	.801
functions sum	6.47	3.727	.904
Total Points	20.12	9.867	2.393
Experimental Group (n=23)			
if/else sum	9.17	5.140	1.072
for sum	3.30	2.976	.621
while sum	2.30	2.401	.501
rep sum	5.61	3.940	.821
functions sum	4.83	3.774	.787
Total Points	19.61	9.272	1.933

As an additional perspective from the post- pre-test gain analysis, the pre-post analysis of covariance yielded similar results to the gain analysis for each of the total and subscale scores. There were no significant difference found between the control and experimental groups for the post-test results while adjusting for the pre-test results. Appendix N provides the resulting analysis of covariance data.

Student End-of-course Evaluations

The student end-of-course evaluations question related to the average number of hours spent on the course provided four different categories for students to choose: 0-5, 6-10, 11-15 and >15 hours per week. Although the experimental group had more students than the control indicating they studied 6-10 hours per week, there were no significant differences between the groups. Appendix O shows the cross-tabulation for each category and group for the average number of study hours for the class along with the chi-square statistical analysis.

When comparing the overall instructor and course evaluations results, the control group showed higher average values for both the instructor and overall course values. The group statistics for the overall instructor and course evaluations are shown in Table 10.

Table 10

Group Statistics for Overall Instructor and Course Evaluations

Variable	M	SD	SE
Control Group (n=19)			
Instructor	4.2100	.66241	.15197
Course	3.7953	.66240	.15196
Experimental Group (n=25)			
Instructor	3.2816	.99882	.19976
Course	3.1916	.99382	.19876

Overall, students in the experimental group were less satisfied with the instructor (t (42) = 3.50, ρ=.001) and the course (t (42) = 2.29, ρ=.027).

Time-On-Task Survey Results

Most students elected to not complete the time-on-task survey form. Table 11 shows the group statistics for each programming assignment for the students who completed this survey. No significant differences were found between the control and experimental groups for the average number of hours students reported working on a programming assignment.

Table 11

Group Statistics for Programming Assignment Time-on-Task

Programming Assignment	N	M	SD	SE
		Control Group		
1	5	3.000	1.9685	.8803
2	5	3.200	2.0187	.9028
3	2	3.500	0.7071	0.5000
		Experimental Group		
1	9	3.000	2.6926	.8975
2	7	2.500	1.2583	.4756
3	8	3.313	3.0815	1.0895

The time-on-task survey included two additional questions (see appendix G) related to the overall experience on the programming assignment and use of tools or additional resources to complete the programming assignment. Although the response rate was low to the time-on-task survey in general, several comments were captured that may provide some insight into some of the findings previously presented. A spreadsheet was used to input the text gathered. A manual review of this data was conducted to extract trends and patterns within the data. Table 12 provides several examples of the types of comments that were captured related to the use of each programming environment for programming assignment 1.

Table 12

Programming Assignment 1 Programming Environment Comments from Control and Experimental Groups

Control Group	Experimental Group
"It was a simple and straightforward exercise."	"The programming environment is okay because it is animated."
"Visual C++ programming is great."	"It was different from using UNIX but because we covered how to use Alice thoroughly in class it wasn't too hard."
"I tried including a while loop but I was missing something and I was unable to figure it out."	"My experience was challenging. I could not make sense of the text output to determine whether my world output was accurate."
"First I had to understand the syntax for C++. Then I start entering the code. It took me couple of steps before I could get a hang of the language. After I get familiar with the language it became easy and before you know it I was done."	"I hated learning the Alice environment; it would have taken me 10 minutes to write the code in C++. I feel like I learned to ride my bike and took the training wheels off."
"Well, since this was my first programming assignment I felt lost at first but then slowly I picked up. I think as the semester progresses and we do more assignments, I should get better."	"I think teachers should give a better overview of the program (Alice) before they assign any projects. I had never used Alice before and was unsure of how to use it."
	"The main problem that I had was not being able to see all of the arithmetic operator options available when defining values for variables (score>=)."

For programming assignment 2, the comments related to both programming environments tended to have a more positive flavor. For example, the control group provided comments such as: "The programming environment was really easy."; "The programming environment is quite straightforward and simple to use.", and "It was easy.''. Positive comments from the experimental group included: "It was a good experience."; "It was not anything difficult to complete."; and "Using Alice in this project helped me gain a better understanding of the role that flow control plays in programming.” .

Some usability issues and frustration continued with students in the experimental group who used Alice for programming assignment 2 as indicated by comments such as: “I don’t really like Alice because of its many limitations and the whole visual aspect of it.”; and, “This was a bit tougher to figure out because I had to discover how to make Alice display integers through the objects.”

The frustration with the Alice environment continued into programming assignment 3 with students entering comments including: “Not good, not intuitive.”; and, “My experience with this assignment was harder than the others. I struggled with geometry and creating projects in Alice. ”. However, the remaining students who participated in the survey provided favorable comments including: “I made the monkey swing.”; “It was comfortable getting this Assignment done.”; “I am getting more comfortable programming in Alice. As a result I was able to enhance the program beyond its basic requirements to make it a little more interesting.”; and, “It was an easy experience. ”.

For all programming assignments, the students in the control and experimental groups listed several tools and resources that were used in addition to the programming environments. The tools and resources listed included Google web searches, course materials such as programming guides, course modules, online conferences, and discussions with the professor and other classmates. For programming assignments 2 and 3 some students mentioned using online math calculators to help verify the output of the programs. For example, “I used an online GCD calculator to make sure that the results that I got from my program where mathematically correct.”; and “The only additional tool I used for this project was the calc.exe program in Windows. This tool helped me to check my arithmetic and ensure that formulas in my functions had been programmed correctly to get the proper output.”

Grade Distributions

More students failed or withdrew from the class in the control group compared to the experimental group. However, the differences were not found to be statistically significant. Grouping the data into pass (students who earned a D or better) and fail (students who earned an F, FN, or W) categories showed 66.7 percent of the experimental group passed the course compared to only 45.7 percent of the control group. This difference was not found to be significant (χ^2 (1) = 3.17, ρ =.075). The grade distribution cross-tabulation is shown in table 13.

Table 13

Grade Distribution Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
A	1	3	4	2.9%	8.3%	5.6%
B	9	12	21	25.7%	33.3%	29.6%
C	4	7	11	11.4%	19.4%	15.5%
D	2	2	4	5.7%	5.6%	5.6%
F	8	4	12	22.9%	11.1%	16.9%
FN	3	4	7	8.6%	11.1%	9.9%
W	8	4	12	22.9%	11.1%	16.9%
Total	35	36	71	100.0%	100.0%	100.0%

Research Questions

Based on the results shown in this chapter, answers can now be formed for each of the four research questions.

Research question 1.

For the first research question: Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on *if/else* control structure related exercises?;   the answer is no.

This answer was determined by using the results of the post-test, the first programming assignment total points and sub-scales for the layout, design, test data and functionality grading components and the course final grades. The students in the experimental group consistently exceeded expectations on the first programming assignment in the area of layout. However, there were no significant differences found between the control and experimental groups in the post-test, total points, test data sub-scale or the final course grades. The control group performed better in the design and functionality sub-scales of the first assignment then the experimental group. Because the experimental group did not clearly outperform the control group in a majority of the assessment areas, there was not sufficient evidence to answer this research question positively.

Research question 2

For the second research question: Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on *for/while* control structure related exercises?;   the answer is also no.

The post-test scores showed no significant differences between the two groups in *for/while* control structures performance. Students who used the Alice environment did outperform the control group in the areas of layout, test data, and total points on this programming assignment. However, there were no significant differences found between the two groups for the sub-scales of design and functionality or for the final course grades. Since the experimental group did not clearly outperform the control group for the majority of the assessment instruments, there was not sufficient evidence to support a positive answer to the second research question either.

Research question 3

For the third research question: Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on function related exercises?; the answer is no.

The post-test scores showed no significant differences between the two groups in function performance. Similar to the results found in the first two programming assignment, students in the experimental group consistently exceeded expectations on the third programming assignment in the area of layout. For this project the total points also favored the experimental group. However, there were no differences found for the remaining sub-scales of design, test and functionality for the third programming assignment or for the course final grades. Insufficient evidence was found to support a positive answer to the third research question.

Research question 4

For the final research question: Is there an increase in the time devoted to a CS0 course for students who use the Alice programming environment compared to those who use a C++ IDE?, the answer is also no.

Using the time-on-task surveys for each programming assignment and the end-of-course evaluation, no evidence surfaced to indicate students in the Alice group spent more time learning how to program by using the Alice tool. The numbers of hours estimated by each group for the programming assignments and average weekly course study time were essentially the same.

Summary

Insufficient evidence was found to support positive responses to each of the four research questions. Some significant differences were found in the layout grading components for each of the programming assignments indicating students who used the Alice programming environment did perform better when it came to the visual appeal and formatting of their code. However, the majority of the findings suggested that Alice and C++ programming environments yield similar performance for fulfilling CS0 course learning objectives related to if/else control structures, for/while control structures, and functions, and the amount of time spent on developing code.

Some frustration was revealed from the experimental group as indicated in comments on the time-on-task surveys and the results of end-of-course evaluations. Students who used the Alice programming environment reported they were less satisfied with the course and the instructor than the control group. These results suggest that a closer look into the environment to determine what aspects caused this frustration and what could be done to mitigate these concerns.

Chapter V. Conclusions

The purpose of this chapter is to discuss the use of Alice in a computer programming environment. The study results and implications for future research will be discussed. This chapter contains five main sections covering the topics of research overview, research questions discussion, other findings discussion, general summary, and recommendations. The research overview provides a summary of the issues students face when they learn to program, the importance of overcoming these obstacles, and how Alice, a microworld programming environment, may help students learn how to write computer programs. Within the research questions discussion section, findings and details related to each research questions along with their relationships to previously published research and specific implications are included. The others findings section provides additional results not directly related to the research question but believed to be significant and possibly useful for other researchers. Finally, the recommendations section describes future research to continue this study and contribute to the improvement of learning environments and approaches associated with teaching introductory programming.

Research Overview

Computer programming and computer science remain viable and important areas of study with a strong demand from industry and government for personnel skilled in these areas *(Bureau of Labor Statistics*, 2008). It is critical that colleges and universities prepare students to pursue these fields. However; traditional approaches have led to substantial attrition and difficulty as students attempt to learn fundamental programming principles. Kelleher and Pausch (2005) found beginning programmers often became overwhelmed with syntax, logic, nomenclature, and design making learning to program very difficult for beginners.

Microworlds offer some potential to help students learn by providing graphical interfaces that improve interactivity and the ability for students to construct knowledge. Designers of microworlds use key components of constructionism as students actively construct their knowledge through meaningful hands-on experiences. Within a microworld, students have the ability to play and experiment.

This study focused on the first course in the programming sequence, CS0, as it provides foundational concepts supporting future needs of computer science students. CS0 is designed for students to learn essential programming fundamentals including *if/else* control structures, *for/while* control structures and functions.

This research aimed to evaluate if entry-level programming students who used a microworld demonstrated a better understanding of fundamental programming concepts than students who used a traditional C++ programming environment. To support this goal, an approach was designed to compare two face-to-face CS0 sections, taught by the same experienced instructor using Alice as the programming environment for one section and the Microsoft Visual C++ IDE for the other section. Approximately 35 students were enrolled in each section.

Programming materials, examples, and instructions were provided to students on how to use the programming environments based on the section in which they were enrolled. The instruments used for this research included a pre-test, demographic questionnaire, three programming assignments, a post-test, course evaluations, and final course grades.

Each of the instruments was analyzed using different statistical techniques to assist in answering one or more of the research questions. Chi-square and *t* test analysis were used on the data from most of the instruments to look for relationships between variables and the groups. Chi-square tests were used when categorical variables were evaluated whereas *t* tests were used when continuous variables needed to be evaluated.

This research employed grading rubrics, programming assignments and other assessment activities aligned with course learning objectives designed to reveal details and challenges associated with learning how to program that may have not been previously reported on a population that included a majority of adult learners.

Research Questions Findings Discussion

This section discusses the major findings discovered during this study associated with the four research questions, along with the relationship to other research studies and possible implications.

Research question 1.

For the first research question: Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on *if/else* control structure related exercises?;   the answer was no.

There were no significant differences found between the control and experimental groups using the post-test assessment activity, the *if/else* programming assignment total points, or the course final grades. The students in the experimental group consistently exceeded expectations on the first programming assignment in the area of layout and outperformed the control group for this programming component area. However, because the experimental group did not clearly outperform the control group in a majority of the assessment areas, there was not sufficient evidence to answer this research question positively.

These results imply that students who use the Alice programming for *if/else* control structures perform at least as good as students who use a traditional C++ programming environment. In addition, since a larger percentage of the experimental group exceeded expectations for the layout grading component compared to the control group for *if/else* control structures, students who use the Alice programming environment have the potential to develop code that is better formatted and more visually appealing than those who use the C++ environment. The proper formatting of code is critical in software engineering disciplines as code would be easier to read and possibly easier to maintain. Code that is easier to read is also easier to debug when students are trying to find problems within their code. Students who struggle creating well-formatted code in a traditional programming environment could use Alice to help them produce visually appealing code.

These results help validate findings by other researchers who suggested Alice reduced programming syntax errors (Bishop-Clark et al., 2006; Kelleher & Pausch, 2005) and a student’s aversion to typing (Bishop-Clark et al., 2006; Conway,1997; Conway et al., 2000). With the Alice programming environment, students can select, and then drag, and drop programming structures into the coding area resulting in the code always being properly formatted eliminating layout and code style formatting issues. For the group that used the C++ programming environment, students had to type and format each line of code making sure they effectively used white space and braces to improve code readability and style.

