The Effects of Adaptive Procedural Levels on Engagement and Performance in an Educational Programming Game

Conference Paper · November 2022

DOI: 10.1145/3555858.3555892

Chaima Jemmali
Northeastern University
12 PUBLICATIONS **195** CITATIONS
SEE PROFILE

Magy Seif El-Nasr
University of California, Santa Cruz
283 PUBLICATIONS **4,424** CITATIONS
SEE PROFILE

The Effects of Adaptive Procedural Levels on Engagement and Performance in an Educational Programming Game

Chaima Jemmali
Northeastern University
United States of America
jemmali.c@northeastern.edu

Magy Seif El-Nasr
University of California at Santa Cruz
United States of America
mseifeln@ucsc.edu

Seth Cooper
Northeastern University
United States of America
se.cooper@northeastern.edu

ABSTRACT

Learners' backgrounds, skills, and knowledge vary as they attempt to learn a new subject. To address this variation and allow learners to progress at their own speed, many researchers are suggesting adaptive learning as a solution. Adaptive content has been successful in learning environments such as intelligent tutoring systems, but it has not been thoroughly researched within video games, especially in terms of adaptive procedural levels. In this paper, we analyze the effects of procedural levels that are generated and inserted at run-time in between pre-designed levels in the educational programming game May's Journey. Our study with 94 Amazon Mechanical Turkers shows that players encountered fewer code-related errors in the adaptive version, however, their engagement levels were similar, if not slightly higher in the non-adaptive version.

CCS CONCEPTS

• **Human-centered computing** → *Empirical studies in HCI*; • **Applied computing** → *Interactive learning environments*.

KEYWORDS

adaptive game, programming game, educational game, adaptivity assessment, debugging

ACM Reference Format:
Chaima Jemmali, Magy Seif El-Nasr, and Seth Cooper. 2022. The Effects of Adaptive Procedural Levels on Engagement and Performance in an Educational Programming Game. In *FDG '22: Proceedings of the 17th International Conference on the Foundations of Digital Games (FDG '22), September 5–8, 2022, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3555858.3555892

1 INTRODUCTION

Game-based learning holds a great promise for deep and innovative learning opportunities. Researchers have identified how games can be be used in multiple educational contexts to teach a variety of subjects. Educational games have been proven to be effective at improving learning outcomes [3, 7, 53]. In addition to enhancing learning, game-based learning has also been suggested to foster student's engagement in learning activities [16]. Despite the prevalence of games to teach programming, there is little evidence of the effectiveness of these games or their learning outcomes [15, 25, 39]. In fact, many of these programming learning environments can be either too simple to effectively teach programming or too complicated for beginners, especially when faced with debugging. For the latter, in the absence of external help, players can get stuck, feel frustrated, and may quit if a challenge is too difficult or if they have not fully mastered the learning content for advanced levels. There is therefore a need to have an internal help or practice mechanism within the game that targets individual players.

The benefits of adaptive content are acclaimed by researchers throughout the online education spectrum, whether for tools, games, or intelligent tutoring systems. In this paper, we discuss adaptivity in terms of level generation, which means that levels are generated at run-time, for a specific purpose, for a particular player according to some in-game heuristics or player model. To the best of our knowledge, the only existing programming adaptive game does not adapt in terms of levels but in aspects related to hints and difficulty [18]. Further, there is very little research that assesses procedurally generated levels that are seamlessly integrated within a game in general. This lack of assessment makes it difficult to decide when and how to integrate such levels, which we discuss in following sections.

In this paper, we attempt to answer the research question "Can targeted debugging practice levels result in stronger engagement, more levels played, and better debugging performance?". In our case, we refer to debugging as fixing compile-time errors. We focus on debugging, since it is one of the greatest hurdles when learning programming [26, 29, 40]. We conducted a study with 94 Amazon Machanical Turkers who played a programming narrative puzzle game called *May's Journey*. Players played one of two versions: adaptive or non-adaptive. In the adaptive version, we integrated a procedural level generator that creates levels similar to existing levels of the game at run-time [22]. The result is a version of the game that provides targeted levels to debug a specific error message that the player was encountering. Our hypotheses are as follows:

- *H1:* Players in the adaptive version will show better in-game performance than players in the non-adaptive version in terms of the average number of errors per level, their ability to respond to error messages, and levels completed.
- *H2:* Players in the adaptive version will have fewer errors on average after encountering a targeted practice level.
- *H3:* Players in the adaptive version will report stronger engagement with the game in the post-game survey.

2 RELATED WORK

We divide our related work into three main sections where we discuss previous research on practice and learning, then more specifically practice for debugging and programming, and finally adaptivity in existing educational games.

2.1 Practice Problems

Practice can improve accuracy and speed of performance in cognitive, perceptual, and motor skills, under certain conditions [12, 13, 51]. One of the conditions for optimal learning is the learners' motivation to attend to the task and exert effort to improve their performance [11]. These findings are also reiterated for programming practice. In a study about the relationship between the voluntary practice of short programming exercises and exam performance, Edwards et al. [10] found that voluntary practice did improve performance on exams, but also that motivation may play an important role. However, merely practicing may not be sufficient and tasks should take into account pre-existing knowledge of the learners and provide adequate feedback [11]. For this reason, we make sure that the adaptive practice levels we introduce target specific error messages that players could not debug.

2.2 Practice Problems for Debugging

Several tools teach debugging, from tutors [6, 28, 32] to games targeting debugging specifically [34, 38] or debugging through domain knowledge [14, 24], which is the type of knowledge targeted by most tools surveyed by Li et al. [35]. Most of these tools are also pre-designed and do not adapt to players' specific needs. Carter's tutor [6] uses a case-based reasoning approach where a case contains a defect and represents a specific troubleshooting setting. The cases are generated dynamically using templates specific to programming topics but do not adapt to learners. To help learners debug C++ programs, Kumar [28] created an Intelligent Tutoring System using Model-based Reasoning for domain modeling. The tutor can generate problems from templates and give feedback in the form of code and state explanations. However, while it theoretically can adapt to students, this feature has not been explained or studied in the paper. On the other hand, Lee et al.'s tool Debug It [32] adapts to player's performance but relies on pre-defined problems which makes the adaptivity less targeted. Further, all these tools use exclusively exercises of the type "Modify" in the taxonomy of the Use-Modify-Create (UMC) framework [33]. In "Use" exercises, students use code and maybe only modify a small thing such as a literal value. In "Modify", they are given incomplete or incorrect code. Finally, in "Create", students will design their own code. It is suggested that students learn better with interleaved exercises than blocked ones [47]. It is therefore important to have the students practice with all types of exercises from the UMC framework. Crescendo [50], a tool for practicing programming successfully used interleaved problems and their preliminary results showed that it can be effective in leading students into independent, motivating, and rewarding programming experience. To the best of our knowledge, there are no existing tools that dynamically generate levels that target specific error messages that a player is struggling with.

2.3 Adaptive Educational Games

Educational games are mostly designed with a fixed progression and predefined levels. However, several researchers calling for dynamic tailoring of difficulty on a per-player basis have emerged [54]. This push for adaptive content in educational games is driven by the success and the popularity of Intelligent Tutoring Systems (ITS) [52] and Adaptive Hypermedia [5].

