Towards Automated Playtesting in Game Development

Nathalya Stefhany Pereira
Coders, Developers and Gamers Hub - CDG Hub
National Institute of Telecommunications - INATEL
Santa Rita do Sapucaí,MG, Brazil
Email: nathalya.stefhany@gec.inatel.br

Phyllipe Lima
Coders, Developers and Gamers Hub - CDG Hub
National Institute of Telecommunications - INATEL
Santa Rita do Sapucaí,MG, Brazil
Email: phyllipe@inatel.br

Eduardo Guerra
Computer Science Faculty
Free University of Bozen-Bolzano - UniBZ
Bolzano, Italy
Email: guerraem@gmail.com

Paulo Meirelles
Computer Science Institute
Federal University of ABC - UFABC
Santo André,SP, Brazil
Email: paulo.meirelles@ufabc.edu.br

***Abstract*—Digital games are also a software product. However, games have a fun requirement and tightly coupled UI, which makes them hard to test. Nevertheless, they are made of code and might also benefit from the advantages that automated tests bring to enterprise software systems. In this paper, we discuss two categories of automated tests for games, focusing on playtesting. We used the Unity Game Engine to build our tests on top of the NUnit framework and the Unity Test Framework. To demonstrate our approach for automated playtesting, we developed a 2D Arkanoid-style game. We also present the steps we took to make the testing feasible.**

***Index Terms*—automated playtesting, games, software engineering, unit testing,**

I. Introduction

Electronic games are also software products. When comparing to board games or card games, they all have mechanics, arts, and stories. However, digital games execute on an electronic device like any other software [1]. For this reason, digital games should also be the subject of software engineering studies and practices, such as unit testing and quality assurance.

However, this has not been the case since digital games are not often used in software engineering research. Likewise, game programmers in the industry do not often apply unit and automated tests during the creation of their code [2]–[6]. They argue that games have a "fun" requirement that is sometimes illogical and dynamic since game designers often alter the mechanics during development. In this scenario, creating unit tests, for instance, poses a challenge since the test code will also change constantly. Another problem is that the UI (User Interface) in games is tightly coupled to the game code, creating another challenge. Furthermore, testing a game usually requires playing the game in a sequential manner which may compromise automated testing [2], [6].

One frequently studied topic by software engineering researchers and practitioners is automated testing. It is also recognized as an essential part of ensuring software quality [7]. Furthermore, an automated suite of tests protects developers from code regression when new features are introduced [8], [9]. However, we need to study approaches to enable game programmers to also benefit from this and implement automated tests in their systems.

According to [6], game playing is a sequential decision-making process where a player needs to make decisions and take actions based on received observations. Extending this idea, *playtesting* is the act of game playing to discover bugs and validate if the requirements are met before releasing it. According to [5], nearly 50% of all testing for a game is concerned with playtesting, usually taking place manually. While only 5.47% of tests performed were automated, and 1.37% were unit/integration testing. Another data about the lack of studies regarding tests in game development is that out of 99 papers presented in the Computing Track at SBGames, between 2016 and 2020, 15 were labeled as "software engineering". Furthermore, only Lovreto *et al.* [4] discussed automated testing in games.

However, several initiatives might bring new insights so that developers can begin acquiring the culture of automated playtesting and benefiting from it. For instance, the Unity Game Engine[1], one of the most popular frameworks for game development, ships with a fully integrated framework for unit testing, the Unity Test Framework.

Another recent initiative is the GameCI [2], which is an open-source suite of tools that enable easy setup for continuous integration for games developed using Unity Game engine. Finally, it is also known that testable software units, such as classes, are easier to evolve and that testable classes have a better overall design.

Using the Unity Test Framework, this work discusses two categories of automated tests in game development. The first is traditional unit testing called "non-gameplay tests", and the second is "automated playtesting". For this last one, we also identify and described the required steps to implement it. We developed a 2D Arkanoid-style game with the Unity Engine

[1]https://unity.com
[2]https://game.ci/docs

and implemented test scenarios to demonstrate these two categories. The majority of the tests were written considering the "automated playtesting". We also configured a CI (Continuous Integration) server to run these tests combining GitHub Actions and the GameCI tool.

II. Background

Unit testing is the process of automatically testing small units of production code, such as classes and methods. They aid in detecting bugs in the early stages of development and verify if the software complies with the requirements. Usually, these tests are written by the developers themselves since they are strictly tied to the code [10]. Writing unit tests during the development process naturally creates a test suite that serves as regression tests. This suite can be executed after every newly added feature to guarantee that the code still works as expected and no bug was introduced. Furthermore, unit tests are update documentation of the code [7], [8].

