RoboBUG: A Serious Game for Learning Debugging Techniques

Michael A. Miljanovic
University of Ontario Institute of Technology
2000 Simcoe Street North
Oshawa, Ontario, Canada
michael.miljanovic@uoit.ca

Jeremy S. Bradbury
University of Ontario Institute of Technology
2000 Simcoe Street North
Oshawa, Ontario, Canada
jeremy.bradbury@uoit.ca

ABSTRACT

Debugging is an essential but challenging task that can present a great deal of confusion and frustration to novice programmers. It can be argued that Computer Science education does not sufficiently address the challenges that students face when identifying bugs in their programs. To help students learn effective debugging techniques and to provide students a more enjoyable and motivating experience, we have designed the RoboBUG game. RoboBUG is a serious game that can be customized with respect to different programming languages and game levels.

CCS CONCEPTS

• **Social and professional topics** → **Computer science education**; • **Software and its engineering** → *Software testing and debugging*; • **Applied computing** → Computer games;

KEYWORDS

debugging, programming, software engineering, computer science, education, serious games, game-based learning

1 INTRODUCTION

Research related to programming and software development often focuses on advancing the state-of-the-art in software developer practices, techniques and tools. Research into programming and software development learning tools is equally important as software developers must first learn best practices before they can effectively perform them.

It is essential that programmers who seek to write reliable, high quality source code be able to efficiently identify and repair bugs in program code [24]. The process of fixing these bugs, *debugging*, has been shown to consume up to 50% of a programmer time in large software projects [5]. Furthermore, the ability to debug code is not easily acquired, and experts have a significant advantage over novices [3]. In addition to experience, experts have knowledge of a variety of debugging techniques including code tracing, instrumentation and the use of breakpoints in debuggers.

Game-based learning has already been implemented and proven effective for computer programming education [10, 16] which suggests that it may also prove useful in debugging education. Serious games should not only help players achieve learning outcomes but should also provide a fun and positive experience. Previous studies have shown that motivated and engaged learners will perform more effectively [8, 18]. Finally, the benefits of serious games suggest they may be an effective way to counter the frustration typically associated with debugging.

First year courses often do not teach debugging explicitly, and expect students to learn it for themselves. This lack of preparedness is compounded by the fact that there is no established set of best practices for teaching debugging [7]. This is compounded by a lack of online resources dedicated to helping novices learn to debug [3]. Even students with a good understanding of how to write programs still struggle with debugging [1]. These students may have the ability to fix bugs in their programs, but only after accomplishing the difficult task of finding the bugs first.

In general, the lack of accessibility to debugging techniques leads students to have a primarily negative experience, even when given the opportunity to learn debugging [20]. This fact is particularly problematic when students conclude that debugging skills are based on aptitude and are unable to be learned [4]. We hope to address the problem of accessibility and frustration with debugging education by creating RoboBUG, a puzzle-based serious game, that is designed to help students achieve debugging learning outcomes in an enjoyable rather than tedious way.

The creation of RoboBUG required us to address a number of challenges, including:

(1) **Game design:** How can debugging activities be represented as game tasks/actions? How can these tasks be connected to produce enjoyable and cohesive gameplay?
(2) **Game learning data:** What debugging topics and learning materials should be used in the game to achieve the desired learning outcomes while minimizing frustration?
(3) **Game evaluation:** How do we design our study to effectively assess debugging learning and level of enjoyment?

In addition, we needed to consider if the combination of game design and learning data challenges will allow a player to retain debugging technique knowledge after the game's completion.

In the remaining sections of our paper we present background on debugging and game-based learning (Section 2), an overview of our RoboBUG serious game (Section 3), the results of a pilot study (Section 4.1) as well as the results of two full studies (Section 4.2 and Section 4.3). Our studies were conducted with undergraduates at the University of Ontario Institute of Technology (UOIT).