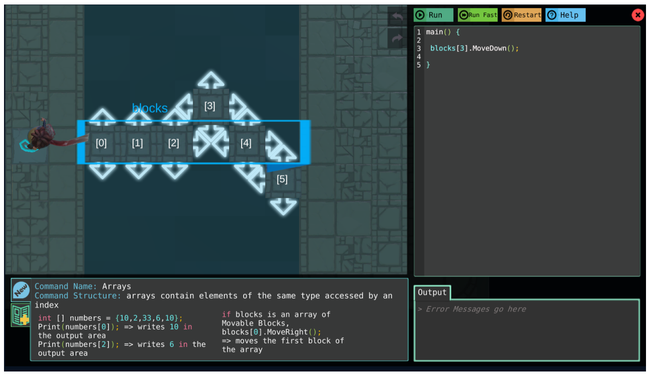
Procedural Content Generation has been used successfully in commercial games such as *Diablo*[1] and *No Man's Sky*[2], however, the usage of PCG within adaptive educational games remains limited, and when available is mostly focused on areas such as Dynamic Difficulty Adjustment, path-finding, and NPC (Non-Playing Character) [8, 48]. In General, the use of PCG in educational games is still limited, and when used the effects of adaptivity are not always tracked. For example, in a study assessing the effects of large-scale campaigns for education, Liu et al. [36] used *DragonBox Adaptive*, an algebra teaching game that provides students with different sets of dynamically generated problems for additional practice depending on how they performed on embedded assessments. However, since there was no data from the original game *DragonBox*, they could not assess the effects of the adaptivity. Besides adaptivity, there are noteworthy instances of non-adaptive applications of PCG in educational games. For example, Refraction [44] an educational game that teaches mathematical skills and problem-solving has procedurally generated puzzles. However, Refraction simply replaces a typical human-designed level with a computer-designed level; it does not attempt to adapt during gameplay. Another example is GrACE [19], an educational game that teaches Computational Thinking (CT) through solving puzzles related to Minimum Spanning Trees. GrACE allows players to request new procedurally generated puzzles at the same difficulty level throughout the game. When testing the effects of the PCG in-game, they found that when PCG is used in an individual context, it increased learning gains as a result and that was especially noticeable for the more abstract puzzles which can promote abstract thinking according to the authors. While these are promising results, the level generation was not integrated adaptively. Autothinking is another game that teaches CT skills [18] through a maze-based environment similar to Pac-Man but with delayed controls. The game uses adaptive content through the manipulation of the NPCs (the cats trying to catch the mouse representing the player) and through providing hints. However, it does not adapt in terms of content or level generation.

In contrast, our approach includes procedurally generated levels created during run-time that are encountered in between predesigned levels to provide more practice before progressing into the game. The generated levels target a specific error message that a player was encountering and are blended in the storyline to make a cohesive transition and preserve flow.

3 METHODOLOGY

In this section, we first introduce the game we are using, the steps of generating an adaptive level and the way we integrate it into the original game.

[1]https://diablo2.blizzard.com/en-us/
[2]https://www.nomanssky.com/



```
main() {

 blocks[3].MoveDown();

}
```

Command Name: Arrays
Command Structure: arrays contain elements of the same type accessed by an index

```
int [] numbers = {10,2,33,6,10};
Print(numbers[0]); => writes 10 in the output area
Print(numbers[2]); => writes 6 in the output area
```

```
if blocks is an array of Movable Blocks,
blocks[0].MoveRight();
=> moves the first block of the array
```

```
> Error Messages go here
```

Figure 1: First level introducing arrays. Players need to move the elements with index 3 and 5 of the array of blocks down and up respectively. Players can type on the right side, where they are given the first command. On the bottom left, they are introduced to arrays with examples and explanations. In the middle they can see their code execute in real-time.

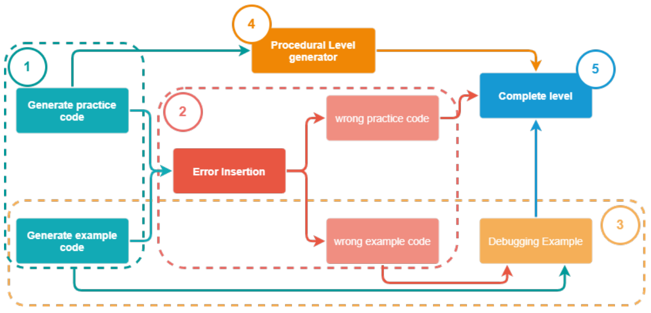


Figure 2: Steps of Generating an adaptive Level. Step 1) Generating code that can solve a level and a similar code to provide an example. Step 2) The last error encountered by the player is inserted to each code. Step 3) The wrong and correct example codes are used to provide an example of how to debug an error. Step 4) The correct practice code is used to generate a level. Step 5) The example and wrong practice code are added to the level interface.

3.1 May's Journey

May's Journey is a 3D narrative puzzle game where players navigate a broken game world that is made of code [21, 23]. Players are tasked to help the main protagonist *May* understand the mystery of the recent events that broke the code of the world. Throughout the journey, players need to code their way out of various puzzles, collecting artifacts, meeting NPCs, and acquiring companions. The most notable character the players meet is the *Oracle*, who is a powerful being, that helped connect *May* with the player and assumes the role of guide and mentor throughout the game. The gameplay can be explained as two interleaving phases: exploration and programming. Various areas of the world are inaccessible until the player solves coding challenges. These coding challenges involve interacting with the objects in the environment through code by moving, rotating blocks and platforms, lighting fire platforms, activating pressure plates, or changing values of variables such as block weights. The game's programming language is a simplified custom object oriented language where the emphasis was made on making error messages simple, understandable, and accessible for beginners.

The learning content progresses from simple instructions, to loops, variables, inputoutput, conditionals, and arrays. An example level can be seen in Figure 1 where players are introduced to arrays. Players type in their code in the coding interface on the right and make two types of errors; code-related or puzzle-related, which can be seen in the output area (bottom right). Code-related errors can be either purely syntactic or semantic such as using the wrong type of variable or function. Puzzle errors can be seen as logic errors: the code compiles with no errors but it does not solve the problem. In this study, we focus on code-related errors.

3.2 Adaptive Level Generation

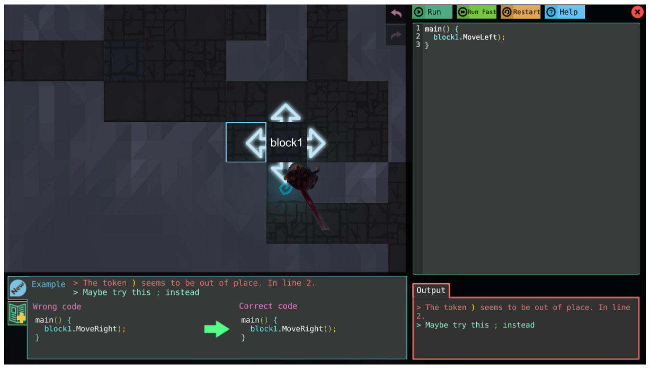
The goal of the level generation is to provide players with an easier level in which they can practice debugging an error they were encountering in a pre-designed level. The idea of an easy practice is to isolate a specific error, since their code could contain more than one, which would make it harder to identify and debug. Figure 2 shows all the steps needed to generate a fully playable adaptive level in an automated way.

