

RoboBUG: A Game-Based Approach to Learning Debugging Techniques

by

Michael A. Miljanovic

A thesis submitted in partial fulfillment
of the requirements for the degree of

Masters of Science

in

Computer Science

University of Ontario Institute of Technology

Supervisor: Dr. Jeremy Bradbury

March 2015

Copyright © Michael A. Miljanovic, 2015

Abstract

Debugging is the systematic process of finding and fixing errors (i.e. bugs) in a computer program, and it is considered a critical skill that should be acquired early in a programmer’s career. In order to learn debugging techniques, it is necessary to understand what bugs are, how they work, how to find them, and how to fix them. Unfortunately, the process of learning debugging is often both difficult and tedious to novices, and is not always adequately covered in the undergraduate computer science curriculum. As an alternative to traditional approaches for learning debugging (e.g. labs, written assignments) we propose the use of a game-based approach for introducing debugging techniques. Our approach is intended to create a more enjoyable learning experience that remains equally as effective as traditional methods at learning debugging concepts. Specifically, we designed a game called RoboBUG in which a player assumes the role of a futuristic programmer trying to find “bugs” in a mechanical suit. We then conducted an evaluation to assess novice programmers playing the RoboBUG game and novices who instead completed a traditional written assignment. Participants were assessed based on their achievement of debugging learning outcomes and on their qualitative interpretation of the learning activity. Our results found that study participants reported a positive attitude towards using games for learning, and those who played the RoboBUG game believed it to be more fun and engaging than previous experiences with written assignments.

Keywords: bugs, debugging, game-based learning, novice programmers, serious games, programming, education

Acknowledgements

I'd like to thank my supervisor, Dr. Jeremy Bradbury, for his help and guidance during my studies as a Master's student under his supervision.

I'd like to thank my thesis committee, including Dr. Lennart Nacke, Dr. Roland van Oostveen, and Dr. Christopher Collins, for their feedback and contributions to my thesis.

I'd like to thank my girlfriend, Veronica, for her patience and understanding during my long hours of work and difficult scheduling.

I'd like to thank all of the prototype testers whose feedback helped me to make RoboBUG more accessible.

I'd like to thank my family and friends for their support and encouragement in everything I seek to accomplish.

Contents

Abstract	i
Acknowledgements	iii
Contents	iv
List of Figures	vii
List of Tables	viii
1 Introduction	1
1.1 Motivation	1
1.2 Problem	3
1.3 Thesis Statement	4
1.4 Contributions	4
1.5 Organization of Thesis	5
2 Background	6
2.1 Overview	6
2.2 Debugging	6
2.2.1 Bugs	8
2.2.2 Practices and Techniques	9
2.2.3 Debugging Tools	13
2.3 Computer Science Education	14
2.3.1 Educational Terminology	15
2.3.2 Traditional Approaches for Learning Debugging	15
2.3.3 Non-traditional Approaches for Learning Debugging	18
2.4 Serious Games and Game-based Learning	21
2.4.1 Designing Game-based Learning	23
2.4.2 Evaluating Game-based Learning	25
2.5 Summary	27

3	The RoboBUG Game	29
3.1	Overview	29
3.2	Game Design	29
3.2.1	Prototype Testing	31
3.2.2	Learning Objectives	33
3.3	Gameplay Mechanics	36
3.4	Level Design	38
3.4.1	Level 1 - Code Tracing	39
3.4.2	Level 2 - Testing	39
3.4.3	Level 3 - Print Statements	41
3.4.4	Level 4 - Divide and Conquer	41
3.4.5	Level 5 - Breakpoints	42
3.5	Summary	44
4	Study Methodology	45
4.1	Overview	45
4.2	Experimental Setup	46
4.2.1	Participants	46
4.2.2	Experiment Environment	46
4.2.3	Data Management	47
4.2.4	Experimental Documents and Tools	47
4.3	Procedure	49
4.3.1	Introduction	49
4.3.2	Learning Activity	50
4.3.3	Quiz	51
4.3.4	Survey	51
4.4	Data Analysis	51
4.4.1	Analysis of Game and Assignment Performance	52
4.4.2	Analysis of Quiz Results and Learning Outcomes	52
4.4.3	Analysis of Exit Survey Data	53
4.5	Ethical Considerations and Risks	53
4.6	Summary	54
5	Results	55
5.1	Overview	55
5.2	Activity Performance	56
5.3	Quiz Performance	57
5.3.1	Traditional Assignment Group	58
5.3.2	Game group	58
5.3.3	Game vs. Traditional	59
5.4	Survey Feedback	59
5.4.1	Impressions of the Learning Activity	61
5.5	Threats to Validity	65

5.5.1	Internal Validity	65
5.5.2	Construct Validity	66
5.5.3	External Validity	66
5.5.4	Conclusion Validity	67
5.6	Summary	68
6	Conclusions	69
6.1	Discussion	69
6.2	Limitations	70
6.3	Future Work	72
6.4	Conclusions	72
	Bibliography	74
A	Appendix 1: Powerpoint Slides	82
B	Appendix 2: Traditional Group Written Assignment	97
C	Appendix 3: Evaluation Test	110
D	Appendix 4: Feedback Survey	115
E	Appendix 5: RoboBUG Media Credits	123

List of Figures

2.1	<i>A program execution as a succession of states.</i> [Zel09]	8
2.2	CMeRun System Original and Augmented Code [Eth04]	13
2.3	Results from Fitzgerald et al. “[<i>Student</i>] <i>Perceptions: What is the hardest part of debugging?</i> ” [FMH ⁺ 10]	17
2.4	Screenshots of TALM [MM10].	20
2.5	Educational Games Model for Self-Learning Introductory Programming [IMMJ10]	23
3.1	First prototype of RoboBUG	30
3.2	RoboBUG Debugging Game	34
3.3	RoboBUG Tools	35
3.4	RoboBUG Level Introduction	36
3.5	RoboBUG Level End Screen	37
3.6	RoboBUG Level 1 - Code Tracing	38
3.7	RoboBUG Level 2 - Testing	39
3.8	RoboBUG Level 3 - Print Statements	40
3.9	RoboBUG Level 4 - Divide and Conquer	42
3.10	RoboBUG Level 5 - Breakpoints	43
4.1	Experimental Procedure	49
5.1	Average Time per Level/Question	56
5.2	Average Score per Question by Group	57
5.3	Total Quiz Scores by Age and Group.	58
5.4	Average Score per Quiz Question by Year.	60
5.5	Study Participants’ Familiarity with Debugging Techniques.	60

List of Tables

5.1	Quiz Averages by Age and Learning Activity.	59
5.2	Average Rating of Levels by Learning Activity.	61
5.3	Results of Survey Feedback Questions.	62

Chapter 1

Introduction

1.1 Motivation

Novice programmers who seek to write reliable, high quality source code need to be able to efficiently identify and repair bugs or errors. The process of identifying bugs, known as debugging, has been shown to take up to 50% of the time of a large software project [Chu09]. Furthermore, programmers who work independently must spend an even more substantial amount of time debugging their own code [Chu09]. This means that programmers who lack the ability to efficiently debug their source code may find themselves spending hours trying to find small errors while spending little time actually writing new source code and implementing new functionality. For a beginner, this can mean that very little progress is made over time as too much time is being spent trying to figure out why their code doesn't work and how to fix it.

The ability to debug code is not easily acquired, and experts have a significant advantage over novices who lack the support or experience with debugging techniques [CB14]. An expert programmer not only has greater experience with debugging

source code, but also an increased familiarity with common errors and what causes them. In comparison, a beginner will encounter unfamiliar errors more frequently and may often be unsure as to how to fix an encountered bug. This creates frustration among novices, leading them to resort to ad-hoc or trial-and-error methods, and even reducing or eliminating their motivation towards the pursuit of computer science [Eth04].

Therefore, an efficient method of helping students learn debugging is a necessity for aiding novice programmers. That method must also minimize frustration and demonstrate important debugging techniques in a manner that novices will find engaging. This may require diverging from traditional assignments and tests that students use to learn debugging, and instead investigating the value of different approaches such as contests or games. For example, competitive exercises [Bry11, EPDJ07] have shown some success in engaging university students who are learning introductory programming¹.

Game-based learning has been proven effective for learning general programming concepts [IMMJ10, MTJV09], which suggests that it may also prove to be useful in debugging education. Currently available games do not have a specific focus on debugging, however, which presents an opportunity to create a game that aids in learning debugging. We have created a game that is designed specifically for students to learn debugging techniques that are applicable to real world software debugging. We believe that this style of game would also prove useful in other areas of computer science, including but not limited to computer networks, mobile development, and parallel programming.

¹However, these approaches do not help the individual programmer who is learning computer science without peer interaction.

1.2 Problem

The creation of a game to assist of learning debugging presents a number of obstacles, including the selection of **learning material**, the **game design**, and the **efficiency evaluation** of the game as a learning tool. A game that is designed to helping with learning the relevant material and is evaluated to be at least as efficient as traditional learning methods can be considered a contribution to the realm of academia.

1. Learning Material

The game must be able to help with learning relevant debugging concepts in such a way that the players retain information after the game's completion. These concepts need to be identified and incorporated into the game itself. There is a limited amount of material that can be introduced before overloading the player with information, so only a subset of debugging techniques will be used in a single game. One of the primary challenges of developing this game will be to select the debugging techniques that are most important to introduce to novice programmers. This learning material must not only be something taught by traditional learning methods, but also material that can be introduced using the limitations of a video game medium.

2. Game Design

Once the learning material has been chosen, a game must be created that appropriately helps the game's players learn the material. This game will have a large number of requirements, including not only featuring an accurate representation of debugging techniques, but also the ability to immerse players into an environment where they will feel like they are playing a game and not completing a tedious debugging task. The game must allow students to learn material in a time-efficient manner, and also include examples that both demonstrate the

application of debugging techniques as well as fit into the theme of the game.

3. Efficiency Evaluation

The value and contribution of the game as a learn tool can only be assessed using a proper evaluation. Quantitative and/or qualitative methods need to be used to evaluate the game-based learning with respect to achieving learning objectives and with respect to users' feelings and impressions of the game. This evaluation may be done in isolation or may be done in comparison to traditional learning approaches.

1.3 Thesis Statement

Thesis Statement: *A game-based approach is a viable approach for the purposes of learning debugging techniques by novices. Introducing new materials through a game-based approach can be a positive experience for students.*

This thesis presents a game that is designed specifically for the purpose of introducing debugging techniques to novice programmers. Ultimately, we seek to make an alternative to traditional written methods that novices will strongly prefer and learn from equally well.

1.4 Contributions

This thesis offers two main contributions to the field of computer science education:

- The RoboBUG Game - We have created a game-based learning tool that helps students learn proper debugging techniques. RoboBUG can be distributed to educational institutions, providing them with a new approach to learning debugging. Since the source code for RoboBUG will be released as an open source

software project, it can also be modified and adapted to extend its usability beyond the realm of debugging. Instructors may find it helpful to alter the RoboBUG game to help their students learn various software topics in addition to debugging.

- The Evaluation - We have conducted a study to examine the use of RoboBUG as a learning tool. This study provided insight into students' impressions and perspectives of game-based learning, and identified problem areas in the design of educational games. Our results may help future research in the realm of game-based learning, as well as guide game developers towards making more effective and enjoyable educational games.

1.5 Organization of Thesis

This chapter has outlined the motivation and problem of learning debugging with games, as well as the thesis statement and contributions. The remainder of the thesis includes:

- Chapter 2: A literature review of work done in computer science education, approaches for learning debugging, as well as gamification and game-based learning.
- Chapter 3: The design of the RoboBUG game-based learning tool.
- Chapter 4: The setup and procedure used to evaluate the RoboBUG game.
- Chapter 5: The results of the experiment that was outlined in Chapter 4.
- Chapter 6: A discussion of the conclusions, contributions, and limitations to the thesis.

Chapter 2

Background

2.1 Overview

This chapter investigates related research on the subjects of debugging, debugging education, and game-based learning. In the first section, we look at the fundamental topics covered in debugging literature and identify the key concepts that need to be introduced to novice programmers. In the second section, we examine debugging as it is currently being taught, and we also discuss studies that investigate the effectiveness of various debugging learning approaches. Finally, the third section introduces the subject of game-based learning with examples from computer science. We also describe how to design and evaluate a game that is intended to help students learn debugging.

2.2 Debugging

Debugging is “*the methodical process of finding and reducing the number of bugs in a program*” [BF14]. This fundamental process is critical to the design of efficient code,

as a program littered with errors is likely to run slowly, give incorrect results, or not run at all. Although formal debugging processes and tools exist, many programmers first learn to debug their programs informally, sometimes with strategies as simple as trial-and-error. The importance of learning proper debugging comes from a need to scale debugging to large programs and reduce the extensive amount of time that is dedicated to debugging rather than writing code. The general subject of debugging covers a broad range of smaller topics, including different types of bugs, formal practices and techniques for debugging, and software debugging tools. In this section, we will examine each of these topics in order to properly identify what is most important for learning, and to determine what elements can be implemented in our study.

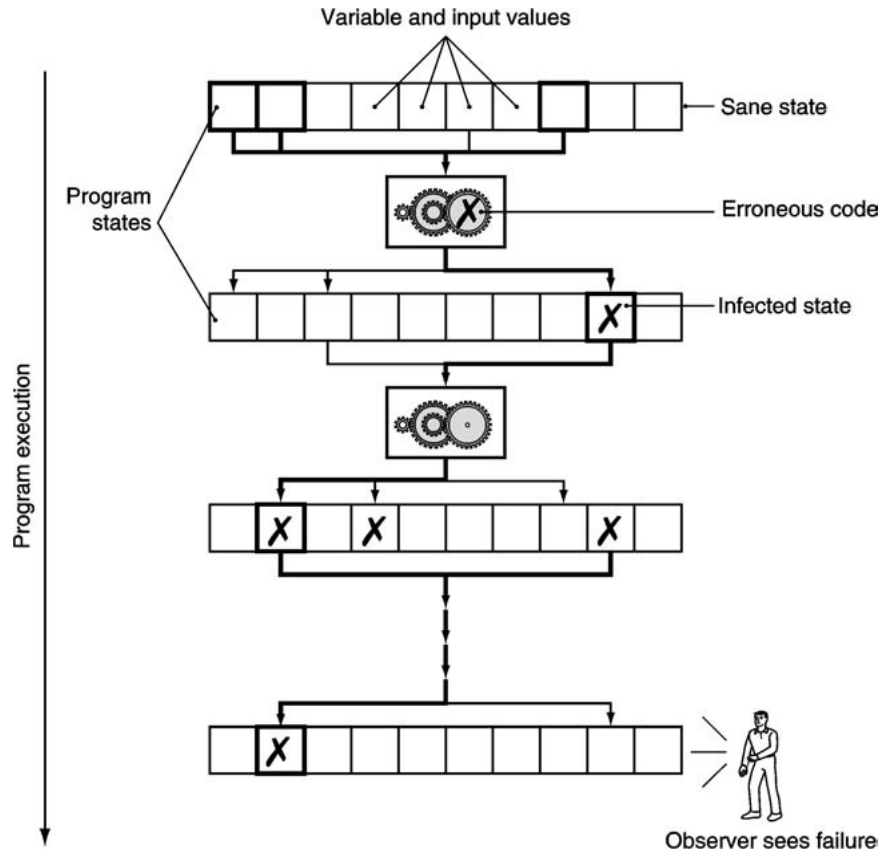


Figure 2.1: “A program execution as a succession of states. Each state determines the following states, and where from defect to failure errors propagate to form an infection chain.” [Zel09]

2.2.1 Bugs

A **bug** refers to incorrect program code, states, or execution that leads to some kind of problem [Zel09]. A bug in a program code is a defect, which leads to an infection in the program state, causing failure during execution (see **Figure 2.1**). The terms bug, defect, error, and fault are often used interchangeably, and ultimately all refer to any kind of error in programming. A bug is often the result of code that contains mistakes, but can also occur due to other issues such as a program receiving invalid input. Once a bug is identified, it is often very easy to fix; however, the main difficulty of debugging is the ability to find all of the bugs that cause a program to fail. Bugs

can be categorized into three different types.

1. **Syntax errors** are any errors that prevent a compiler or interpreter from parsing a statement. Examples of syntax errors include missing semicolons, extra brackets, or misspellings of variable names. These errors are not difficult to find due to the fact that compilers will often specifically catch and identify them during compilation. As a result, syntax errors are the least common errors found in student written code [HLB12].
2. **Data errors** are input errors caused by data that is different from what the program expects, or caused by the program processing the wrong data. This kind of problem does not always mean that the program itself is incorrect; it may be due to the data not being formatted in a way that the program can read it. For example, a program that is designed to read a file filled with integers will encounter a data error if that file contains floating point numbers.
3. **Logic errors** are semantic errors that cause incorrect computations or program behaviors. Unlike syntax and data errors, it is very difficult to check for these types of errors using software. A program might encounter a logic error if, for example, it tries to access the first element of an array, but doesn't check to make sure the array is non-empty. As logic errors are the most common type of error, they often require developers to apply multiple types of manual techniques to identify and fix. We will exclusively be using logic errors for the purposes of this study.