The Unity Game Engine comes fully integrated with tools that will allow us to write unit tests that execute automatically. Our testing strategies depend on such tools. First, we have the NUnit, which is an open-source unit testing framework for C#. NUnit is a metadata-based framework and offers a set of attributes[3] to allow developers to configure their testing method, fixture, and tear down process. To run the tests, the Unity Game Engine comes with the Unity Test Framework, which invokes the class methods written with the NUnit and offers a friendly GUI to verify and run the tests.

The Unity Test Framework offers two paths for automated testing, the Edit Mode and the Play Mode. The former allows us to run tests that do not require the game to be running, and the latter can run tests that require playing the game. No external library is required since the Unity Package Manager handles all necessary dependencies to have the NUnit and Unit Test Framework ready to be used.

III. Related Work

This section presents results and work performed by other researchers and practitioners regarding software testing in game development.

Lovreto *et al.* [4] investigates how to automate gameplay testing for mobile games. The authors investigate the feasibility and efficiency of using scripts to test mobile games automatically. They selected 16 popular games with varying genres from the GooglePlay store. A Python script was created for each game using Appium tests, the OpenCV library, and the UIAutomatorViewer. Each study was conducted in three phases. The first step was an exploratory analysis to understand the game mechanics better. Then they planned two test cases, one focused on actions that simulate a player interacting with the game and another focused on simulating interaction with the game's menu. Finally, in the third phase, they wrote the steps required to code the automated test. They concluded that the performance of Appium and OpenCV made the test scripts feasible. The majority of tests cases ran in under 60 seconds. One difference of [4] to ours is that we are investigating how to automate gameplay testing using a framework integrated into the development environment instead of an external script. This way, we have access to the game's production code.

Llopis and Houghton [11] researched how TDD (Test Driven Development) can be applied to game development. They encountered several challenges, one of them being that games are usually developed for several platforms. To overcome this, they had to write small programs using the target system API to run the executable from the command line and catch the return values of methods. The other challenge was the randomness present in games, such as a ball assuming random speed values. These tests needed to be conducted with deterministic values. Furthermore, the UI in game is usually something developers do not know how to handle gracefully. However, isolating the code with different responsibilities in separate modules or libraries can aid in writing the tests. The authors concluded they reach a flexible, robust, and loosely coupled code even with these challenges. Our work intends to minimize the cross-platform issue by using a game engine that already deploys to multiple platforms and is packed with a unit testing framework, such as Unity.

Zheng *et al.* [6] proposes an automated testing framework for online combat games that they named *Wuji*. The tool combines EMOO (Evolutionary multi-objective optimization) with DRL (Deep Reinforcement Learning) to learn how to play the game, test, and explore space. To evaluate *Wuji*, they studied 1349 real bugs from industry games, used two real-world games to test their tool, and were able to find 3 unknown bugs.

Song *et al.* [12] proposes an automated game testing technique. The author combines AIRL (Adversarial inverse reinforcement learning) with EMOO to design a framework for testing mini-games in WeChat platform. The author did not conduct any evaluation, and their tool is focused only on games available in WeChat.

In these recent studies [6], [12], they propose an approach using AI techniques where we might not have access to the game code, which is a different strategy that we are currently investigating in this paper. In our complete work, we aim to discuss these differences and how they can complement each other.

IV. Arkanoid 2D Game

This section presents the game that we developed to serve as a target for our automated testing approach. It is a 2D Arkanoid-style game, publicly available on GitHub[4].

Taito Corporation originally developed Arkanoid for the arcade in 1986. It became popular and still inspires new games. The goal is to destroy blocks on the scene using a ball while keeping it bouncing off a platform that moves horizontally. The player controls the platform, and if the ball falls off the screen, the game is over.

Our version of Arkanoid keeps the central mechanic as the original one, i.e., destroying blocks and not letting the ball fall. We implemented several types of blocks, and they are differentiated by their color. There are four common blocks that award points when destroyed and one indestructible block

[3]Feature of languages such as Java and C# to allow customization of metadata

[4]https://github.com/NathalyaStefhany/IC-2020

that serves as an obstacle. Finally, there are special power-up blocks that give the player a special ability when destroyed. Fig. 1 presents our implementation of Arkanoid.