- *Step 1:* Generate 2 codes; one will be input to the procedural level generator described below (practice code, top-right code in figure 3), and the other will be used to create an example to the player (example code, bottom left code in figure 3).
- *Step 2:* Given the last error that the player has encountered, insert that error to both practice and example codes.
- *Step 3:* Given the wrong and correct example codes, create an example for the player to follow when solving the level.
- *Step 4:* Generate the level given the input practice code from *Step 1.*
- *Step 5:* Populate the generated level's coding interface with the wrong practice code that the player needs to correct and provide the debugging example generated in *Step 3.*

These steps are explained in further detail in the following subsections. In figure 3 the practice code, example code, and the error messages associated with them are showcased as they would appear in the game.

3.2.1 *Step 1: Code Generation.*

Several tools exist that automatically generate programming problems, and most of them use either a template-based approach or Context-Free Grammars (CFG). Template-based approaches [27, 41, 42, 49] provide the generator with a structure to fill or with a learning objective containing metadata. The advantage of this approach is that it gives finer control over the generated problems, and it also makes it easier to ensure that the code generated is well-formed and executable. However, this tight control will also limit the variety of problems generated. CFGs [1, 17] are similar to the template-based approaches, where the structure of the desired input is specified with a context-free



```
main() {
  block1.MoveLeft);
}
```

Example > The token) seems to be out of place. In line 2.
> Maybe try this ; instead

Wrong code

```
main() {
  block1.MoveRight);
}
```

Correct code

```
main() {
  block1.MoveRight();
}
```

Output

```
> The token ) seems to be out of place. In line
2.
> Maybe try this ; instead
```

Figure 3: Example of a procedurally generated level coding interface, with the example on the bottom left and the initial code on the right.

grammar G. Then, the generator reads G and uses its rule derivations to produce $L(G)$, which is the language accepted by G. While this methodology ensures that the code is well-formed, it requires extensive derivation rules and must be very specific to a particular problem or language.

Our generator uses a combination of these two techniques where depending on the construct needed, it will either use a grammar or a template. Our templates are simple and represent codes that include if statements and while loops. We choose templates for these types of codes because the way these levels exist in the game is very specific and generating more varied codes is not useful in our case since they can't be used in the game. For simple instructions and for loops, we use also a simple grammar where the rules represent the type of object and the command applied to it and arguments if any. The number of objects and commands applied to each can be pre-selected or random. Both of these approaches do not generate the code directly but rather generate an Abstract Syntax Tree (AST) that is then translated into code. Similar to our level generator, this step is added to keep our generator modular and applicable to different languages and situations.

3.2.2 Step 2: Error Insertion. To create practice levels for specific error messages, we need to be able to insert these errors into our correct solution code. During some initial data processing, we identified the most common error type to be "missing X", which in general refers to missing ';' or missing '}'. This example is the easiest to create an erroneous code that contains this error by simply removing one of these tokens. However, one of the most common errors was "The token X is out of place", which could mean very different errors. We have manually analyzed the data and for each error that can be introduced more than once, we looked for how it had occurred. Table 1 shows the errors we can insert in our levels and how they can be inserted. For example, if the error is the attribute does not exist, it is not necessarily a syntax error. In many cases, players do not add the parentheses of the command which leads to the compiler treating it as an attribute.

3.2.3 Step 3: Example Creation. Given the correct example code generated in step 1 and its erroneous version generated in step 2,

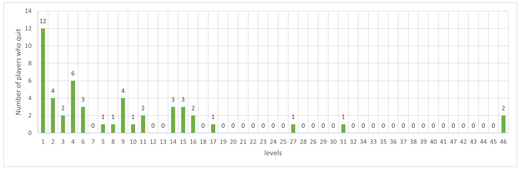


Figure 4: The number of players quitting at each level

we provide the players with an example to follow that contains the same error as the one they need to fix. The example can be seen in Figure 3 on the bottom left side of the screen where we can first see the error message, the wrong code and an arrow pointing to the correct code.

3.2.4 Steps 4 & 5: Level Generation. To generate the levels we use a grammar-based level generator that works backwards from a solution code to generate a level that can be solved with that code [22]. The generator insures solvability by construction. The original generator attempts to have levels non-solvable by shorter code solutions, considered as easier solutions, but it still happens in 21% of the cases according to the creators [22]. Since the generation time is short (0.38 seconds), we test for such cases and regenerate if a level can be solved with an easier solution.

Once a satisfying level is created, we populate the coding interface on the right with the practice code that players need to fix, and the example interface on the bottom left with the wrong example code, the error message associated and the equivalent correct example code as can be seen in Figure 3.

3.2.5 Timing & Integration. Timing, whether for adaptive content [45] or for feedback [43], is considered highly important. In fact, it is very important to provide the learner with timely content to avoid frustration or annoyance. However, we could not find best practices or guidelines on "when is the right time to provide adaptive/personalized content?" in an educational programming game. And the question is rarely specifically discussed in the literature.

In the absence of a formal procedure, we decided to analyze previous data to decide on an appropriate threshold to present an adaptive level to the player. The data was collected from a previous study conducted on the latest version of the game. The data included 50 players aged between 9 and 14 years, who made at least one code submission. There were 19 female participants, 24 males, 1 non-binary, and 6 students who preferred not to answer. When asked about their programming experience, most of the players (33) reported having experience with block-based programming, 6 players had experience with some text-based programming language, 5 used an object-oriented language, while 6 had no experience at all. When analyzing players' data, we were interested in the quit behavior. Specifically, when do players quit, and how many errors do they encounter before they quit in each level. Our goal is to be able to intervene and offer a procedurally generated level before a player decides to quit.

From our data sample, we could not find obvious patterns after which players quit. In fact, when analyzing their last action in the game, half the time they were in the exploration phase rather than the programming phase. However, we did observe that most of

Table 1: A list of error messages for which a PCG level is created and the ways it can be introduced.

Error	Ways it can be introduced
Missing X	The most common occurrence of this error is with tokens ';' and '}' , but can also be triggered with ')' and '('
The token X seems to be out of place	This error can be triggered in multiple ways: - Missing main method - Code outside of main method - Main method missing both parentheses - Main method missing opening curly bracket - Extra curly brackets after commands
The variable X does not exist in this scope	This can be either a syntax error (messing up capitalization or a typo by adding/forgetting a character) or it can be a semantic error by using a variable name from another level/example
The command X does not exist	Syntax error in the command
The attribute X does not exist	Syntax error in the attribute or removing parenthesis from a command
The command X does not take arguments between parentheses	Adding arguments to a command that doesn't take any, like Move or Stop
The object X cannot be Y	Applying the wrong command on an object (e.g rotating a movable object)
X must be called on an object	Having a correct command without an object

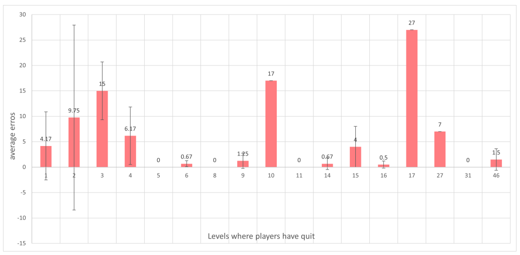


Figure 5: Mean number errors players made before quitting each level

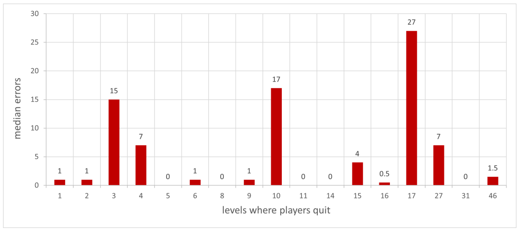


Figure 6: Median number errors players made before quitting each level

our players quit during the first 10 levels of the game. Figure 4 shows the number of players quitting at each level. We can see that 37 of the players quit in the first 10 levels of the game with 12 players quitting in the first level where players are first introduced to programming.