2 BACKGROUND

2.1 Debugging

As mentioned in the previous section, debugging is the processing of finding and fixing problems (bugs) in a program. Debugging can involve the use of dedicated debugger tools and can use a combination of both static static and dynamic debugging techniques. Static techniques are those that do not require execution of the program

Conference'17, Washington, DC, USA
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Figure 1: A screenshot of the RoboBUG game

The RoboBUG game interface is divided into two regions: (1) The code region (left) in which the user controls an avatar to navigate the source code while using different debugging tools to eventually find bugs, (2) The sidebar region (right) in which the user can view information about available debugging tools, the currently active tool and the time remaining in the level. In addition to these two regions, the RoboBUG game also utilizes dialogs (bottom) to provide contextual information to the user including feedback from debugging activities.

while dynamic techniques rely on run-time information. Common debugging techniques include code tracing, print statements, divide-and-conquer and breakpoints. **Code tracing** is a static technique in which the programmer reads through code to make sure it is behaving properly. **Print statements** are a code injection approach to inserting output statements into the program that display internal program status information as output. **Divide-and-conquer** is a debugging strategy that allows a programmer to systematically separate source code into sections in an effort to isolate a bug. **Breakpoints** are a common feature in modern debuggers that allows the execution of a program to be paused in order to allow the programmer to view the internal value of variables at specific execution points.

After conducting a review of debugging techniques we decided to select the above four techniques for inclusion in the RoboBUG game. Other debugging techniques, such as black-box testing, were excluded in order to constrain the duration of the game. Although the chosen techniques are only applied in specific levels of the game, there is an overarching theme of program comprehension. Experts are faster and more effective debuggers than novices due to superior program comprehension strategies [14], which is why we chose to emphasize the importance of comprehension by having players debug code they did not write themselves.

In addition, we selected types of bugs that are common in student-written code. As syntax errors are often identified by compilers, they tend to be less problematic for students than logic or data errors [9]. Thus, we chose to include only logic and data errors in our tool, and because we believe the techniques we selected are best explained through their abilities to find these types of errors.

2.2 Game-based Learning

Serious games for Computer Science is an active research area [17, 23], especially with respect to learning how to write computer programs. Games such as Code Hunt [21] help learners develop their skills through puzzle tasks that require players to write programs in order to solve a specific problem. Serious programming games usually focus on the act of creating programs by writing source code, or alternatively by using a drag-and-drop interface, as seen with Program Your Robot [11]. A nonstandard example of a puzzle-based programming game is Robot ON! [15], that does not require players to write any programs. Instead, Robot ON! focuses on program understanding and comprehension by requiring players to read source code written by someone else.

Some games such as Gidget [13] have been designed with an emphasis on helping players learn about debugging. However, Gidget is meant to introduce general debugging, and uses unique pseudo-code instead of a common language such as C++ or Java. In addition,

Figure 2: A RoboBUG comic storyboard used to advance the game plot and setup a new game level

The RoboBUG game utilizes a plot to engage the user in the debugging learning tasks. Each level begins with a four panel comic that provides plot details and connects each level with an overall story. For example, in the above comic, alien bugs have infected the player's mech suit and caused the vision system to malfunction. In the corresponding level the user is tasked with detecting bugs in the mech suit's vision system source code.

we did not find any serious games in the literature that specifically focus on learning debugging techniques.

3 THE ROBOBUG GAME

RoboBUG[1] (see Figure 1) is a serious game intended to be played by first-year computer science students who are learning to debug for the first time. We chose to design RoboBUG as a puzzle-type game, as puzzles have been shown to be effective for both helping to learn material as well as demonstrating higher level concepts such as critical thinking and problem-solving [19].

RoboBUG was implemented in C# using the Unity game engine[2] and open source media elements. Although the standard version of RoboBUG is based on debugging in C++, it also includes a framework that allows instructors to create their own levels using other programming languages. New levels are specified using XML and can be customized with respect to different aspects of a level, including time limit, available tools, output text, and source code. These new levels can be inserted into the game with minimal effort.

In the RoboBUG game a player takes the role of a scientist whose world is under attack from an alien bug world. The alien world has sent an advanced army of tiny bugs that infect technology in the scientist's world – including the scientist's 'Mech Suit' (a robotic suit of armour). In order to help save the world from the alien bugs, the scientist must first purge bugs from the infected 'Mech Suit'. Bugs are purged by the scientist by virtually entering the infected source code to find all of the alien bugs. In each level the player (taking the role of the scientist) must fix a particular part of the Mech Suit (e.g. the vision system) by figuring out where the bug is hiding (see Figure 2). the game is finished once the player has completed all levels, found all of the bugs, and has a working Mech Suit. The actions required by different debugging techniques are represented as tools that the player can aim at lines of code. For example, a Breakpointer tool can be aimed at a line of code to insert a break point and a Warper tool can be aimed at a function call to jump to another part of the code (the function definition).