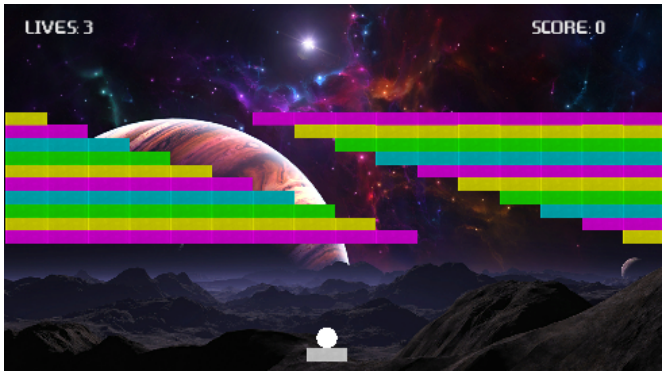

Fig. 1. Example Scene of Arkanoid Game

The common blocks take different amounts of hits to be destroyed and award a proportional number of points. To serve as an obstacle, we implemented indestructible blocks. They are gray-colored, and as the name suggests, cannot be destroyed. Their only purpose is to stand in the way of the player. Finally, we implemented a power-up system in the game. They are obtained by destroying special blocks and award a temporary advantage to the player. To implement these power-ups, we used the Strategy [13] design pattern. These different types of blocks and power-ups will each serve as individual test scenarios to demonstrate our testing strategy.

V. Automated Playtesting

In this section, we describe the two categories of automated tests that we identified in game development. The main goal is the automated playtest, but we feel it would be incomplete if the other tests were ignored. We are proposing two classes of tests. The first is basic unit testing, and we named it *Non-gameplay Tests*. It is the traditional unit testing found in enterprise software, where we verify the return value of a method or if a method was indeed called.

The second type of test is concerned with automatically playing the game and running the tests. The *Automated Playtests* are the most challenging ones since they will tests if the mechanics are correctly implemented while simulating players' input. We also create separate scenes for testing purposes. Just like the test code is separate from the productions code, our scenes follow the same pattern. We can also execute tests concerned with the initial setup of the scene, i.e., we want to verify if the correct objects are present when a given scene is loaded. One of our goals is to describe the steps we are taking to implement these types of tests.

These two classes fit very well with the Unity Test Framework's Edit Mode and Play Mode tests.

A. Non-gameplay Test

The *Non-gameplay Tests* goals are to verify the return of methods values and if methods are being called without involving any gameplay and objects in the scene. They were implemented with a black-box testing approach, similar to how non-game software systems execute unit testing. To better demonstrate this category, we have the `TestSortScore()` example. In Arkanoid, we created a score table to inform players of the highest scores in descending order. To sort the score, we created a `SortScore()` method that receives as a parameter a list of objects (composed of the player's name, level, and score) and returns the same list but with the objects sorted by score.

Fig. 2 shows the implementation of the test. In the beginning, we have the `Init()` method that will be invoked to do the necessary initialization. From lines 27 to 28, the objects are added to the list and then passed to the `SortScore()` method. Once this is done, the method's return is stored in the `highScores` variable and asserted through the Assert.AreSame() method in lines 33-35.

```
1  public class TestAddScore{
2      private AddScore addScore;
3      private HighScoreEntry score1,
4              score2;
5      private HighScores highScores;
6
7      [SetUp]
8      public void Init(){
9          // Class containing the sorting method
10         addScore = new AddScore();
11
12         score1 = new HighScoreEntry {
13             name = "NA", round = 1, score = 1500
14         };
15         score2 = new HighScoreEntry {
16             name = "JO", round = 2, score = 3000
17         };
18         highScores = new HighScores();
19
20         // List that stores the objects
21         highScores.highScoreEntryList = new
22             List<HighScoreEntry>();
23     }
24
25     [Test]
26     public void TestSortScore(){
27         highScores.highScoreEntryList.Add(score1);
28         highScores.highScoreEntryList.Add(score2);
29
30         highScores = addScore.
31             SortScore(highScores);
32
33         Assert.AreSame(score2,
34             highScores.highScoreEntryList[0]);
35         Assert.AreSame(score1,
36             highScores.highScoreEntryList[1]);
37     }
38 }
```

Fig. 2. Unit Test for AddScore Class

B. Automated Playtest

As mentioned, playtesting is the act of game playing to discover bugs and observe the results of actions taken by the player. In our Arkanoid example, if the player hits a destructible block, we need to check if the block is destroyed and the block's point is added to the score. We want to do this automatically and from a unit testing perspective to access the game's production code. Combining these elements enables to test the game's functionalities without having a person manually play the game and assert the results.

We can start creating tests that verify if certain elements are in the game scene. We also find this kind of testing in web applications that check if a component is rendered correctly on the screen.