It is reasonable that the Alice group would consistently outperform the C++ group as the Alice programming environment, by its very design, eliminates formatting errors and mistakes. At the end of the semester, students in the C++ group were not able to match the performance demonstrated by the Alice group for the layout grading component as 50 to 75 percent in the control group still earned below the "exceeds expectations" level. Using Alice in addition to a C++ environment may help students develop code that is easier to debug and maintain reducing some of the frustration introductory programmers experience.

Research question 2.

For the second research question: Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on *for/while* control structure related exercises?;   the answer was also no.

There were no significant differences found between the control and experimental groups using the post-test assessment activity or the course final grades. The students in the experimental group consistently exceeded expectations on the second programming assignment in the areas of layout, test data and total points for the *for/while* programming assignment. As with research question 1, there was not enough evidence to suggest Alice outperforms C++ for students using *for/while* control structures.

These results imply that students who use the Alice programming for *for/while* control structures perform at least as good as students who use a traditional C++ programming environment. Similar to research question 1, since a larger percentage of the experimental group exceeded expectations for the layout grading component compared to the control group for *for/while* control structures, students who use the Alice programming environment have the potential to develop code that is better formatted and more visually appealing than those who use the C++ environment. Students who struggle creating well-formatted code in a traditional programming environment could use Alice to help them produce visually appealing code.

These results help validate the findings of other researchers who suggested Alice reduced programming syntax errors (Bishop-Clark et al., 2006; Kelleher & Pausch, 2005) and a student's aversion to typing (Bishop-Clark et al., 2006; Conway,1997; Conway et al., 2000).

Research question 3.

For the third research question: Is there an increase in grades for CS0 students who use the Alice programming environment compared to those who use a C++ IDE on functions?; the answer was also no.

There were no significant differences found between the control and experimental groups using the post-test assessment activity or the course final grades. The students in the experimental group consistently exceeded expectations on the third programming assignment in the areas of layout and total points for the functions programming assignment. As with research questions 1 and 2, there was not enough evidence to suggest Alice outperforms C++ for students using functions.

These results imply that students who use the Alice programming for functions perform at least as good as students who use a traditional C++ programming environment. A clear theme emerged through the results of the first three research question results: students who struggle creating well-formatted code in a traditional programming environment could use Alice to help them produce visually appealing code.

Research question 4.

For the final research question: Is there an increase in the time devoted to a CS0 course for students who use the Alice programming environment compared to those who use a C++ IDE?, the answer was also no.

Using the time-on-task surveys for each programming assignment and the end-of-course evaluation, no significant differences were found between the control and experimental groups for the number of hours students reported working on each programming assignment on the time-on-task surveys or on the end-of-course evaluation forms.

This implies that a programming environment, even if it is promoted as being more motivating, may not result in increased study time for the students. Students, particularly adult-learners, may not have large amounts of time to learn how to use a programming environment. Additional materials with step-by-step guidelines for using the environment may be needed to help optimize the time students put into their programming assignments.

These results differ from other researchers results. For example, Sykes (2007) found that students in the Alice group reported spending up to four times as much time as the control groups on their assignments. Dougherty (2007) found students in a CS0 course enjoyed creating and put significantly more time into their worlds than students who used a text-based programming environment. The differences between the present study and the findings of Dougherty and Sykes could partially be explained by taking a closer look at each of these studies. The findings of both Sykes and Dougherty were based on qualitative, not quantitative surveys. Also, the context of Sykes and Dougherty was related to the motivation of using a fun, yet, at times frustrating environment. Their students explained even though the Alice programming environment was frustrating to use, it was also fun to use thereby increasing student motivation and time commitments within the environment.

Even though there were no differences found between the control and experimental groups in the current research in terms of time-on-task, some qualitative comments gathered did suggest some frustration resulted from using the Alice programming environment. Comments such as “My experience was challenging. I could not make sense of the text output to determine whether my world output was accurate”; and “I hated learning the Alice environment; it would have taken me 10 minutes to write the code in C++. ”, suggested that some increased time resulting from frustration using the Alice programming environment may have been present.

Other Findings Discussion

There were a number of other findings resulting from this research study that may not be considered major but do have some implications on the field of computer programming and computer science. The other findings have been grouped into the following areas: general code testing deficiency, weak repetition control structures prerequisite knowledge, Alice interface concerns, study population, student end-of-course evaluations and the use of rubrics. The following sub-sections describe the findings, related research, and implications for each of these areas.

General code testing deficiency.

For most programming assignments, the majority of students in both groups did not even meet the expectations for properly testing their code throughout the semester.

All programming assignments described the importance of creating a test plan, yet more than 80 percent of the students did not create a test plan or provide any evidence their code was tested.

This implies that, regardless of the programming environment used in a CS0 course, more emphasis is needed to ensure students properly test their code. Additional examples and test cases could be provided during the lectures and discussions to improve testing. Detailed feedback could also be given to students after each programming assignment when a faculty member noticed testing of code was not properly conducted or documented. In this manner, by the time the final programming assignment was given, most students should have adapted better testing skills and habits regardless of the programming environment used.

The deficiency in testing of code is not new. Murphy, Lewondowski, McCauley, Simon, Thomas and Zander (2008) found that, although novice computer science students were conducting some testing, this testing was usually limited to a minimal number of test cases. Olan (2003) suggested testing was rarely done as beginning programmers often measured success by the fact that the code compiled without syntax errors.

Although a majority of students for the second programming assignment did not meet the testing expectations, 25 percent of the students who used the Alice programming environment exceeded expectation for testing their code compared to 0 percent of students who used the C++ programming environment. The second programming assignment required students to use *for/ while* control structures. Although more data would be required to fully explain these findings, one possible explanation could be that the graphical environment within the Alice programming environment may have provided some additional ease-of-use for testing and debugging. Researchers (Calloni & Bagert,1997; Carlisle et al., 2005; Myers, 1986) suggested that a graphical, easy-to-use environment can help inspire students to learn and improve performance. Alice does have a graphical interface where students can assign an image to perform repetitive tasks using *for/ while* loops and other code development activities. However, in the current study no comments were captured suggesting that testing was easier using the Alice environment. In addition, one student in the experimental group mentioned that an online calculator was used to help test their output for this programming assignment. It could be those students who tested more thoroughly did so because of their resourcefulness and attention to detail of the programming assignment requirements as opposed to the graphical and visual interface within the Alice environment. Regardless of the reasons or justification for this deficiency, more emphasis on testing of code by novice programmers is required in any programming environment adopted for use in a CS0 class.

Weak repetition control structures prerequisite knowledge.

There were no significant differences found between the control and experimental groups in the prerequisite knowledge areas of *if/else* control structures, *for/while* control structures and functions. However, students demonstrated more prerequisite knowledge using *if/else* statements than they did for functions and *for/while* repetition loops. On average, students correctly answered 32 percent of the *if/else* questions, 22 percent of the function questions, and only 6 percent of the *for/while* repetition loop questions on the pre-test.

These findings imply more emphasis should be placed on teaching *for/while* control structures as students appear to have less prerequisite knowledge compared to *if/else* control structures for this population in a CS0 course. Currently the CS0 course at this institution dedicates one week to *if/else* control structures and one week to *for/while* control structures. Adding an additional week to study the *for/while* control structures may be needed to overcome this deficit present in student's prerequisite knowledge. The course could be rearranged to compress some other topics such as the Web page development or Unix commands to spend more time on the course content related to learning *for/while* control structures.

Other researchers have shown *for/while* control structures are more challenging than *if/else* control structures. Ehlert and Schulte (2009) found selection control structures (*if/else* control structures) were easier than looping control structures (*for/while* control structures). Similarly, Dale (2006) found looping control structures were listed more frequently than selection control structures as a difficult topic in an introductory programming course.

Alice interface concerns.

The Alice interface, although very good for reducing syntax errors (Bishop-Clark et al., 2006; Kelleher & Pausch, 2005), improving layout and visual appeal, and reducing the amount of code students have to type (Bishop-Clark et al., 2006; Conway,1997; Conway et al., 2000), may cause some issues for novice programmers. For the first programming assignment, students needed to create nested *if/else* control structures as the most efficient solution for the problem. Students who did not nest their *if/else* control structures would have resulted in correct functionality (the program would work properly) but a less than desirable design as the code would not run as quickly since more selection control structures would have to be processed each time the code was executed. Students who used the C++ programming environment consistently created nested *if/else* control structures for the first programming assignment.

Although no details were captured as to why some students struggled nesting their *if/else* statements in the Alice programming environment and the users of the C++ programming did not, it is believed that the drag and drop interface of Alice may have misled some students. The drag and drop feature of Alice provides a visual color code indicating correct locations where components can be dropped. A green color indicates the component can be dropped at the location whereas a red color indicates the component cannot be dropped. When creating a nested structure, the Alice programming environment considers both inside and outside of an existing *if/else* structure as valid drop points. Some students could have seen the green color and dropped the component at the wrong location resulting in less efficient coding. Students coding in the C++ environment could have also incorrectly typed their code in the wrong place, but did not seem to do so for this study. In-depth student interviews after each project or focus group discussions may have added some insight into this finding.

The Alice interface could also have been the origin of another problem where more students in the experimental group provided incorrect solutions for one of the functions-related questions on the post-test compared to the control group. This question provided the input parameters in the order of x, z, y where as the call to the function was listed in the order of x, y, z. If a student was not paying attention to the order of the parameters, the wrong answer would have been submitted even if the student understood the overall process of using functions. More students in the experimental group reversed two of the input parameters in their answer.

Although no data were gathered that confirmed why more students in the experimental group missed this problem, it is believed the Alice interface may have contributed to this problem. Once a method or function is created in Alice, parameters are input using an easy-to-use drop down selection menu. This prevents input entry errors in terms of acceptable value and order of the parameters. The control group who used the C++ programming environment had to type in every line and character of code. This may have made them more aware of the details and importance of the order of the parameters in a function call.

This finding implies educators need to be aware of the trade-offs associated with the easy-to-use interface of Alice and other programming environments. Although students may produce code that has less syntax errors and involves less typing, exercises and activities dedicated to typing and reviewing code may be needed to ensure students are acquiring the level of detail needed to properly design efficient code and recognize the importance of parameters and the order of those parameters in functions and sub-routines.

Other researchers have expressed concerns about the Alice interface restricting a student’s ability to learn how to correctly type code. Powers et al. (2007) observed students who used the Alice programming environment did not pay attention to syntax. Mullins et al. (2009) noted students who used Alice lacked the ability to successfully transition to IDE’s where more typing was required. A level of detail was lost as students did not learn how to properly type their code as they progressed in courses.

Most modern IDEs, including Microsoft Visual C++, have features that help minimize code typing by completing code on common syntax as it is typed. Some drop down menus and graphical icons that can be dragged and dropped to a location are also available to help reduce typing. However, as Cheung et al. (2009) noted students often struggle learning how to use these features of the IDE adding to the learning curve associated with programming.

Study population.

Another set of findings and observations arose from the analysis of the demographic questionnaire data. Analysis revealed a large percentage of male, adult learners participated in this study. Each of these characteristics will be discussed as to their uniqueness compared to other research studies and possible instructional implications.

Male learners.

The demographics questionnaire showed 85 percent of the research population were male. This statistic revealed a shortage of females in the computing programs at this university. This finding is not new (Widnall, 1988; CRA Statistics) but remains a problem.

This finding implies that the shortage of females in the computer field must continuously be addressed. This is a national problem that needs attention through marketing and other approaches to attract more females to the program. Instructors must encourage participation and openly welcome both females and males to the program. Creating a positive atmosphere by including women to lead computer science meetings and facilitate seminars would send a message that females are welcome and active participants in university functions. Encouraging females to become teaching assistants may help other female students in a classroom feel better represented. Staffing qualified women to teach these courses may also help to inspire more women to pursue fields in computing.

Comparing the gender statistic from this institution to other institutions conducting research using Alice showed many with similar gender demographics and a few with different statistics. This differs from Conway (1997) and Rodger (2002) who reported 58 percent female and 42 percent male, and 80 percent female and 20 percent male, respectively. Studies by Hundhausen and Brown (2005), and Dodds, Libeskand-Hadas, Alvarado and Kuenning (2008) reported male ratios similar to the current study. Typically, the number of male students either pursuing or considering pursuing a degree in computing is larger than female students (Cohoon, 2003).

Adult learners.

Over 70 percent of students in this study reported their age as greater than 25. This finding implies that instructors, course designers and curriculum specialists should be aware of the differences between adult learners and traditional learners to optimize learning within the course. Knowles (1970) suggested adult learners are internally self-motivated, self-directed, and have a readiness to learn skills and knowledge that are immediately relevant to their personal needs. Adult students are problem-focused. Baker (2009) referred to older software engineers as having both intrinsic and extrinsic motivation. He stated the extrinsic motivation comes from their desire to continue in their careers whereas the intrinsic motivation may occur because of their pride and desire to be competitive with younger software engineers.

The fact that adult learners are self-directed and motivated does not necessarily rule out the use of a microworld. A microworld such as Alice still could be used to demonstrate the visual layout, and use of graphics to assist with testing code. Instructors and curriculum designers should use the known strengths of a microworld specifically related to learning how to program and find other ways to engage adult learners such as assigning projects aligned with their current or perspective careers. As suggested by Sung and Shirley (2004) “courses designed for adult students should draw from the students' life experience and strengths in real-world problem solving” (p. 2).

Many other researchers using Alice as their programming environment reported traditional college-aged students falling in the range from 18 to 22 (Anewalt, 2008; Dougherty,2007; Conway, 1997; Moskal et al., 2004; Mullins et al., 2009; Rodger, 2002; Sykes, 2007) or high school students typically less than 18 years old (Cooper et al., 2000; Rodger et al., 2009; Adams, 2007). One exception was found in Klassen (2006) where the computer science department at the California Lutheran University used Alice for their CS0 courses for both traditional and adult learners. Klassen concluded adult learners were already serious about their education and didn’t need a tool like Alice to motivate them. Although Klassen’s research was limited to approximately 22 evening students, the results do reflect some of the same concerns expressed by some Alice users in this research. For example, the comments from one student such as “I hated learning the Alice environment; it would have taken me 10 minutes to write the code in C++. I feel like I learned to ride my bike and took the training wheels off”, reflected this sentiment. The few students that were frustrated about the environment seemed to stay in the course regardless of their concerns.