When looking at the average and the median number of errors players make before quitting in each level, there were also no patterns and the variance was high between levels and players. Figures 6 and 5 show respectively the median and mean average errors made by players who quit in that level. Levels that are not shown don't have any players that quit and levels with 0 mean that players quit without getting any error. If we look at the first level as an example, we can see that players make on average 4.17 errors with a large standard deviation. The median number of errors for that level is 1 which means that half of the players quit after only one error. Levels with a high median like 5, 12, or 20 represent only one or two players. Averaging across levels, players make on (mean) average about 5 errors before quitting with a standard deviation of 7.5.

Given all this information, we have decided to introduce a procedurally generated level after 3 errors the first time and randomly between 3 and 5 errors the following times. The error count is reset to 0 after each level and is not activated during the generated level. Further, a level is given only if a player has made at least one code-related error and won't be activated if the errors are all puzzle-related. We have chosen 3 to make sure that we intervene quickly enough the first time, and then added a bit of randomness to make the adaptivity less obvious and less expected.

3.2.6 In-game integration. To make the level generation as seamless as possible and part of the game, we chose to integrate it as a glitch. In fact, since the game world's code is breaking, a glitch is fitting. When a player is coding and the threshold for generating a level is reached, the coding interface disappears, a glitch effect appears for 2 seconds and the player is transported to the generated level where they are greeted by the *Oracle* who explains that the game world is breaking faster than expected and she needed the player to help her fix a program in another area of the world.

Since players are mostly quitting in the first few levels of the game, we limited the application of adaptivity to the first 15 levels of the game. After a small pilot study, we have also discovered that introducing adaptive levels in the first level of the game can be more confusing than helpful, which is why we introduce them starting from level 2. We have also limited the scope of the code generator to only generate code for 1 object with a simple command. Commands are unlocked as the player learns them in the game. The same limitation is applied to the example code. Since the purpose of the generated level is to practice debugging an error in an easier environment, the generated level is created in a way that would minimize any other overload. An example can be seen in Figure 3 where the error is "the token X is out of place". Both the example and the practice code show the same trigger for this error, which is removing the opening parenthesis of a function. If the error is "missing X" the token to remove is chosen randomly in each case and can be the same or different in the practice and example code.

4 USER STUDY

To study the effects of the targeted levels, we conducted a between-subject study with Amazon Mechanical Turkers. We chose them as our participants for their growing popularity amongst researchers looking for a wide range of individuals to generalize their findings [2, 30, 31]. Moreover, through one of our previous user studies, we noticed that they constitute a good base of users who don't usually have programming experience [21]. We limited the recruitment to participants that are located in the United States to limit language issues.

Participants were randomly assigned to one of the game conditions, adaptive or non-adaptive. These conditions differed in whether or not the adaptive practice levels are triggered. In the adaptive version, the levels are triggered as discussed in the previous section, but in the non-adaptive version, nothing additional happens. Each user was asked to sign-up, and provide some demographic information such as age, gender, and programming experience. Then, they would play the game for at least 30 minutes, during which all their game actions are logged. Finally, they can access a short post-game survey where they can provide their impressions of the game. Participants were compensated with $5 for their time.

5 PARTICIPANTS

Our inclusion criteria for the data was that players make at least 2 code submissions. We chose 2 since in the first level players are asked to click run and submit the code given to them as part of the tutorial. This inclusion criterion is also meant to minimize the instances where participants did not actively play the game and simply waited for the timer. These participants were excluded from the study. We ended up with 94 eligible participants with 49 in the adaptive version and 45 in the static version. The gender distribution is as follows: 70 males, 22 females, 1 non-binary, and 1 who preferred not to answer. The mean age of the participants was 35.71 with a standard deviation of 8.96. Surprisingly, when asked about their programming experience, only 32 participants reported no experience while most of them reported being either a professional programmer (25) or pursuing a degree in CS or a related field (28). It is unclear whether this question was always answered correctly, or could have been misunderstood by some players whose data did not reflect their expertise.

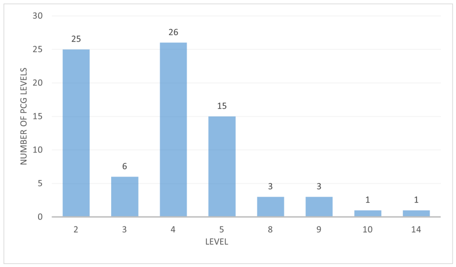


Figure 7: The distribution of how many adaptive practice levels were introduced in each level

Table 2: The errors that had practice levels generated and the number of times they have been generated.

error message	number of levels
The token X seems to be out of place	22
The command X does not exist	16
Missing X before Y	15
The variable X does not exist in this scope	12
Missing X	7
no error	5
The command X does not take arguments between parenthesis	3

6 RESULTS

To understand how players' performance may have been affected in the adaptive version, we first looked at in-game metrics. Our hypothesis is that players in the adaptive version would have fewer errors in general, fewer code-related errors (excluding puzzle errors), and would solve more levels (including procedurally generated levels).

6.1 Adaptive Levels Statistics

First, we take a look at how the adaptive levels were distributed across players. Out of 49 players in the adaptive version, only 26 players have encountered a targeted practice level. A total of 80 levels were generated with an average of about 3 per player. Figure 7 shows when adaptive levels were introduced to players. The highest occurrences appear in levels 4 and 6, which represent respectively the second programming level where players move two blocks and the fourth programming level in which players need to align stairs. As the game progresses, fewer practice levels are introduced both because errors tend to be more puzzle-related at this point and because most players have not made it that far into the game.

In Table 1, we can see which errors were introduced in the PCG levels. The most popular are consistent with popular errors in similar programming languages found in previous work specifically in Java [4, 9, 20, 46] "The token X seems out of place" or "Missing X before Y". There are five levels with no error which means that the error encountered did not have a rule on how to introduce it. In this case, players would receive a correct code that they would just need to run.

6.2 Adaptive vs Non-adaptive

We compare the two versions of the game in terms of four variables; levels attempted, average number of overall errors, and code-related errors, and time spent playing (which refers to the time actively playing, excluding the time where the game browser tab is not in focus). We calculate the mean and median across players for each version and report the Mann-Whitney U test since our data is not normally distributed.