For the Arkanoid game, we can verify that a predefined amount of destructible blocks are shown on scenes. One might overlook these tests arguing they are too simplistic. However, they help us guarantee that these scenes are ready for more complex automated playtesting. Also, some games use initial elements on the scene to control the difficulty. Hence, testing if a scene is ready can also control these and other game design decisions and rules.

Fig. 3 shows the test that checks the number of destructible blocks. First, we load the scene (line 4). Then, the blocks that contain the `Destructible` tag are searched and stored in an array. Finally, we assert if the size of the array is equal to the expected size, i.e., 57 blocks (line 10) for Level 1 of the Arkanoid game.

```
1  public class TestBlocks{
2      [Test]
3      public void TestNumberBlocksSceneLevel1(){
4          EditorSceneManager.OpenScene(
5              "Assets/Scenes/Level1.unity");
6
7          GameObject[] blocks = GameObject.
8              FindGameObjectsWithTag("Destructible");
9
10         Assert.AreEqual(57, blocks.Length);
11     }
12 }
```

Fig. 3. Unit Test for Blocks in Level 1

Afterward, we can perform more complex automated playtesting. The gameplay needs to be simulated to perform these tests so that everything happens in an automated fashion. Lets first list the steps required to play Arkanoid manually:

1) Press the left mouse button to start the game, which releases the ball from the platform;
2) Once the game starts, the player must move the platform horizontally to prevent the ball from falling more than three times.
3) To progress, the player must use the platform to bounce the ball back and destroy every destructible block on the scene.

As for step 1 (release the ball), we can generalize that we need to handle the player's input. To automatically playtest, we need to encapsulate this code in different methods. This way, we can invoke the method by both the test code and the player's input when regularly playing the game. Consider the hypothetical code snippet on Fig. 4. The code to "release the ball" was encapsulated in the `ThrowBall()` method. This reinforces that making software testable contributes to a cleaner code.

As for step 2 (move the platform), we need to automatically make the platform move with the ball's x coordinate in tests requiring specific blocks to be destroyed. In other words, we need to keep the game running in a *auto mode*. However, for tests that require letting the ball fall, we switch back to manual mode. To implement this, we created a boolean

```
1      if(Input.MouseOnClick()){//Mouse button pressed
2        //The code to throw the ball should
3        //encapsulate it in a separate method.
4
5        ThrowBall();
6        //The ThrowBall() method can
7        //also be called from test code
8      }
```

Fig. 4. Example of the Method Releases the Ball

variable that lets the test code determine if the game should run automatically or manually. Also, the code that moves the platform is encapsulated to be easily called from other parts of the code.

No player input is required for step 3 (bounce back) since the Unity Game Engine physics handles this process. From a test perspective, no extra effort was necessary. The *auto mode* and initial position of the platform were enough.

After implementing the above steps, we began creating specific test scenes for our test scenarios. As known, unit tests should be executed fast, so we created test scenes with only the necessary number of blocks. As a demonstration, consider the scenario containing four destructible blocks that should reward a total of 100 points after being destroyed.

After we create the scene, we begin writing the test code. Fig. 5 presents the initial setup ([UnitySetUp] attribute on line 1) for this test. It loads the scene (line 3), store a reference to every block (line 7) and set the platform to *auto mode* (line 12)

```
1  [UnitySetUp]
2  public IEnumerator SetUp(){
3      SceneManager.LoadScene("LevelTestScore");
4
5      yield return new WaitForSeconds(1);
6
7      blocks = GameObject.
8          FindObjectsOfType<Block>();
9
10     platform = GameObject.
11         FindObjectOfType<Platform>();
12     platform.AutoPlay = true;
13
14     ball = GameObject.FindObjectOfType<Ball>();
15 }
```

Fig. 5. Initial Setup for the Test

After the initial setup, we can execute the test method presented in Fig. 6. This test aims to assert the correct number of points rewarded after the destruction of the four blocks. To make the test execute faster, we loop through the array with the references to blocks (line 5) on the scene (populated during the initial setup), then we check the number of hits required to destroy this block (line 7). Then we create another loop that forces the ball to be thrown in the direction of the block (lines 14-19) the number of times required to destroy it. We repeat this process for every block. Finally, after being destroyed, we assert if the number of points obtained is correct.