These findings suggest caution should be used when selecting the programming environment for adult learners. Adult learners want an environment that they can use in their work environment and may feel frustrated to have to learn within an environment which doesn’t fit into their near or long term career goals.

Student end-of-course evaluations.

The results of the student evaluations were very telling. Students who used the Alice programming environment gave both the instructor and the overall course a significantly lower evaluation than students who used the C++ programming environment.

This finding implies that end-of-course evaluations remain a valuable resource for determining strengths and weaknesses within a course or program. As programming tools and course materials are modified, instructors and administrators should monitor the evaluations closely and conduct focus groups or additional interviews as needed to determine underlying causes of discontent. The lower course and instructor evaluations from the students who used the Alice programming environment suggest a different approach may be needed. Perhaps a blending of a microworld and a more traditional programming environment may improve overall evaluations scores. However, more data are needed to determine root causes of these findings.

When survey results were provided from other literature reviewed for this study, the presentation was usually qualitative in nature with a handful of questions about the satisfaction with Alice or other aspects of the course being answered and summarized. Many results of these surveys were favorable (Anewalt, 2008; Cooper et al., 2000; Dougherty, 2007; Rodger, 2002; Sykes, 2007) although some (Klassen, 2006; Powers et al, 2007) did provide some concerns and limitations about the environment. The significantly lower course and instructor evaluations from this study for the Alice section combined with the negative comments from a few students in the Alice section suggest the experimental group did not like the Alice programming environment as much as the control group liked the C++ programming environment.

Use of rubrics.

Through the use of a rubric, it was determined a larger percentage of control group students exceeded expectations for the design and functionality grading components compared to the experimental group students and a larger percentage of experimental group students exceed expectation for the layout grading component for the programming assignment that required the students to use *if/else* control structures. For the second programming assignment that required the students to use *for/while* control structures, a larger percentage of control group students exceeded expectations for the layout and test grading components compared to the experimental group students. For the third programming assignment that required the students to use functions, a larger percentage of control group students exceeded expectations for the layout grading components compared to the experimental group students.

These findings imply variations may occur between programming environments based on specific learning outcomes and specific types of skill sets such as layout, design, functionality and testing. To capture these variations a rubric may be useful to help to identify strengths of different programming environments by dividing the grading into smaller, finer subscales. Continuing to collect and analyze this level of detail in future research studies and in classrooms in general may help designers and developers of introductory programming environments utilize the strengths from different programming environments and integrate these features into an environment optimized for learning how to program.

General Summary

Several overall implications can be summarized and presented based on the implications presented earlier. First, a variety of programming tools, and assessment methods are required to optimize student learning in a CS0 class. The combination of a microworld, like Alice and an IDE like Microsoft's visual C++ may be required to help students properly format code and reduce syntax errors while also supporting the need of students to practice typing code to build skills needed when migrating to professional programming environments. The incorporation of assessment techniques such as using rubrics to grade programming assignments help identify strengths and weaknesses of the different programming environments available. Details provided from these rubrics may help in the design and creation of future, more powerful programming environments.

Second, instructors and administrators may need to be trained to support a population that includes a large percentage of adult learners. To better prepare to teach this diverse population, faculty need to be versed in approaches to teach self-directed, motivated adult learners. This could result in the inclusion of more real-world projects designed by the students or continuously presenting a positive atmosphere supporting and welcoming female and minority students. Administrators can help in this area by assigning qualified female and minority faculty to teach or serve as teaching assistants for the courses.

Third, a number of curriculum changes may be needed to address these findings. In addition to adding more real-world projects into the course, additional time dedicated to teaching *for/while* control structures is needed. Students had the least amount of prerequisite knowledge in this area compared to *if/else* control structures or functions. The additional time may allow students to gain more knowledge of *for/while* control structures. Testing of code was also determined to be deficient overall. Because of this finding, additional time and examples may be required so students fully understand and demonstrate how to properly test the code they developed for a course.

Finally, continuous monitoring of student success rates, end-of-course evaluations, programming assignment rubrics, and retention in future courses are needed to determine the impact of changes made in programming environments, course materials, instructors and the curriculum. As changes are made to a course or program, monitoring will help determine if the changes resulted in positive outcomes in student learning.

Recommendations

Based on these findings, the Alice programming environment will not solve all issues associated with novice programmers learning how to use programming structures. However; there are other tools including a new version of Alice that may be worth conducting follow-up studies. Alice 3.0 adds additional features to include better alignment with professional IDEs and object-oriented programming languages for smoother transition to more advanced courses. This along with a graphical programming tool such as Raptor may be worth reviewing and conducting a future follow-up study.

The lessons learned from this research should also be incorporated in CS0 courses to enhance learning. Instructors should work closely with students to identify issues with control structures and testing. Materials could be extracted from the textbook and other resources to allow more practice with fundamental programming concepts. More time for capturing student reflection could also help improve the course. Focus groups and detailed interviews could be incorporated into future research to capture details as to why students did not like the Alice programming environment and where they specifically struggled while learning control structures. Students also may need more practice actually typing code as opposed to just selecting, dragging and then dropping control structures. Having students practice more formatting code using an IDEs or text-editor is also recommended to improve their visual layout of their code throughout the semester.

Another area that needs to be researched is the impact of these environments and other programming environments on online courses. This research used students enrolled in face-to-face classes only. Multiple online sections with more students may provide additional insight into differing platforms and possible advantages of visual and graphical tools for a different population of students.

How the selection and use of programming environment in a CS0 course impacts student success and learning in subsequent programming courses needs to be considered for this type of population. CS1 and CS2 often have a more object-oriented flavor to the courses suggesting tools that prepare students for this jump may provide an advantage to CS1 and CS2 students (Lorenzen & Sattar, 2008). Monitoring a cohort of students as they move from CS0 to CS1 to CS2 is recommended.

Recognizing other factors can greatly influence student success and retention, more data are needed on Alice, other microworlds, and visual tools to provide additional insight into trends in retention for introductory computer programming courses. Retention and student success continue to be important factors driving an academic program. The better we can determine the impact of the programming environment on student success and retention the better we can predict student enrollments that in turn help us justify the resources we need to run successful programs.

Finally, the results of this study do add support that Alice is a good programming environment for novice programmers. In particular, the Alice programming environment does an excellent job of forcing students to prepare well-formatted code. It does not seem to improve code design or functionality. As evidenced by the lower course and instructor evaluations for this research population, novice programmers did not like the Alice programming environment as much as they did the visual C++ environment. A small number of students in the Alice group continued to be frustrated with the environment throughout the semester. These users wanted to move beyond Alice to work in a productive programming environment. Designers are needed to continue to improve existing programming environments that support beginners entry into the computer programming field as well as fully support their advancement to other courses and job opportunities.

APPENDICES

Appendix A. Pilot Study

The pilot study took place for the Fall 2008 semester directly prior to the full study that was conducted in Spring 2009. The purpose of the pilot study was ensure the programming assignments and the post-test questions were compatible with the Alice and the C++ programming environments. It also helped the instructor become comfortable using the Alice programming environment and review any associated hand-outs and training materials. Some analysis on the programming assignments and post-test were also performed to determine if the output data would be sufficient for answering the proposed research questions.

Approximately 70 students participated in the pilot with 35 using the Alice programming environment and 35 using the C++ programming environment. Results of the pilot showed the programming assignments were mostly compatible with Alice. However, the second programming assignment directions were modified slightly to include a revised Greatest Common Divisor algorithm. This was modified because Alice did not support a modulus operator. The directions worked around this issue making sure both groups of students used a repetition look in place of the modulus operator.

The pilot also helped determine that additional resolution in grading the programming assignments might be needed to better differentiate the strengths and weaknesses of the two programming environments. It was decided to incorporate a grading rubric with components of layout, design, test data and functionality. In addition, results helped determine adding a pre-test and questionnaire would be useful to better characterize each group and determine the learning gain.

Appendix B. Questionnaire

Instructions:

There are two sections to this pre-test. In section I, please circle the response that best describes your demographic data. In section II, answer the question in the space provided.

Section I: Demographic data.

1. What is your Gender?
    a. Male
    b. Female
2. What is your age?
    a. <20
    b. 20-25
    c. 26-30
    d. 31-40
    e. >40
3. What is your major?
    a. Computer and Information Science (CMIS)
    b. Computer Science (CMSC)
    c. Information Systems Management (IFSM)
    d. Computer Information Technology (CMIT)
    e. Computer Studies (CMST)
    f. Other ____________________
4. What is your ethnicity?
    a. African American

   b. Asian
   c. Caucasian
   d. Hispanic
   e. Native American
   f. Other ____________________

5. How many courses, including this one, are you taking this semester?
   a. 1
   b. 2
   c. 3
   d. 4
   e. >4
6. How many previous computer programming courses have you taken at this or any other higher education institution within the last 5 years?
   a. 0
   b. 1
   c. 2
   d. 3
   e. >3
7. How many previous math courses have you taken at this or any other higher education institution within the last 5 years?
   a. 0
   b. 1
   c. 2

d. 3

e. >3

Appendix C. Pre-test and Post-test Questions

Multiple Choice

Identify the choice that best completes the statement or answers the question.

____ 1. Given the following pseudocode, select the correct output after the segment is run.

```
Set X = 4
If X < 5 Then
    Write “Excellent Work”
Else
    Write “Try again!”
End if
Write “ Thanks”
```

a. Thanks
b. Try again!
c. Excellent Work
d. Excellent Work Thanks

Short Answer

2. What is the output of the code corresponding to the following pseudocode?

```
Set y = 0
For i = 0 Step 3 to 6
  For j = 0 Step 5 to 15
     Set y = y + 1;
  End For (j)
End For (i)
Output y
```

3. What is the output of the code corresponding to the following pseudocode?

```
Set x = 4
Set y = 2
If (x > y) Then
  Output “big X”
Else
  Output “big Y”
End If
```

4. What is the output when code corresponding to the following pseudocode is run?

```
Main Program
  Set X = 1
  Set Y = 2
  Set Z = 3
  Call Display(X, Y, Z)
End Program
Subprogram Display(X, Z, Y)
  Write Z, X, Y
End Subprogram
```

5. What is the output when code corresponding to the following pseudocode is run?

```
Main
  Set X = 4
  Set Y = 10
  Set Z = Product(X, Y)
  Write Z
```

```
End Program
Function product (x, y) as Integer
   set Z = X * Y
end Function
```

6. What is the output of code corresponding to the following pseudocode if Amount = -5?

```
  If Amount > 0 Then
    Write Amount
 End If
 Write Amount
```

7. Using the code shown below, what will be displayed for each of the following input numbers?

   a. 0 ____________

   b. 50 ___________

   c. -4 ___________

```
Input Number
If Number < 0 Then
   Write “1”
Else
    If Number = 0 Then
       Write “2”
    Else
       Write “3”
     End If
```

```
 End If
Write “Completed”
```

8. What numbers will be displayed if code corresponding to the following pseudocode is run?

```
Set Number = 1
While Number < 3
    Write 2  * Number
    Set Number = Number + 1
End While
```

9. What numbers will be displayed when code corresponding to the following pseudocode is run?

```
set N = 3
For K = N Step 1 To 5
   Write N, “ “, K
End For
```

10. What numbers will be displayed if code corresponding to the following pseudocode is run?

```
Set Number = 0
While Number < 4
    Write Number
    Set Number = Number + 1
End While
```

11. What is the output of the following pseudocode?

```
For K = 2 Step 1 To 8
  Write K
End For
```

12. What is the output of code corresponding to the following pseudocode?

```
Main
  Write F(2, 3)
  Write G(-2)
End Program
Function F(X, Y) As Integer
    Set F = 5 * X + Y
End Function
Function G(X) As Integer
    Set G = X * X
End Function
```

13. Suppose a program contains the following function:

```
Function G(X, Y) As Integer
   Set G = X + Y
End Function
```

What is displayed when the following statement is called in the main program?:

```
Write G(4,12)
```

Appendix D. Experimental Group Programming Assignments

Programming Assignment 1: if/else programming assignment in Alice

Task Overview

Your task is to implement the algorithm below using Alice. In addition to the algorithm below, you should prompt the user for their test score and display the resulting letter grade. Pick an object of your choice to visually interact with. For example, you could select a chicken, to ask the user for their score and then say their letter grade based on the logic below.

Once you have a complete program, you should fully test and **submit your Alice world to your instructor using the WebTycho assignments folder as appropriate.** You should save your World using your name and Module3Assignment1. For example, if your name is John Smith, you should save your World as "JohnSmithModule3Assignment1.a2w". Note the extension of ".a2w" is automatically part of the name when you save using Alice version 2.

Please make sure you have successfully conducted all of the exercises and readings leading up this assignment prior to attempting this exercise.

The Algorithm

The following algorithm, written in pseudocode, could be used determine a letter grade based on a test score.

```
if (score >= 85) then
  Set grade = 'A'
else
  if (score >= 75) then
```

```
    Set grade = ‘B’
  else
    if (score >= 65) then
      Set grade = ‘C’
    else
      if (score >= 55) then
        Set grade = ‘D’
      else
        Set grade = ‘F’
      end if // score >= 55
    end if // score >= 65
  end if // score >= 75
end if // score >= 85
```

Test Plan

As previously noted, a test plan should accompany any code you submit. The plan contains data and strategies you have used to test your code. Test your code with the following numbers:

85.0, 65.0001, 54.99999, –33.3, 100, 90, 150

Programming Assignment 2: repetition loop programming assignment in Alice

Task Overview

Your task is to implement the algorithm below using Alice. This algorithm determines the greatest common divisor of two positive integers. As in previous projects, feel free to pick an object of your choice to visually interact with. For example, you could select a mythical creature to ask the user for to input their two positive numbers.