Table 3 shows the results for the four variables we looked at for the participants as a whole. As can be seen from the p values of the tables, none of the results are statistically significant, which means there wasn't a notable difference between the two conditions, (A) for adaptive and (NA) for non-adaptive. Levels attempted is the number of levels in which a player has made a code submission. This number also includes targeted practice levels in the adaptive version (A). From the mean values, we can see a higher average for version (A) but also a very high standard deviation, which means players in the adaptive version had larger differences. However, the medians are exactly the same for levels attempted. A similar trend can be seen in the average number of errors which is the total number of errors divided by the number of levels attempted. We chose to compare the average rather than the absolute value for the number of errors since the number of levels played was not the same and players who reach higher levels would naturally have more errors. Finally as expected, the time spent in the game was very similar with an average of ~27 and a median of ~29 minutes for both versions which is close to the required time they had to play. Some players, however, did not play the full 30 minutes while others played for longer times, even if they didn't have to. The longest play-time was 1h45 minutes and the shortest was ~7 minutes.

Looking at participants as a whole did not show any major differences; however, as mentioned in the previous section, only 26 players in the adaptive version have actually experienced a practice level. Given this information, we decided to compare players who have gotten a practice level against players in the non-adaptive version who *would* have gotten such a level if they were assigned to the adaptive version. The criteria for selection was that players had passed the first level, and had at least 3 errors in one level (one of which should be a code-related error). The criteria guarantee that the player had or would have encountered at least one practice level depending on the game version they were assigned to. This subgroup would represent the players who needed more practice. After the filtering, 47 players were retained, 26 of which were in the adaptive version vs 21 in the non-adaptive version. Table 4 shows the results for the same game variables for this subset of the players. We can see from the table that players in the adaptive version have attempted more levels 15 vs 4 ($p < 0.001$), had less average errors 1.78 vs 4 ($p < 0.001$) and less average code errors 0.89 vs 1.38 ($p = 0.006$). All these results were statistically significant. This may be explained by the fact that the adaptive practice levels are made to be easier to solve. To verify this explanation, we looked at the same variables but without including the practice levels. The results are reported in Table 5 where when using the whole data, players in the non-adaptive version have fewer errors on average including code errors, and these results are statistically significant. However, when we only look at the filtered data, there is no difference between the two conditions. This indicates that the practice levels are easier to solve and were the reason why players in the adaptive version had fewer errors when these levels were included.

Besides the number of errors, we also wanted to compare how players responded to error messages. Specifically, a slightly different version of %fix and an exact replica %bad presented by Marceau et al. [37]. In our case, the measure %fix is calculated by dividing the number of times an error was fixed by the number of times it has occurred.%bad is not the opposite of %fix but rather a measure of when players responded poorly to an error message and is measured by summing the number of times an unrelated or a partial fix was made and dividing by the the sum of unrelated, partial, and full fixes. In Marceau's work the focus was on assessing the error messages but since our focus is the player, instead of calculating these values for each error, we calculated them for each player. This means once we have %fix and %bad for each error message in each level, we average across errors encountered rather than across players. We will call these new estimates %fix_p and %bad_p with p referring to player. This number can be between 0 and 1 and represents the percentage of responding well or poorly to error messages. The results can be seen in Table 6 in the two first rows. No statistically significant difference can be observed between the two versions. Again, we looked at the subgroup of players that had/would have had a practice level, but in this case, no difference was found between the two versions. The results are reported in the last two rows of the table as %fix_p_f and %bad_p_f with f referring to filtered.

Given these results, *H1* holds partially since players who have encountered a practice level in the adaptive version did perform better than players who would have gotten one in the static game version in terms of levels attempted (including practice levels) and average errors, however, they did not perform better in terms of responding to errors.

6.3 Adaptive version: Before & After

In this section, we look at the players in the adaptive version and how their performance changed before and after receiving a practice level. Since we are looking at players' behavior before and after a practice level, only the players who have encountered one are included which is 26. Specifically, we look at the average errors, the fix, and the bad estimates. From Table 7, we can observe that players had fewer errors on average and improved their %fix ratio after encountering a practice level, but these results were not statistically significant. However, players' %bad ratio has also increased after a practice level which means they were answering worse on average and that result is statistically significant. This result may be due to the fact that players improve on certain errors but not on others, especially ones that they have not encountered, or ones that are generally more difficult to fix. Further, as the game evolves more constructs are encountered and more difficult errors are introduced as a result. To further investigate this, we looked at the error message players have received a practice level for. Specifically, we want to examine whether those errors appeared less after practice. We define before and after a practice level as all the levels that happened since the beginning/last practice level until the next practice level/end. We found that the median was the same, 1, before and

Table 3: Mean, median, and Mann Whitney U test for in-game variables for all players including practice levels. A represents the adaptive version and NA is the static non-adaptive version.

All players, all levels including practice

	mean A	mean NA	median A	median NA	U	p
levels attempted	9 ± 11.66	4.46 ± 2.99	4	4	970	0.319
avg errors	5.45 ± 11.45	4.18 ± 4.72	2.45	3	1097	0.969
avg code errors	3.94 ± 11.72	1.6 ± 2.2	1	0.75	1006.5	0.473
time spent	25.73 ± 10.2	27.92 ± 14.53	29.24	29.52	1081	0.87

Table 4: Mean, median, and Mann Whitney U test for in-game variables for filtered players including practice levels. A represents the adaptive version and NA is the static non-adaptive version.

Filtered players, all levels including practice

	mean A	mean NA	median A	median NA	U	p
levels attempted	16 ± 12.28	5.90 ± 3.4	15	4	104	$<$ **0.001**
avg errors	1.91 ± 0.8	5.96 ± 5.28	1.78	4	68.5	$<$ **0.001**
avg code errors	0.91 ± 0.56	2.26 ± 1.83	0.89	1.38	148	**0.006**
time spent	30.55 ± 7.83	32.49 ± 17.92	29.99	29.67	227	0.334

Table 5: Mean, median, and Mann Whitney U test for in-game variables for all players and filtered players, excluding practice levels. A represents the adaptive version and NA is the static non-adaptive version. The three top rows represent unfiltered data and the three bottom rows represent filtered data.

All players, without practice levels

	mean A	mean NA	median A	median NA	U	p
levels attempted	3 ± 2.65	4.47 ± 2.99	3	4	851.5	0.058
avg errors	7.55 ± 11.79	4.18 ± 4.72	4.5	3	780	**0.014**
avg code errors	4.81 ± 11.68	1.62 ± 2.2	1.6	0.75	832.5	**0.041**

Filtered players, without practice levels

levels attempted	4 ± 2.83	5.90 ± 3.40	4	4	221	0.273
avg errors	5.56 ± 5.73	5.96 ± 5.29	4	4	253	0.679
avg code errors	2.34 ± 2.57	2.27 ± 1.83	1.55	1.38	262.5	0.840

Table 6: Mean, median, and Mann Whitney U test for responses to error percentages %fix_p and %poor_p for all players and %fix_p_f and %poor_p_f for filtered players. A represents the adaptive version and NA is the static non-adaptive version.