2.2.2 Practices and Techniques

The process of debugging takes several steps, and a variety of techniques can be applied to a program in order to effectively find and repair bugs. In order to debug

a program, it is necessary that the program can be **tested**; that is, executed with intent to fail. Doing this requires the creation of **test cases**, which are sets of input designed to execute the functionality of the program or reproduce a problem. However, discovering no failures in testing does not imply there are no defects: when a bug is known to exist, testing can show its presence, but not a bug's absence. Therefore, testing can only be used to provide confidence in the correctness of the program.

A test case that fails allows for isolation of a bug inside of the source code; applying a **divide and conquer** strategy of testing leads to the section of code containing the bug to be identified in order to reduce the area that needs to be searched for bugs. For example, if several test cases are run and they only fail when a particular function is executed, the developer learns that a bug exists within that function. This approach is not foolproof, as a cause and effect chain may exist that leads to program failure due to a bug that doesn't become visible until a later time in the program's execution. It's possible that a bug only occurs when a specific combination of functions are run in some sequence. These chains of cause and effect may be difficult to discover manually, but some automated techniques exist that aid in their detection.

Manual vs. Automated Techniques

Manual debugging involves a developer, sometimes with the aid of debugging tools, attempting to find and repair bugs without the aid of other software. This can involve manual testing and execution of the code, or **code tracing**, in which the source code is manually traced line-by-line and variable values are tracked to ensure their correctness. Many of the tasks that can be completed manually can instead be **automated**, saving a large amount of time for the programmer, particularly with programs that have large amounts of source code [Zel09]. For example, the comparison of variables

to correct values can be automated in the form of assertions inserted into the program, which are lines of code that confirm the state of a program’s execution. Other techniques that can be automated include program slicing, observing program states, anomaly detection, and identification of cause and effect chains [Zel09]. Although there are many tools (see **Section 2.2.3**) available to help with bug triage, localization, validation, and even confirmation, generating repairs remains a predominantly manual, and thus expensive process [LFW13]. Some exceptions to this include genetic algorithms used in automatic bug repair [KJB13].

Static vs. Dynamic Techniques

The techniques associated with debugging can generally be split into two different categories:

1. **Static techniques** explore abstractions of all possible program behaviors, and do not require code to be tested or executed [HP04]. It is common for this to take the form of manual **code review**, however static techniques range in complexity and effectiveness at bug detection. Automatic static analysis tools such as Lint or FindBugs are readily available to developers, but are relatively underused due to issues such as false positive bug reporting [JS13]. The most complicated and extensive technique for finding bugs is a formal proof of correctness which guarantees that no bugs exist, however the difficulty in constructing such a proof makes it infeasible for most programs. In particular, for very large systems, bugs will almost certainly exist because of the sheer size of the program, which can include over a million lines of code [BBC⁺10].
2. **Dynamic techniques** for debugging rely on the runtime behavior of a program. Debuggers are widely used by programmers and enable them to monitor

a program's execution, stop it, restart it, set breakpoints to pause at specific locations, change values in memory, or even go back to a previous program state. This flexibility greatly enhances an individual's ability to find bugs and reduces the time spent trying to find them using more tedious techniques.

<pre> if(ans == "Yes") { var1=var2+var3; } // Original Code </pre>
--

<pre> if (ans == "Yes") { cout <<"-->if(ans<" << ans << "> == ""Yes"")" << endl; var1 = var2 + var3; cout << "-->var1<" << var1 << "> = var2<" << var2 << "> + var3<" << var3 << ">;" << endl; } // Augmented Code </pre>
--

Figure 2.2: CMeRun System Original and Augmented Code [Eth04]

2.2.3 Debugging Tools

The existence of debugging tools is a great help to novice programmers, and decreases their reliance on trial-and-error or other inefficient techniques. Learning and using tools designed to test programs as simple as “*Hello World*” should not require extensive debugging knowledge, but can be a first step to improve programming practices for beginners and increase their knowledge about what debugging is and how to do it [RK97]. The development of better testing tools can alleviate the problem of producing programs that are well debugged, which are especially difficult for object oriented languages given that novices must realize the need to test each class individually. Many programming Integrated Development Environments (IDEs), such as Eclipse [Ecl] and Microsoft Visual Studio [VS1], include debuggers with them to allow developers to use dynamic debugging techniques. However, existing debuggers can be overcomplicated and take more time to learn than they do to actually use, depending

on the task at hand. There are a wide variety of different tools that exist to aid with more specific debugging challenges, for varying skill levels of users. For example, the tool CMeRun is a simple tool designed for use by novices, and allows them to observe program output, statements, and variable values as they are executed line by line [Eth04]. CMeRun requires syntactically correct code, but is able to create a new version of the same program that outputs all of the statements and values so that the programmer can make sense of what is happening (see **Figure 2.2**). This helps introduce the effects of programming errors, while keeping the interface simple and avoiding the complexity of more advanced debuggers. In addition, CMeRun serves as an example of the importance of introducing debugging to novices slowly, without overwhelming them with the extensive functionality of IDE debuggers.

2.3 Computer Science Education

Students enrolled in university-level Computer Science, Software Engineering, or Information Technology programs may start their studies with little to no programming experience and even less debugging experience. While formal training and experience with debugging both contribute to a student’s ability to debug, instructors are not always aware of how much assistance students need with debugging [Chu09]. Despite a large amount of investigation done to examine novice debugging, there are no established best practices for introducing the material [MLM⁺08]. Many students who do have some knowledge about debugging techniques apply them incorrectly or ineffectively, and do not know how to systematically search their source code for bugs. In order to appropriately address the difficulties students face with debugging, we will examine some approaches to how debugging can be taught. We consider not only how debugging has been traditionally taught using lectures, assignments, and

formal methods, but also innovative approaches that include digital media or class activities [MM10, Bry11]. By examining these approaches, we can combine the most useful aspects of traditional and innovative approaches into a multifaceted study that compares the benefits of both.

2.3.1 Educational Terminology

Before investigating the application of learning techniques to debugging, it is important to clearly define the relevant educational terminology. **Learning** can be defined as gaining “*knowledge or understanding of, or skill in, by study, instruction or investigation.*” [Moo73] In particular, our goal is to help students enrolled in their first year of computer science learn through instruction in their course tutorials or laboratories. There is a subtle difference between our desired approach and **teaching methods**, which are “*structures deliberately designed to foster learning.*” [Moo73] The important distinction is that we focus less on “*providing instruction*” than we should on “*producing learning.*” [BT95] Improving teaching methods is a secondary goal; our primary goal is to help students learn in whatever way possible. We are looking to help students using a **learning tool** that is accessible and able to help students **achieve learning outcomes**, that is, successfully acquire knowledge, understanding, and skill in debugging.

2.3.2 Traditional Approaches for Learning Debugging

Early training in formal debugging helps students to increase their debugging competency at a faster rate [CL03]. This means that debugging education should occur early in a program, so that novices are not struggling to debug their programs using improvised and haphazard methods. Unfortunately, although debugging is considered an

important part of the computer science curriculum, very few university courses offer that are exclusively related to learning debugging [oCC13, CC08], and few curricula actually emphasize its importance by offering formal training at an early stage [CL03]. Debugging methods are typically taught as part of general computer programming and software engineering courses, but the extent to which it is covered remains highly inconsistent.

Part of the issue with traditional debugging learning methods is that standard programming textbooks have sparse coverage of debugging, and there is no set of established best practices or pedagogies (educational theories) to guide instructors [FMH⁺10]. Novices struggling with debugging may ultimately resort to inefficient techniques such as random modification or non-functional reformatting, which only serves to further confuse and frustrate the programmer. Struggling programmers often seek information or learning materials online, but there is a lack of available debugging information that is geared specifically for helping novices about how to debug [CB14]. Novices often find that debugging error messages are unclear [NPM08], and a lack of experience with proper testing techniques contributes to the problem of efficiently changing their source code. It may be that there is an assumption that students will simply ‘pick up’ debugging skills as a by-product of learning debugging techniques, but this risks leading students to develop their own ineffective strategies.

REPORTED DIFFICULTY WITH TROUBLESHOOTING STAGES

Troubleshooting stage	Subjects who found this stage most difficult
Understanding the code	4 (19%)
Testing	3 (14%)
Finding the problem	12 (57%)
Fixing the problem	2 (10%)

Figure 2.3: Results from Fitzgerald et al. “[Student] Perceptions: What is the hardest part of debugging?” [FMH⁺10]

Although there are few resources to refer to regarding formal debugging learning methods, there are a number of studies [CWL13, AEH05, FMH⁺10] that examine the behaviour of novices who perform debugging activities in the classroom. Introductory programming students, regardless of skill level, are sometimes completely unaware of debugging tools and their use, and their knowledge of debugging in general is fragile. Many students who have a good understanding of programming often do not acquire skills to debug programs effectively [AEH05]. A recurring issue is the inability of students to properly understand the programs that they are trying to debug. Furthermore, for many students, the difficulty of debugging is not in repairing errors but rather in troubleshooting and finding the source of problems; they struggle to understand the system, test the system, and locate errors. More than anything else, students find the most difficult part of debugging is actually finding what is wrong (see **Figure 2.3**). Experts are more likely to try to understand a program before debugging, while novices focus on finding and fixing an error without fully comprehending program functionality. The end result is that novices enjoy the achievement of resolving bugs, but dislike the challenge of debugging, sometimes believing that

debugging skills are based on aptitude and are unable to be learned [CWL13].

2.3.3 Non-traditional Approaches for Learning Debugging

The numerous issues with traditional approaches for learning debugging have led researchers to examine some non-traditional alternatives. These alternatives include a wide variety of activities, including web based tutorials, tutoring tools, competitive classroom exercises, and some game-based approaches. Non-traditional approaches to help learn debugging find varying levels of success, but seem to generally be evaluated in a vacuum (they do not compare their results to traditional methods like we seek to do in this study). Despite the lack of comparative evaluations, these examples still provide useful data on how students can learn debugging outside of classroom lectures and written assignments.

Web-based approaches seek to remedy a number of problems with learning debugging traditionally by providing tools that are self-paced, configurable, provide immediate feedback, allow for extensive practice, and automatically assess the learner [EPDJ07]. By learning about debugging online, novices can avoid a number of problems, including a lack of properly trained instructors, lack of physical resources, diversity of student goals and skill levels, and being overwhelmed by the already large amount of material included in introductory programming courses. One advantage of web-based approaches is that while it is difficult to incorporate testing early into the computer science curriculum, offering an online option allows students to learn about debugging at their own leisure. However, online tutoring systems designed to assist novice programmers in learning debugging have not received widespread adoption, possibly stemming from a number of inherent downsides [CB14]. Systems with few applications will not be adequate at helping learners, while systems with a large number of applications may create a dependency for the learner. There is a risk that pro-

grammers will become over-reliant on these large systems, and learn more about the intricacies of the learning tool than actual debugging techniques. Once a programmer is in a situation where the system is unavailable or unable to assist them, they will be less capable than programmers who have learned proper debugging techniques. Since debugging is a skill that does not immediately follow from being able to write code, it may be better to introduce tools designed to aid in debugging after students have already learned about proper techniques [FS12].

There are also options for learning debugging with non-traditional approaches that remain in the classroom setting. In particular, competitive exercises that focus on students trying to fix bugs more efficiently than their peers have been found to increase interest in software testing, increase awareness of bugs, and decrease the percentage of time students spend debugging [Bry11, STF⁺07]. The idea behind the exercises is that students find competitive debugging programs more fun in comparison with being forced to debug their own code in isolation. The competition aspect also helps motivate students to perform better so that they can prove themselves against their peers. Although this approach has shown success, there are issues in that human intervention cannot be provided on a 24/7 basis. The exercises themselves may help students, but when a student is struggling to debug their own code and not able to attain timely assistance, they will not be practicing proper debugging techniques and instead learning that programming is difficult, confusing, and lonely [STF⁺07].

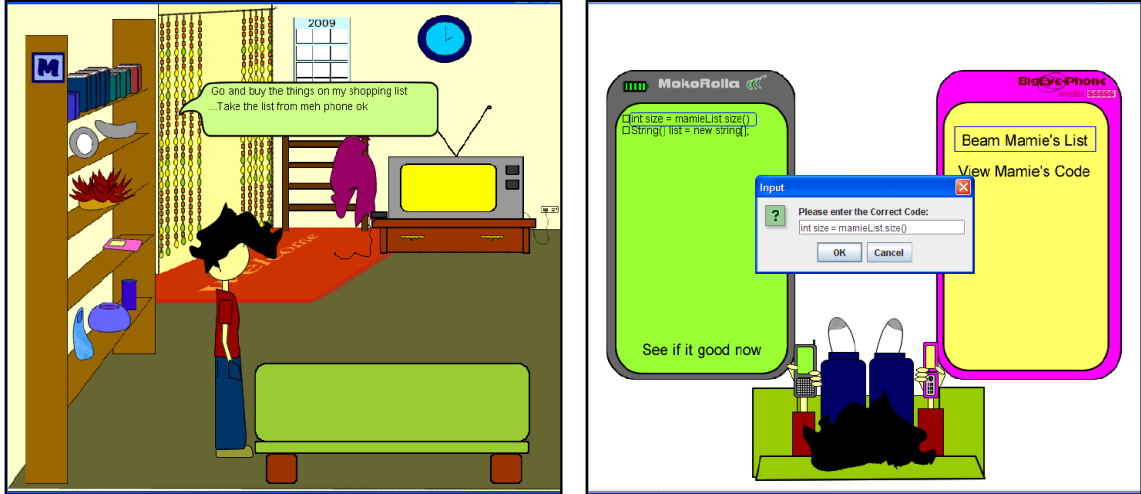


Figure 2.4: Screenshots from Mohammed and Mohan’s Trinbago Adventures of L. Macawell (TALM). On the left: “Opening scene where a Non-Player Character (NPC) talks to the hero from behind a curtain.” On the right: “Programming exercise scene showing the two phones and the player’s attempt at fixing the code on the hero’s phone.” [MM10]

Trying to find an effective, convenient, and fun approach for learning debugging points towards examples of game-based learning. Digital games attract players’ attentions for length periods of time and ask them to repeat a set of actions constantly, which is ideal for learning programming and debugging. An effective game for learning computer programming and debugging was developed by Mohammed and Mohan [MM10], which combined cultural aspects and contextualized situations where the outcomes of code snippets would be observable and make sense (see **Figure 2.4**). The game increased students’ confidence by diffusing frustration and providing hidden rewards. A strong relationship was found between the length of time a student played the game and the increase in their analytical skills. The success of this game at learning debugging suggests that game-based learning can be an effective approach, but it is unclear how it compares to traditional approaches.

2.4 Serious Games and Game-based Learning

Among the non-traditional approaches for learning debugging, the use of computer games seems to offer the greatest opportunity in terms of flexibility and level of entertainment. The use of computer games for learning (i.e. (Digital) Game-based learning) is a potentially powerful way to combine entertainment (to combat the perceived tedium of debugging) with a medium that should be able to properly help students learn the same learning outcomes that are normally taught in computer science classes. Motivation is a large problem among today’s students [Pre05], and traditional methods do not have the same ability as games to keep students engaged and interested. In addition, the average student has substantial experience with playing video games and is unlikely to struggle with gameplay in the same way as an older person [Blu07]. Although this presents problems for students in the 40+ age range, most students in our target population (first/second year university) are in the 18-25 age range and are almost certainly familiar with video games, if they do not already play them on a regular basis.

A game can be defined as “*an activity that is voluntary and enjoy-able, separate from the real world, uncertain, unproductive in that the activity does not produce any goods of external value, and governed by rules* [GAD02].” However, we are particularly interested in serious games, which are “*a mental contest, played with a computer in accordance with specific rules, that uses entertainment to further government or corporate training, education, health, public policy, and strategic communication objectives*” [MTJV09]. These definitions, as well as additional information from studies in game-based learning [Shu11, IMMJ10, GAD02], will help to identify the dimensions of games that are essential aspects that distinguish a game from more generic educational software.