The default version of RoboBUG includes four levels that teach different debugging techniques in C++ (see Table 1). Each of these levels includes: a tutorial that introduces new debugging tools, 2-3 subproblems that contain small debugging tasks and partial source code and a final challenge that combines the tools introduced in the tutorials and the knowledge gained from the subproblems. The final challenge will typically involve detecting a bug in the full program.

A player's progress through the game is recorded in a set of log files that allow gameplay to stopped and resumed. Prototype testing has shown that the game with the default levels takes approximately 30 minutes to complete. RoboBUG has been used with the four default levels during an introductory programming course at UOIT.

3.1 Game Levels

In developing the default version of RoboBUG, we thought about the order and content that should be included in the game levels. We chose to first introduce code tracing as it can be used in combination with other techniques and it is not too time-consuming due to the short length of the example programs. Next, we selected the use of print statements in order to help novices learn to identify program behavior at run-time. This was followed by the strategy of divide-and-conquer, where novices can reduce the search space for bugs by commenting out code that is guaranteed to contain no bugs. Finally, we adapted some features of a debugger so that novices can learn the concepts of breakpoints and the value of observing a program's execution state during run-time.

The following subsections provide a brief walkthrough of the game levels in RoboBUG. Each level includes several parts that build upon each other with the final part requiring the player to debugging a subsystem of the 'Mech Suit'.

3.1.1 *Level 1: Code Tracing.*

- *Subproblem A:* A mathematical function definition is provided to the player that includes input, output, and behavior. The function is intended to return an average of a set of values (a floating point number), and the player's goal is to trace through the code and identify that the 'avgf' variable (responsible for storing the average) has a type (boolean) that doesn't match its intended use.
- *Subproblem B:* The source code from Subproblem A is expanded to calculate the average of a list of numbers using a loop; however, the player needs to identify that the loop adds the 'avgf' variable to a running total sum rather than properly averaging the numbers.
- *Subproblem C:* The source code is expanded again and now contains the full function for calculating the average of

[1] https://github.com/sqrlab/robobug
[2] https://unity3d.com/

Table 1: An overview of the levels and tools in RoboBUG

Level	Tools	Description
Level 1	Bugcatcher	The Mech Suit is unable to stand because it cannot correctly calculate the physical forces acting upon it. The player must practice **code tracing** by identifying bugs while reading through source code that calculates the average value of a set of physical forces.
Level 2	Bugcatcher, Activator	The Mech Suit is failing to correctly identify the most dangerous creatures that appear in its viewing area. The player needs to use **print statements** to identify program bugs in an algorithm that sorts the externally viewable bugs from most to least dangerous (threat assessment).
Level 3	Bugcatcher, Activator, Commenter, Warper	The Mech Suit vision system has been infected and no longer functions at all. To fix it, the player must search for a bug in the robot's visual color database. Since the database is large, the player will need to employ a **divide-and-conquer** strategy and comment out different blocks of source code.
Level 4	Bugcatcher, Activator, Breakpointer, Warper	The Mech Suit is not able to correctly calculate which creatures are closest in proximity. This is the most challenging level, requiring the player to use several debugging tools to locate bugs across multiple functions. This includes the use of **breakpoints** to display variable values and program state at run-time while locating the bug in a distance calculation function.

physical forces acting upon the 'Mech Suit'. However, an extraneous line of source code is present that increments the average after it has been calculated correctly.

3.1.2 *Level 2: Print Statements.*

- *Subproblem A:* The player is presented with a function that should swap two numbers; however, the final line of code performs the operation in reverse, and assigns the values incorrectly. This behavior can be identified when the player enables the appropriate print statements which show that the final result only includes one correctly swapped value.
- *Subproblem B:* The swap function from the Subproblem A has been incorporated into a function that sort a list of numbers that unfortunately includes an out-of-bounds indexing error. Using print statements the player can discover this area and identify that the bug is in the loop condition (a '>=' sign is used instead of a '>').
- *Subproblem C:* The final expanded source code is part of the threat assessment component in the 'Mech Suit' and contains a function that sorts a list of threat rankings. The print statements in this level show the state of the list at different stages of sorting. Observing each print statement should help the player realize that the first element of the list is accidentally not sorted and leads to an incorrect list.