Once you have a complete program, you should fully test and **submit your Alice world to your instructor using the WebTycho assignments folder as appropriate.** You should save your World using your name and Module3Assignment2. For example, if your name is John Smith, you should save your World as "JohnSmithModule3Assignment2.a2w". Note the extension of ".a2w" is automatically part of the name when you save using Alice version 2.

Make sure you have successfully conducted all of the exercises and readings leading up this assignment prior to attempting this exercise.

The Algorithm

The following algorithm, written in pseudocode, could be used determine the Greatest Common Divisor for two positive numbers.

```
Declare a, b, gcd
Input a
Input b
if a = 0
  gcd = b
end if
else
  while b != 0
    if a > b
       a = a – b
```

```
      else
          b = b – a
    end while
    gcd = a
  end else
  print gcd
```

Test Plan

As previously noted, a test plan should accompany any code you submit. The plan contains data and strategies you have used to test your code. Please test your code with the following numbers:

X	Y	Expected Answer
1	1	
2	2	
5	5	
20	20	
2	10	
10	2	
3	4	
4	3	
36	48	
48	36	
90	390	
252	108	
1024	256	
23205	1638	

Programming Assignment 3: functions programming assignment in Alice

Task Overview

You need to create and test two different functions in Alice. The first function should be called traparea(). This function should take three numbers and return the area of the trapezoid represented by the three numbers. Recall that given, a, b and h, where a and b represent the lengths of the parallel sides and h represents the distance between these sides, the formula for the area of a trapezoid is 0.5*(a+b) * h.

The second function should take two numbers and return the perimeter of a rectangle. You should call this function rectperimeter(). Recall that given two numbers, a and b, representing the sides of a rectangle, the perimeter is calculated from this formula:

Perimeter = 2*a + 2*b.

You should test your functions by calling them from your World method. Make sure you prepare at least 5 tests for each function.

Once you have a complete program, you should fully test and **submit your Alice world to your instructor using the WebTycho assignments folder as appropriate.** You should save your World using your name and FunctionsAssignment. For example, if your name is John Smith, you should save your World as "JohnSmithFunctionsAssignment.a2w". Note the extension of ".a2w" is automatically part of the name when you save using Alice version 2.

Make sure you have successfully conducted all of the exercises and readings leading up this assignment prior to attempting this exercise.

Appendix E. Control Group Programming Assignments

Programming Assignment 1: if/else programming assignment in C++

Task Overview

Your task is to implement the algorithm below using C++. In addition to the algorithm below, you should prompt the user for their test score and display the resulting letter grade

Once you have a complete program, you should fully test and **submit your C++ code to your instructor using the WebTycho assignments folder as appropriate.** You should save your project using your name and Module3Assignment1. For example, if your name is John Smith, you should save your program as “JohnSmithModule3Assignment1.cpp.

Make sure you have successfully conducted all of the exercises and readings leading up this assignment prior to attempting this exercise.

The Algorithm

The following algorithm, written in pseudocode, could be used determine a letter grade based on a test score.

```
if (score >= 85) then
  Set grade = ‘A’
else
  if (score >= 75) then
     Set grade = ‘B’
   else
     if (score >= 65) then
```

```
      Set grade = ‘C’
    else
      if (score >= 55) then
        Set grade = ‘D’
      else
        Set grade = ‘F’
      end if // score >= 55
    end if // score >= 65
  end if // score >= 75
end if // score >= 85
```

Test Plan

As previously noted, a test plan should accompany any code you submit. The plan contains data and strategies you have used to test your code. Test your code with the following numbers:

85.0, 65.0001, 54.99999, –33.3, 100, 90, 150

Programming Assignment 2: repetition loop programming assignment in C++

Task Overview

Your task is to implement the algorithm below using C++. This algorithm determines the greatest common divisor of two positive integers.

Once you have a complete program, you should fully test and **submit your C++ program to your instructor using the WebTycho assignments folder as appropriate.** You should save your program using your name and Module3Assignment2. For example, if your name is John Smith, you should save your program as "JohnSmithModule3Assignment2.cpp".

Make sure you have successfully conducted all of the exercises and readings leading up this assignment prior to attempting this exercise.

The Algorithm

The following algorithm, written in pseudocode, could be used determine the Greatest Common Divisor for two positive numbers.

```
Declare a, b, gcd
Input a
Input b
if a = 0
  gcd = b
end if
else
  while b != 0
    if a > b
       a = a – b
    else
       b = b – a
  end while
  gcd = a
end else
print gcd
```

Test Plan

As previously noted, a test plan should accompany any code you submit. The plan contains data and strategies you have used to test your code. Please test your code with the following numbers:

X	Y	Expected Answer
1	1	
2	2	
5	5	
20	20	
2	10	
10	2	
3	4	
4	3	
36	48	
48	36	
90	390	
252	108	
1024	256	
23205	1638	

Programming Assignment 3: functions programming assignment in C++

Task Overview

You need to create and test two different functions in C++. The first function should be called traparea(). This function should take three numbers and return the area of the trapezoid represented by the three numbers. Recall that given, a, b and h, where a and b represent the lengths of the parallel sides and h represents the distance between these sides, the formula for the area of a trapezoid is 0.5*(a+b) * h.

The second function should take two numbers and return the perimeter of a rectangle. You should call this function rectperimeter(). Recall that given two numbers, a and b, representing the sides of a rectangle, the perimeter is calculated from this formula:

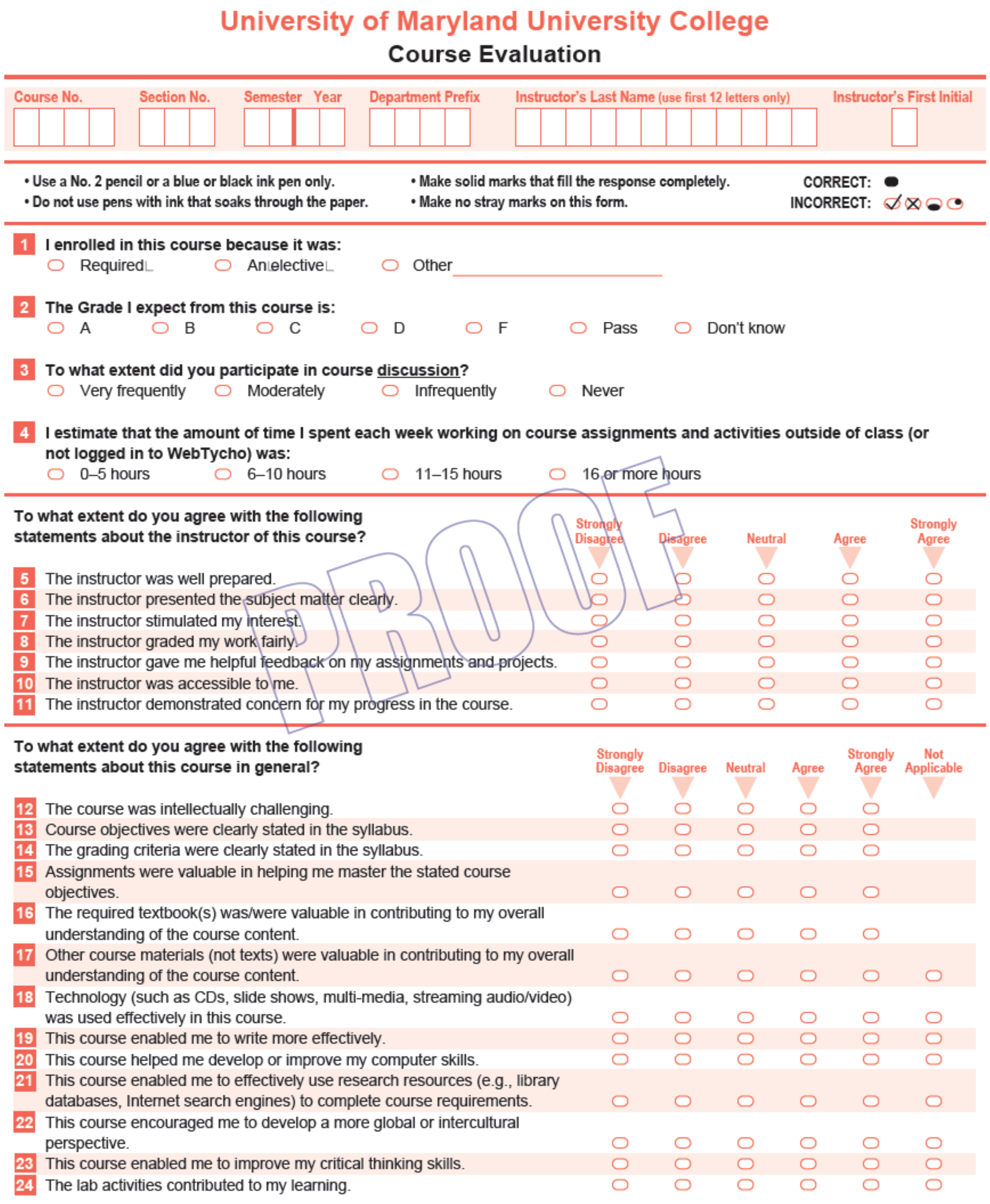
Perimeter = 2*a + 2*b.

You should test your functions by calling them from your main method. Make sure you prepare at least 5 tests for each function.

Once you have a complete program, you should fully test and **submit your C++ to your instructor using the WebTycho assignments folder as appropriate.** You should save your World using your name and FunctionsAssignment. For example, if your name is John Smith, you should save your C++ as "JohnSmithFunctionsAssignment.cpp".

Make sure you have successfully conducted all of the exercises and readings leading up this assignment prior to attempting this exercise.

Appendix F. Course Evaluation Form (Page 1)



University of Maryland University College

Course Evaluation

Course No.	Section No.	Semester Year	Department Prefix	Instructor's Last Name (use first 12 letters only)	Instructor's First Initial

- Use a No. 2 pencil or a blue or black ink pen only.
- Do not use pens with ink that soaks through the paper.
- Make solid marks that fill the response completely.
- Make no stray marks on this form.

CORRECT: ●
INCORRECT: ✓ ⊗ ◒ ◔

1 I enrolled in this course because it was:
○ Required ○ An elective ○ Other ____________

2 The Grade I expect from this course is:
○ A ○ B ○ C ○ D ○ F ○ Pass ○ Don't know

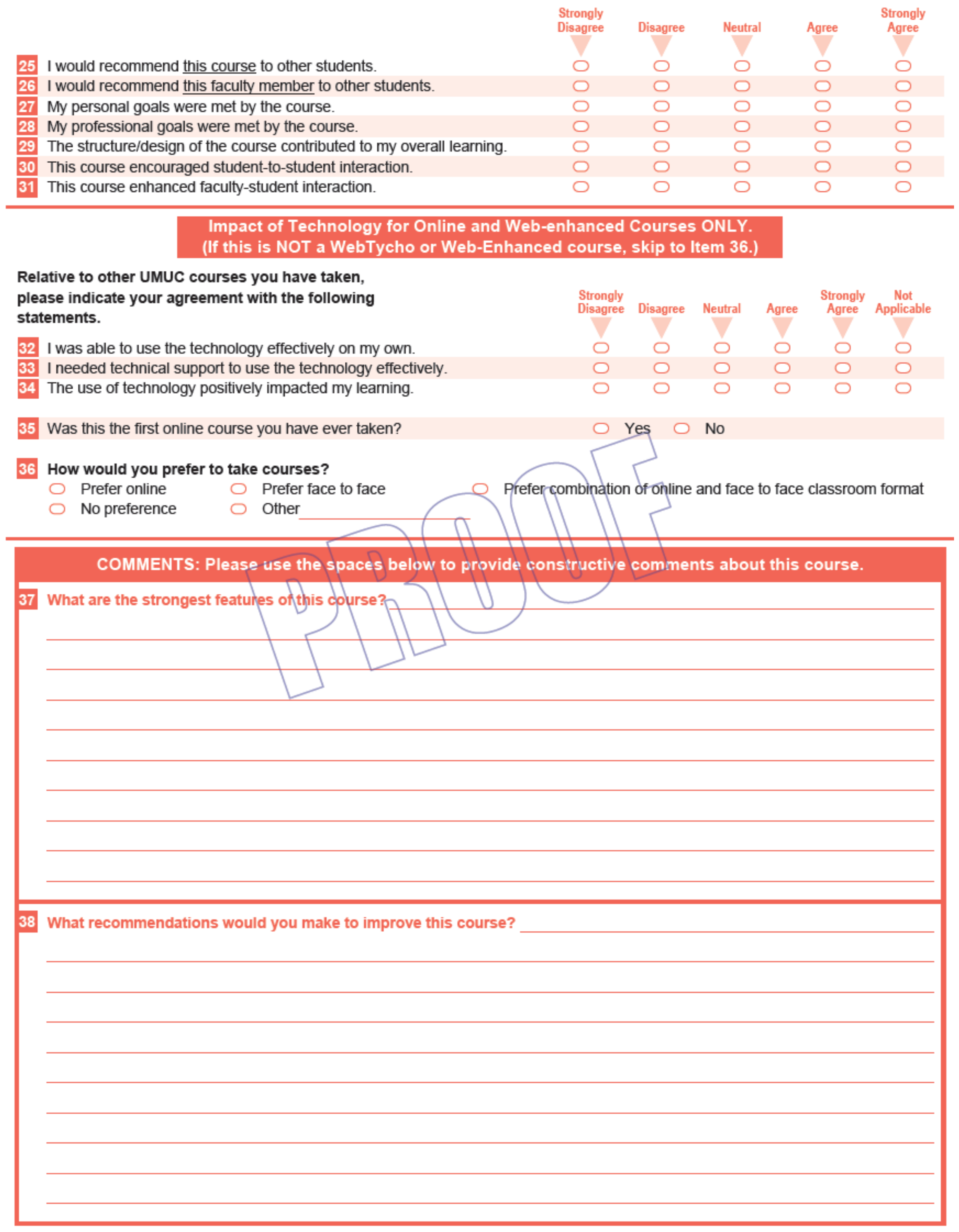
3 To what extent did you participate in course discussion?
○ Very frequently ○ Moderately ○ Infrequently ○ Never