Regarding the for-loop in test code, we are aware that we should not write these kinds of loops in test code to make them more readable. However, we should keep in mind that we have different requirements for game development than enterprise

```
1  [UnityTest]
2  public IEnumerator TestScoreWhenDestroyAllBlocks(){
3     int num;
4
5     foreach (Block block in blocks){
6        if (block.tag == "Destructible"){
7           if (block.points == 10)
8              num = 1;
9           else if (block.points == 30)
10             num = 2;
11          else
12             num = 3;
13
14          for(int i = 0; i < num; i++){
15             ball.transform.position = new Vector2(
16                block.transform.position.x - 0.5f,
17                platform.transform.position.y);
18
19             ball.ThrowBall();
20
21             yield return new WaitForSeconds(0.5f);
22          }
23       }
24    }
25
26    Score score = GameObject.
27       FindObjectOfType<Score>();
28
29    Assert.AreEqual(100, score.getPlayerPoints());
30 }
```

Fig. 6. Test Method to Destroy 4 Blocks

software and tighter coupling with UI and graphical elements, which makes writing these tests codes challenging. Since this is still a work in progress, there is still much groundwork and room for improvements in the test code.

VI. Conclusion

In this paper, we discussed two types of automated testing that can be executed during game development. The first type is the "non-gameplay tests". The second is "automated playtesting". In this last one, we want the game to play automatically while tests are running, asserting the expected behavior. To support the execution of these tests, we needed to write not only the test code itself but also mechanisms that made it possible for the production code to be called from the test code and executed in *auto mode*. We also describe the actions required to implement these mentioned mechanisms.

We used an automated playtesting approach that is built with the game code itself. No external script was used to test the game. With this approach, we can test the game as it is being developed and create a suite of unit tests to protect the code from regression. We could also successfully configure a CI server to run these tests using GitHub Actions and the GameCI tools.

A. Threats to Validity

Our approach is built on top of tools that enable the implementations of our testing code, i.e., a unit testing framework. Unity offers excellent integration with the NUnit framework and allows the execution through the Unity Test Framework. Therefore, our solution is coupled to these tools and might not be straightforward to apply to other game engines or game development environments. Also, in this paper, we developed a simple 2D game to demonstrate. More complex games should also be considered to improve our testing methods.

B. Future Work

There is much groundwork regarding automated testing in game development. The next step would be to implement these approaches, precisely the gameplay, in more complex games. One good starting point would be a 2D side-scrolling game. To automate the test as we did for the Arkanoid game, we need to implement a solution for the player to dodge obstacles, recognize enemies, and so forth. In short, we need to combine testing techniques with artificial intelligence techniques that learn how to play and test a game. Also, we intend to incorporate the UI aspect alongside our testing code, given that this is one of the challenges that game programmers face.

Acknowledgment

The authors would like to thank FINATEL for financial support for this work.

References

[1] J. Schell, *The Art of Game Design: A Book of Lenses*, 2nd ed. A. K. Peters, Ltd., 2014.

[2] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" in *Proceedings of the 36th International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 1–11.

[3] S. Aleem, L. F. Capretz, and F. Ahmed, "Critical success factors to improve the game development process from a developer's perspective," *Journal of Computer Science and Technology*, 2018.

[4] G. Lovreto, A. T. Endo, P. Nardi, and V. H. S. Durelli, "Automated tests for mobile games: An experience report," in *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, 2018.

[5] J. N. de Oliveira Neto, D. Viana, E. Sá, L. Rivero, R. F. Lopes, and F. Silva, "Is there time for software testing in the indie games development? a survey with practitioners of the game industry," in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, 2019.

[6] Y. Zheng *et al.*, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 772–784.

[7] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review: Why and how developers review tests," in *Proceedings of the 40th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018.

[8] M. Mayeda and A. Andrews, "Evaluating software testing techniques: A systematic mapping study," in *Advances in Computers*, A. R. Hurson, Ed. Amsterdam, The Netherlands: Elsevier Science, 2021, vol. 123, ch. 2, pp. 41–141.

[9] Z. Peng, X. Lin, M. Simon, and N. Niu, "Unit and regression tests of scientific software: A study on swmm," *Journal of Computational Science*, vol. 53, pp. 1–13, 2021.

[10] M. F. Aniche and M. A. Gerosa, "How the practice of tdd influences class design in object-oriented systems: Patterns of unit tests feedback," in *2012 26th Brazilian Symposium on Software Engineering*, 2012, pp. 1–10.

[11] N. Llopis and S. Houghton. (2006) Backwards is forward: Making better games with test-driven development. [Online]. Available: http://www.convexhull.com/articles/tdd_gdc06.pdf

[12] Z. Song, "An automated framework for gaming platform to test multiple games," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2020, pp. 134–136.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.