There is a subtle distinction between game-based learning and gamification, which is “*the idea of using game design elements in non-game contexts*”. [DDKN11] There is an opportunity for gamification in the context of helping students learn debugging by adding game elements to the actual debugging process. This might involve computer science course instructors modifying courses to accommodate certain game elements. However, we have chosen to focus on creating a serious game with a focus on learning debugging. The difference here is that rather than trying to add entertainment to debugging, we are instead looking at designing, creating, and evaluating a new serious game that will help students to learn.

Although there is a noticeable lack of games specifically for learning debugging, there is a wealth of literature in the more generic fields of game-based learning design and evaluation. The work already done in these fields is vital for understanding how students learn course materials, as well as how to determine the efficacy of educational games. Computer programming has been an especially popular area for educational game development, with games including Code Hunt [TB14], Cleogo [CB98], Scratch [CWB11], WISE [CHYH04], Alice2 [KCC02], RoboCode [KMG06], Prog & Play [MDTV12], and StarLogo [Res96]. The success of games that help students learn introductory programming gives hope to similar possibilities for debugging games.

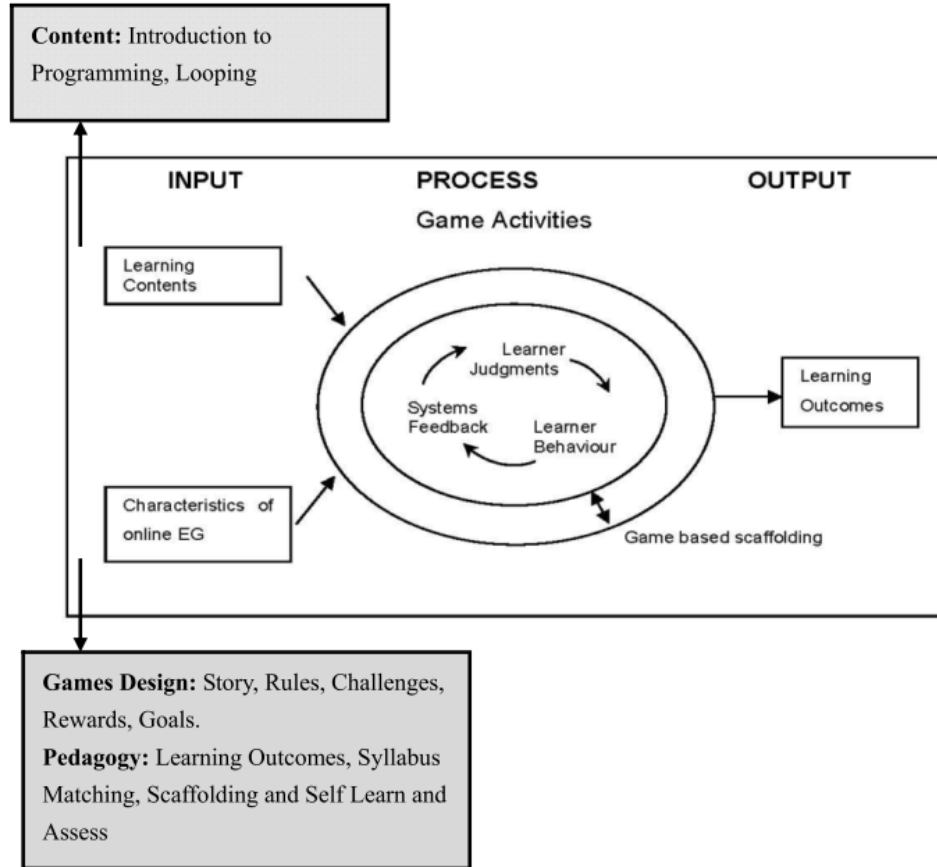


Figure 2.5: Educational Games Model for Self-Learning Introductory Programming [IMMJ10]

2.4.1 Designing Game-based Learning

Although our ultimate goal is to create a game for learning debugging, the quality of the game is critical to its efficacy as a learning tool. Numerous studies [Sia03, Shu11, GAD02] point out that learners who are motivated and engaged will perform better and achieve higher standards than those who are bored or disinterested with learning material. This is particularly relevant when considering that over 50% of students in computer science abandon their program before its completion [MTJV09]. By making a game that is engaging and fun as well as informative, we can make students more motivated to learn while simultaneously helping them learn material

that would be covered in a course lecture. In addition, we will need to acquire a sufficient understanding of the art and science behind game design so that we do not create an inferior product that contains all of the learning material but fails as a game [Eck06]. Using a process that properly combines learning content with educational game elements, we can help students to reach the appropriate learning outcomes (see **Figure 2.5**).

The first definition of a game introduces the idea of fantasy, where players escape reality and take roles that differ from their real selves. The second description mentions of rules, which have a twofold purpose: first, to allow players to perceive feedback discrepancies between their goals and their current situation, as well as to provide limited but flexible options for game play [GAD02]. A third dimension of games is the use of sensory stimuli, consisting of the sounds and graphics that appeal to game players and grab attention, as well as condition players to learn when to respond with the appropriate input or strategy [Sia03]. Challenge must be present in order for a game to feature meaningful difficulties that are neither too easy and thus trivial, or too difficult and thus impossible to overcome. Part of the learning that comes with games is the presence of mystery to players, whose curiosity inspires them to pursue knowledge. Finally, games must allow players to exhibit a great amount of control over their virtual environment, as even control over instructionally irrelevant parts of an activity create a more positive experience.

There is not a strong consensus on the specific dimensions of a game, but these six elements set the groundwork for what can be manipulated in order to make a game specifically suited to our needs. On a broader scale, there are a number of game genres that encompass the different possible games that could be designed. Although there is essentially no limit to the number of potential game genres that exist, there are seven categories that describe the vast majority: Action, Adventure, Fighting,

Role-Playing, Simulation, Sports, and Strategy [Gro07]. Some of these are clearly more suited to different learning tasks than others, and historically adventure and simulation games have been focused upon for use in game-based learning [GAD02]. The choice of genre and level of game complexity should be tailored to the learning material; a more complicated game can allow development of more broad knowledge and understanding [MDM08], but we instead have chosen to create what could be considered a series of ‘mini-game’ simulations that cover a more specific subject area. These games are primarily puzzles, which are effective tools for helping students understand the learning material as well as the more high level concepts of problem solving and critical thinking [Sia03].

2.4.2 Evaluating Game-based Learning

There are numerous difficulties with evaluating the efficacy of game-based learning, especially due to the lack of empirical studies that evaluate serious games [PDS03, MDM08, Gro07, Ke09]. This suggests that a greater contribution can be made by empirically evaluating the serious game we create for learning debugging. In order to do this, we need to examine it from two different perspectives: first, as a game independent of its educational content, and secondly, as a learning tool for learning debugging. The former can be done by evaluating the game’s **heuristics** to determine its playability and whether or not players will actually find it engaging and fun. The latter is more difficult, and requires an assessment of learning outcomes.

Heuristics are an accepted method of evaluating the usability and playability of video games [DCT04]. By evaluating a game based on its game play, game story, game mechanics, and game usability, we can assess whether or not a game is easy to play (although potentially difficult to complete), and whether players will find playing the game worthwhile. Game play focuses primarily on player goals and motivations,

and looks at a player's perception of the game as a whole. Game story refers to on the element of fantasy in the game and how the player experiences the game's narrative and characters. Game mechanics describes the controls and input options available to the player, and how the game responds with feedback. Finally, game usability is a game's ease of use and simplicity, in that a player should be able to learn how to play a game quickly and spend more time figuring out what to do rather than how to do it.

The value in identifying these heuristics is that we have something to present to the game's testers for them to rate on a qualitative level. This is because ultimately, there is no substitute for user testing and user studies in order to properly evaluate the efficacy of a serious game [DCT04]. Like other human-computer interaction based software, it is difficult if not impossible to use automated testing to determine the usability of an interface. This means we will need to resort to human testers in order to determine whether or not players will find our game fun to play and easy to pick up and use without struggling to learn how to use it.

Once we have determined that a game is sufficiently playable, we must then consider how useful it is as a learning tool. This will also require the game to be played by users, who presumably fit in the target audience for our study. The effects of gender and ethnicity in the target audience are unclear, however it is suspected that older players (aged 40 and higher) will not find our game as effective for learning as younger students [Blu07]. With this in mind, we are still interested in introducing our game to beginners in order to perform a user study similar to those performed in many other game-based learning studies [Ke09]. To do this, we need to complement our game with a **debriefing** activity which will help to tie the game and its learning material back to the real world in line with curriculum objectives [dFO06]. This will include the necessary inquisition into the players' qualitative observations

about the game and hopefully provide us with insight as to what went well, what went wrong, and how to improve our game to make it better in the event that we find it is successful for learning.

The desired outcomes of a user study on game-based learning can be categorized into a list of different types of skills that players develop through game play [GAD02]. One of the key advantages of learning using games is that we are able to develop important cognitive skills in addition to simply delivering learning material to students. These skills include **Technical and Motor Skills** (developed primarily in games that require dexterity and quick reflexes), **Declarative Skills** (pertaining to facts and data), **Procedural Skills** (instructing players how to perform a given task), **Strategic Skills** (especially critical thinking), and **Affective Skills** (boosting an individual's confidence). Of these, we are less interested in developing Technical and Motor Skills or Declarative Skills, but instead want to focus on how well players develop in the skills that are procedural, strategic, and affective. These pertain specifically to debugging: we want players to learn how to debug, what strategies to employ in debugging, and we want to boost their confidence with debugging. This suggests that we should be interested in querying students as to their experiences with debugging and what levels of confidence they feel they have.

2.5 Summary

In the background chapter we have examined the role of debugging in computer science education, as well as traditional and non-traditional methods for learning debugging. This allows us to identify the key learning objectives to be included in our learning activities, and to determine what knowledge we will need to test during our evaluation. We also have a starting point for designing and evaluating our game-

based learning tool, which will have similar traits to other computer science games but will specialize in targeting the area of debugging education.

Chapter 3

The RoboBUG Game

3.1 Overview

RoboBUG is a game designed to teach debugging to novice programmers. The player takes the role of a scientist who is tasked with exterminating all of the bugs that have infected his or her robotic ‘Mech Suit’, using various techniques that reflect real debugging strategies. Through the controlling of a robot avatar, the player traverses C++ source code in his or her hunt for bugs. The avatar runs, falls, climbs, and can even transport through functions and segments of source code. To complete the game, a player must use various tools such as the Breakpointer tool to set breakpoints or the Warper tool to jump from function to function. The player reaches the end of the game by completing five levels that each introduce a specific debugging technique.

3.2 Game Design

RoboBUG was built using the Unity game creation system and is written in the C# programming language. Media elements, including the sound and graphics, were

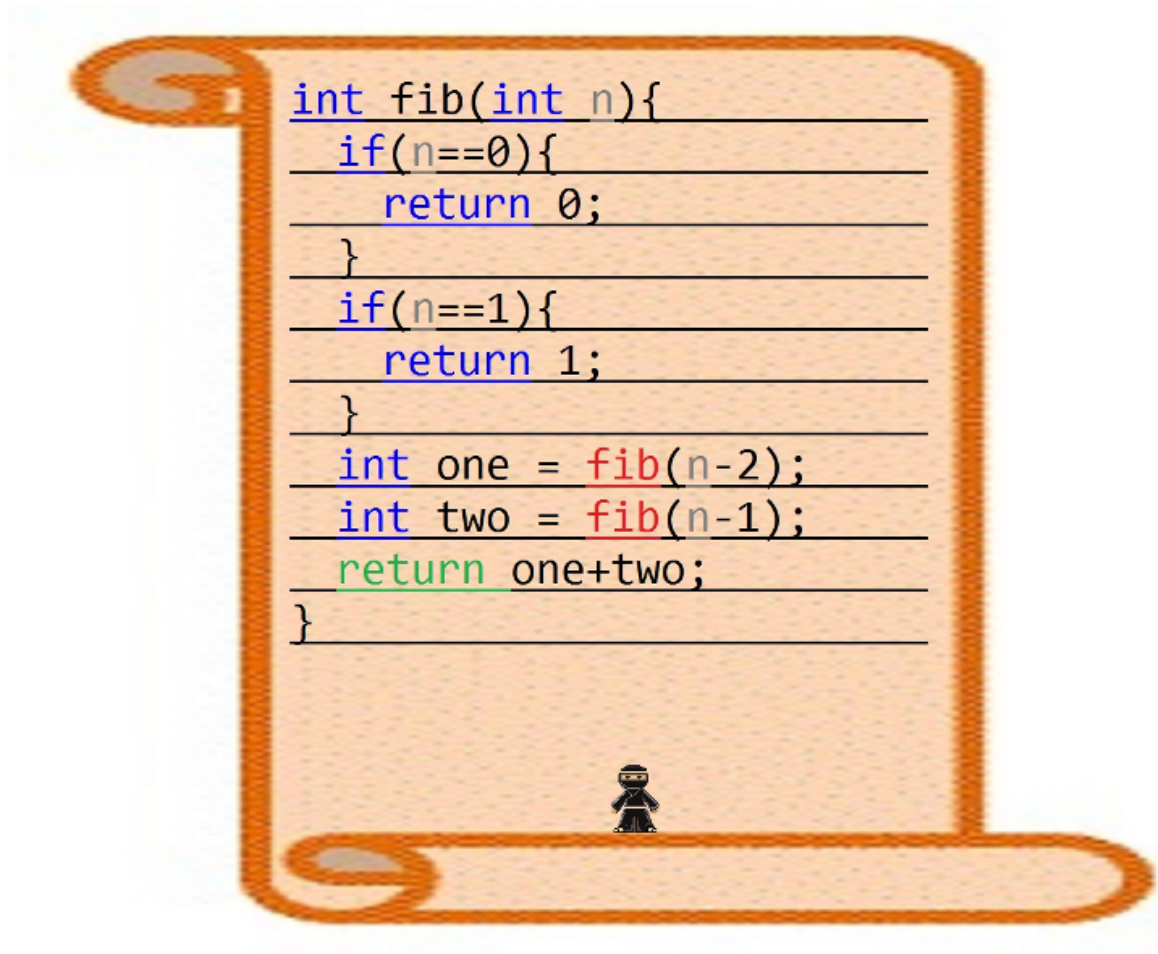


Figure 3.1: First prototype of RoboBUG

selected from open source art projects available online (see **Appendix E**). The game itself followed an iterative design process, and originally involved a ninja character traversing a scroll in the search of other ninjas (see **Figure 3.1**). The theme was later changed to be more oriented towards technology and computers, as the ninjas and scrolls were replaced by a more robotic theme, and the name RoboBUG was chosen. The original version also had a strict time limit and a limited number of ninja stars (which later became tools), providing additional challenge to players. These features were removed in the interest of ensuring participants in the user study would all be able to complete the game. The game was originally designed with seven levels,

however the level regarding error messages was later combined with the divide-and-conquer level to shorten the game length, and the final seventh level was removed from the game due to its length and difficulty from combining all of the previous techniques. Some features that were considered but ultimately did not make it to the final version included enemies, jumping, varying levels of difficulty, and strict time constraints.

3.2.1 Prototype Testing

To evaluate the quality and playability of the game, several game testers played the game and provided feedback based on their experience. The most common issues were encountered in the second level (testing), as one of the earlier versions of the game required players to locate the bug present in each of the three functions. This was later changed in order to adapt to time constraints and to minimize the frustration that players experienced trying to find unique and elusive bugs in the source code. Several of the game testers gave up after being unable to complete the second level after up to twenty minutes. Other players reported spending up to an hour attempting to finish the game, which was a concern due to the experimental method planned for the game that would only allow for a fifty minute play session. The issue of time constraints and desire to keep the game flowing led to the creation of the hints that appear after ten minutes being spent on a single level.

The second most problematic issue encountered by the prototype testers was the ability to control their character. Since the game introduces additional tools over time, players did not always recall the controls introduced at the beginning of the game, so additional reminders were added so that players would not get lost after forgetting how to do what they wanted to do. All of the testers also felt like the game's movement controls were unresponsive and shaky, which prompted a change in the game source

code that determined how the character in the game moved. Originally, the character had to 'jump' down through the source code using a combination of keys, but this was found to be unintuitive and players were often unable to descend to desired lines of source code. Eventually, the controls were changed to simple arrow key inputs that matched the movement of a cursor through a text document. This way, players could simply press up, down, left, and right to navigate through the source code without any delays or obstacles. In addition, the original penalty for false positive bug catching was changed from a decrease in the character's movement speed to a time delay before the player could reuse one of their tools.

