Patrícia Monteiro Negrão

Bachelor in Computer Science Engineering

Automated Playtesting In Videogames

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Fernando Birra, Professor,
FCT NOVA University of Lisbon

Co-adviser: Guilherme Santos, CEO,
ZPX - Interactive Software

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

July, 2020

Automated Playtesting In Videogames

This document was created using the (pdf)LaTeX processor, based on the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Acknowledgements

This dissertation would not have been possible without the help and support of the ZPX company. I want to thank specially to Guilherme Santos, Tiago Fernandes and Francisco Raposo who taught me a lot about a completely new area that I had no prior knowledge and had a lot of patience with my doubts and curiosities.

I want to professor my teacher Fernando Birra who made this dissertation possible and guided me.

I want to thank João Pires for the love and support across this year of making this dissertation and in the years of me going through university and motivating me.

I want to thank my parents and my friends whom I consider family for believing in me and being by my side all of these years in university.

Finally, I want to thank my twin sister, Raquel Negrão, who read my dissertation over and over again to make sure everything was correct. The one whom I kept bothering to make sure the english was correct and never complained.

Thank you to all of these people who supported me and made it all possible.

Abstract

Game industry has recently emerged as a major software development industry. The number of games being developed has increased exponentially over the last few years [20]. And as these numbers grow there is also a need to test these games. Related to the game development workflow in diverse areas (AI, networking, graphics, engines, etc.) a unit testing and play testing component should be associated, which nowadays is rare or even nonexistent. Right now a common approach to testing games is hiring *Testers* to manually play a large number of potential scenarios that end users may exercise. Current game testing practices are labor intensive and become tedious and monotonous with the passage of time[3]. Furthermore, it gets quite expensive to pay someone to test these sort of games over and over again. Automated testing simplifies and makes these repetitive tasks efficient and automatic.

This dissertation consists in the implementation and definition of a framework for a system that simplifies the creation of unit tests and automatic playtesting. The playtests test the integrity of the level and are able to determine whether it is possible to exploit the game in some way. This dissertation has a partnership with the ZPX company that provided a level to use and experiment the framework and manual playtesting in order to get results. The results obtained from the use of this framework are compared with the results of the manual testing performed by testers. These results consist of the time each testing approach takes, how many bugs were found in total and the quality of the report for the game level designer. These results uncover the advantages and disadvantages of the framework and the manual playtesting. One advantage to creating this framework in a highly general and modular way is that this framework can be applied to different games and the tedious work from the developer can be taken away since there is not a need to implement the more general playtests for each new game in development.

Keywords: Games, Unit Testing, Automated Testing, Bots, Unreal Engine, Frameworks

Resumo

A indústria de jogos tem emergido recentemente como uma grande área da indústria de desenvolvimento de software. O número de jogos que estão a ser desenvolvidos aumentaram exponencialmente ao longo dos últimos anos[20]. Com estes números a crescer, existe uma crescente necessidade de testar esses mesmos jogos. Associado ao workflow das áreas de desenvolvimento de jogos (AI, networking, gráficos, engines etc) deveria estar relacionado uma componente de unit testing e play testing que neste momento é rara ou inexistente. Uma abordagem comum de testar os jogos é contratar *Testers* de modo a que estes possam jogar manualmente e repetidamente vários cenários possíveis, e que futuros utilizadores possivelmente irão jogar. Estas técnicas resultam em bastante trabalho, são tediosas e monótonas com o passar do tempo[3]. Para além disso, acaba por ficar bastante caro pagar a alguém que teste os jogos uma e outra vez. Testes automatizados simplificam e fazem com que estas tarefas repetitivas e monótonas sejam feitas de modo automático.

Esta dissertação consiste em implementar e definir uma ferramenta que simplifique a criação de testes unitários e de play testing automático. Os playtests testam a integridade de um nível e são capazes de determinar se é possivel o jogador tirar partido dos bugs. Esta dissertação é realizada em parceria com a empresa ZPX que forneceu um nível para se experimentar a framework e testar manualmente de modo a obter-se resultados. Os resultados obtidos a partir do uso da framework são comparados com o playtest manual e dessa comparação podemos inferir as vantages e desvantagens de cada uma das abordagens, tanto a automática como a manual. Uma vantagem de criar a framework mais geral e modular é que esta framework pode ser aplicada a vários jogos diferentes e retirar o trabalho tedioso dos developers uma vez que não têm de implementar os playtests mais gerais para cada novo jogo em desenvolvimento.