3.1.3 *Level 3: Divide and Conquer.*

- *Subproblem A:* The player is presented with a large list of integer red-green-blue (RGB) color tuples, which each range between 0 and 255; these represent different colors in the 'Mech Suit' vision subsystem. A print statement at the beginning of the function indicates that one of the green values is out of bounds. By using a divide-and-conquer approach to commenting out the different code sections for different colors, the player can identify that one of the colors has an out of range green value.
- *Subproblem B:* The code in this level consists of multiple large lists of RGB colors, and a print statement at the beginning of the code indicates that there is an invalid blue color value. The player must use the commenter tool to comment out each list of colors until the error is located.
- *Subproblem C:* This level is similar to Subproblem B, except the source code color lists are divided across multiple files that must be checked separately. The player uses the commenting tool to comment out each file until they discover which file contains the error. Using their 'warper' tool, they can then warp to that file and isolate the invalid value.

3.1.4 *Level 4: Breakpoints.*

- *Subproblem A:* The source code in this part takes two pairs of numbers representing a location (x and y coordinates), and indicates which pair has a lower magnitude. The player uses code tracing to discover that there is a '=' instead of a '==' in a comparison statement. While this part of the level does not use breakpoints it is included to show the benefits of breakpoints in the subsequent parts of the level.
- *Subproblem B:* This part of the level requires the player to use breakpoints to check the values of coordinates used in each function, and identify a small logic error.
- *Subproblem C:* This part is similar to Subproblem B, but contains an error where a variable is unintentionally reassigned instead of used in a calculation.
- *Subproblem D:* The source code in the final subproblem of this level contains a small logic error in a purposefully obfuscated calculation. The error is obfuscated to enhance the benefits of breakpoints in finding the bug. This source code is from the 'Mech Suit' subsystem that calculates locations to target.

3.2 Game Mechanics

In order to complete a level, the player must navigate the avatar in the code region of the game interface (see Figure 1) using the arrow keys. Once a bug has been found, the player uses the *bugcatcher* tool to "capture" the bug at a specific line of code. If the location is incorrect, the player fails the level and must start it again. The player can also fail if he or she expends all of the available tools, or does not complete the level within the time allotted. As the game progresses,

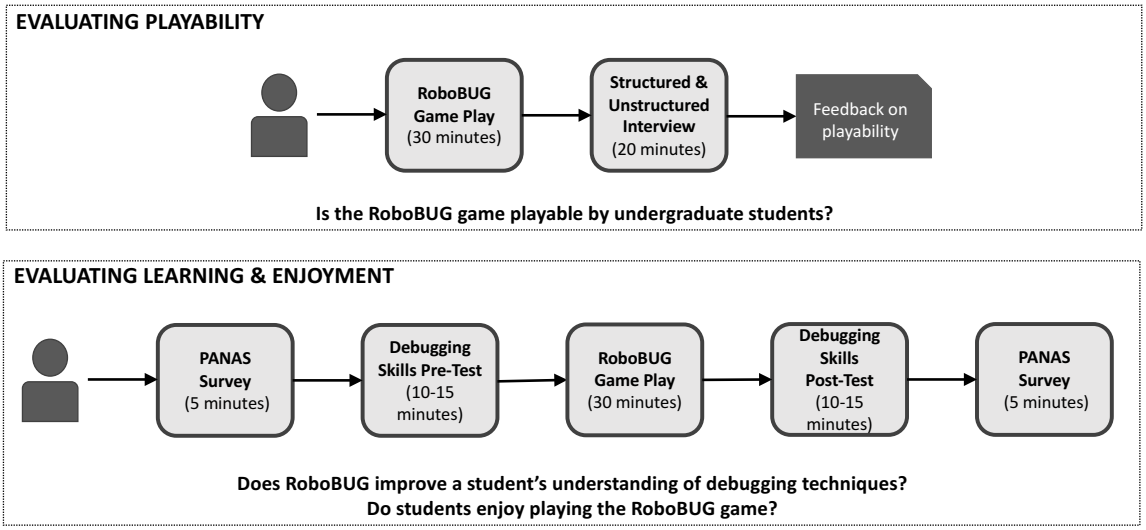

Figure 3: The RoboBUG evaluation methodology

the player is given access to additional kinds of tools that can activate print statements (*activator* tool), comment out source code (*commenter* tool), set and trigger breakpoints (*breakpointer* tool) and jump to different regions of the program (*warper* tool). Completion of each level requires the player to use the tools available before using the 'bugcatcher' tool to complete the level.