4 I estimate that the amount of time I spent each week working on course assignments and activities outside of class (or not logged in to WebTycho) was:
○ 0–5 hours ○ 6–10 hours ○ 11–15 hours ○ 16 or more hours

To what extent do you agree with the following statements about the instructor of this course?	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
5 The instructor was well prepared.	○	○	○	○	○
6 The instructor presented the subject matter clearly.	○	○	○	○	○
7 The instructor stimulated my interest.	○	○	○	○	○
8 The instructor graded my work fairly.	○	○	○	○	○
9 The instructor gave me helpful feedback on my assignments and projects.	○	○	○	○	○
10 The instructor was accessible to me.	○	○	○	○	○
11 The instructor demonstrated concern for my progress in the course.	○	○	○	○	○

To what extent do you agree with the following statements about this course in general?	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	Not Applicable
12 The course was intellectually challenging.	○	○	○	○	○	
13 Course objectives were clearly stated in the syllabus.	○	○	○	○	○	
14 The grading criteria were clearly stated in the syllabus.	○	○	○	○	○	
15 Assignments were valuable in helping me master the stated course objectives.	○	○	○	○	○	
16 The required textbook(s) was/were valuable in contributing to my overall understanding of the course content.	○	○	○	○	○	
17 Other course materials (not texts) were valuable in contributing to my overall understanding of the course content.	○	○	○	○	○	○
18 Technology (such as CDs, slide shows, multi-media, streaming audio/video) was used effectively in this course.	○	○	○	○	○	○
19 This course enabled me to write more effectively.	○	○	○	○	○	○
20 This course helped me develop or improve my computer skills.	○	○	○	○	○	○
21 This course enabled me to effectively use research resources (e.g., library databases, Internet search engines) to complete course requirements.	○	○	○	○	○	○
22 This course encouraged me to develop a more global or intercultural perspective.	○	○	○	○	○	○
23 This course enabled me to improve my critical thinking skills.	○	○	○	○	○	○
24 The lab activities contributed to my learning.	○	○	○	○	○	○

Appendix F. Course Evaluation Form (page 2)



		Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
25	I would recommend this course to other students.	○	○	○	○	○
26	I would recommend this faculty member to other students.	○	○	○	○	○
27	My personal goals were met by the course.	○	○	○	○	○
28	My professional goals were met by the course.	○	○	○	○	○
29	The structure/design of the course contributed to my overall learning.	○	○	○	○	○
30	This course encouraged student-to-student interaction.	○	○	○	○	○
31	This course enhanced faculty-student interaction.	○	○	○	○	○

Impact of Technology for Online and Web-enhanced Courses ONLY.
(If this is NOT a WebTycho or Web-Enhanced course, skip to Item 36.)

Relative to other UMUC courses you have taken, please indicate your agreement with the following statements.

		Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	Not Applicable
32	I was able to use the technology effectively on my own.	○	○	○	○	○	○
33	I needed technical support to use the technology effectively.	○	○	○	○	○	○
34	The use of technology positively impacted my learning.	○	○	○	○	○	○

35 Was this the first online course you have ever taken? ○ Yes ○ No

36 How would you prefer to take courses?

○ Prefer online ○ Prefer face to face ○ Prefer combination of online and face to face classroom format

○ No preference ○ Other ______________

COMMENTS: Please use the spaces below to provide constructive comments about this course.

37 What are the strongest features of this course? ______________

38 What recommendations would you make to improve this course? ______________

Appendix G. Time-on-task Collection Form

1. How many total hours would you estimate you spent on preparing this project for submission to your instructor for grading? Please do you best to estimate the time it took from when you first started working on the project to when you completed it.

2. Describe your experience using the programming environment during this project.

3. Describe any additional tools beyond the programming environment you used to help complete this project.

Appendix H. Cross-tabulation Results for Questionnaire

Table H1

Gender Counts for Control (C), Experimental (E), and Total (T) Groups

Gender	Count			% within Group		
	C	E	T	C	E	T
Male	15	19	34	88.2%	82.6%	85.0%
Female	2	4	6	11.8%	17.4%	15.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table H2

Gender chi-square Tests

Method	Value	*df*	*p*
Pearson chi-square	.243[a]	1	.622
Likelihood Ratio	.248	1	.619

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is 2.55.

Table H3

Age Counts for Control (C), Experimental (E), and Total (T) Groups

Gender	Count			% within Group		
	C	E	T	C	E	T
<20	0	0	0	0.0%	0.0%	0.0%
20-25	1	9	10	5.9%	39.1%	25.0%
26-30	8	4	12	47.1%	17.4%	30.0%
31-40	6	7	13	35.3%	30.4%	32.5%
>40	2	3	5	11.8%	13.0%	12.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table H4

Age chi-square Tests

Method	Value	*df*	*p*
Pearson chi-square	7.274[a]	3	.064
Likelihood Ratio	8.095	3	.044

a. 3 cells (37.5%) have expected count less than 5. The minimum expected count is 2.13.

Table H5

Major Counts for Control (C), Experimental (E), and Total (T) Groups

Major	Count			% within Group		
	C	E	T	C	E	T
CMIS	0	7	7	0.0%	30.4%	17.5%
CMSC	0	1	1	0.0%	4.3%	2.5%
IFSM	6	4	10	35.3%	17.4%	25.0%
CMIT	9	9	18	52.9%	39.1%	45.0%
CMST	1	0	1	5.9%	0.0%	2.5%
Other	1	2	3	5.9%	8.7%	7.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table H6

Major chi-square Tests

Method	Value	*df*	*p*
Pearson chi-square	9.037[a]	5	.108
Likelihood Ratio	12.316	5	.031

a. 9 cells (75.0%) have expected count less than 5. The minimum expected count is .43.

Table H7

Ethnicity Counts for Control (C), Experimental (E), and Total (T) Groups

Ethnicity	Count			% within Group		
	C	E	T	C	E	T
African-American	8	11	19	47.1%	47.8%	47.5%
Asian	5	2	7	29.4%	8.7%	17.5%
Caucasian	1	5	6	5.9%	21.7%	15.0%
Hispanic	2	4	6	11.8%	17.4%	15.0%
Other	1	1	2	5.9%	4.3%	5.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table H8

Ethnicity chi-square Tests

Method	Value	*df*	*p*
Pearson chi-square	4.289[a]	4	.368
Likelihood Ratio	4.491	4	.344

a. 8 cells (80.0%) have expected count less than 5. The minimum expected count is .85.

Table H9

Number of Courses Counts for Control (C), Experimental (E), and Total (T) Groups

Number of Courses	Count			% within Group		
	C	E	T	C	E	T
1	2	2	4	11.8%	8.7%	10.0%
2	8	5	13	47.1%	21.7%	32.5%
3	0	4	4	0.0%	17.4%	10.0%
4	5	5	10	29.4%	21.7%	25.0%
>4	2	7	9	11.8%	30.4%	22.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table H10

Number of Courses chi-square Tests

Method	Value	*df*	*p*
Pearson chi-square	6.721[a]	4	.151
Likelihood Ratio	8.282	4	.082

a. 6 cells (60.0%) have expected count less than 5. The minimum expected count is 1.70.

Table H11

Number of Previous Computer Courses Counts for Control (C), Experimental (E), and Total (T) Groups

Computer Courses	Count			% within Group		
	C	E	T	C	E	T
0	10	12	22	58.8%	52.2%	55.0%
1	5	5	10	29.4%	21.7%	25.0%
2	1	0	1	5.9%	0.0%	2.5%
3	1	2	3	5.9%	8.7%	7.5%
>3	0	4	4	0.0%	17.4%	10.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table H12

Number of Previous Computer Courses chi-square Tests

Method	Value	*df*	*p*
Pearson chi-square	4.721[a]	4	.317
Likelihood Ratio	6.550	4	.162

a. 7 cells (70.0%) have expected count less than 5. The minimum expected count is .43.

Table H13

Number of Previous Math Courses Counts for Control (C), Experimental (E), and Total (T) Groups

Math Courses	Count			% within Group		
	C	E	T	C	E	T
0	3	6	9	17.6%	26.1%	22.5%
1	8	5	13	47.1%	21.7%	32.5%
2	1	3	4	5.9%	13.0%	10.0%
3	5	4	9	29.4%	17.4%	22.5%
>3	0	5	5	0.0%	21.7%	12.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table H14

Number of Previous Math Courses chi-square Tests

Method	Value	*df*	*p*
Pearson chi-square	7.062[a]	4	.133
Likelihood Ratio	8.904	4	.064

a. 6 cells (60.0%) have expected count less than 5. The minimum expected count is 1.70.

Appendix I. Pre-Test Chi-square and *t* test Results

Table I1

Pre-test Question 1 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	11	16	27	64.7%	69.6%	67.5%
3	6	7	13	35.3%	30.4%	32.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table I2

Pre-test Question 1 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	.105[a]	1	.746
Likelihood Ratio	.105	1	.746

a. 0 cells (.0%) have expected count less than 5. The minimum expected count is 5.53.

Table I3

Pre-test Question 2 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	16	23	39	94.1%	100.0%	97.5%
3	1	0	1	5.9%	0.0%	2.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table I4

Pre-test Question 2 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	1.388[a]	1	.239
Likelihood Ratio	1.746	1	.186

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is .43.

Table I5

Pre-test Question 3 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	9	14	23	52.9%	60.9%	57.5%
3	8	9	17	47.1%	39.1%	42.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table I6

Pre-test Question 3 chi-square Tests

Method	Value	*df*	*p*
Pearson chi-square	.251[a]	1	.616
Likelihood Ratio	.251	1	.616

a. 0 cells (.0%) have expected count less than 5. The minimum expected count is 7.23.

Table I7

Pre-test Question 4 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	9	14	23	52.9%	60.9%	57.5%
1	7	9	16	41.2%	39.1%	40.0%
3	1	0	1	5.9%	0.0%	2.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table I8

Pre-test Question 4 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	1.470[a]	2	.479
Likelihood Ratio	1.829	2	.401

a. 2 cells (33.3%) have expected count less than 5. The minimum expected count is .43.

Table I9

Pre-test Question 5 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	13	14	27	76.5%	60.9%	67.5%
3	4	9	13	23.5%	39.1%	32.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table I10

Pre-test Question 5 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	1.085[a]	1	.298
Likelihood Ratio	1.107	1	.293

a. 0 cells (.0%) have expected count less than 5. The minimum expected count is 5.53.

Table I11

Pre-test Question 6 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	15	19	34	88.2%	82.6%	85.0%
3	2	4	6	11.8%	17.4%	15.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table I12

Pre-test Question 6 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	.243[a]	1	.622
Likelihood Ratio	.248	1	.619

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is 2.55.

Table I13

Pre-test Question 7 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	6	13	19	35.3%	56.5%	47.5%
3	4	2	6	23.5%	8.7%	15.0%
6	5	6	11	29.4%	26.1%	27.5%
9	2	2	4	11.8%	8.7%	10.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table I14

Pre-test Question 7 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	2.493[a]	3	.477
Likelihood Ratio	2.508	3	.474

a. 5 cells (62.5%) have expected count less than 5. The minimum expected count is 1.70.

Table I15

Pre-test Question 8 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	16	21	37	94.1%	91.3%	92.5%
3	1	2	3	5.9%	8.7%	7.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table I16

Pre-test Question 8 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	.112[a]	1	.738
Likelihood Ratio	.114	1	.735

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is 1.28.

Table I17

Pre-test Question 9 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	15	21	36	88.2%	91.3%	90.0%
1	1	1	2	5.9%	4.3%	5.0%
3	1	1	2	5.9%	4.3%	5.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table I18

Pre-test Question 9 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	.102[a]	2	.950
Likelihood Ratio	.101	2	.951

a. 4 cells (66.7%) have expected count less than 5. The minimum expected count is .85.

Table I19

Pre-test Question 10 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	14	21	35	82.4%	91.3%	87.5%
3	3	2	5	17.6%	8.7%	12.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table I20

Pre-test Question 10 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	.716[a]	1	.397
Likelihood Ratio	.707	1	.400

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is 2.13.

Table I21

Pre-test Question 11 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	16	22	38	94.1%	95.7%	95.0%
1	0	1	1	0.0%	4.3%	2.5%
3	1	0	1	5.9%	0.0%	2.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table I22

Pre-test Question 11 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	2.094[a]	2	.351
Likelihood Ratio	2.821	2	.244

a. 4 cells (66.7%) have expected count less than 5. The minimum expected count is .43.

Table I23

Pre-test Question 12 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	14	19	33	82.4%	82.6%	82.5%
1	2	1	3	11.8%	4.3%	7.5%
3	1	3	4	5.9%	13.0%	10.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table I24

Pre-test Question 12 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	1.218[a]	2	.544
Likelihood Ratio	1.243	2	.537

a. 4 cells (66.7%) have expected count less than 5. The minimum expected count is 1.28.

Table I25

Pre-test Question 13 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	14	15	29	82.4%	65.2%	72.5%
3	3	8	11	17.6%	34.8%	27.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table I26

Pre-test Question 13 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	1.440[a]	1	.230
Likelihood Ratio	1.489	1	.222

a. 1 cells (25.0%) have expected count less than 5. The minimum expected count is 4.68.