All players, all levels including practice

	mean A	mean NA	median A	median NA	U	p
%fix_p	0.32 ± 0.25	0.36 ± 0.32	0.29	0.36	1057.5	0.637
%bad_p	0.65 ± 0.26	0.62 ± 0.32	0.61	0.64	1090.5	0.821

Filtered players, all levels including practice

%fix_p_f	0.45 ± 0.17	0.50 ± 0.14	0.46	0.48	232	0.368
%bad_p_f	0.51 ± 0.16	0.48 ± 0.14	0.53	0.50	247.5	0.575

after $U = 458.5$ and $p = 0.7691$. This result can also be expected since a player would make between 3 and 5 errors in the future and most of them would be puzzle-related errors so the number of times the specific error appears does not change in a significant way. We did not look at %fix and %bad ratio since many times the error does not appear at all and in that case, it is not clear what value should be attributed since it will sway the data in one direction or another. Further, since only 26 players are included, the sample size might be too small to see a significant difference.

Given these findings, *H2* cannot be supported and players do not have fewer errors after solving a practice level.

7 POST-GAME RESULTS COMPARISON

7.1 Quantitative data

Out of the original 94 participants, 61 (32 in the adaptive version vs 29 in the static) have completed the post-game survey which contained 5 closed questions and 2 to 3 open questions. The results

Table 7: Mean, median, and Mann Whitney U test for responses to error percentages %fix_p and %poor_p, before and after a practice level has been encountered. A represents the adaptive version and NA is the static non-adaptive version.

	mean before	mean after	median before	median after	U	p
avg errors	3.65 ± 4.1	2.7 ± 1.4	3	2.5	1719.5	0.198
%fix_p	0.09 ± 0.64	0.37 ± 0.27	0.25	0.33	1466.5	0.08
%bad_p	0.24 ± 0.70	0.61 ± 0.26	0.5	0.59	1345.5	**0.017**

for the closed questions can be seen in Table 8. We can observe a trend where players in the non-adaptive version seemed to like the game more and wanted to play more but with no statistical significance. Players in the non-adaptive version also thought that the game provides more help with a median of 5 vs 4 (p=0.048) and were more likely to think that the game can be an effective way to learn programming with a median of 5 vs 4 (p=0.002) and these results are statistically significant. Players in both versions thought of the game as equally difficult. When analyzing the post-game results from the filtered subset of players discussed earlier, we observe a similar trend as can be seen in Table 9. In this subset, there were only 36 players, 21 in the adaptive version and 15 in the non-adaptive one. In this table, we can see that players liked the game the same but wanted to play more in the static version with a median of 5 vs 4 (p=0.042). Players in the non-adaptive version also thought that the game was easier although not statistically significant. The same trend was observed for the questions about the efficiency of the game as a learning tool and whether it provides enough help with the latter having a larger gap with a median of 5 vs 3. Given these results, our *H3* does not hold and in fact, the opposite is true, players in the non-adaptive version seem to report stronger engagement with the game. Although it is important to note that players in both versions did like the game and mostly reported to want to play more, however, players in the non-adaptive version perceived it as more helpful and more effective.

7.2 Qualitative data

To attempt to understand the reason behind the unsupported hypothesis, we analyzed the open-ended questions. These questions ask the players what did they like/dislike most about the game. Players are also asked if they encountered glitches in the game that transported May into new levels. If they respond yes, they receive another open-ended question asking them what did they think about these levels. These questions are purposefully left vague to not influence the players' response. When analyzing the responses to the question "What did you like most about the game?", a single coder assigned a theme to each response, and responses that could not be categorized were discarded. Such responses did not include anything in particular that the player liked about the game, examples of these include "good", "extremely like this game", or "its very interesting so i like the game". Given these criteria 40 responses were retained, 20 from each version. In a first pass through the responses, 19 categories have been identified. These categories have been merged into 5 themes listed below:

- **Learning:** programming, learning, puzzle, creativity, strategy
- **Aesthetic:** music, story, graphics, character
- **Achievement:** achievement, challenge, object collection
- **Gameplay:** gameplay, progression, movement, adventure
- **Misc:** unique, smooth, minimal instruction

Creativity and strategy were included with learning because it is in the context of solving the puzzles. Table 10 shows the distribution of responses over these categories. It is important to note that one response could count in multiple categories, for example, a player mentioned "I like the gameplay, character and graphics of the game." which would count in Gameplay once and Aesthetic twice. It is interesting to see that most players like the game because of its learning aspect mentioning specifically the coding aspect, the puzzles, the challenge, and the sense of achievement. Examples of quotes from players in the adaptive version are "As someone with little coding experience, I thought it was an interesting way to begin learning a new skill" and "It seemed like an interesting idea and I liked figuring out the needed code". In the non-adaptive versions, examples are "I liked that the game made you think and wanted you to solve puzzles by thinking creatively. It was fun and interesting" and "I enjoyed the coding aspect and the artstyle was whimsical and childlike". More players in the adaptive version have mentioned the learning and achievement aspects while the ones in the static version have mentioned gameplay and aesthetics more. However, when mentioning the gameplay specifically, that could also count towards learning since the main gameplay is programming.

When analyzing answers to the question "What did you dislike most about the game?", we applied the same strategy as for the previous question to eliminate answers that did not mention a specific aspect, with the exception of answers that would fit in the category *Nothing* which included answers that ranged from "none" to "nothing" to "There's nothing to be disliked". With this criteria, 46 answers were retained with 24 and 22 in the adaptive and static versions respectively. In these answers, we identified 16 categories, however, 8 of them appeared only once and these include music, interface, movement, control, etc. The most popular response was nothing with 8 in the adaptive version and 10 in the static. However, the biggest difference between the answers was that players in the adaptive versions mentioned the lack of instructions and the lack of helpful hints 9 times compared to only 2 in the static version. This may explain why the players in the static version perceived the game as more helpful. The instructions however mean different things, some players wanted more directions for the tasks "Not enough instructions. I wasted a lot of time in one room because there were no commands or hints I could use to get to the next room" and "It is not very self explanatory. The stair problem i had no idea right away they were different heights. It was a massive amount of trial and error". Other players wanted directions on where to go next "I sometimes wasn't sure of where I needed to go to progress in the game", and other players wanted more hints if they are stuck "The game was fun, but I wish there was more hints available for

Table 8: median, and Mann Whitney U test for post-game survey questions. A represents the adaptive version and NA is the static non-adaptive version. The first question is a 7-pt likert scale and the rest are 5-pt likert.

Question	median A	median NA	U	p
Did you like the game?	6	7	360.5	0.140
Would like to play more of the game?	4	5	356.5	0.125
How difficult was the game for you?	4	4	352.5	0.110
Do you feel that the game provides enough help?	4	5	326.5	**0.048**
Do you think this game can be an effective way to learn programming?	4	5	256.5	**0.002**

Table 9: median, and Mann Whitney U test for post-game survey questions with filtered players. A represents the adaptive version and NA is the static non-adaptive version. The first question is a 7-pt likert scale and the rest are 5-pt likert.

Question	median A	median NA	U	p
Did you like the game?	6	6	115.5	0.191
Would like to play more of the game?	4	5	94	**0.042**
How difficult was the game for you?	3	4	113	0.160
Do you feel that the game provides enough help?	3	5	96	**0.049**
Do you think this game can be an effective way to learn programming?	4	5	57	$<$ **0.001**

Table 10: The number of time each category was mentioned in players' open-ended response to the question "What did you like the most about the game?"