4 ROBOBUG EVALUATION

Serious games, including those in the Computer Science education literature, tend to be published without a proper evaluation [12] making it difficult to assess their impact on learning. The most effective type of evaluation to determine the efficacy of a game for learning is a user study [6]. In our evaluation of RoboBUG we have conducted three separate user studies with 23, 5 and 14 participants respectfully. Our first study (Section 4.1) was a pilot study of a first prototype of RoboBUG, our second study (Section 4.2) assessed the playability of the refined version of RoboBUG and our third study (Section 4.3) assessed the enjoyability and the achievement of learning outcomes in the refined version of RoboBUG.

4.1 Pilot Study

A pilot study of an early RoboBUG prototype assessed the value of the game in comparison with traditional assignment-based learning. All participants were undergraduate students at UOIT with knowledge of the C++ programming language. Participants were split into two random groups: a control group of 12 participants who completed a short assignment, and an experimental group of 11 participants who played the RoboBUG game. Both the assignment and the game were based on the same debugging techniques (see Section 2.1) and included similar source code – each level in the RoboBUG game had a corresponding assignment question.

The evaluation found no significant difference with regards to achieving learning outcomes between the assignment-based learning activity and the RoboBUG prototype. We believe this result

1 Very Slightly / Not at all	2 A Little	3 Moderately	4 Quite a bit	5 Extremely

1. Interested	_____	5. Strong	_____
2. Distressed	_____	6. Guilty	_____
3. Excited	_____	7. Scared	_____
4. Upset	_____	8. Enthusiastic	_____

Figure 4: Positive-Negative Affect Scale (PANAS) [22]

The Positive-Negative Affect Scale is a self-evaluation assessment of affect based on words that associate with positive or negative emotions. Total affect is calculated by adding all of the positive or negative item ratings, with higher scores representing higher affect levels.

was impacted by the fact that study participants who played the RoboBUG prototype found it complicated, and some participants were not able to complete the game without hints. Despite this, participants who played RoboBUG tended to find it to be more 'fun'. After the results of the pilot study RoboBUG was updated to address these issues, by subdividing levels, reducing complexity, and providing opportunities for players to fail and replay the levels.

4.2 Evaluating Playability

To evaluate the current version of RoboBUG, we first conducted a user study to address the following research question:

- *Is the RoboBUG game playable by undergraduate students?*

This was an important research question to answer first because not identifying and addressing issues with playability could serious impact our ability to assess the learnability and enjoyment of RoboBUG. In other words, we don't want design and technical issues to confound our evaluation of RoboBUG's potential as a learning tool for debugging.

Our evaluation involved the participation 5 first year Computer Science students at UOIT who were familiar with C++. Participants were between the ages of 18 and 25, with mixed demographics. Participants individually took part in a 1 hour session during which they were observed playing the RoboBUG game for at least 30 minutes (see Figure 3). Following the game play, the participants completed a 20 minute interview where they answered both structured and unstructured questions about their experience, including:

- What did you learn about debugging that you didn't know before?
- What aspect/part of the game was most enjoyable?
- What aspect/part of the game was the most frustrating?
- What aspect/part of the game was most innovative?
- What aspect/part of the game would you like to see improved?

During the interview, participants provided feedback about parts of the game where they became stuck or frustrated. The goal was for RoboBUG to be playable before measuring its efficacy as a game.

Overall, the game was viewed positively by the participants, who particularly enjoyed the game elements that differentiated RoboBUG gameplay from real debugging tasks. During the interviews, the participants gave the following opinions:

- *"[I enjoyed] trying to test my skills with how good I am with debugging."*
- *"It's a great tool, that's what I can say."*
- *"The way that the divide and conquer was set up was pretty cool."*
- *"The inclusion of breakpoints was kind of innovative."*
- *"[The warper tool] was interesting because I thought all of the code would be in one class."*
- *"I think that the warper/commenting, being able to zip between different segments of code was really good."*

While participants enjoyed playing RoboBUG, our study did identify some important playability issues with the game, including control problems and concerns with some of the game's levels. In particular, participants in all our evaluations had significant challenges trying to debug a level containing an off-by-one index bug. This bug was cited by participants to be especially frustrating, due to players having trouble identifying print statements that would help them find the bug. There was also some confusion with the way that the game handled commenting out source code, as players did not realize that a persisting error meant the bug was **not** commented out. A frequently requested change to the game was the idea of a 'hint' system that would provide better feedback to players who become frustrated or fail to complete levels.

RECALL – WHAT?

What can **print statements** be used for in debugging?

a. Outputting the value of a particular variable
b. Indicating the code that is not run during execution
c. Printing a fixed version of buggy code
d. Separating buggy code from bug-free code using text

UNDERSTANDING – WHEN?