Table I27

Group Statistics for Pre-test Summation Variables

Variable	M	SD	SE
Control Group (n=17)			
If sum	6.35	5.601	1.358
For sum	.59	2.181	.529
While sum	.71	1.687	.409
Rep sum	1.29	3.670	.890
Functions sum	2.12	3.276	.795
Total Points	9.76	11.060	2.682
Experimental Group (n=23)			
If sum	5.22	5.946	1.240
For sum	.22	.736	.153
While sum	.52	1.729	.360
Rep sum	.74	1.815	.378
Functions sum	3.04	3.747	.781
Total Points	9.00	10.846	2.262

Table I28

Pre-test Summation Variables t test

Variable	*t*	*df*	*p*
If sum	.612	38	.544
For sum	.762	38	.451
While sum	.336	38	.738
Rep sum	.630	38	.532
Functions sum	-.814	38	.421
Total Points	.219	38	.828

Appendix J. Project 1 Statistics

Table J1.

Project 1Test Rubric Component Counts for Control (C), Experimental (E), and Total (T) Groups

Category	Count			% within Group		
	C	E	T	C	E	T
Does Not	17	21	38	100.0%	91.3%	95.0%
Meets	0	0	0	0.0%	0.0%	0.0%
Exceeds	0	2	2	0.0%	8.7%	5.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table J2

Project 1 chi-square Testing Component

Method	Value	*df*	*P*
Pearson chi-square	1.556[a]	1	.212
Likelihood Ratio	2.291	1	.130

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is .85.

Table J3

Group Statistics for Project 1 Total

Variable	M	SD	SE
	Control Group (n=17)		
Total Points	9.59	.507	.123
	Experimental Group (n=23)		
Total Points	9.22	1.085	.226

Table J4

Project 1 Total Points Variable t test

Variable	*t*	*df*	*p*
Total Points	1.304	38	.200

Appendix K. Project 2 Cross-tabulation and Chi-square statistics

Table K1

Project 2 Rubric Component Counts for Control (C), Experimental (E), and Total (T) Groups

Grading	Count			% within Group		
	C	E	T	C	E	T
			Design Component			
Does Not	0	1	1	0.0%	4.3%	2.5%
Meets	2	0	2	11.8%	0.0%	5.0%
Exceeds	15	22	37	88.2%	95.7%	92.5%
Total	17	23	40	100.0%	100.0%	100.0%
			Functionality Component			
Does Not	1	1	2	5.9%	4.3%	5.0%
Meets	0	2	2	0.0%	8.7%	5.0%
Exceeds	16	20	36	94.1%	87.0%	90.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table K2

Project 2 Design chi square statistics output

Method	Value	*df*	*P*
Pearson chi-square	3.503[a]	2	.174
Likelihood Ratio	4.588	2	.101

a. 4 cells (66.7%) have expected count less than 5. The minimum expected count is .43.

Table K3

Project 2 Functionality chi square statistics output

Method	Value	*df*	*P*
Pearson chi-square	1.580[a]	2	.454
Likelihood Ratio	2.315	2	.314

a. 4 cells (66.7%) have expected count less than 5. The minimum expected count is .85.

Appendix L. Project 3 Cross-tabulation and Chi-square statistics

Table L1

Project 3 Rubric Component Counts for Control (C), Experimental (E), and Total (T) Groups

Grading	Count			% within Group		
	C	E	T	C	E	T
Design Component						
Does Not	0	1	1	0.0%	4.3%	2.5%
Meets	1	2	3	5.9%	8.7%	7.5%
Exceeds	16	20	36	94.1%	87.0%	90.0%
Total	17	23	40	100.0%	100.0%	100.0%
Test Component						
Does Not	14	19	33	82.4%	82.6%	82.5%
Meets	0	0	0	0.0%	0.0%	0.0%
Exceeds	3	4	7	17.6%	17.4%	17.5%
Total	17	23	40	100.0%	100.0%	100.0%
Functionality Component						
Does Not	2	0	2	11.8%	.0%	5.0%
Meets	0	4	4	.0%	17.4%	10.0%
Exceeds	15	19	34	88.2%	82.6%	85.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table L2

Project 3 Design chi square statistics output

Method	Value	*df*	*P*
Pearson chi-square	.898[a]	2	.638
Likelihood Ratio	1.268	2	.530

a. 4 cells (66.7%) have expected count less than 5. The minimum expected count is .43.

Table L3

Project 3 Test chi- square statistics output

Method	Value	*df*	*P*
Pearson chi-square	.000[a]	1	.983
Likelihood Ratio	.000	1	.983

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is 2.98.

Table L4

Project 3 Functionality chi- square statistics output

Method	Value	*df*	*P*
Pearson chi-square	5.699[a]	2	.058
Likelihood Ratio	7.886	2	.019

a. 4 cells (66.7%) have expected count less than 5. The minimum expected count is .85.

Appendix M. Post-Test Chi-square and T test results

Table M1

Post-test Question 1 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	2	4	6	11.8%	17.4%	15.0%
1	0	0	0	0.0%	0.0%	0.0%
2	0	0	0	0.0%	0.0%	0.0%
3	15	19	34	88.2%	82.6%	85.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table M2

Post-test Question 1 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	.243[a]	1	.622
Likelihood Ratio	.248	1	.619

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is 2.55.

Table M3

Post-test Question 2 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	5	13	18	29.4%	56.5%	45.0%
1	6	2	8	35.3%	8.7%	20.0%
2	3	3	6	17.6%	13.0%	15.0%
3	3	5	8	17.6%	21.7%	20.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table M4

Post-test Question 2 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	5.274[a]	3	.153
Likelihood Ratio	5.378	3	.146

a. 6 cells (75.0%) have expected count less than 5. The minimum expected count is 2.55.

Table M5

Post-test Question 3 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	2	1	3	11.8%	4.3%	7.5%
1	0	0	0	0.0%	0.0%	0.0%
2	0	0	0	0.0%	0.0%	0.0%
3	15	22	37	88.2%	95.7%	92.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table M6

Post-test Question 3 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	.775[a]	1	.379
Likelihood Ratio	.769	1	.381

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is 1.28.

Table M7

Post-test Question 5 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	1	3	4	5.9%	13.0%	10.0%
1	0	0	0	0.0%	0.0%	0.0%
2	0	0	0	0.0%	0.0%	0.0%
3	16	20	36	94.1%	87.0%	90.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table M8

Post-test Question 5 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	.557[a]	1	.455
Likelihood Ratio	.588	1	.443

a. 2 cells (50.0%) have expected count less than 5. The minimum expected count is 1.70.

Table M9

Post-test Question 6 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	6	8	14	35.3%	34.8%	35.0%
1	3	2	5	17.6%	8.7%	12.5%
2	0	0	0	0.0%	0.0%	0.0%
3	8	13	21	47.1%	56.5%	52.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table M10

Post-test Question 6 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	.794[a]	2	.672
Likelihood Ratio	.787	2	.675

a. 2 cells (33.3%) have expected count less than 5. The minimum expected count is 2.13.

Table M11

Post-test Question 7 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	1	1	2	5.9%	4.3%	5.0%
5	1	0	1	5.9%	0.0%	2.5%
6	7	10	17	41.2%	43.5%	42.5%
8	1	1	2	5.9%	4.3%	5.0%
9	7	11	18	41.2%	47.8%	45.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table M12

Post-test Question 7 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	1.553[a]	4	.817
Likelihood Ratio	1.911	4	.752

a. 6 cells (60.0%) have expected count less than 5. The minimum expected count is .43.

Table M13

Post-test Question 8 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	2	8	10	11.8%	34.8%	25.0%
1	7	7	14	41.2%	30.4%	35.0%
2	2	2	4	11.8%	8.7%	10.0%
3	6	6	12	35.3%	26.1%	30.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table M14

Post-test Question 8 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	2.762[a]	3	.430
Likelihood Ratio	2.951	3	.399

a. 3 cells (37.5%) have expected count less than 5. The minimum expected count is 1.70.

Table M15

Post-test Question 9 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	4	10	14	23.5%	43.5%	35.0%
1	10	8	18	58.8%	34.8%	45.0%
2	1	1	2	5.9%	4.3%	5.0%
3	2	4	6	11.8%	17.4%	15.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table M16

Post-test Question 9 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	2.619[a]	3	.454
Likelihood Ratio	2.655	3	.448

a. 4 cells (50.0%) have expected count less than 5. The minimum expected count is .85.

Table M17

Post-test Question 10 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	5	9	14	29.4%	39.1%	35.0%
1	2	1	3	11.8%	4.3%	7.5%
2	2	4	6	11.8%	17.4%	15.0%
3	8	9	17	47.1%	39.1%	42.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table M18

Post-test Question 10 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	1.332[a]	3	.722
Likelihood Ratio	1.334	3	.721

a. 4 cells (50.0%) have expected count less than 5. The minimum expected count is 1.28.

Table M19

Post-test Question 11 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	4	8	12	23.5%	34.8%	30.0%
1	4	3	7	23.5%	13.0%	17.5%
2	2	3	5	11.8%	13.0%	12.5%
3	7	9	16	41.2%	39.1%	40.0%
Total	17	23	40	100.0%	100.0%	100.0%

Table M20

Post-test Question 11 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	1.050[a]	3	.789
Likelihood Ratio	1.051	3	.789

a. 4 cells (50.0%) have expected count less than 5. The minimum expected count is 2.13.

Table M21

Post-test Question 12 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	3	5	8	17.6%	21.7%	20.0%
1	2	3	5	11.8%	13.0%	12.5%
2	2	6	8	11.8%	26.1%	20.0%
3	10	9	19	58.8%	39.1%	47.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table M22

Post-test Question 12 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	1.895[a]	3	.594
Likelihood Ratio	1.949	3	.583

a. 6 cells (75.0%) have expected count less than 5. The minimum expected count is 2.13.

Table M23

Post-test Question 13 Counts for Control (C), Experimental (E), and Total (T) Groups

Points	Count			% within Group		
	C	E	T	C	E	T
0	1	2	3	5.9%	8.7%	7.5%
1	2	0	2	11.8%	0.0%	5.0%
2	1	1	2	5.9%	4.3%	5.0%
3	13	20	33	76.5%	87.0%	82.5%
Total	17	23	40	100.0%	100.0%	100.0%

Table M24

Post-test Question 13 chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	2.985[a]	3	.394
Likelihood Ratio	3.705	3	.295

a. 6 cells (75.0%) have expected count less than 5. The minimum expected count is .85.

Table M25

Group Statistics for Post-Test Summation Variables

Variable	Mean	STD	STD Error Mean
Control Group (n=17)			
If sum	13.82	4.290	1.040
For sum	4.00	2.449	.594
While sum	3.47	2.211	.536
Rep sum	7.47	3.826	.928
Functions sum	8.59	2.694	.653
Total Points	29.88	9.130	2.214
Experimental Group (n=23)			
If sum	14.39	3.963	.826
For sum	3.52	2.998	.625
While sum	2.83	2.387	.498
Rep sum	6.35	4.764	.993
Functions sum	7.87	3.152	.657
Total Points	28.61	9.926	2.070

Table M26

Post-test Summation Variables t test

Variable	*t*	*df*	*p*
If sum	-.433	38	.668
For sum	.538	38	.594
While sum	.871	38	.389
Rep sum	.799	38	.429
Functions sum	.757	38	.454
Total Points	.415	38	.681

Appendix N. Pre- and Post-test Analysis of Covariance Results

Table N1

Total score Adjusted Means ANCOVA Results for Control (C) and Experimental (E)

Adjusted Means (F = .121, ρ = .721)	
Control	Experimental
29.66	28.77

Table N2

If/else score Adjusted Means ANCOVA Results for Control (C) and Experimental (E)

Adjusted Means (F = .623, ρ = .435)	
Control	Experimental
13.61	14.54

Table N3

For/while score Adjusted Means ANCOVA Results for Control (C) and Experimental (E)

Adjusted Means (F = .283, ρ = .598)	
Control	Experimental
7.19	6.55

Table N4

Functions score Adjusted Means ANCOVA Results for Control (C) and Experimental (E)

Adjusted Means (F = 1.18, ρ = .284)	
Control	Experimental
8.74	7.75

Appendix O. Student Evaluation Average Weekly Study Hours

Table O1

Student Evaluation Study Hour Counts for Control (C), Experimental (E), and Total (T) Groups

Avg. Study Hours	Count			% within Group		
	C	E	T	C	E	T
0-5	8	10	18	44.4%	40.0%	41.9%
6-10	5	13	18	27.8%	52.0%	41.9%
11-15	2	0	2	11.1%	0.0%	4.7%
>15	3	2	5	16.7%	8.0%	11.6%
Total	18	25	43	100.0%	100.0%	100.0%