Category	Overall	Adaptive	Non-adaptive
Learning	28	16	12
Aesthetics	12	3	9
Achievement	8	5	3
Gameplay	7	1	6
Misc	3	1	2

when you're stuck". One player also mentioned the hints that are given in between scene transitions "Some of the text that was giving you hints or instructions disappeared too quickly and it was hard to read it all". These hints include instructions such as "Use ctrl-c and ctrl-v to copy/paste your code" or "press the button help to get a hint" etc. These kinds of instructions are popular in video game loading screens. However, if the scene loads too fast, there is not enough time to read them.

Besides the lack of instructions, 2 players mentioned the adaptive levels specifically as the thing they disliked the most about the game, the answers are "Being moved to another area" and "It was not always clear what the goal was. Especially when taken from the main castle to a room with the white figure I could put in the code to move the block but it was not always clear where I was trying to move it to". The latter response showcases the main difference between a procedurally generated level and a static one. In a hand-crafted level, the lighting and setting of the level make it easier for the player to identify their goal, while that's not always the case in a procedurally generated one. Further, hand-designed levels include specific hints on how to solve that puzzles which are difficult to generate automatically.

Finally, the last open-ended question was only available if the players answered yes to this question "Did you experience instances where the game seemed to glitch and May got transported to another level in the game?". Out of 61 players, only 21 have answered yes to this question. Interestingly, only 5 of those players have actually encountered an adaptive level. The other players were split equally between the static version and the adaptive version but without experiencing it. Of the 5 who answered 2 disliked them "It was slightly annoying" and "They were annoying and felt like a punishment for not doing the code right" while the other two liked them "I only got to experience one, but it was interesting and informative" and "i think its good to play". The last person simply answered "good". It is unclear why so many players answered this question incorrectly, it could be because the wording was ambiguous, or they were not paying attention, or that these levels seemed natural to them and didn't seem like something out of the ordinary in the game. In fact, 21 players who encountered a practice level have answered "no" to the question.

8 DISCUSSION

From the combination of our results, it seems that players in the adaptive version performed better in terms of the number of error messages per level, and levels reached and attempted (including practice levels), however, they seemed to think that the game is not as helpful and lacks instruction. An interesting finding was that if we account for practice levels, players in the adaptive version perform significantly better than players in the non-adaptive, and this is likely due to the practice levels being easier to solve, which is in contrast with the fact that players found the adaptive version less helpful. There were no other notable differences between the two game versions that were statistically significant.

One of the main reasons for the unsupported hypotheses is that players were playing for only 30 minutes and it therefore may be difficult to observe many differences. Further, since our participants are Amazon Mechanical Turkers, many of them are multi-tasking, which would make the game even more difficult for them given that they are not completely focused. Further, many were not fully engaged with the game or the surveys. Some of the answers to the survey questions are contradictory such as a player answering that they definitely disliked the game but would definitely want to play more. Also, many have answered the open-ended questions with one word only "good" for all the questions. There is also doubt about their reported level of expertise in programming which is not supported by their in-game behavior and their skills as logged in the data. In fact, these types of participants are more accustomed to short tasks that can be performed in a fast, repetitive way.

Additionally, while our focus is on debugging, it is not necessarily what the player was struggling with. In that case, the targeted practice levels for debugging might be more annoying than helpful. However, some players did like those levels which may mean that depending on the type of player or depending on the context, these levels may or may not be enjoyable. We did not have enough data to draw any conclusions about what such a context or player could be like. This is especially important because the adaptive levels seem to increase performance but if some players perceive them as annoying or frustrating, they might not continue playing, especially in a voluntary context. In terms of timing or framing of these adaptive levels, it is unclear what would be the best approach to strike the balance between helpful and enjoyable. It seems that the answer to that will also depend on the individual player, their performance, and their preferences. Finally, debugging practice levels alone may not be sufficient and different types of adaptivity in terms of hints, scaffolding techniques, instructions, and practice levels with specific learning constructs are needed.

9 CONCLUSION

There are many variables and small details that play a role in the adaptive version of the game beyond practicing debugging. In fact, the timing and the way the levels are introduced can affect how players perceive it. Some of our hypotheses were not supported by our experiments, namely that players would enjoy the adaptive version more, and that they will perform better after receiving a practice level. It is unclear if this result is the product of how and when the levels were introduced, the design of the levels themselves, or player preferences. More work needs to be done in relation to adaptive procedural levels in learning environments to shed light on these issues. In this paper, we shed some light on the difficulties of integrating adaptive levels in an existing game. In the future, we plan to vary how and when levels are presented and gather more data to be able to match a player model to a specific approach.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under grant no. 1810972.

REFERENCES

[1] Abejide Ade-Ibijola. 2018. Syntactic generation of practice novice programs in Python. In *Annual Conference of the Southern African Computer Lecturers' Association*. Springer, 158–172.

[2] Michael B Armstrong and Richard N Landers. 2017. An Evaluation of Gamified Training: Using Narrative to Improve Reactions and Learning. *Simulation & Gaming* (2017), 1046878117703749.

[3] Richard Blunt. 2007. Does game-based learning work? Results from three recent studies. In *Proceedings of the Interservice/Industry Training, Simulation, & Education Conference*. 945–955.

[4] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 223–228.

[5] Peter Brusilovsky. 2001. Adaptive hypermedia. *User modeling and user-adapted interaction* 11, 1 (2001), 87–110.

[6] Elizabeth Emily Carter. 2014. An intelligent debugging tutor for novice computer science students. (2014).

[7] Brianno D Coller and Michael J Scott. 2009. Effectiveness of using a video game to teach a course in mechanical engineering. *Computers & Education* 53, 3 (2009), 900–912.

[8] C Darryl. 2003. Enhancing Gameplay: Challenges for Artificial Intelligence in Digital Games. *Journal DiGRA* 2 (2003).

[9] Thomas Dy and Ma Mercedes Rodrigo. 2010. A detector for non-literal Java errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. 118–122.

[10] Stephen H Edwards, Krishnan P Murali, and Ayaan M Kazerouni. 2019. The Relationship Between Voluntary Practice of Short Programming Exercises and Exam Performance. In *Proceedings of the ACM Conference on Global Computing Education*. 113–119.

[11] K Anders Ericsson, Ralf T Krampe, and Clemens Tesch-Römer. 1993. The role of deliberate practice in the acquisition of expert performance. *Psychological review* 100, 3 (1993), 363.

[12] Paul M Fitts and Michael I Posner. 1967. Human performance. (1967).

[13] Eleanor Jack Gibson. 1969. Principles of perceptual learning and development. (1969).

[14] Lindsey Ann Gouws, Karen Bradshaw, and Peter Wentworth. 2013. Computational thinking in educational activities: an evaluation of the educational game light-bot. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 10–15.

[15] Casper Harteveld. 2011. *Triadic game design: Balancing reality, meaning and play.* Springer Science & Business Media.

[16] Leslie J Hinyard and Matthew W Kreuter. 2007. Using narrative communication as a tool for health behavior change: a conceptual, theoretical, and empirical overview. *Health Education & Behavior* 34, 5 (2007), 777–792.