3.2.2 Learning Objectives

Most of RoboBUG’s features were designed specifically for introductory computer science students learning debugging for the first time. Early in development, emphasis was placed on ensuring that players were responsible for learning how to find the bugs rather than the nuances of fixing them. We wanted students to play the game and by doing so, learn about standard debugging techniques, as well as how to use them. We also did not want to be teaching new programming concepts to students, so the source code that players would have to debug needed to be simple to understand so that there wasn’t an over-abundance of time spent reading and not debugging. Levels had to be designed in such a way that bugs would be present, but not obvious, and players needed to apply specific debugging techniques to find bugs. Five different debugging techniques were chosen for learning material: **source code tracing**, **testing**, **print statements**, **divide and conquer**, and **breakpoints**. These techniques were each introduced in a different level, and players were required to use the techniques in the context of the game to advance and reach the end. We also designed the gameplay to discourage poor debugging techniques such as brute force debugging (e.g. manually checking every single line of source code). This ensures that players are learning debugging through playing the game and not simply finding bugs through tedious and repetitive tasks. Not only must students learn different debugging techniques in the game, they must also learn how to use them properly and when they are most effective. Although students may have the option to attempt code tracing with multiple tasks, it is not always the most effective approach, so they should be able to adapt to different debugging challenges with the appropriate technique. For example, students should learn where it is most useful to place print statements in order to debug source code, or where to insert breakpoints to follow and examine the behaviour of a program.

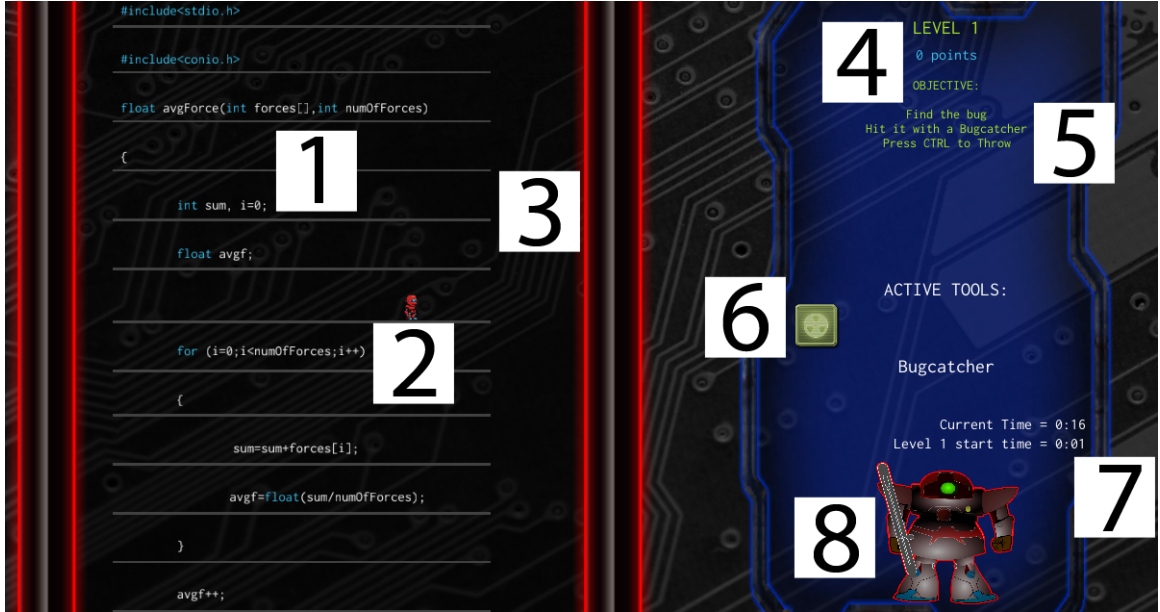


Figure 3.2: RoboBUG Debugging Game

#	Name	Description
1	Main Code Screen	The environment that the player traverses
2	Player Avatar	The avatar that the player controls in the game
3	Column Space	Occasionally contains game elements such as breakpoints
4	Points	Shows total points accumulated during the game thus far
5	Objective	Gives instructions on what to do to complete the level
6	Tools	Shows the available tools and highlights the active tool
7	Timer	Displays the elapsed time and level start time
8	Mech Suit	Highlights the 'buggy' areas of the Mech Suit






Tool Icon	Name	Levels Used	Description
	Breakpointer	1, 3, 4, 5	'Catches' bugs that it is thrown at (if present), otherwise penalizes the player for false positive debugging.
	Tester	2, 5	Runs code with the goal of hopefully encountering a bug during run-time.
	Activator	1, 3, 4, 5	Activates level specific elements such as print statements or comments.
	Breakpointer	5	Triggers breakpoints that can be toggled on or off before testing.
	Debugger	1, 3, 4, 5	Allows the player to teleport from function to function in order to examine different parts of a program.

Figure 3.3: RoboBUG Tools



Figure 3.4: RoboBUG Level Introduction

3.3 Gameplay Mechanics

RoboBUG is a two dimensional game where the player takes control of an avatar that runs, jumps, and falls through lines of source code on the screen (see **Figure 3.2**). This design was chosen in order to mimic real programming environments, and players are able to maneuver through the source code using the arrow keys in a similar way to moving a cursor through text in an IDE or word processor. In a given level, the player is tasked with finding a hidden bug, and given instructions that reflect a specific debugging technique. The player has access to a number of different tools, some of which are only available on specific levels. The player can cycle through available tools at any time, and simply presses a button for their avatar to use the selected tool. These tools include a **Bugcatcher**, a **Tester**, an **Activator**, a **Breakpointer**, and a **Warper** (see **Figure 3.3**).

With the exception of the second level (Testing), use of the Bugcatcher is necessary



Figure 3.5: RoboBUG Level End Screen

in order to proceed through the game. Once the Bugcatcher has been used on the line of source code in a level that features a bug, the bug is killed and the player is congratulated (see **Figure 3.4**). However, if the player uses the Bugcatcher on a line of source code with no bugs, they lose points and are prevented from using their tools for up to a minute, so that they can re-evaluate the source code and figure out where the bug really is. This feature is also used to prevent players from completing levels by simply checking every line of source code for bugs with the Bugcatcher via brute force. Points are awarded to the player based on the amount of time it took for them to complete the level (see **Figure 3.5**). After a ten minute period, if a player has still not completed a level, text hints appear over the bugs to indicate to them where they should use the Bugcatcher and why. At the end of the game, the player is shown the total time taken and their final score.



Figure 3.6: RoboBUG Level 1 - Code Tracing

3.4 Level Design

Each level of the game involves the player trying to debug source code that contains a single bug. The player is given instructions about the source code functionality, and are given instructions that allow them to attempt actual debugging techniques. Correctly applying the appropriate technique in a level will lead to players being able to locate which line of source code contains the bug, and then use the Bugcatcher tool to indicate the bug's location. The only exception to this is in the second level, where the bug is found via testing as soon as the player uses a test case that demonstrates its existence. Each level is progressively more difficult, and requires players to examine source code thoroughly before attempting to figure out the problem. Later levels give the player access to more tools that help them to identify bugs that are hard to find via manual source code tracing or trial-and-error.



Figure 3.7: RoboBUG Level 2 - Testing

3.4.1 Level 1 - Code Tracing

The first level is an exercise in a player's ability to read and understand source code (see **Figure 3.6**). The player is told that their 'Mech Suit' cannot calculate the average force acting on it. This is due to an error in the source code that calculates an average. The player is told to read through the source code and try and identify what the problem is manually, without running the source code or using any tools. The bug is located on a line where the average is incremented for no reason. Once the player realizes that this line of source code is out of place, they simply use the Bugcatcher tool there to catch the bug and finish the level.

3.4.2 Level 2 - Testing

This level is very distinct from the other four levels in the game (see **Figure 3.7**). The Mech Suit's weapon is not being appropriately calibrated by a number of functions,



Figure 3.8: RoboBUG Level 3 - Print Statements

and the player is tasked with using testing to identify a bug in the source code. Unlike the other levels, the bug in the source code is not visually available to the player, and the Bugcatcher tool is not used. They are only able to proceed by typing out test cases (by providing pairs of input and output) and running those tests to see if the expected output is calculated. There are three different functions that the player can test, and each one has a unique flaw. The player can proceed by finding a flaw in any one of these functions. The first function calculates a sum of distances, and contains a bug when the values contain any floating point variables (instead of integers). The second function calculates a median of 'power ratings', which fails if the list of input values is not in a sorted order. The final function calculates an average temperature, which does not work when one of the test values is negative.

3.4.3 Level 3 - Print Statements

This is the first level where the player has the ability to change the tools they have access to (see **Figure 3.8**). The player is told that their Mech Suit is unable to prioritize external threats. The Mech Suit uses a sorting algorithm to sort these threats by rank, based on how dangerous they are to the Mech Suit. Until this source code is fixed, the Mech Suit will not be able to identify the most critical threats. The source code in this level is a very simple implementation of bubble sort, where the bug is caused by the first index initialized to 1 instead of 0, which leaves the first element in place during the sorting. In order to observe this behaviour, there are a number of locations where the player can use their activator tool to enable print statements in the source code. By using another tool, the Warper, they can go visit a different screen where the print statement output is visible. By making print statements in appropriate locations, they can look at the output and see clearly that the first element remains unchanged through the program's execution. Using this information, they can return to the source code and find the line where the indices are initialized, then use the Bugcatcher tool to complete the level.

3.4.4 Level 4 - Divide and Conquer

The source code in this level is substantially long which creates a problem for anyone seeking to manually debug it (see **Figure 3.9**). In the story, the character's Mech Suit has a vision system that relies on a database of color value vectors that are represented in RGB form (red/green/blue, ranging from 0 to 255). This database is represented in a large number of tables, which would take an extensive period of time to check individually for bugs. Instead, the player is informed that they can use the Activator tool to comment out blocks of source code and then execute the function.



Figure 3.9: RoboBUG Level 4 - Divide and Conquer

Although the source code normally gives an undescriptive error message, when a certain segment of source code is commented out, the program suddenly functions again. This leads the player to identify the bug based on the segment of source code that changes the error message based on whether or not it has been commented out. Further examination of the table contained in that source code leads the player to discovering that one of the colors has an invalid value; the green value of one of the colors is greater than 255. Once the player finds this color, they finish the level by using the Bugcatcher tool.

3.4.5 Level 5 - Breakpoints

The final and most difficult level gives the player access to a wide variety of tools that are used as part of an in-game debugger system (see **Figure 3.10**). The Mech Suit's final problem is an issue with calculating distance based on coordinates; it is unable to determine which of two objects is closer, due to a mathematical error in



Figure 3.10: RoboBUG Level 5 - Breakpoints

one of the distance calculations. The bug is very subtle and is simply present due to an extra negative sign in one of the calculation lines. In order to identify this problem, the player must run the source code through several for loops that gather coordinates and calculate which of two objects is closer to coordinates (0,0,0). This is done by the player setting breakpoints inside the loop using the Breakpointer tool, and then using the Tester tool to continue running the source code until a breakpoint is reached. While the source code is paused, the player can observe the values of each of the program's variables and use these to determine that the x coordinates are not correctly being calculated. Once this is discovered, the player can use the Warper tool to examine the function that calculates the distances, and use the Bugcatcher tool on the x coordinate calculation line. This completes both the fifth level and the entirety of the game.

3.5 Summary

We have created a game that is designed to teach students the debugging learning objectives we discussed previously in Chapter 2. Prototype testing has shown us that RoboBUG can be completed by players within a reasonable amount of time. This gives us a tool that we can use for teaching debugging to students, and also for use in evaluating the efficacy of game-based learning for teaching debugging.

Chapter 4

Study Methodology

4.1 Overview

This section outlines the details of the evaluation used to assess the task of learning debugging techniques with a game-based approach in comparison to learning using a written assignment. An experiment was conducted where study participants were randomly divided into one of two groups: one group would complete a traditional written assignment as their learning activity, and the other group would play the RoboBUG game. All participants were given a short presentation to introduce debugging, followed by one of the two learning activities (game-based vs. traditional), a multiple choice evaluation, and finally a feedback questionnaire. Participants' data was collected regarding performance on the learning activity, evaluation, and questionnaire. Since participants in each group only completed one of the two learning activities, we are unable to directly compare the two groups. However, the insight gained from observing students in both groups and the qualitative data provided during the questionnaire allow us to assess the overall impression of students to RoboBUG and game-based learning.

4.2 Experimental Setup

4.2.1 Participants

Participants for the study included students enrolled in computer science programs at the University of Ontario Institute of Technology (UOIT). These students had mixed demographics, including age, gender, and race. All students had familiarity with C++ (the programming language used in both learning activities), and had various levels of familiarity with debugging. The participants were randomly divided into two groups of approximately equal size. Participants in the first (control) group were given the written learning activity, while participants in the second (experimental) group were given the RoboBUG game instead. All participants were offered a chance to win one of two \$25 gift cards for their participation in the study, and all participants additionally received a \$10 gift card.

4.2.2 Experiment Environment

The study took place during several sessions in the same physical place over the course of a month. The experiment resources were made available to the participants via the UOIT’s Software Quality Research Lab (SQR Lab) website. Participants were able to complete the study independently using their school-provided laptops. The room was monitored by one of the researchers. Participation was limited to the duration of the session, and participants did not need to complete any activities or surveys on their own time.

4.2.3 Data Management

All data gathered during the study was recorded and stored online using Limesurvey [LS1]. Limesurvey is an online survey tool that allows data to be exported to Comma-Separated Values (CSV) files, which can later be reviewed and evaluated. The time taken by each participant to complete each level of the RoboBUG game was stored in a text file on each participants' computer, which was required to be submitted to Limesurvey during the course of the study. The CSV files were only accessed and stored on a computer in the University's Software Quality Research Laboratory (SQR Lab). Anonymity of the participants was ensured through use of identification numbers that link participants to their survey and activity results. Only one researcher had access to information regarding the email addresses of students, and only for the purposes of providing feedback as well as presenting the prizes for the draw.

4.2.4 Experimental Documents and Tools

Participants in the study were presented with four different documents:

1. The introduction presentation was a short ten minute Powerpoint presentation where students were introduced to the definition of a bug, the concept of debugging, and some examples of considerations and techniques for use in debugging (see **Appendix A**).
2. Each participant completed one of two different learning activities:
 - The first of the two learning activities, which was given to the control group, was a set of five written questions that covered five debugging techniques: Code Tracing, (Black Box) Testing, Print Statements, Divide and Conquer, and Breakpoints (see **Appendix B**).

- The second activity, which was given to the experimental group, was the RoboBUG game (see **Chapter 3**). It introduced the same techniques as the written assignment, in the same order, with similar code, however participants were required to perform the techniques in the medium of the game rather than write down their answers directly.
3. The evaluation quiz was a set of ten multiple choices questions that covered the techniques used in the learning activity (see **Appendix C**). These questions required participants to not only recall definitions of techniques, but also to apply problem-solving skills in order to evaluate their ability to complete debugging tasks.
 4. The exit survey included a number of Likert scale questions as well as some open ended questions about improvements to the experiment design and general feedback (see **Appendix D**). The three open ended questions were:
 - (a) What was one positive thing about the way you learned debugging?
 - (b) What was one thing you want to see improved about the way you learned debugging?
 - (c) Please add any additional feedback you may have.

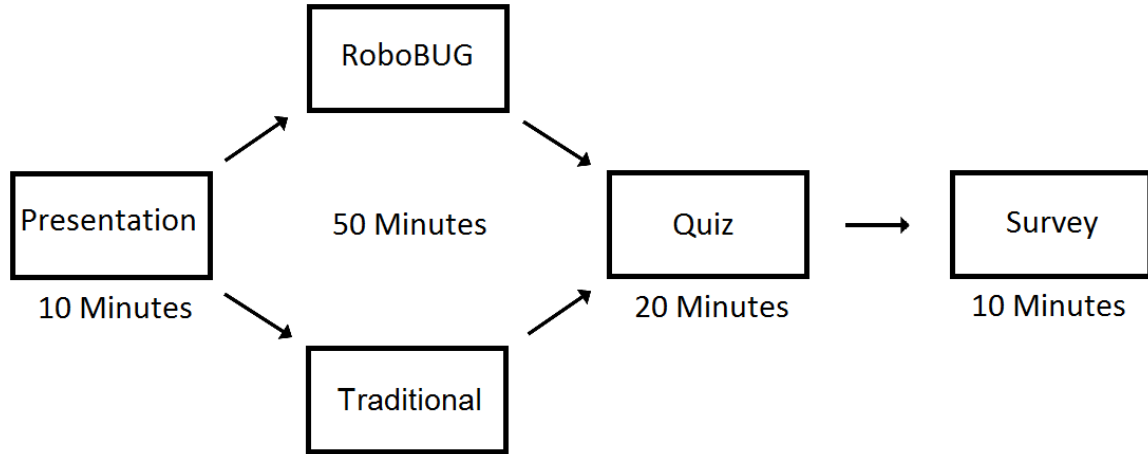


Figure 4.1: Experimental Procedure

4.3 Procedure

Participants were informed of the study by an email sent to all students in the computer science undergraduate program. Upon arrival to the experiment session, each participant was given a paper consent form that outlined the study’s intent and procedure, and included a unique random participant identification number (PID). Participants were able to consent to the study using an online version of the consent form that stored their PIDs and their emails for the purposes of the draw and for study feedback. Participants in the control group were given four digit PIDs, while participants in the experimental group were given five digit PIDs, purely for the purposes of identifying which group a participant belonged to.