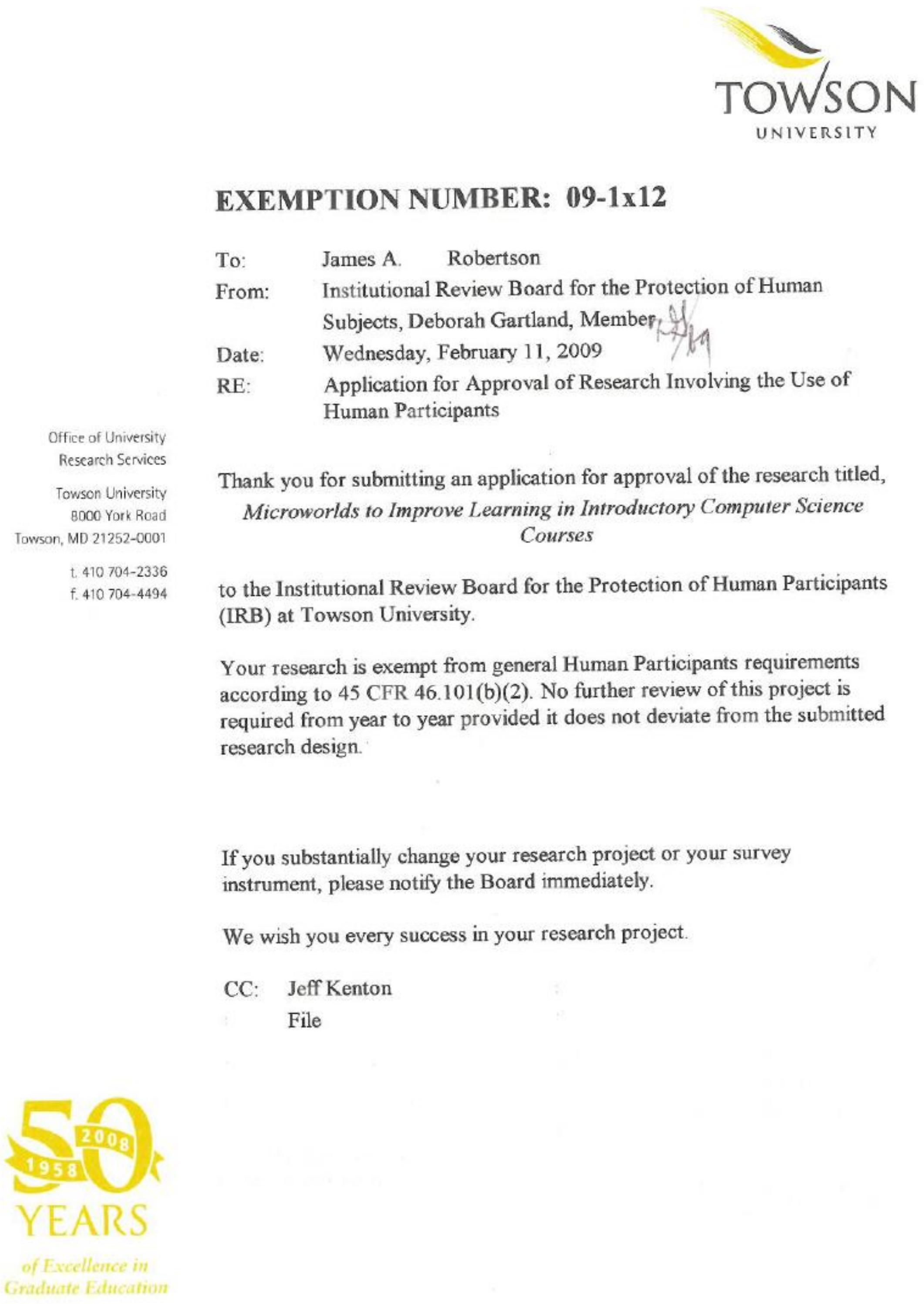
Table O2

Student Evaluation Study Hour chi-square Tests

Method	Value	*df*	*P*
Pearson chi-square	4.970[a]	3	.174
Likelihood Ratio	5.735	3	.125

a. 4 cells (50.0%) have expected count less than 5. The minimum expected count is .84.

Appendix P. IRB Approval



TOWSON UNIVERSITY

EXEMPTION NUMBER: 09-1x12

To:	James A. Robertson
From:	Institutional Review Board for the Protection of Human Subjects, Deborah Gartland, Member [illegible]
Date:	Wednesday, February 11, 2009
RE:	Application for Approval of Research Involving the Use of Human Participants

Office of University Research Services
Towson University
8000 York Road
Towson, MD 21252-0001
t. 410 704-2336
f. 410 704-4494

Thank you for submitting an application for approval of the research titled,
Microworlds to Improve Learning in Introductory Computer Science Courses

to the Institutional Review Board for the Protection of Human Participants (IRB) at Towson University.

Your research is exempt from general Human Participants requirements according to 45 CFR 46.101(b)(2). No further review of this project is required from year to year provided it does not deviate from the submitted research design.

If you substantially change your research project or your survey instrument, please notify the Board immediately.

We wish you every success in your research project.

CC: Jeff Kenton
File

50 YEARS 1958 2008
of Excellence in Graduate Education

Appendix Q. Informed Consent Form

INFORMED CONSENT FORM

James Robertson, Researcher

The Computer and Information Science Department (CMIS) within the School of Undergraduate Studies at UMUC is conducting research on learning outcomes of introductory programming courses. As part of your enrollment, you will be asked to write and submit working computer programs. These programs will be scored based on a rubric. Your individual scores will be grouped with the other students in this section and compared with another section of this course that is completing the same assignments. Based on the outcomes of these section-by-section comparisons, we may make changes to the way the course is taught for future students. Your participation in the research study is voluntary, though you are still required to complete all mandatory course assignments.

During this research, you will do what you normally do in an introductory computer programming course. That is, you will learn fundamental computer programming design and implementation techniques. You will be adhering to the standard CMIS102 syllabus and participating in pre-tests, post-tests, projects, assignments, reading and class participation. There are no known risks associated with this research.

All information will remain strictly confidential. Although the descriptions and findings may be published, at no time will your name be used. The participants' data will remain protected at all times. During the analysis phase, only student randomly assigned identification numbers will be used to separate records. If small subgroups are identified the results will not be publicly distributed in reports or findings. You are at liberty to withdraw your consent to the experiment and discontinue participation at any time without prejudice. If you do choose to not participate or to discontinue participation your decision will not have an effect on your grade in the course. However, all of the projects and activities still need to be completed. In these cases, the individuals' responses will be withheld from contribution to the research.

If you have any questions after today, please feel free to call 240-582-2846 and ask for James Robertson (jrobertson@umuc.edu), or contact Dr. Marie Cini (mcini@umuc.edu), the Dean of School of Undergraduate Studies, or Dr. Patricia Alt, Chairperson of the Institutional Review Board for the Protection of Human Participants at Towson University at (410) 704-2236.

I, ___________________________________, affirm that I have read and understood the above statement and have had all of my questions answered.

Date: _____________________

Signature: ____

Witness: ____

Appendix R. Programming Assignment Grading Rubric

Rubric description	Does not meet expectations (1 point)	Meets expectations (2 points)	Exceeds expectations (3 points)
Program Layout (Visual appeal)	- Proper naming conventions were rarely used - Internal documentation was not provided - Use of white space, indenting and braces was inconsistent	- Proper naming conventions were used most of the time - Internal documentation provided good descriptions - Use of white space, indenting and braces was good	- Proper naming conventions were used all of the time - Internal documentation provided excellent descriptions - Use of white space, indenting and braces was excellent.
Program Design	- Use of control structures was usually incorrect or inappropriate - Algorithms were inefficient - Input/Output was unclear or not formatted	- Use of control structures was correct and appropriate most of the time. - Algorithms were efficient most of the time. - Input/Output was clear and well formatted most of the time.	- Use of control structures was correct and appropriate all of the time. - Algorithms were efficient all of the time. - Input/Output was clear and well formatted all of the time.
Test data	No test data was provided	Test data was provided for most code functionality.	Test data was provided for all code functionality.
Functionality	Program did not meet any of the required functionality	Program met most of the required functionality	Program met or exceeded all of the required functionality

References

Adams, J. (2007). Alice, middle schoolers & the imaginary worlds camps. *ACM SIGCSE Bulletin*, 39(1), 307-311.

Alice (2009). Retrieved February 8, 2009 from http://alice.org/index.php?page=sponsors/sponsors.

Anderson, E., & McLoughlin, L. (2007). Critters in the classroom: a 3D computer-game-like tool for teaching programming to computer animation students. In *International Conference on Computer Graphics and Interactive Techniques: ACM SIGGRAPH 2007 educators program (*Article 7). New York: ACM.

Andrade, H. (2005). Teaching with rubrics: the good, the bad, and the ugly. *College Teaching*, 53(1), 27-31.

Anewalt, K. (2008). Making CS0 fun: an active learning approach using toys, games and Alice. *Journal of Computer Sciences in Colleges*, 23(3), 98-105.

Areias, C., & Mendez, A. (2007). A tool to help students to develop programming skills. In B. Rachev,A. Smrikarov, & D. Dimov (Eds.), *ACM International Conference Proceeding Series: Vol. 285. Proceedings of the 2007 international conference on Computer systems and technologies* (Article 89). New York: ACM.

Baker, K., (2009). Learning theory and the re-education of older software engineers. *ACM SIGITE Newsletter archive*, 6(2), 2-10.

Becker, K. (2003). Grading programming assignments using rubrics. In D. Finkel (Ed.) *Annual Joint Conference Integrating Technology into Computer Science Education: Proceedings of the 8th annual conference on Innovation and technology in computer science education* (pp. 253-253). New York: ACM.

Ben-Ari, M. (1998). Constructivism in Computer Science. In *Technical Symposium on Computer Science Education Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education* (pp. 257-261). New York: ACM.

Bers, M. (2007). *Blocks to robots: learning with technology in the early childhood classroom.* New York: Teachers College Press.

Biddle, R. & Tempero, E. (1998). Java pitfalls for beginners. *ACM SIGCSE Bulletin*, 30(2),48-52.

Bishop-Clark, C., Courte, J., & Howard, E. (2006). Programming in pairs with Alice to improve confidence, enjoyment and achievement. *Journal of Educational Computing Research,* 34(2), 213-228.

Bishop-Clark, C., Courte, J., Evans, D., & Howard, E. (2007). A quantitative and qualitative investigation of using Alice programming to improve confidence, enjoyment and achievement among non-majors. *Journal of Educational Computing Research*, 37(2), 193-207.

Brouwer, N., Muller, G., & Rietdijk, H. (2007). Educational designing with microworlds. *Journal of Technology and Teacher Education*, 15(4), 439-462.

Brown, P. (2008). Some field experience with Alice. *Journal of Computing Sciences in Colleges*, 24(2), 213-219.

Brooks, M. & Brooks, J. (1999). The courage to be constructivist. Educational Leadership. 57(3) 1-10.

Bureau of Labor Statistics (2008). Retrieved September 20, 2008 from http://www.bls.gov/news.release/ecopro.toc.htm.

Calloni, B., & Bagert, D. (1997). Iconic programming proves effective for teaching the first year programming sequence. In J. Miller (Ed.) *Technical Symposium on Computer Science Education: Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education* (pp. 262-266). New York: ACM.

Carlisle, M., Wilson, T., Humphries, J., & Hadfield, S. (2005). RAPTOR: a visual programming environment for teaching algorithmic problem solving. In *Technical Symposium on Computer Science Education: Proceedings of the 36th SIGCSE technical symposium on Computer science education* (pp. 176-180). New York: ACM.

Cheung, J., Ngai, G., Chan, S., & Lau, W. (2009). Filling the gap in programming instruction: a text-enhanced graphical programming environment for junior high students, In *Technical Symposium on Computer Science Education: Proceedings of the 40th ACM technical symposium on Computer science education* (pp. 276-280). New York: ACM.

CITE Demographic Data (2009). [2009 CMIS demographic data]. Unpublished raw data.

CITE Grade Distribution Data (2008). [2008 CS0 f2f grade distribution data]. Unpublished raw data.

Clements, D. (1989). *Computers in elementary mathematics education*. Englewood Cliffs, NJ. Prentice Hall.

Close, R., Kopec, D., & Aman, J. (2000). CS1: perspectives on programming languages and the breadth-first approach. *Journal of Computing Sciences in Colleges*, 15(5), 228–234.

Cohoon, J. (2003). Must there be so few?: Including women in CS, In International Conference on Software Engineering: Proceedings of the 25th International Conference on Software Engineering (pp. 668-674). Washington, D.C: IEEE Computer Society.

Computer Curricula (1978). ACM Curriculum Committee on Computer Science. Curriculum '78: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, 22(3).

Computer Curricula (2001). Retrieved October 15, 2008 from http://www.acm.org/education/education/education/curric_vols/cc2001.pdf.

Conway, M. (1997). Alice: Easy-to-Learn 3D Scripting for Novices. (Doctoral dissertation, University of Virginia, 1997).

Conway, M., Audia, S., Burnette, T., Cosgrove, D., & Christiansen, K (2000). Alice: lessons learned from building a 3D system for novices. In *Conference on Human Factors in Computing Systems: Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 486-493). New York: ACM.

Cooper S., Dann, W., & Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5), 108-117.

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. In *Technical Symposium on Computer Science Education: Proceedings of the 34th SIGCSE technical symposium on Computer science education* (pp. 191-195). New York: ACM.

CRA Statistics retrieved Oct 10, 2009 from http://www.cra.org/info/taulbee/women.html.

Crotty, M. (1998). *Foundations of Social Research: Meaning and Perspective in the Research Process.* Thousand Oaks, CA: Sage Publications.

Dale, N. (2006). Most difficult topics in CS1: results of an online survey to educators. *Inroads: the SIGCSE Bulletin*, 38(2), 49-53.

de Raadt, M., Watson, R., & Toleman (2004). Introductory programming: what's happening today and will there be any students to teach tomorrow? In R. Lister & A. Young (Eds.) *ACM International Conference Proceeding Series: Vol. 57. Proceedings of the sixth conference on Australasian computing education* (pp. 277-282). New York: ACM.

Dingle, A. & Zander, C. (2000). Assessing the ripple effect of CS1 language choice. *Journal of Computing Sciences in Colleges*, 16(2), 85-93.

diSessa, A. (2000). *Changing minds: computers, learning, and literacy*. Cambridge, MA, MIT Press.

Dodds, Z., Libeskind-Hadas, R., Alvarado, C., & Kuenning G. (2008). Evaluating a breadth-first CS 1 for scientists. In *Technical Symposium on Computer Science Education: Proceedings of the 39th SIGCSE technical symposium on Computer science education* (pp. 266-270). New York: ACM.

Dougherty, J. (2007). Concept visualization in CS0 using Alice. *Journal of Computing Sciences in Colleges*, 22(3), 145-152.