[17] Daniel Malcolm Hoffman, David Ly-Gagnon, Paul Strooper, and Hong-Yi Wang. 2011. Grammar-based test generation with YouGen. *Software: Practice and Experience* 41, 4 (2011), 427–447.

[18] Danial Hooshyar, Liina Malva, Yeongwook Yang, Margus Pedaste, Minhong Wang, and Heuiseok Lim. 2021. An adaptive educational computer game: Effects on students' knowledge and learning attitude in computational thinking. *Computers in Human Behavior* 114 (2021), 106575.

[19] Britton Horn, Christopher Clark, Oskar Strom, Hilery Chao, Amy J Stahl, Casper Harteveld, and Gillian Smith. 2016. Design insights into the creation and evaluation of a computer science educational game. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 576–581.

[20] Matthew C Jadud. 2006. *An exploration of novice compilation behaviour in BlueJ.* Ph. D. Dissertation. University of Kent.

[21] Chaima Jemmali, Sara Bunian, Andrea Mambretti, and Magy Seif El-Nasr. 2018. Educational game design: an empirical study of the effects of narrative. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*. ACM, 34.

[22] Chaima Jemmali, Carter Ithier, Seth Cooper, and Magy Seif El-Nasr. 2020. Grammar Based Modular Level Generator for a Programming Puzzle Game.. In *AIIDE Workshops*.

[23] Chaima Jemmali and Zijian Yang. 2016. *May's Journey: A serious game to teach middle and high school girls programming.* Master's thesis. Worcester Polytechnic Institute.

[24] Cagin Kazimoglu. 2013. *Empirical evidence that proves a serious game is an educationally effective tool for learning computer programming constructs at the computational thinking level.* Ph. D. Dissertation. University of Greenwich.

[25] 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.

[26] Andrew J Ko, Brad A Myers, and Htet Htet Aung. 2004. Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on.* IEEE, 199–206.

[27] Amruth Kumar. 2005. Rule-based adaptive problem generation in programming tutors and its evaluation. In *Proc. of Int. Conf. on Artificial Intelligence in Education.* Citeseer, 36–44.

[28] Amruth N Kumar. 2002. Model-based reasoning for domain modeling in a web-based intelligent tutoring system to help students learn to debug c++ programs. In *International Conference on Intelligent Tutoring Systems.* Springer, 792–801.

[29] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. 2005. A study of the difficulties of novice programmers. In *Acm Sigcse Bulletin*, Vol. 37. ACM, 14–18.

[30] Richard N Landers and Tara S Behrend. 2015. An inconvenient truth: Arbitrary distinctions between organizational, Mechanical Turk, and other convenience samples. *Industrial and Organizational Psychology* 8, 2 (2015), 142–164.

[31] Richard N Landers and Rachel C Callan. 2014. Validation of the beneficial and harmful work-related social media behavioral taxonomies: development of the work-related social media questionnaire. *Social Science Computer Review* 32, 5 (2014), 628–646.
[32] Greg C Lee and Jackie C Wu. 1999. Debug It: A debugging practicing system. *Computers & Education* 32, 2 (1999), 165–179.
[33] Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, and Linda Werner. 2011. Computational thinking for youth in practice. *Acm Inroads* 2, 1 (2011), 32–37.
[34] Michael Jong Lee. 2015. *Teaching and engaging with debugging puzzles.* Ph. D. Dissertation.
[35] Chen Li, Emily Chan, Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2019. Towards a framework for teaching debugging. In *Proceedings of the Twenty-First Australasian Computing Education Conference*. 79–86.
[36] Yun-En Liu, Christy Ballweber, Eleanor O'rourke, Eric Butler, Phonraphee Thummaphan, and Zoran Popović. 2015. Large-scale educational campaigns. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–24.
[37] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. 499–504.
[38] Michael A Miljanovic. 2015. *RoboBUG: a game-based approach to learning debugging techniques.* Ph. D. Dissertation.
[39] Michael A Miljanovic and Jeremy S Bradbury. 2018. A review of serious games for programming. In *Joint International Conference on Serious Games*. Springer, 204–216.
[40] Roy D Pea and D Midian Kurland. 1984. On the cognitive effects of learning computer programming. *New ideas in psychology* 2, 2 (1984), 137–168.
[41] Ricardo Queirós and José Paulo Leal. 2011. Pexil: Programming exercises interoperability language. In *Conferência Nacional XATA: XML, aplicações e tecnologias associadas, 9. ª*. ESEIG, 37–48.
[42] Danijel Radošević, Tihomir Orehovački, and Zlatko Stapić. 2010. Automatic on-line generation of student's exercises in teaching programming. In *Radošević, D., Orehovački, T., Stapić, Z:" Automatic On-line Generation of Students Exercises in Teaching Programming", Central European Conference on Information and Intelligent Systems, CECIIS*.
[43] Valerie J Shute. 2008. Focus on formative feedback. *Review of educational research* 78, 1 (2008), 153–189.
[44] Adam M Smith, Erik Andersen, Michael Mateas, and Zoran Popović. 2012. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*. 156–163.
[45] Geng Sun, Tingru Cui, Ghassan Beydoun, Jun Shen, and Shiping Chen. 2016. Profiling and supporting adaptive micro learning on open education resources. In *2016 International Conference on Advanced Cloud and Big Data (CBD)*. IEEE, 158–163.
[46] Emily S Tabanao, Ma Mercedes T Rodrigo, and Matthew C Jadud. 2011. Predicting at-risk novice Java programmers through the analysis of online protocols. In *Proceedings of the seventh international workshop on Computing education research*. 85–92.
[47] Kelli Taylor and Doug Rohrer. 2010. The effects of interleaved practice. *Applied Cognitive Psychology* 24, 6 (2010), 837–848.
[48] Richard Van Eck. 2007. Building artificially intelligent learning games. In *Games and simulations in online learning: Research and development frameworks*. IGI global, 271–307.
[49] Akiyoshi Wakatani and Toshiyuki Maeda. 2016. Evaluation of software education using auto-generated exercises. In *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. IEEE, 732–735.
[50] Wengran Wang, Rui Zhi, Alexandra Milliken, Nicholas Lytle, and Thomas W Price. 2020. Crescendo: Engaging Students to Self-Paced Programming Practices. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 859–865.
[51] Alan Traviss Welford. 1968. Fundamentals of skill. (1968).
[52] Etienne Wenger. 2014. *Artificial intelligence and tutoring systems: computational and cognitive approaches to the communication of knowledge.* Morgan Kaufmann.
[53] Wee Ling Wong, Cuihua Shen, Luciano Nocera, Eduardo Carriazo, Fei Tang, Shiyamvar Bugga, Harishkumar Narayanan, Hua Wang, and Ute Ritterfeld. 2007. Serious video game effectiveness. In *Proceedings of the international conference on Advances in computer entertainment technology*. ACM, 49–55.
[54] Alexander Zook, Stephen Lee-Urban, Mark O Riedl, Heather K Holden, Robert A Sottilare, and Keith W Brawner. 2012. Automated scenario generation: toward tailored and optimized military training in virtual environments. In *Proceedings of the international conference on the foundations of digital games*. 164–171.