4.3.1 Introduction

To present the concepts of debugging, the introduction of study is a live Powerpoint presentation that gives the students an idea of the fundamental concepts behind debugging. The concepts of program faults, differences between fixing and finding bugs, and the five techniques used in the study are introduced to students by the

researcher. This part of the study was designed to take approximately ten minutes, but is necessary so that students understand terminology and what they are expected to accomplish through debugging. After the presentation, the students are given a link to the online consent form, which then leads to the learning activity.

4.3.2 Learning Activity

Each group is given one of two activities: The RoboBUG game, or a written assignment. They are both available to students online, and it is not until this point that students discover which of the two activities they will be completing.

Game-based Learning

Students in the RoboBUG game group are given a download link after they have completed the consent form. They must download the game and play it till it is completed, upon which time the game outputs a log file to their desktop which they enter into the same website where they found the link. Students are given about 50 minutes to complete this part of the section, and each of the five levels provides an obvious hint to the solution if it takes a participant more than 10 minutes on a particular level.

Traditional Learning via a Written Assignment

The other group completes the same online consent form as the RoboBUG group, but then must complete a set of five questions that correspond to the same techniques taught in the RoboBUG game. Like the game, participants were given 50 minutes (10 per question) to complete this section of the study. If students do not complete a section within ten minutes, the online form automatically proceeds. Each question

is followed by its solution so that participants can see what the correct approach is in the event that they were mistaken.

4.3.3 Quiz

Both groups were led to a new online form once they have completed the learning activity. This form was a simple multiple choice quiz with ten questions that pertain to the same material the participants just learned. Questions included asking definitions of black-box testing and the best location for a breakpoint in a given piece of code.

4.3.4 Survey

The final online form that participants completed was a feedback survey that asked some demographic questions about students' familiarity with debug prior to the study, as well as their experience in the study. Most of the questions asked the student to rate their experience on a Likert scale, although there were a few open ended questions for suggestions about improving the study or what was most successful.

4.4 Data Analysis

All of the data the students entered into the various online forms could be retrieved in Microsoft Excel documents, and participants' data sets could be connected between the forms using their unique identification numbers.

4.4.1 Analysis of Game and Assignment Performance

The performance data gathered from the RoboBUG game was the time spent in seconds by each participant in each level. Each level actively focused on a specific debugging technique. This data was stored in a log file that the student uploaded once they completed the game. The purpose of this data was to both analyze what parts of the game were most difficult/simple for students, and to ascertain how students performed in the game in comparison with one another. The data gathered from the assignment group was simply the answers that each participant provides to the five questions. Each of these five questions corresponded to the same technique that was included in the respective RoboBUG level. Their answers were graded and a total score was calculated for each student. Like the RoboBUG group, their results were used to determine which questions were most difficult/simple and how well each student performed. Although we cannot compare the results of the two groups, we have designed the activities with a controlled experiment setup in mind. In particular, the code used in both activities is nearly identical, the questions evaluate similar tasks, and both activities were completed in the same environment.

4.4.2 Analysis of Quiz Results and Learning Outcomes

All participants completed the same ten question evaluation, which is simple to evaluate. Each question has only one correct answer, so each participant had their results totaled to come up with their overall performance evaluation. This metric allowed us to compare the participants in the two groups, and to determine which group performed better in the study. Simply viewing the average of each group offers one possible analysis of the data, however it is also possible to determine if the participants' scores on the game and assignment correlate with their results on the test. If

a correlation is found between the results of the learning activity and the evaluation performance, then it can be concluded that the learning activity contributes to each participants' debugging knowledge, and we can see if students in the RoboBUG group tend to perform better than students in the written group.

4.4.3 Analysis of Exit Survey Data

The exit survey is primarily to assess participants' demographics as well as their opinions regarding the use of game-based learning (see **Appendix D**). We hope to see a strong support from students for the use of games in teaching debugging, which supports our belief that games are a more enjoyable and fun way to learn. The open ended questions at the end of the survey are important for qualitatively assessing the participants' overall impression of the study and the debugging techniques. We hope to see a generally positive response from participants towards the RoboBUG game.

4.5 Ethical Considerations and Risks

Ethical considerations and mitigation of potential risks to participants is especially important due to all participants being students at UOIT. Participants may feel at risk that their performance in the study will be witnessed by other participants in the lab. They may also lack confidence about their performance in the tasks, and have a negative reaction to poor test results. Participants may also feel stressed about being asked to participate in the study under a time limit. There is a risk that participants may become aware of the other participants' rank or score in the study. Participants may be able to see how well others are doing in the event that they look at another participant's screen. Participants may feel coerced into participating in the research. This is due to being asked to participate by researchers who may

have been their instructors in courses they took at UOIT. There is an additional risk that participants will become unable to withdraw from the study if they lose their participant identification number (PID). Participants will be reassured that their performance in the study does not reflect upon themselves and will not be graded as part of the course. They will also be told that although there is a time limit on the study, it is not necessary that they complete the material perfectly in that period. Participants will be physically spaced apart during the study to ensure that there is no collaboration or loss of privacy between participants. Participants' performance results will only be viewed by the researchers and will not be released publicly. Participants will be informed that participation in the study is optional and does not have any effect on their enrollment in the course. They will also be told that they may choose to opt out at any time with no repercussions. Students will also be informed that the course instructor will not be aware of their participation, presence during the study, or have access to raw data (only anonymized data). To minimize the risk of losing a participant's PID, the number has been attached to the first page of the consent form, which participants will keep after consenting or choosing not to consent to the study. After the completion of the study, the solutions to the assignment and a copy of the RoboBUG game will be provided to all students in the course, including non-participants. The RoboBUG game itself will be made available as an open-source project once the results have been obtained.

4.6 Summary

We have created a detailed methodology for conducting an evaluation that will assess participants' knowledge of debugging with respect to one of two learning tasks. The evaluation will give us some insight as to how well the RoboBUG game was designed.

Chapter 5

Results

5.1 Overview

A total of 23 students at the University of Ontario Institute of Technology (UOIT) took part in one of several experimental sessions. These students were selected from computer science undergraduates in the first three years of study. Of these students, 11 were students in the second semester of their third-year, and 12 were students in the second semester of their first-year. Each student was randomly assigned to either the Game group or the Traditional Assignment group (Traditional group). The Game group consisted of 7 third-year students and 4 first-year students, while the Traditional group contained 4 third-year students and 8 first-year students. Second year students took part in an earlier trial of the study, but data corruption caused their data to be lost and unable to be included in the results.

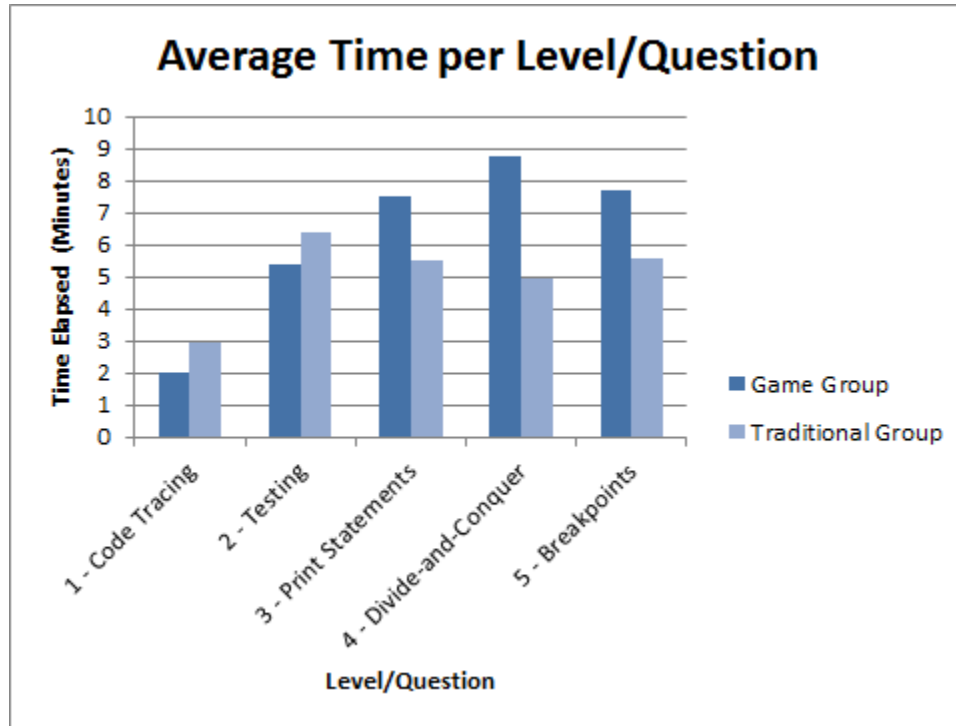


Figure 5.1: Average Time per Level/Question.

The Game group spent more time on each level as the game became more complicated. The time taken on each of the Traditional questions did not significantly change after the first question.

5.2 Activity Performance

Recall that each participant completed one of two different learning activities, each intended to take approximately 50 minutes. The time taken by each participant to complete the learning activity was recorded, including the amount of time spent on each level or question (see **Figure 5.1**). Participants spent the least time on the first level/question (on average, 2 minutes for the Game group and 3 minutes for the traditional group), but took longer on later levels/questions. Some participants took more than 10 minutes on the last 3 levels, indicating they needed an in-game hint in order to complete the stage. Participants in the Traditional group did not need

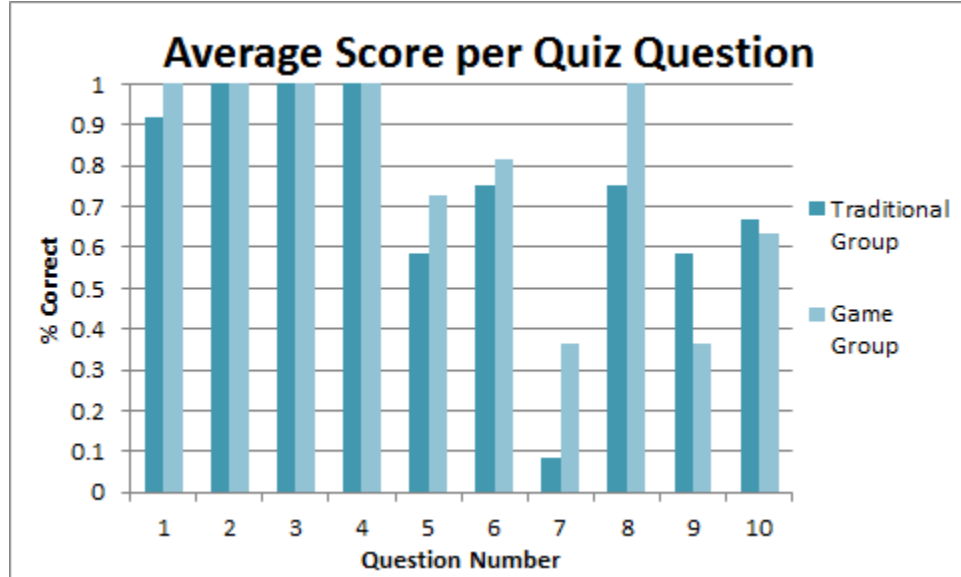


Figure 5.2: Average Score per Question by Group (see **Appendix C**).

Questions 1-4 asked students for definitions of debugging techniques, with almost all participants answering them correctly. Question 7 was the most difficult question, where few participants correctly identified the usefulness of the divide-and-conquer approach. Third-year students performed significantly better than first-year students, but there was no significant difference between the two learning activities.

to answer their questions correctly to proceed, and thus were able to complete the activity faster.

5.3 Quiz Performance

Recall that an evaluation was conducted to assess the learning outcomes for each participant (see **Figure 5.2**). Each participant completed a ten question multiple choice quiz after completing the learning assignment. Participants were scored based on the number of correct answers out of 10, and all participants were able to complete the quiz within the time limit.

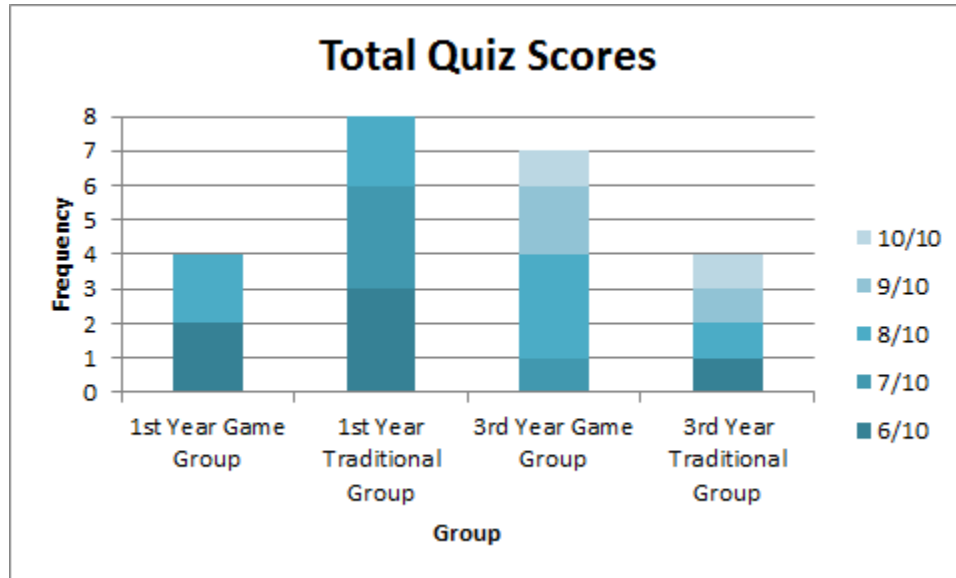


Figure 5.3: Total Quiz Scores by Age and Group.

The Game group spent more time on each level as the game became more complicated. The time taken on each of the Traditional questions did not significantly change after the first question.

5.3.1 Traditional Assignment Group

The Traditional group's quiz scores ranged from 6 to 10, with a mean of 7.33 and a standard deviation of 1.3. First-year students' scores ranged from 6 to 8 with a mean of 6.875, while the third-year students' scores ranged from 6 to 10 with a mean of 8.25.

5.3.2 Game group

The Game group's quiz scores ranged from 6 to 10, with a mean of 7.91 and a standard deviation of 1.22. First-year students' scores ranged from 6 to 8 with a mean of 7, while the third-year students' scores ranged from 8 to 10 with a mean of 8.429.

Evaluation Test Averages by Group and Age				
	Game Group Average	Written Group Average	Overall Average	Weighted Average
1st Year Average	7.00	6.88	6.92	6.94
3rd Year Average	8.43	8.25	8.36	8.34
Overall	7.91	7.33		
Weighted	7.71	7.56		

Table 5.1: Quiz Averages by Age and Learning Activity.

Overall averages were calculated using the average of all members of a group or year. Weighted averages were calculated using the average of the two groups/years.

5.3.3 Game vs. Traditional

An unpaired t-test was conducted to compare the results of the two groups. The results found that the Game group's results were not significantly higher, with a probability of $p=0.29$ ($t=-1.09$, $sdev=1.26$). A second unpaired t-test comparing the results of the first-year students to the third-year students found that the third-year students performed significantly higher ($p=0.0036$, $t=-3.28$, $sdev=1.06$), independently of the group to which they were assigned. These results suggest that the difference in means between the two groups is likely due to the presence of a higher percentage of third-year students in the Game group than the Traditional group. A comparison exclusively between the first-year students in each group found no significant difference ($p=0.83$). Similarly inconclusive results were found when comparing third-year students in each of the two learning activity groups.