Driscoll, M., (2000). *Psychology of learning for instruction.* Needham Heights, MA, Pearson.

Edwards, L.D. (1995). Microworlds as representations. In A. diSessa, C. Hoyles, R. Noss, & L. Edwards (Eds.), *Computers and exploratory learning* (pp. 127-154). New York: Springer-Yerlag.

Ehlert, A., & Schulte, C. (2009). Empirical comparison of objects-first and objects-later. In International Computing Education Research Workshop: Proceedings of the fifth international workshop on Computing education research workshop (pp. 15-26). New York: ACM.

Ellis, H. (2002). Andragogy in a web technologies course. In *Technical Symposium on Computer Science Education Proceedings of the 33rd SIGCSE technical symposium on Computer science education* (pp. 206-210). New York: ACM.

Fischer, G., Giaccardi, E., Ye, Y., Sutcliffe, A.G., & Mehandjiev, N. (2004). Meta-design: a manifesto for end-user development. *Communications of the ACM*, 47(9), 33-37.

Fry, C. (1997). Programming on an already full brain. *Communications of the ACM*, 40(4), 55-64.

Garner, S., Haden, P., & Robins, A. (2005). My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. *In A. Young & D. Tolhurst (Eds.) ACM International Conference Proceeding Series; Vol. 106. Proceedings of the 7th Australasian conference on Computing education* (pp. 173-180). Darlinghurst, Australia: Australian Computer Society, Inc.

Garris, R., Ahlers, R., & Driskell, J. (2002). Games, motivation, and learning: a research and practice model. *Simulation & Gaming*, 33(4), 441-467.

Gay, L. & Airasian, P. (2003). *Educational Research*. Columbus, Ohio:Pearson.

Goldweber, M., Bergin, J., Lister, R. & McNally, M. (2006). A comparison of different approaches to the introductory programming course. In D. Tolhurst, & S. Mann (Eds.) *ACM International Conference Proceeding Series: Vol. 165. Proceedings of the 8th Australian conference on Computing education* (pp. 11-13). Darlinghurst, Australia: Australian Computer Society, Inc.

Gomes, A. & Mendes A. (2007). An environment to improve programming education. In B. Rachey, A. Smrikarov, & D. Dimov (Eds.) *ACM International Conference Proceeding Series: Vol. 285. Proceedings of the 2007 international conference on Computer systems and technologies* (Article 8). New York: ACM.

Gross, P., & Powers, K. (2005). Evaluating assessments of novice programming environments. In *International Computing Education Research Workshop: Proceedings of the first international workshop on Computing education research* (pp. 99-110). New York: ACM.

Hadjerrouit, S. (1998). Java as first programming language: a critical evaluation. *ACM SIGCSE Bulletin*, 30(2), 43-47.

Henriksen, P. & Kölling, M. (2004). Greenfoot: combining object visualization and interaction. In *Conference on Object Oriented Programming Systems Languages and Applications: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (pp. 73-82). New York: ACM.

Hogle, J. (1995). Computer worlds in education: catching up with Danny Dunn. (ERIC Document No. ED425738)

Hoover, W. (1996). The Practice Implications of Constructivism. *SEDL Letter* , 9(3) retrieved January 15, 2010 from http://www.sedl.org/pubs/sedletter/v09n03/practice.html.

Hu, H. (2008). A summer programming workshop for middle school girls. *Journal of Computer Sciences in Colleges*, 23(6), 194-202.

Hundhausen, C., & Brown, J. (2005). Personalizing and discussing algorithms within CS1 studio experiences: An observational study. In *International Computing Education Research Workshop: Proceedings of the first international workshop on Computing education research* (pp. 45-56). New York: ACM.

Jonassen, D. (1991). Objectivism versus constructivism: Do we need a new philosophical paradigm? *Educational Technology Research & Development*, 39(3), 5-14.

Jonassen, D. (2002). Learning as activity. *Educational Technology*, 42(2),45-51.

Kahn, K. (1996). Drawings on napkins, video game animation, and other ways to program computers. *Communications of the ACM*, 39(8), 49-59.

Kato, Y. (2006). An XML-based microworld simulator for business modeling education. In *Proceedings of the Fourth International Conference on Creating, Connecting and Collaborating through Computing* (pp. 232-239). Washington, DC: IEEE Computer Society.

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

Kelleher, C., Pausch, R. & Kiesler, S. (2007). Storytelling Alice motivates middle school girls to learn computer programming. In Conference on Human Factors in Computing Systems: Proceedings of the SIGCHI conference on Human factors in computing systems (pp. 1455-1464). New York: ACM.

Kieran-Greenbush, K. (1991). Reaching the adult learner: adult learner and computer training. In *Proceedings of the 19th annual ACM SIGUCCS conference on User services* (pp. 177-182). New York: ACM.

Klassen, M. (2006). Visual approach for teaching programming concepts. *9th International Conference on Engineering Education*, July 2006. T1A-1-T1A-6.

Knowles, M. (1970). *The Modern Practice of Adult Education. Andragogy versus pedagogy*, Englewood Cliffs: Prentice Hall/Cambridge.

Kolling, M. & Rosenberg, J. (2000). Objects first With Java and BlueJ. In S. Haller (Ed.) *Technical Symposium on Computer Science Education: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education* (pp. 429-429). New York: ACM.

Koski, M., Kurhila, J., & Pasanen, T. (2008). Why using robots to teach computer science can be successful theoretical reflection to andragogy and minimalism. *In International Conference on Computing Education Research: Proceedings of the 8th International Conference on Computing Education Research* (pp. 32-40). New York:ACM.

Kubota, R. (2001). Voices from the margin: second and foreign language teaching approaches from minority perspectives. *The Canadian Modern Language Review*, 54(3), 394-412.

Lahtinen, E., AlaMutka, K., & Järvinen, H. (2005). A study of the difficulties of novice programmers. In *Annual Joint Conference Integrating Technology into Computer Science Education: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education* (pp. 14-18). New York: ACM.

Lui, A., Kwan, R., Poon, M. & Cheung, Y. (2000). Saving weak programming students: applying constructivism in a first programming course. *Inroads: the SIGCSE Bulletin*, 36(2), 72-76.

Lorenzen, T & Sattar, A. (2008). Objects first using Alice to introduce object constructs in CS1. *SIGSE Bulletin*, 40(2), 62-64.

Markoff, J. (2009). Computer science programs make a comeback in enrollment. Retrieved November 10, 2009 from http://www.nytimes.com/2009/03/17/science/17comp.html

Masterson, T. & Meyer, R. (2001). SIVIL: A true visual programming language for students. *Journal of Computer Sciences in Colleges*, 16(4), 74-86.

McGrath, V. (2009). Reviewing the evidence on how adult learners learn: An examination of Knowles' Model of Andragogy. In T. Fleming (Ed.) *The Adult Learner 2009* (pp. 99-110). AONTAS, Dublin.

McKinney, A. (2003). A recent radical graphical approach to programming. *Journal of Computer Sciences in Colleges*, 18(6), 28-35.

Meyer,R. & Masterson, T. (2000). Towards a better visual programming language: Critiquing prograph's control structures. *Journal of Computer Sciences in Colleges*, 15(5), 183-196.

Moskal, B., Lurie, D., & Cooper, S. (2004). Evaluating the effectiveness of a new instructional approach. In *Technical Symposium on Computer Science Education: Proceedings of the 35th SIGCSE technical symposium on Computer science education* (pp. 75-79). New York: ACM.

Muhlhauser, M. & Gecsei, J. (1996). Services, frameworks and paradigms for distributed multimedia applications. *IEEE Multimedia*, 3(3), 48-61.

Mullins, P., Whitfield, D., & Conlon, M. (2009). Using Alice 2.0 as a first language. *Journal of Computing Sciences in Colleges*, 24(3), 136-143.

Murphy, L., Lewandowski,G., McCauley,R.,Simon,B., Thomas, L., & Zander, C. (2008). Debugging: the good, the bad, and the quirky–a qualitative analysis of novices' strategies. In *Technical Symposium on Computer Science Education: Proceedings of the 39th SIGCSE technical symposium on Computer science education* (pp. 163-167). New York: ACM.

Myers, B. (1986). Visual programming, programming by example, and program visualization: a taxonomy. In M. Mantei & P. Orbeton (Eds.) *Conference on Human Factors in Computing Systems: Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 59-66). New York: ACM.

Olan, M. (2003).Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, 19(2), 319-327.

Open World Learning. retrieved January 15, 2010 from http://www.openworldlearning.org/mia/welcome.htm.

Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic.

Papert, S. (1980b) . Computer-based microworlds as incubators for powerful ideas. In R. Taylor (ed.) *The computer in the school: Tutor, tool, tutee* (pp. 203-210). New York. Teacher's College Press.

Papert, S. (1991). *Situating Constructionism*. In I. Harel & S. Papert (Eds.), Constructionism (pp. 1-12). Norwood, NJ: Ablex.

Pattis, R. (1981). Karel *the Robot: A gentle introduction to the art of programming. (2nd ed.).* Chicago: John Wiley & Sons.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., & Bennedsen, J. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, 39(4), 204-223.

Petre, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communication of the ACM*, 38(6),55-70.

Piaget, J. (1973). *The Child and Reality*. Harmondsworth: Penguin.

Powers, K., Ecott, S. & Hirschfield, L. (2007). Through the looking glass: Teaching CS0 with Alice. *ACM SIGCSE Bulletin*, 39(1), 213-217.

President's Information Technology Committee (2005). Computational Science: Ensuring America's Competitiveness. Presidents report, June 2005.

Reiber, L. (2004). Microworlds. In D. Jonassen (Ed.) *Handbook of Research on education communications and technology* (pp. 583-603). London. Lawrence Erlbaum Associates.

Rigas, G., Carling, E., & Brehmer, B. (2002). Reliability and validity of performance measures in microworlds. *Intelligence,* 30(2002), 463-480.

Robertson, J. (2007). [Summer 2007 CS0 Loop and Function Analysis]. Unpublished raw data.

Rodger, S. (2002). Introducing computer science through animation and virtual worlds *ACM SIGCSE Bulletin*, 34(1), 186-190.

Rodger, S., Hayes, J., Lezin, G. Qin, H., Nelson, D. ,Tucker, R., & Lopez, M. et al. (2009). Engaging middle school teachers and students with Alice in a diverse set of subjects. ACM SIGCSE Bulletin, 41(1), 271-275.

Roussou, M. (2004). Learning by doing and learning through play: an exploration of interactivity in virtual environments for children. *Computers in Entertainment*, 2(1), 10-20.

Ruehr, F. (2008). Tips for teaching types and functions. In *International Conference on Functional Programming: Proceedings of the 2008 international workshop on Functional and declarative programming in education* (pp. 79-90). New York: ACM.

Scott, J. (1987). Training adult learners - a new face in end users. In *Proceedings of the 15th annual ACM SIGUCCS conference on User Services* (pp. 54-55). New York: ACM.

Seidman, R. (2009). Alice first: 3D interactive game programming. In *Annual Joint Conference Integrating Technology into Computer Science Education Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education* (pp. 345-345). New York: ACM.

Seymour, E. & Hewitt, N. (1997*). Talking about leaving*. Boulder: Westview Press.

Soloway, E. & Spohrer, J. (1989). *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, New Jersey.

Sung, K., & Shirley, P. (2004). Algorithm analysis for returning adult students. *Journal of Computing Sciences in Colleges*, 20(2), 62-69.

Sykes, E. (2007). Determining the effectiveness of the 3D Alice programming environment at the computer science I level. *Journal of Educational Computing Research*, 36(2), 223-244.

Tiobe (2009). Retrieved April 2, 2009 from http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html/

University data (2009). Retrieved January 21, 2009 from http://www.umuc.edu/ip/ataglance.shtml

Venit, S. & Drake, E. (2009). *Prelude to Programming Concepts & Design*. Addison-Wesley, Boston, Massachusetts.

Vesgo, J. (2007). Continued drop in CS bachelor's degree production and enrollments as the number of new majors stabilizes. *Computing Research News*, 19(2).

Widnall, S . E . (1988). AAAS presidential lecture:
Voices from the pipeline. *Science*, 241,1740-1745.

Winslow, L. (1996). Programming pedagogy: a psychological overview. *ACM SIGCSE Bulletin* , 28(3),17-22.

Xing, C. (2008). Enhancing the learning and teaching of functions through programming in ML. *Journal of Computing Sciences in Colleges*, 23(4), 97-104.

Curriculum Vita

NAME: James A. Robertson

PERMANENT ADDRESS: Severna Park, MD, 21146

PROGRAM OF STUDY: Instructional Technology

DEGREE AND DATE TO BE CONFERRED: Doctor of Education, May 2010

Collegiate Institutions Attended	Dates	Degree	Date of Degree
Towson University Major: Instructional Technology	2003-2010	Ed.D.	May, 2010
University of Dayton Major: Electro-Optical Engineering	1990-1995	M.S.	May, 1995
University of Houston-Clear Lake Major: Electro-Optics	1985-1989	B.S.	May, 1989
Salisbury University Major: Medical Technology	1981-1983	B.S.	May, 1983

Professional Publications:

Sadera, W, Robertson, J., Song, L., & Midon, N. (2009). Success in online learning and the role of community. *Journal of Online Learning and Teaching*, 5(2), 1-8.

Bennett, K. & Robertson, J. (2009). Multimodal signature file formats and performance in computational environments. *Proceedings of the SPIE*, 7324 (2009), 1-12.

Professional Positions held:

University of Maryland University College, Adelphi, MD Academic Director and Collegiate Faculty	April 2001 - present
Oracle Corporation, Arlington, VA Principal Consultant	Jan 2000 – April 2001
IIT Research Institute, Lanham, MD Research Engineer	June 1989 – Jan 2000
University of Texas Medical Branch, Galveston, TX Medical Technologist	May 1983- June 1989