Suppose you are debugging code where you need to know the values of variables during run-time. Which methods are appropriate?

e. Print statements or breakpoints
f. Breakpoints or divide-and-conquer
g. Divide-and-conquer or print statements
h. Print statements, breakpoints or divide-and-conquer

APPLICATION – HOW?

In the code below, where is the best place for a breakpoint if you want to find out the array values during each iteration of the sort?

i. Line 5
j. Line 7
k. Line 9
l. Line 11

```
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. }
```

Figure 5: Sample Skill-Testing Questions

*Listed in this figure are three of the ten questions included in the skill test given to participants before and after gameplay. The test was divided into three categories of questions: **recall** about **what** the techniques were, **understanding when** to use each technique, and **application** of **how** to use debugging techniques.*

Positive Keyword	Average Change	Negative Keyword	Average Change
Interested	-0.64	Anxious	-0.21
Enthusiastic	-0.64	Nervous	-0.14
Alert	-0.50	Guilty	-0.07
Excited	-0.36	Stressed	-0.07
Determined	-0.36	Depressed	-0.07
Attentive	-0.36	Scared	0.00
Proud	-0.14	Distressed	0.07
Inspired	-0.14	Hostile	0.07
Happy	-0.07	Jittery	0.29
Confident	0.00	Afraid	0.29
Active	0.07	Irritable	0.36
Strong	0.14	Upset	0.43
		Ashamed	0.43

Figure 6: Positive-Negative Affect Scores

*This graph shows the average change of affect for all participants based on each question on the PANAS. Positive scores indicate that participants associated **more** with that emotion after playing RoboBUG, and negative scores indicate that players associate **less** with that emotion after playing RoboBUG.*

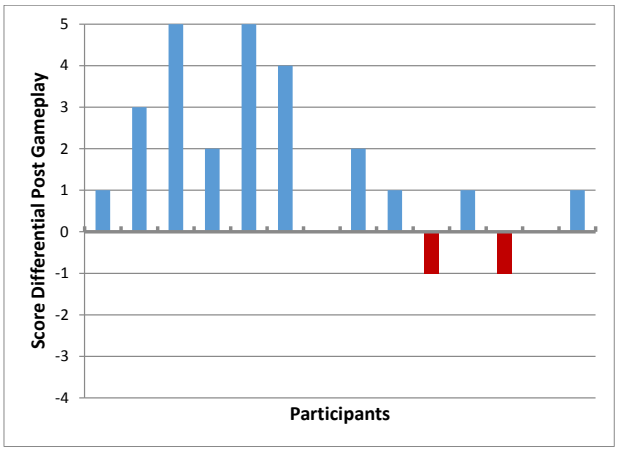

Figure 7: Change in Skill Test Scores by Participant after playing RoboBUG

Each bar represents a single participant's score differential on the Skill Test after playing RoboBUG. The participants are ordered from left to right based on increasing initial Skill Test scores (before playing RoboBUG). An interesting observation is that the largest positive score differentials were achieved by students with the lowest initial Skill Test scores.

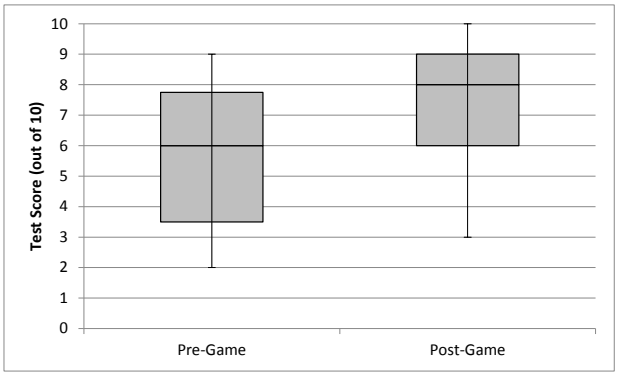

Figure 8: Box plot of the Skill Test Scores before and after playing RoboBUG

4.3 Evaluating Learning and Enjoyment

To further evaluate the current version of RoboBUG, we conducted a second user study to address the following research questions:

- *Does RoboBUG improve a student's understanding of debugging techniques (i.e., achieve learning outcomes)?*
- *Do students enjoy playing the RoboBUG game?*

Our evaluation again involved the participation of first year Computer Science students at UOIT between the ages of 18 and 25, with mixed demographics, gender, and race.