5.4 Survey Feedback

Recall at the end of the study, participants were asked to complete a questionnaire to assess their feelings about debugging and the study. All 23 participants completed the final feedback survey, and 17 of them reported that they already had some level of

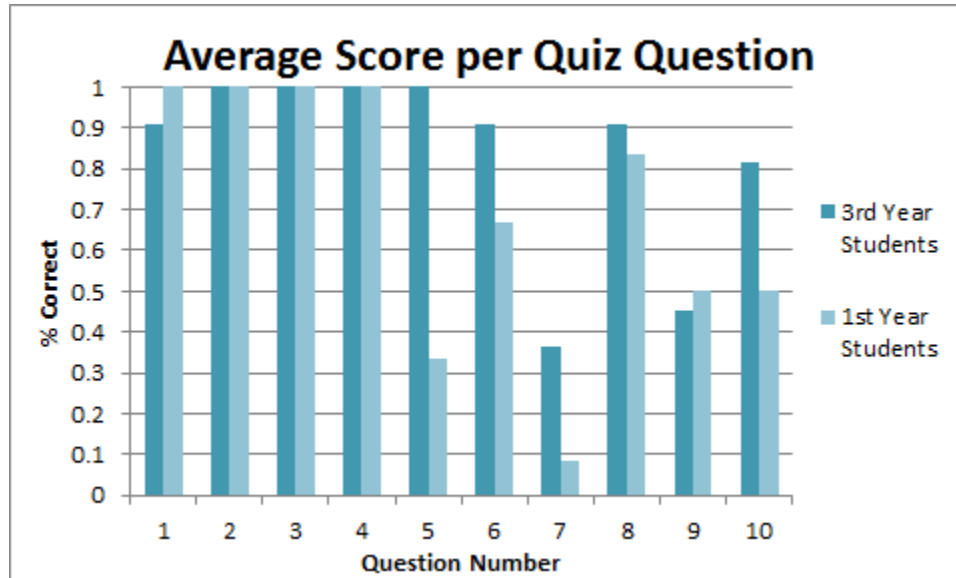


Figure 5.4: Average Score per Quiz Question by Year.

Third-year students scored higher than first-year students on all but a few questions.

Participants' Previous Experience with Debugging							
Group	Have Previous Experience	Confidence with Debugging (1 = Lowest Confidence, 5 = Highest Confidence)					Total
1 st Year	7	1	3	2	6	0	12
%	58%	8%	25%	17%	50%	0%	100%
3 rd Year	10	0	0	2	7	2	11
%	82%	0%	0%	18%	64%	18%	100%
Game	7	1	1	2	5	2	11
%	58%	9%	9%	18%	45%	18%	100%
Written	10	0	2	2	8	0	12
%	83%	0%	17%	17%	67%	0%	100%

Figure 5.5: Study Participants' Familiarity with Debugging Techniques.

familiarity with debugging. Of the six who were not familiar, five of them were first-year students, and only one was a third-year student. The majority of the questions asked students to answer on a Likert scale, with 1 representing "strongly disagree" and 5 representing "strongly agree". The average confidence level with debugging reported by students on a scale from 1 (lowest confidence) to 5 (highest confidence) was 3.52,

Average Rating of each Level by Group					
	Level 1	Level 2	Level 3	Level 4	Level 5
	Code Tracing	Testing	Print Statements	Divide/Conquer	Debugger
Written	3.4	3.3	3.6	3.9	3.4
Game	3.8	3.5	2.8	3.4	2.9

Table 5.2: Average Rating of Levels by Learning Activity.

and was approximately the same between both the Traditional group and the Game group (mean=3.55 and 3.5 respectively). Students disagreed that the Powerpoint at the start of the experiment taught them new information (mean=2), but they agreed that it was a useful reminder to them (mean= 4.1). Although both groups were neutral tending towards agreeable regarding game-based learning for debugging, the Traditional group was more interested in learning with games than the Game group (mean=3.75 and 3.36 for assignment group and Game group respectively). The Game group had a higher opinion of the first two lessons (code tracing and testing) than the Traditional group, however Game participants tended towards disliking the print statement and debugger levels far more than the Traditional group (mean=2.8 and 2.9 for each respective level). The Traditional group did not feel as prepared as the Game group regarding the ten quiz questions (mean=3.58 and 4 respectively). Participants in the game and written group were asked for feedback on the experiment at the end of the session. Most participants enjoyed the learning activity, felt that the difficulty was appropriate, and believed games could enhance the learning process. When asked if their confidence with debugging had improved, participants had a neutral response, with a tendency towards agreeing.

5.4.1 Impressions of the Learning Activity

The game-based learning activity and the traditional written activity were both designed to be interchangeable – the source code in both activities was almost identical,

Participant Feedback on Experiment using Likert Scale (1 = Strongly Disagree, 5 = Strongly Agree)					
Group	Participant enjoyed the learning activity				
Rating	1	2	3	4	5
1 st Year	1	1	2	6	2
%	8%	8%	17%	50%	17%
3 rd Year	0	2	1	4	4
%	0%	18%	9%	36%	36%
Game	1	2	2	4	2
%	9%	18%	18%	36%	18%
Written	0	1	1	6	4
%	0%	8%	8%	50%	33%
Group	Participant felt the difficulty was appropriate				
Rating	1	2	3	4	5
1 st Year	0	1	6	4	1
%	0%	8%	50%	33%	8%
3 rd Year	1	1	2	3	4
%	9%	9%	18%	27%	36%
Game	1	2	2	3	3
%	9%	18%	18%	27%	27%
Written	0	0	6	4	2
%	0%	0%	50%	33%	17%
Group	Participant feels more confident in debugging				
Rating	1	2	3	4	5
1 st Year	0	2	6	4	0
%	0%	17%	50%	33%	0%
3 rd Year	3	0	4	3	1
%	27%	0%	36%	27%	9%
Game	2	2	4	2	1
%	18%	18%	36%	18%	9%
Written	1	0	6	5	0
%	8%	0%	50%	42%	0%
Group	Participant believes games did/could enhance learning				
Rating	1	2	3	4	5
1 st Year	0	2	5	4	1
%	0%	17%	42%	33%	8%
3 rd Year	0	1	3	4	3
%	0%	9%	27%	36%	27%
Game	0	3	3	3	2
%	0%	27%	27%	27%	18%
Written	0	0	5	5	2
%	0%	0%	42%	42%	17%

Table 5.3: Results of Survey Feedback Questions.

and each level of the game corresponded to a similar question in the written activity. Students from both groups identified several issues with the overall design of the debugging activities:

1. **Clarity of Learning Activities:** The primary negative responses from both groups were directed at the confusion and lack of clarity in the learning activity. Six participants in the Traditional group were concerned with the wording of the questions, requesting “*more explanation/some form of example or guidance on how to use each technique.*” Five participants in the Game group commented that “*instructions have to be clearer*” and “*the description on some of the levels are difficult to understand*”. A few participants also suggested that a useful addition would be “*A tutorial or example before it started to know how the controls worked.*”
2. **Flexibility and Independence:** Participants in both groups felt that there were too many restrictions on what they are able to do in the learning activity. Of the four participants in the Traditional group who commented on this theme, two wished that the learning activity “*could have been a little more interactive*”, and another felt that “*usually when I’m debugging, being able to run the code is my comfort zone*”. Two participants in the Game group made similar comments, and one participant said that “*I might be old fashioned, but I prefer being given buggy code and having to compile it*”

Participants in both groups also identified some tasks as being straightforward. Three participants in the Traditional group and two participants in the Game group gave positive feedback regarding the straightforwardness of the activity. The responses from the Traditional group pointed out that “*it was quick and simple*”, “*very short and too the point*”, and “*it was presented in a easy to read manner.*” The Game group

did not make the same comments about shortness, but some participants felt that “*the information was straightforward*” and “*you go in knowing exactly what you need to do.*”

Impressions of the RoboBUG Game

1. **Fun:** Five participants in the Game group described the game as ‘fun’ (e.g. “*The fact that it was a game makes it very fun to do and doesn’t feel time consuming at all.*”) or mentioned that the game gave them “*a sense of accomplishment after each level*”. In contrast, none of the participants in the Traditional group mentioned fun or enjoyment in their responses.
2. **Engaging:** Three participants from the Game group commented on their level of engagement in the activity. RoboBug “*was actually fun and engaging*”, and one participant felt that “*using a game based approach kept my interest.*” In comparison, only one participant in the Traditional group made a positive reference to engagement, in which they stated, “*You have to take your time and make sure you look over everything and cover your ground when trying to take on debugging.*”
3. **Control Issues:** Five participants in the Game group responded negatively about the game’s controls (“*The controls on the roboBUG felt off*”). One of the more specific issues was that “*The vertical movement controls were a little bit frustrating, and having to use tab multiple times to switch between break point and testing was a little annoying.*” It’s possible that introducing the aforementioned tutorial might have resolved some of these issues.

Impressions of the Traditional Written Assignment

The Traditional group's responses were mostly directed towards the debugging techniques that they learned in the study (e.g. *"I liked the chance to try out the different techniques and see which one I preferred."*). Six Participants in the Traditional group listed the techniques (either specifically or as a whole) as a positive thing about learning debugging. Only one participant in the Game group had made any reference towards learning the debugging techniques. Three participants in the Traditional group explicitly said that they felt they improved in debugging in general. One of the Third-Year students commented that the Traditional activity was a *"fairly good introduction to debugging, none of it was any new to me at least but I feel that people who have had less experience with debugging would certainly learn from it."* Overall, the Traditional group had no real comments on the actual assignment; most of the focus was on the underlying debugging tasks and improvements to debugging ability.

5.5 Threats to Validity

5.5.1 Internal Validity

The primary threat to internal validity was the effect of level of experience on the participants who took part in the study. The combination of a small sample size and random group assignments resulted in a large proportion of third-year computer science students playing the RoboBUG game while a large proportion of the first-year computer science students completed the written assignment. This led the quiz average of the Game group to be higher, however ultimately there was no significant effect of Game or Traditional assignment on the quiz score of the participants.

5.5.2 Construct Validity

The quiz method used to assess the knowledge of participants after the learning activity was a simple multiple choice test. This test may not have been an accurate instrument for measuring knowledge, as it was found that several of the questions were answered correctly by 100% of students. This may be due to both methods being equally effective at teaching students, or it may be because the questions themselves were not designed in a manner that allowed different levels of knowledge to be differentiated and exposed. The primary risk to validity is due to the fact that this was the only instrument used to measure their knowledge of debugging, and no practical tests (e.g. asking them to debug real code after the learning activity was completed) were conducted.

Despite this risk, the results seem to indicate that the test was a sufficient measurement. Students in the third-year of study are expected to have a greater knowledge of debugging than their first-year peers, and the test results showed this was clearly the case. The fact that the third-year students scored higher suggests that the test was able to accurately measure debugging knowledge of individual students, but was not able to identify significant differences between the Game and Traditional groups.

5.5.3 External Validity

The specific population used in this study included only first and third-year computer science students, which created a wide gap in the participant quiz scores. Without any second or fourth year students, it is difficult to conclude how effective the learning methods would be for students of different ages. It is possible that second year students may have found that the RoboBUG game was more targeted towards them as an audience, as they would have a higher level of understanding of programming

than first-year computer science students while still being less familiar with debugging than third-year computer science students.

5.5.4 Conclusion Validity

The most major threat to conclusion validity is the fact that both groups did not complete both tasks. Therefore, we can't make definite claims regarding the benefits of RoboBug compared to Traditional techniques. Due to this threat, we have focused on an independent evaluation of each group using descriptive statistics. With a larger sample size and both groups completing both learning tasks, we may still not be able to assess the effectiveness of RoboBUG with respect to achieving learning outcomes. It is possible that there are too many variables that need to be controlled in order for a proper assessment to be made, or it could be that it simply isn't feasible to compare the RoboBUG game to a Traditional assignment. A large number of educational studies have found no significant difference when comparing media in education [Rus97].

The only statistically significant result of this study was the difference between first-year computer science students and third-year computer science students. First-year students did not perform as well on the quiz as third-year students, regardless of which group they were part of. It is reasonable to expect that third-year students have more familiarity and experience with debugging, which explains why they scored higher on the quiz. Third-year students have taken more courses on computer science and are more likely to have already used some of the debugging techniques in this study. By contrast, first-year students are more likely to have never been introduced to formal debugging, and lack the advantages of their more experienced and older peers.

5.6 Summary

This chapter presents the results of our controlled experiment examining the differences between game-based and traditional approaches to learning debugging. We have gathered data about the performance of first and third-year students in both the Game and Traditional written assignment groups. The results indicated that playing RoboBUG did not significantly affect participants' performance in comparison with a written assignment. Participants in both groups expressed interest in game-based learning and indicated a preference for game-based learning over traditional assignments.

Chapter 6

Conclusions

6.1 Discussion

This thesis presents RoboBUG, a game for learning debugging techniques. Overall, student participation in an evaluation found RoboBUG to be fun, new, and different. Unfortunately, no evidence was found to suggest that RoboBUG was more effective than traditional written assignments in achieving debugging learning outcomes. However, an important consideration is the fact that the experiment took place over the course of only one 90 minute session. It is possible that the effects of the learning activities were not as noticeable as they would be in a longitudinal study. Several of the previous studies in non-traditional teaching methods [FS12,FCJ04,EPDJ07,Ast04,SF06] either proposed or conducted studies that took place over the course of a semester or longer. RoboBUG was not designed for a study. In fact, RoboBUG was designed for use in a lab environment and was intended to be used as a supplement to the traditional learning process and not a replacement. As an interesting avenue for future research we would like to extend RoboBUG for use in a longitudinal study as it would also allow us to assess the benefits of failure in the learning process. Currently,

a user of RoboBUG can fail during an individual level of the game, but a user will not find any value in repeating the levels of the game over and over again.

Other studies [CB14, Bry11, MM10] that took place in a single session did not manage to show that non-traditional teaching techniques are superior to traditional techniques, but focused instead on student feedback and on whether the activity helped achieve learning outcomes. Interestingly, a 2010 study by Mohammed and Mohan noted that “*the increases in test scores were minimal largely because of issues with the usability of the game and insufficient instructional guidance*” [MM10]. Students playing the RoboBUG game had similar comments regarding usability in their feedback, and it is possible that a game designed for a single session might not be able to teach complicated debugging tasks and other advanced concepts. This is supported by some of the qualitative feedback students gave regarding the levels in the RoboBUG game: earlier and simpler stages were positively reviewed, but students were particularly frustrated by the complicated final debugger level. Advanced topics may be better taught by dividing them into smaller and simpler subtasks, as students felt overwhelmed by the amount of new information and unsure of what steps they needed to take. A large number of students needed in-game hints in order to complete the final stage of the RoboBUG game. Meanwhile, participants in the Traditional Assignment group would receive the correct answer immediately after answering a question incorrectly. This difference in design may have contributed to the differences in feelings of frustration between participants in the two groups.

6.2 Limitations

The primary limitation of the experimental results was due to the relatively small participant sample size and the limited variation of students (only first and third-

year students in computer science). Although 23 participants took part in the study, the significant difference between third and first-years was found to be the primary cause for difference in scores. Taken separately, each experience group did not have significant differences between those who played RoboBUG and those who completed the written assignment. The two groups of 12 first-year students and 11 third-year students may not have been independently large enough to discover any kind of significant result. Unfortunately, this study was unable to recruit fourth-year students, who may have provided additional insight and different results. Second-year students participated in an earlier pilot of the study, but their data was lost due to a server crash.

The RoboBUG game also suffers from a number of limitations due to its design and implementation. At present, the design does not support large-scale debugging well; multiple files would require the avatar to spend a great deal of time navigating from place to place, which makes it especially difficult to understand how large programs work. The current implementation of RoboBUG also does not allow for multiple scenarios and source code to be run at each level. Changing the RoboBUG source code so that each level can be easily adapted to accept different programs may assist learning. RoboBUG could also be better designed to facilitate learning through better use of failure as a learning element. Modifying the game so that each level is slightly different during each game session would provide an opportunity for participants to learn through repetition, and to improve their skills if they fail to complete a level.

Finally, the experimental design did not allow for students to complete both learning tasks. Due to time constraints, participants were only able to complete either the RoboBUG Game or the Traditional activity. Participants were informed that they would gain access to the RoboBUG game after it was released as an open-source project.

6.3 Future Work

Future work in this area should consider the effects of game-based learning over a longer time period. Debugging is a problem encountered at all levels of computer programming, so it is valuable to have continuous exposure to proper debugging techniques. A game that programmers can play for multiple hours may be a motivating learning experience that improves their skills through play. A game that can be played over the course of a week, or even over a semester, might be a more valuable tool to those who encounter frequent debugging challenges. Alternatively, changes to the RoboBUG game may help to target other learning objectives and potentially achieve different results. Software concepts such as parallelization and version control might be incorporated into games to be introduced to novice programmers. This could be done in a similar style to RoboBUG, or with an entirely different game design depending on the problems being addressed. For example, a game teaching parallelization might present players with a piece of serial code, then ask them to locate where processes should fork and join, or where deadlocks and data races are likely to occur. With regards to debugging, RoboBUG might be improved by adding new functionality, including new tutorial stages or options for students to compete with one another or work cooperatively to solve problems.