In this study we evaluated the game's ability to help students achieve learning outcomes as well as the user experience. This study involved a larger sample size than the accessibility study (14 students) and took approximately 1 hour to complete. Participants in this study first completed the Positive and Negative Affect Scale (PANAS) [22], which assesses the user's feelings (see Figure 4). The PANAS scale in our study contained 12 positive words and 13 negative words. After completing PANAS, participants completed a debugging skills pre-test. This pre-test included ten multiple choice questions about the four debugging techniques used in the game (see Section 2.1). These questions tested participant abilities to recall, understand, and apply the debugging techniques. The first four questions tested knowledge about the usage of each debugging technique. The next four questions tested understand of when the techniques should be used. Finally, the last two questions involved the application of the techniques themselves. Examples of these questions can be observed in Figure 5. Once the pre-test was complete, participants played the RoboBUG game for approximately 30 minutes. After the game, participants completed the debugging skills test and the PANAS questionnaire again.

The results of both the pre- and post-game PANAS data and skill test data were analyzed to identify any changes in positive and negative effect as well as debugging skills. The analysis was conducted using a paired t-test (see Figure 9). Our results indicate that RoboBUG helps students to achieve the debugging learning outcomes (see Figure 7 and Figure 8). Players became familiar with the nature of the debugging techniques, and were able to practice debugging and solve problems in a satisfying manner. In addition, the largest improvements in test scores were observed for participants with low initial test scores, suggesting that the game is most helpful for participants who are in the greatest need of assistance. Unfortunately, there was a non-statistically significant decrease in positive affect and a non-statistically significant increase in negative affect, indicating that the game still led to some user frustrations. It is possible that the game remains less frustrating than real debugging tasks, but our observations of participants suggest that the lack of

Paired-samples t-test				
	Mean	Std. Dev.	t(13)	p value
Test Scores			3.0970	0.0085
Pre-Game	5.71	2.46		
Post-Game	7.36	2.10		
Positive Affect			2.1272	0.0531
Pre-Game	45.93	5.40		
Post-Game	42.93	8.92		
Negative Affect			0.7555	0.4634
Pre-Game	18.86	5.67		
Post-Game	20.21	10.64		

Figure 9: Paired-samples t-test

There was a significant increase in debugging test scores after the game was played. No significant changes in positive or negative affect scores were observed.

a hint system and the difficulty of the tasks were major challenges. Ultimately, our game still requires participants to debug code and completely removing frustration related to debugging remains an open problem.

5 SUMMARY & CONCLUSIONS

We have presented the RoboBUG game as a serious game solution to the challenge of learning debugging in first year Computer Science courses. RoboBUG was evaluated for playability, learning benefits and enjoyment. Our evaluation of RoboBUG showed that the game helps students to achieve learning outcomes, but has a non-statistically significant impact on enjoyment (positive and negative affect). In addition, the game seemed to be most effective at aiding students who were not initially skilled at debugging. The RoboBUG game and source code are available online at https://github.com/sqrlab/robobug.

The short length of the game, chosen to fit within the experiment time frame, meant that we had to limit the amount of content we could include. It is possible that different effects on learning benefits and enjoyment might be observed with an extended play session, or with added new content. RoboBUG is designed to make the addition of levels accessible for instructors, but introducing new game mechanics requires further work from the game developers.

Since the completion of our study, we have continued to improve RoboBUG by updating the interface design elements, implementing a hint system to reduce frustration and enhancing the ability to extend RoboBUG with new levels. In addition, we have also developed a prequel game that will allow RoboBUG to be played by users who have limited programming experience [15]. Future areas of work include the addition of a points system for competitive play, adding a cooperative multi-player mode, and improving the replay-ability by using program mutation [2] to generate random bugs each time a level is played.

In addition to enhancing the RoboBUG game we are also focusing on additional evaluation. Specifically, we are in the process of conducting a longitudinal study of RoboBUG in a first year programming course at UOIT and believe this larger in-class study will complement the information from our controlled experiments.

6 ACKNOWLEDGMENTS

This research was partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC). We thank the reviewers for their thoughtful comments and suggestions.

REFERENCES

[1] Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. 2005. An analysis of patterns of debugging among novice computer science students. In *Proc. of 10th SIGCSE Conf. on Innovation and Technology in Comp. Sci. Education (ITICSE '05)*. 84–88.

[2] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proc. of International Conference on Software Engineering 2005 (ICSE '05)*. 402–411.