6.4 Conclusions

The qualitative data gathered in our evaluation supports our belief that the RoboBUG debugging game is an entertaining alternative to learning via traditional methods, but is not a significantly superior approach to achieving learning outcomes. The best use of RoboBUG may now be as a learning supplement, providing a short lesson on effective ways that students can debug their code. RoboBUG's current value is not

only as a learning tool, but also may be a starting point for the creation of a better game for teaching debugging or other programming concepts. We plan to release RoboBUG as an open source game and use user feedback to evolve it into a more valuable learning tool.

Bibliography

- [AEH05] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin*, 37(3):84, September 2005.
- [Ast04] O.L. Astrachan. Non-competitive programming contest problems as the basis for just-in-time teaching. In *Proceedings of the 34th Annual Frontiers in Education (FIE 2004)*, pages 442–446, October 2004.
- [BBC⁺10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
- [BF14] P. Bourque and R.E. Fairley. *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society, 2014.
- [Blu07] Richard Blunt. Does game-based learning work? Results from three recent studies. *Proceedings of the Interservice/Industry Training, Simulation, & Education Conference*, pages 945–955, 2007.
- [Bry11] Renee Bryce. Bug Wars: a competitive exercise to find bugs in code. *Journal of Computing Sciences in Colleges*, pages 43–50, 2011.

- [BT95] RB Barr and J Tagg. From teaching to learning—A new paradigm for undergraduate education. *Change: The magazine of higher learning*, 27(6):13–26, 1995.
- [CB98] Andy Cockburn and Andrew Bryant. Cleogo: collaborative and multi-metaphor programming for kids. In *Proceedings of the 3rd Asia Pacific Computer Human Interaction*, pages 189–194, 1998.
- [CB14] Elizabeth Carter and G.D. Blank. Debugging Tutor: Preliminary Evaluation. *Journal of Computing Sciences in Colleges*, pages 58–64, 2014.
- [CC08] Jaruek Chookittikul and Wajee Chookittikul. Six sigma quality improvement methods for creating and revising computer science degree programs and curricula. In *Proceedings of the 38th Annual Frontiers in Education Conference*, pages F2E–15–F2E–20, 2008.
- [Chu09] Du Chuntao. Empirical Study on College Students’ Debugging Abilities in Computer Programming. In *Proceedings of the 1st International Conference on Information Science and Engineering*, pages 3319–3322. IEEE, 2009.
- [CHYH04] D.J. Cook, Manfred Huber, Ramesh Yerraballi, and L.B. Holder. Enhancing computer science education with a wireless intelligent simulation environment. *Journal of Computing in Higher Education*, 16(1):106–127, 2004.
- [CL03] Ryan Chmiel and MC Loui. An integrated approach to instruction in debugging computer programs. In *Proceedings of the 33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 3, pages S4C_1–S4C_6. IEEE, 2003.

- [CWB11] R. Collobert, Jason Weston, and L. Bottou. Natural language processing (almost) from Scratch. *The Journal of Machine Learning Research*, 12:2493–2537, 2011.
- [CWL13] Mei-Wen Chen, Cheng-Chih Wu, and Yu-Tzu Lin. Novices’ Debugging Behaviors in VB Programming. *Learning and Teaching in Computing and Engineering*, pages 25–30, March 2013.
- [DCT04] Heather Desurvire, Martin Caplan, and Jozsef A. Toth. Using heuristics to evaluate the playability of games. In *Proceedings of the Conference on Human Factors and Computing Systems (CHI ’04) - Extended Abstracts*, pages 1509–1512, 2004.
- [DDKN11] Sebastian Deterding, Dan Dixon, R. Khaled, and Lennart Nacke. From game design elements to gamefulness: defining gamification. 2011.
- [dFO06] Sara de Freitas and Martin Oliver. How can exploratory learning with games and simulations within the curriculum be most effectively evaluated? *Computers & Education*, 46(3):249–264, April 2006.
- [Eck06] Richard Van Eck. Digital game-based learning: It’s not just the digital natives who are restless. *EDUCAUSE review*, 41(2):1–16, 2006.
- [Ecl] Eclipse - The Eclipse Foundation open source community website. <https://eclipse.org/>. [Accessed March 19th, 2015].
- [EPDJ07] Sebastian Elbaum, Suzette Person, Jon Dokulil, and Matt Jorde. Bug Hunt: Making Early Software Testing Lessons Engaging and Affordable. In *Proceedings of the 29th International Conference on Software Engineering (ICSE’07)*, pages 688–697, 2007.

- [Eth04] Jim Etheredge. CMeRun. *ACM SIGCSE Bulletin*, 36(1):22, March 2004.
- [FCJ04] T. Flowers, C. Carver, and J. Jackson. Empowering students and building confidence in novice programmers through Gauntlet. In *Proceedings of the 34th Annual Frontiers in Education (FIE 2004)*, pages 433–436, 2004.
- [FMH⁺10] Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. Debugging From the Student Perspective. *IEEE Transactions on Education*, 53(3):390–396, August 2010.
- [FS12] Joel Fenwick and Peter Sutton. Using Quicksand to improve debugging practice in post-novice level students. In *Proceedings of the 14th Australasian Computing Education Conference*, volume 12, pages 141–146, 2012.
- [GAD02] R. Garris, R. Ahlers, and J. E. Driskell. Games, Motivation, and Learning: A Research and Practice Model. *Simulation & Gaming*, 33(4):441–467, December 2002.
- [Gro07] Begoña Gros. Digital Games in Education. *Journal of Research on Technology in Education*, 40(1):23–38, September 2007.
- [HLB12] Morgan Hall, Keri Laughter, and Jessica Brown. An empirical study of programming bugs in CS1, CS2, and CS3 homework submissions. *Journal of Computing Sciences in Colleges*, 28(2):87–94, 2012.
- [HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notes*, 39(12):92–106, 2004.
- [IMMJ10] Roslina Ibrahim, Rasimah Che Mohd Yusoff, Hasiah Mohamed@Omar, and Azizah Jaafar. Students Perceptions of Using Educational Games to

- Learn Introductory Programming. *Computer and Information Science*, 4(1):205–216, December 2010.
- [JS13] Brittany Johnson and Yoonki Song. Why don’t software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.
- [KCC02] Caitlin Kelleher, Dennis Cosgrove, and David Culyba. Alice2: programming without syntax errors. In *Proceedings of the 15th Annual Symposium on the User Interface Software and Technology*, pages 3–4, 2002.
- [Ke09] Fengfeng Ke. A qualitative meta-analysis of computer games as learning tools. *Handbook of Research on Effective Electronic Gaming in Education*, 2009.
- [KJB13] David Kelk, Kevin Jalbert, and Jeremy S Bradbury. with ARC. In *Proceedings of the 1st International Conference on Multicore Software Engineering, Performance, and Tools (MUSEPAT 2013)*, pages 73–84, 2013.
- [KMG06] Jackie O. Kelly, N.U.I. Maynooth, and J. Paul Gibson. RoboCode & Problem-Based Learning : A non-prescriptive approach to teaching programming. *Innovation and Technology in Computer Science Education (ITiCSE ’06)*, (June):26–28, 2006.
- [LFW13] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, June 2013.
- [LS1] LimeSurvey - the free and open source survey software tool. <https://www.limesurvey.org/en/>. [Accessed March 19th, 2015].

- [MDM08] Alke Martens, Holger Diener, and Steffen Malo. Game-based learning with computers—learning, simulations, and games. *Transactions on Entertainment I*, 5080:172–190, 2008.
- [MDTV12] Mathieu Muratet, Elisabeth Delozanne, Patrice Torguet, and Fabienne Viallet. Serious game and students’ learning motivation: effect of context using Prog&Play. *Intelligent Tutoring Systems*, pages 123–128, 2012.
- [MLM⁺08] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. Debugging. In *Proceedings of the 39th Technical Symposium on Computer Science Education (SIGCSE ’08)*, pages 163–167, 2008.
- [MM10] Phaedra Mohammed and Permanand Mohan. Combining Digital Games with Culture: A Novel Approach towards Boosting Student Interest and Skill Development in Computer Science Programming. In *Proceedings of the 2nd International Conference on Mobile, Hybrid, and On-Line Learning*, pages 60–65, 2010.
- [Moo73] M.G. Moore. Toward a theory of independent learning and teaching. *The Journal of Higher Education*, pages 661–679, 1973.
- [MTJV09] Mathieu Muratet, Patrice Torguet, Jean-Pierre Jessel, and Fabienne Viallet. Towards a Serious Game to Help Students Learn Computer Programming. *International Journal of Computer Games Technology*, 2009:1–12, 2009.
- [NPM08] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages. In *Proceedings of the 39th Technical Symposium on Computer Science Education (SIGCSE ’08)*, pages 168–172, 2008.

- [oCC13] ACM/IEEE-CS Joint Task Force on Computing Curricula. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, January 2013.
- [PDS03] M. Pivec, O. Dziabenko, and I. Schinnerl. Aspects of game-based learning. In *Proceedings of the 3rd International Conference on Knowledge Management (I-KNOW '03)*, July 2003.
- [Pre05] M. Prensky. Computer games and learning: Digital game-based learning. *Handbook of Computer Game Studies*, 2005.
- [Res96] M. Resnick. StarLogo: An environment for decentralized modeling and decentralized thinking. In *Conference Companion on Human Factors in Computing Systems (CHI '96)*, pages 11–12, 1996.
- [RK97] John Rosenberg and Michael Kölling. Testing object-oriented programs. *ACM SIGCSE Bulletin*, 29(1):77–81, March 1997.
- [Rus97] Thomas L. Russell. *The "No Significant Difference" phenomenon as reported in 248 research reports, summaries, and papers*. North Carolina State University, 1997.
- [SF06] Jonathan Sykes and Melissa Federoff. Player-centred game design. *CHI '06 extended abstracts on Human factors in computing systems - CHI EA '06*, page 1731, 2006.
- [Shu11] V.J. Shute. *Stealth assessment in computer-based games to support learning*. 2011.

- [Sia03] A.C. Siang. Theories of learning: a computer game perspective. In *Proceedings of the 5th International Symposium on Multimedia Software Engineering*, pages 239–245, December 2003.
- [STF⁺07] Beth Simon, Lynda Thomas, Sue Fitzgerald, Renée McCauley, Susan Haller, John Hamer, Brian Hanks, Michael T. Helmick, Jan Erik Moström, and Judy Sheard. Debugging assistance for novices. In *Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '07)*, pages 137–151, 2007.
- [TB14] Nikolai Tillmann and Judith Bishop. Code hunt: Searching for Secret Code for Fun. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*, pages 23–26, 2014.
- [VS1] Visual Studio - Microsoft Developer Tools. <https://www.visualstudio.com/>. [Accessed March 19th, 2015].
- [Zel09] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2/e edition, 2009.

Appendix A

Appendix 1: Powerpoint Slides

An Introduction to Debugging

Michael Miljanovic

MSc Student

Faculty of Science (Computer Science)



Welcome To Our Debugging Study!

- Study Activities
 - Consent Form
 - Learning Activity
 - Assessment Questions
 - Questionnaire
- Consent
 - Participation is **voluntary** and **you can leave at any time**
 - Participants are eligible to receive one of two **\$25 gift cards**

What is a Software Bug?

“something that went wrong”*

*A failure, fault or error in the design or implementation of your software. It can result in incorrect output, crashes and more.

What is Debugging?

- The process of **finding** and **fixing** software bugs
- In order to find a bug you often have to reproduce it. Why?
 - So that you know it exists
 - So that you know when you have fixed it
- The larger the software, the harder it may be to find and fix bugs
- **Nobody is perfect**; everyone makes mistakes, and it's worth learning how to properly rectify them before you spend tons of time trying to fix them!

Reproducing Bugs – Testing

- To debug a program, it needs to be **testable**
 - It has to be able to run on any given input so you need tests that are examples of the different kinds of possible input
- Bugs may not show up all the time – you may need specific test cases to see a bug
 - In fact, part of debugging comes from being able to choose good test cases that can identify bad behavior

Reproducing Bugs – Testing

- Test cases should be tailored to your program – there is no *'formula'* to follow to make a good set of test cases
 - Different input types
Example: String vs. int
 - Empty inputs
Example: empty list []
 - Inputs that cause different behaviours
Example: If the program behaves differently for negative vs. positive numbers, be sure to check both cases

The Limits of Testing

“Testing only allows you to identify and reproduce a bug. It does not tell you which part of the source code caused the bug”

Finding Bugs

- The location of bugs can sometimes be hinted at by error messages
 - Looking at the error message line number can help to identify where the bug is located
 - Remember that a bug may be introduced on an earlier line than it is first identified
- You can also add **print statements** to your program to help find a bug

Finding Bugs

- It's not always necessary to test an entire program to find a bug
 - You can run parts of your program one at a time to make sure each is running properly - **Divide and Conquer**

Finding Bugs

- Modern Integrated Development Environments (IDEs) include a tool for helping you find bugs - the **debugger**
 - **Code Tracing** - you can walk through your program line-by-line to see the values of variables and identify when a problem occurs (*this can also be done manually outside the debugger*)
 - **Breakpoints** - if you have a large program you can use breakpoints to identify places in the program where you want to stop execution and look at the variable values

Fixing Bugs

- Fixing bugs can be **tricky**
 - If you find a variable that has an incorrect value it may not be obvious how you can fix it
- **Simple** solutions are best!
- Some complicated bugs may arise from chains of causes and effects
 - **Example:** variable x was 42, therefore p became 0, and then the program failed

Fixing Bugs – More Testing

- Once you have fixed a bug you need to test your program again to make sure the bug is really fixed
- You also need to make sure that fixing a bug hasn't introduced other bugs!

References

- Andreas Zeller, **Why Programs Fail: A Guide to Systematic Debugging**, Morgan Kaufmann, San Francisco, CA, 2009

Consent Form Link

- <http://bit.ly/debug-experiment-2015>
- Use 1280x720 resolution for the activity, and 'fantastic' graphics quality
- Don't forget to return to the online survey when you are finished!

Appendix B

Appendix 2: Traditional Group Written Assignment

Debugging Experiment

This study covers five different concepts related to debugging:

- **Code Tracing**
- **Testing (Black Box Testing)**
- **Print Statements**
- **Divide-and-Conquer**
- **Breakpoints and Debuggers**

You will be given a 50 minute time period to answer the questions that relate to each topic. Please read through the instructions before you attempt to answer the questions. YOU WILL NOT BE GRADED ON THIS MATERIAL.

Code Tracing

Code tracing requires a programmer to read through code and keep track of key variable values. You can use code tracing to find bugs in short snippets of code by simply looking for typing errors or statements that appear out of place.

Find the **bug** in the function below.

```
1. //Finds the average of input numbers
2. //Input : Integer numbers
3. //Output : The mean of the input
4.
5. #include<stdio.h>
6. #include<conio.h>
7. int main()
8. {
9.     int n,a[100],sum=0,i;
10.    float avg;
11.    cout<<"Count of numbers to be averaged : ";
12.    cin >> n;
13.    {
14.        cout<<"Enter " << n << " numbers";
15.        for(i=0;i<n;i++)
16.        {
17.            cin >> a[i];
18.        }
19.        cout<<"Average of "<< n <<" numbers entered is ";
20.        for(i=0;i<n;i++)
21.        {
22.            sum=sum+a[i];
23.        }
24.        avg=(float)sum/n;
25.        avg++;
26.        cout<<avg;
27.    }
28. }
```

The error is in line number: 25 – there is no reason to increment the average.

Testing

One method of bug detection involves running code with the intention of generating a bug or defect. Testing for bugs allows programmers to identify incorrect function behaviour, perform a repair, and then test to ensure that the fix was successful. In the case of **Black Box Testing**, the tester is not able to view the specific code being tested, and is only told what its intended behaviour is.

For the function header below, write **AT LEAST FIVE** different test cases that check for errors in the code. With each test case, explain what the purpose of the test case is.

```
1. //Finds the median of an array of numbers
2. //Input : An array of numbers
3. //Output : The median (middle value) of those numbers
4. //The median of an even-sized array is the average (mean) of the middle two numbers.
5.
6. float median(float[] numarray);
```

Good Examples:

Assert(median([]) == 0) // empty list

Assert(median([6]) == 6) // single element

Assert(median([2,6,8,999] == 7) // even number of elements

Assert(median([3,1,5]) == -3) // unsorted array

Assert(median([-4,-2,0,1]) == -1) // negative values

Print Statements

To get an idea of what is going on internally while a program is running, you can insert print statements into your code. Print statements allow you to see the values of variables that change without having to rely solely on a function's returned result.