[3] Elizabeth Carter and G.D. Blank. 2014. Debugging Tutor: preliminary evaluation. *J. of Computing Sciences in Colleges* (2014), 58–64.

[4] Mei-Wen Chen, Cheng-Chih Wu, and Yu-Tzu Lin. 2013. Novices' debugging behaviors in VB programming. In *Proc. of Learning and Teaching in Comp. and Eng. (LaTiCE 2013)*. 25–30.

[5] Du Chuntao. 2009. Empirical study on college students' debugging abilities in computer programming. In *Proc. of 1st Int. Conf. on Info. Sci. and Eng. (ICISE 2009)*. 3319–3322.

[6] Heather Desurvire, Martin Caplan, and Jozsef A. Toth. 2004. Using heuristics to evaluate the playability of games. In *Proc. of 2004 Conference on Human Factors in Computing Systems (CHI '04) - Extended Abstracts*. 1509–1512.

[7] Sue Fitzgerald, Renée McCauley, Brian Hanks, Laurie Murphy, Beth Simon, and Carol Zander. 2010. Debugging from the student perspective. *IEEE Trans. on Education* 53, 3 (2010), 390–396.

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

[9] Morgan Hall, Keri Laughter, and Jessica Brown. 2012. An empirical study of programming bugs in CS1, CS2, and CS3 homework submissions. *J. of Comp. Sci. in Colleges* 28, 2 (2012), 87–94.

[10] Roslina Ibrahim, Rasimah CM Yusoff, Hasiah M Omar, and Azizah. Jaafar. 2010. Students perceptions of using educational games to learn introductory programming. *Comp. and Info. Sci.* 4, 1 (2010), 205–216.

[11] Cagin Kazimoglu, Mary Kiernan, Liz Bacon, and Lachlan Mackinnon. 2012. A serious game for developing computational thinking and learning introductory computer programming. *Procedia - Social and Behavioral Sciences* 47 (2012), 1991–1999.

[12] Fengfeng Ke. 2009. A qualitative meta-analysis of computer games as learning tools. *Handbook of Research on Effective Electronic Gaming in Education* (2009).

[13] Michael J Lee and Andrew J Ko. 2014. A demonstration of gidget, a debugging game for computing education. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*. IEEE, 211–212.

[14] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18, 2 (2008), 67–92.

[15] Michael A Miljanovic and Jeremy S Bradbury. 2016. Robot ON!: a serious game for improving programming comprehension. In *Proc. of the 5th International Workshop on Games and Software Engineering*. ACM, 33–36.

[16] Mathieu Muratet, Patrice Torguet, Jean-Pierre Jessel, and Fabienne Viallet. 2009. Towards a serious game to help students learn computer programming. *Int. J. of Comp. Games Tech.*, 1–12.

[17] Jackie O'Kelly and J. Paul Gibson. 2006. RoboCode & problem-based learning : A non-prescriptive approach to teaching programming. In *Proc. of 11th SIGCSE Conf. on Innovation and Technology in Comp. Sci. Education (ITICSE '06)*. 217–221.

[18] Valerie J Shute. 2011. Stealth assessment in computer-based games to support learning. In *Computer Games and Instruction*, Vol. 55. 503–524.

[19] A.C. Siang. 2003. Theories of learning: a computer game perspective. In *Proc. of 5th Int. Symp. on Multimedia Soft. Eng. (ISMSE 2003)*. 239–245.

[20] Beth Simon, Sue Fitzgerald, Renée McCauley, Susan Haller, John Hamer, Brian Hanks, Michael T Helmick, Jan Erik Moström, Judy Sheard, and Lynda Thomas. 2007. Debugging assistance for novices. In *Working Group Reports on Innovation and Tech. in Comp. Sci. Education (ITiCSE-WGR '07)*. 137–151.

[21] Nikolai Tillmann and Judith Bishop. 2014. Code Hunt: searching for secret code for fun. In *Proc. of 7th Int. Work. on Search-Based Soft. Testing (SBST 2014)*. 23–26.

[22] David Watson, Lee a. Clark, and Auke Tellegen. 1988. Development and validation of brief measures of positive and negative affect: The PANAS scales. *J. of Personality and Social Psychology* 54, 6 (1988), 1063–1070.

[23] Wai-Tak Wong and Yu-Min Chou. 2007. An interactive Bomberman game-based teaching/learning tool for introductory C programming. In *Proc. of 2nd Int. Conf. on Edutainment*. 433–444.

[24] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.