The code below is a simple sorting algorithm that reorders a list of numbers from highest to lowest. **Add in print statements (e.g. 'cout<<i') in locations that might help you identify bugs in the code.**

```
1. //Sorts a list of numbers from
2. //Input : List of numbers
3. //Output : Sorted list
4. //Note : cout<<array will print the contents of array
5.
6. void BubbleSort (int array[],int size)
7. {
8.     int i = 0;
9.
10.    int temp;
11.
12.    bool swapped = true;
13.
14.    while(swapped){
15.
16.        swapped = false;
17.
18.        while(i<size-1){
19.
20.            if(array[i]<array[i+1]){
21.
22.                temp = array[i];
23.
24.                array[i] = array[i+1];
25.
26.                array[i+1] = temp;
27.
28.                swapped = true;
29.
30.            }
31.
32.            i++;
33.
34.        }
35.
36.    }
37.
38. }
```

The best location for a print statement would be line 28, to examine what swap occurred and if it worked. Line 32 is also a useful location although it may create a large amount of duplicate output.

Print statements should at the very least include 'cout<<array', and optionally 'swapped' and 'i' depending on their location.

Divide and Conquer

To determine the location of a bug in a piece of code, it can be helpful to separate the code into sections so that the bug can be isolated to a certain region. One method of doing this involves commenting out blocks of code to determine which block contains a bug.

The code below has a single faulty function, but it is unclear which is causing the problem. Use the function and its output below to determine which function is not behaving correctly. The buggy function is: **<WRITE YOUR ANSWER HERE>**

```
1. //Finds the value of y in the equation y=ax^3+bx^2+cx+d
2. //Input : integers a,b,c,d,x
3. //Output : integer y in equation y=ax^3+bx^2+cx+d
4.
5. #include<powers.h>
6.
7. int Equation (int a, int b, int c, int d, int x)
8. {
9.     int term1, term2, term3, term4, output;
10.
11.
12.
13.     term1 = Cube(x,a);
14.     term2 = Square(x,b);
15.     term3 = Mult(x,c);
16.     term4 = Zero(x,d);
17.
18.     output = term1+term2+term3+term4;
19.     return output;
```

The correct answer, 26, is attained when Square is commented out and replaced with direct calculation; this indicates that Square is the buggy function.

```

1. //Finds the value of y in the equation  $y=ax^3+bx^2+cx+d$ 
2. //Input : integers a,b,c,d,x
3. //Output : integer y in equation  $y=ax^3+bx^2+cx+d$ 
4.
5. #include<powers.h>
6.
7. int Equation (int a, int b, int c, int d, int x)
8. {
9.     int term1, term2, term3, term4, output;
10.
11.     term1 = x*x*x*a;
12.     term2 = x*x*b;
13.
14.     //term1 = Cube(x,a);
15.     //term2 = Square(x,b);
16.     term3 = Mult(x,c);
17.     term4 = Zero(x,d);
18.
19.     output = term1+term2+term3+term4;
20.     return output;

```

```

>>Equation(1,2,3,4,2)

```

```

1. //Finds the value of y in the equation  $y=ax^3+bx^2+cx+d$ 
2. //Input : integers a,b,c,d,x
3. //Output : integer y in equation  $y=ax^3+bx^2+cx+d$ 
4.
5. #include<powers.h>
6.
7. int Equation (int a, int b, int c, int d, int x)
8. {
9.     int term1, term2, term3, term4, output;
10.
11.     term3 = x*c;
12.     term4 = d;
13.
14.     term1 = Cube(x,a);
15.     term2 = Square(x,b);
16.     //term3 = Mult(x,c);
17.     //term4 = Zero(x,d);
18.
19.     output = term1+term2+term3+term4;
20.     return output;

```

```

>>Equation(1,2,3,4,2)

```

```

1. //Finds the value of y in the equation  $y=ax^3+bx^2+cx+d$ 
2. //Input : integers a,b,c,d,x
3. //Output : integer y in equation  $y=ax^3+bx^2+cx+d$ 
4.
5. #include<powers.h>
6.
7. int Equation (int a, int b, int c, int d, int x)
8. {
9.     int term1, term2, term3, term4, output;
10.
11.     term2 = x*x*b;
12.     term3 = x*c;
13.
14.     term1 = Cube(x,a);
15.     //term2 = Square(x,b);
16.     //term3 = Mult(x,c);
17.     term4 = Zero(x,d);
18.
19.     output = term1+term2+term3+term4;
20.     return output;

```

```

>>Equation(1,2,3,4,2)

```

Breakpoints and Debuggers

A **debugger** is a tool used specifically for finding bugs in programs. Debuggers allow you to insert **breakpoints** into your code, where you can pause during testing and observe the program's current state. This allows you to look at variable values without resorting to using print statements.

The code below was run using a debugger. At separate breakpoints, the values of the variables were observed and can also be seen below. Use these observations in combination with the source code to identify the bug.

Main Code

```
1. //Identifies the vector with the highest sum of coordinates
2. //Input : two vectors of the form (x,y,z)
3. //Output : true if vector1 has a higher sum than vector2, otherwise false
4.
5. bool Compare (int x1, int y1, int z1, int x2, int y2, int z2)
6. {
7.     int xval=0, yval=0, zval = 0, sum=0;
8.
9.     xval = xval+x1-x2;
10.    yval -= y2-y1;
11.    zval += -(z1-z2);
12.
13.    sum = add.total(xval,yval,zval);
14.    return sum>0;
```

Input Variable Values

x1 = 10	x2 = 30
y1 = 20	y2 = -50
z1 = -10	z2 = 90

Output Variable Values

xval = -20	yval = 70	zval = 100
sum = 150		

Although the xval and yval numbers were calculated by taking x1/y1 and subtracting x2/y2, the zval is calculated using $-(z1-z2) = z2-z1$. This means line 11 is incorrect.

Main Code

```
1. //Identifies the vector with the highest sum of coordinates
2. //Input : two vectors of the form (x,y,z)
3. //Output : true if vector1 has a higher sum than vector2, otherwise false
4.
5. bool Compare (int x1, int y1, int z1, int x2, int y2, int z2)
6. {
7.     int xval=0, yval=0, zval = 0, sum=0;
8.
9.     xval = xval+x1-x2;
10.    yval -= y2-y1;
11.    zval += -(z1-z2);
12.
13.    sum = add.total(xval,yval,zval);
14.    return sum>0;
```

Input Variable Values

x1 = 10	x2 = 30
y1 = 20	y2 = -50
z1 = -10	z2 = 90

Output Variable Values

xval = 0	yval = 0	zval = 0
sum = 0		

Main Code

```
1. //Identifies the vector with the highest sum of coordinates
2. //Input : two vectors of the form (x,y,z)
3. //Output : true if vector1 has a higher sum than vector2, otherwise false
4.
5. bool Compare (int x1, int y1, int z1, int x2, int y2, int z2)
6. {
7.     int xval=0, yval=0, zval = 0, sum=0;
8.
9.     xval = xval+x1-x2;
10.    yval -= y2-y1;
11.    zval += -(z1-z2);
12.
13.    sum = add.total(xval,yval,zval);
14.    return sum>0;
```

Input Variable Values

x1 = 10	x2 = 30
y1 = 20	y2 = -50
z1 = -10	z2 = 90

Output Variable Values

xval = -20	yval = 70	zval = 100
sum = 0		

Main Code

```
1. //Identifies the vector with the highest sum of coordinates
2. //Input : two vectors of the form (x,y,z)
3. //Output : true if vector1 has a higher sum than vector2, otherwise false
4.
5. bool Compare (int x1, int y1, int z1, int x2, int y2, int z2)
6. {
7.     int xval=0, yval=0, zval = 0, sum=0;
8.
9.     xval = xval+x1-x2;
10.    yval -= y2-y1;
11.    zval += -(z1-z2);
12.
13.    sum = add.total(xval,yval,zval);
14.    return sum>0;
```

Input Variable Values

x1 = 10	x2 = 30
y1 = 20	y2 = -50
z1 = -10	z2 = 90

Output Variable Values

xval = -20	yval = 70	zval = 100
sum = 150		

Appendix C

Appendix 3: Evaluation Test

Post Experiment Evaluation

This evaluation will review the same concepts that you just finished completing in the previous activity. You will be given 20 Minutes to complete ten multiple choice questions about debugging.

Multiple Choice

Choose the best answer for each question.

- 1) What is **code tracing**?
 - a. **Reading through code to make sure it is behaving properly**
 - b. Separating code by section to isolate a bug
 - c. Observing code during run-time and checking the internal value of variables
 - d. Running code with inputs and ensuring the output gives the desired result
- 2) What is **testing**?
 - a. Reading through code to make sure it is behaving properly
 - b. Separating code by section to isolate a bug
 - c. Observing code during run-time and checking the internal value of variables
 - d. **Running code with inputs and ensuring the output gives the desired result**
- 3) What is a **divide-and-conquer** approach?
 - a. Reading through code to make sure it is behaving properly
 - b. **Separating code by section to isolate a bug**
 - c. Observing code during run-time and checking the internal value of variables
 - d. Running code with inputs and ensuring the output gives the desired result
- 4) What are **breakpoints** used to do?
 - a. Reading through code to make sure it is behaving properly
 - b. Separating code by section to isolate a bug
 - c. **Observing code during run-time and checking the internal value of variables**
 - d. Running code with inputs and ensuring the output gives the desired result
- 5) What is **black box** testing?
 - a. Testing code while looking at it during run-time
 - b. **Testing code without seeing its internal behavior**
 - c. Testing code that requires a very large number of test cases
 - d. Testing code that has unknown behavior
- 6) Suppose you are debugging code where you need to know the values of variables during run-time. Which of these methods is appropriate? **(Choose up to 3)**
 - a. **Adding Print statements**
 - b. **Using Breakpoints**
 - c. Black box testing
- 7) When is a divide-and-conquer approach useful? **(Choose up to 3)**
 - a. **Some of the code is faulty, but most of it is functional**
 - b. **Code that is known to be functional does not need to be run, or can be easily replaced**
 - c. **There is a large amount of code that makes code tracing infeasible.**
- 8) Which of the following techniques do not require you to actually run any code?
 - a. **Code tracing**
 - b. Black box testing
 - c. Debugging with breakpoints
 - d. Divide-and-conquer approach

- 9) Given the code below, which of these test cases is most appropriate for determining if there is an error in the maximum function?

```
1. int Foo (int a, int b, int c)
2. {
3.     if(a>b){
4.         return minimum(a,b,c);
5.     }
6.     else if(c>a){
7.         if(c>b){
8.             return c;
9.         }
10.        else {
11.            return maximum(a,b,c);
12.        }
13.    }
14.    return a;
15. }
```

- a. Foo(3,2,1)
- b. Foo(0,0,0)
- c. Foo(3,1,0)
- d. Foo(0,0,1)
- e. Foo(0,2,1)**

10) In the code below, where is the best place for a breakpoint if you want to understand more about the array values during each iteration of the sort?

- a. Line 11
- b. Line 14
- c. Line 15
- d. Line 17**

```
1. //Sorts a list of numbers
2. //Input : List of numbers
3. //Output : Sorted list
4.
5. void BubbleSort (int array[],int size)
6. {
7.     int i = 0;
8.     int temp;
9.     bool swapped = true;
10.    while(swapped){
11.        swapped = false;
12.        while(i<size-1){
13.            if(array[i]<array[i+1]){
14.                temp = array[i];
15.                array[i] = array[i+1];
16.                array[i+1] = temp;
17.                swapped = true;
18.            }
19.            i++;
20.        }
21.    }
22. }
```

Appendix D

Appendix 4: Feedback Survey

Exit Survey

Please complete this short survey once you have completed the study. This survey will help us to identify any issues that you encountered during the study and to properly evaluate the efficacy of the learning experience.

Personal Experience

These questions pertain to your own personal experience with debugging.

Do you have any experience with debugging (finding and fixing bugs in computer code)?

Yes

No

If you answered yes to the above question, have you used any debugging tools in the past? List any you have used below:

On a Likert scale from 1 to 5, with 1 being very unconfident and 5 being very confident, rate your confidence with debugging.

1 2 3 4 5

Powerpoint

These questions pertain to the Powerpoint presentation you saw at the start of the experiment.

Rate your opinion on the following questions using a Likert scale from 1 to 5 (1 – Strongly Disagree, 2 – Disagree, 3 – Neutral, 4 – Agree, 5 – Strongly Agree)

The material in the Powerpoint slides was completely new to me.

1 2 3 4 5

The Powerpoint slides provided an appropriate introduction to the topics I learned about in the activity.

1 2 3 4 5

Learning Activity – RoboBUG Game

These questions pertain to the second part of the study, where you played the RoboBUG debugging game.

Rate your opinion on the following using a Likert scale from 1 to 5 (1 – Strongly Disagree, 2 – Disagree, 3 – Neutral, 4 – Agree, 5 – Strongly Agree)

I enjoyed learning debugging with RoboBUG.

1 2 3 4 5

The gameplay difficulty in RoboBUG was appropriate for learning how to debug.

1 2 3 4 5

I am more confident in my ability to debug code thanks to RoboBUG.

1 2 3 4 5

I feel like a game-based approach to learning (e.g. RoboBUG) enhanced my learning experience in comparison to traditional written assignments.

1 2 3 4 5

Learning Activity - Assignment

These questions pertain to the second part of the study, where you completed the five question debugging assignment.

Rate your opinion on the following using a Likert scale from 1 to 5 (1 – Strongly Disagree, 2 – Disagree, 3 – Neutral, 4 – Agree, 5 – Strongly Agree)

I enjoyed learning debugging in the assignment.

1 2 3 4 5

The difficulty of the assignment was appropriate for learning how to debug.

1 2 3 4 5

I am more confident in my ability to debug code thanks to the assignment.

1 2 3 4 5

I think a game-based approach to learning would enhance my learning in comparison with the assignment I completed.

1 2 3 4 5

Learning Debugging – Five Levels

Rate each level on a Likert scale from 1 to 5 (1 – Very Poor, 2 – Poor, 3 – Average, 4 – Good, 5 – Very Good) based on how well it prepared you for each debugging task.

Level 1 – Code Tracing (Robot’s Average Force Function)

1 2 3 4 5

Level 2 – (Black Box) Testing (Three Laser Functions)

1 2 3 4 5

Level 3 – Print Statements (Robot’s Threat Assessment)

1 2 3 4 5

Level 4 – Divide-and-Conquer (Robot’s Color Database)

1 2 3 4 5

Level 5 – Breakpoints and Debuggers (Robot’s Distance Calculator)

1 2 3 4 5

Learning Debugging – Five Questions

Rate each question on a Likert scale from 1 to 5 (1 – Very Poor, 2 – Poor, 3 – Average, 4 – Good, 5 – Very Good) based on how well it prepared you for each debugging task.

Question 1 – Code Tracing

1 2 3 4 5

Question 2 – (Black Box) Testing

1 2 3 4 5

Question 3 – Print Statements

1 2 3 4 5

Question 4 – Divide-and-Conquer

1 2 3 4 5

Question 5 – Breakpoints and Debuggers

1 2 3 4 5

Summary

Rate your opinion on the following using a Likert scale from 1 to 5 (1 – Strongly Disagree, 2 – Disagree, 3 – Neutral, 4 – Agree, 5 – Strongly Agree)

I feel like I was adequately prepared for the ten evaluation questions.

1 2 3 4 5

Please answer the following questions with short sentence or paragraph.

What was one positive thing about the way you learned debugging?

What was one thing you want to see improved about the way you learned debugging?

Please add any additional feedback you may have.

Appendix E

Appendix 5: RoboBUG Media Credits

RoboBUG was created using open source media gathered from a variety of sources. These works are licensed under a Creative Commons Attribution-ShareAlike3.0 Unported License. We would like to acknowledge the following websites, works, and artists for their contributions:

- <http://www.amon.co>
- <http://www.opengameart.org>
- <http://opsound.org/>
- <https://openclipart.org>
- <https://www.freesound.org/>
- Gui-Set by Rawdanitsu
- Stephen Redshrike Challener - graphic artist

- William.Thompsonj - contributor.
- A robot by Anton Yu. From 0.18 OCAL database.
- glyph.png by Jinn (Submitted by Andrettin)
- FREE Keyboard and controllers prompts pack by xelu
- Space Gui in various colors by Rawdanitsu
- Icons by phaelax
- angled metal tracks on an electronic circuit board from creative103.com
- (IT) ANTI-MATTER(S) by LDX#40
- THIRTY by _AA_
- ACD8 by PERAMIDES
- FILTHYFILTER by Kid Cholera's VASCULOID
- NIGHTTIME by Kid Cholera's VASCULOID
- ON THE DOWNLOAD by Kid Cholera's VASCULOID
- SPIDERTWO by DAVE HOWES
- alien_screecn_1.wav by CosmicD
- concrete_step_3.wav by movingplaid
- zoom up 1 (quicker delay).wav by Chriddof
- Whoosh_Swish_03.wav by mich3d
- Error.wav by Autistic Lucario