Palavras-chave: Testes automatizados, Jogos, Testes unitários,Bots, Unreal Engine

Contents

Glossary

BluePrints	Blueprints Visual Scripting. In UE4, it will be often found that objects defined using Blueprint are colloquially referred to as just "Blueprints."
CommandLet	Commandlet (command-line applet) class. Commandlets are executed in a "raw" environment, in which the game isn't loaded, the client code isn't loaded, no levels are loaded, and no actors exist.
Gameplay	The features of a video game, such as its plot and the way it is played, as distinct from the graphics and sound effects.
Minecraft	Minecraft is a sandbox video game created by Swedish game developer Markus Persson and released by Mojang in 2011. The game allows players to build with a variety of different blocks in a 3D procedurally generated world, requiring creativity from players. Other activities in the game include exploration, resource gathering, crafting, and combat.
Rare	Rare Limited is a British video game developer and a subsidiary of Xbox Game Studios based in Twycross, England.
Super Mario World	Super Mario World (JP) is a SNES game by Nintendo. The game has been hailed as one of the many various Super Mario Bros. Has been remade or re-released on various platforms including the Game Boy Advance and the Wii,Wii U and 3Ds Virtual Console.

Acronyms

AI	Artificial Intelligence
APF	Automated Playtest Framework
API	Application Program Interface
GDC	Game Developer Conference
PIE	Program Information Exchange
UE4	Unreal Engine 4
UI	User Interface

Chapter 1

Introduction

This chapter presents the context involving automatic software tests and automatic playtesting, the motivation and objectives behind this subject. This subject is considered to be not only new and recent but also exciting in the game development area.

1.1 Motivation

Games are naturally complex, mixing many entertainment genres in diverse platforms into a seamless and enjoyable interactive experience. Game development is a very intricate process with numerous iterations. Sometimes these iterations include various changes that have a significant impact on the diverse game components. Video games are highly interactive and also have a large user base. In most cases, the same game can be available on multiple platforms such as PC, PlayStation, and XBOX. Dealing with all of these factors can lead to a considerable amount of unexpected and unforeseen mistakes. Such mistakes can easily be overlooked without proper testing.

Delivering a game with bugs will lead to criticism from the end-users. From a customer's point of view, encountering bugs may be a sign that the publisher of the game was not concerned about the quality of the product they released and did not respect the customer enough. This situation may lead to loss of customers and money. Therefore, it is essential to pay attention to the diverse changes that the various iterations perform, especially because some of these changes may be major, and it may be necessary to run regression tests in all the systems.

A way to test efficiently and quickly is by using automated tests and automated playtesting. Automated testing allows fastening the process of validation every time a change is made to the system, including when a new feature is added. Also, by using automated tests, wasting time is diminished because the developer would not have to

test the same features over and over again. After all, it is an automatic process, and if something fails, the developer will know what failed. Automated playtesting will be useful to see, for example, if a level works or if the interactions are running or even if the level is passable. All of this can contribute to better game quality and, in the end, for better enjoyment of the game from the clients.

1.2 Context

Writing good gaming software tests is difficult, and not every software developer's favorite occupation[27]. There is a lack of qualified professionals in the testing areas[13], and there is little support for trying to test a game using automated tests (unit tests, integration tests, regression tests). Manual testing is slow, laborious, and not cost-effective and automated playtesting is the opposite of manual testing[8]. The company that promotes this dissertation is a service provider company. ZPX is a full-range development studio offering design, art, coding, and other services to game developers across the world. At the moment, it holds a partnership with Funcom, a Norwegian game development company[1]. This dissertation was conceived to help the development team achieve better results in creating games, managing time efficiently, and making the testing system a little less stressful on the developer.

1.3 Problem

The biggest issue is that automated testing and playtesting in the game development field is not well researched; therefore, not every studio is aware of the advantages brought by automated testing and playtesting. It is hard to demonstrate the future uses and improvement it can provide since not many studios have dwelt in it or can show the results of it directly in-game production. This is the main issue, since there are not many examples of successful integration of automated playtesting it becomes difficult to convince studios that this is an improvement and not an added task. It can be an educational process for the studios and it takes time to show results. In addition, when starting the development of a new game, an issues arises when implementing similar automated tests or playtests that were used in previous game developments. Since the tests always have a few distinct details, in the beginning of the game development they cannot use the tests previously developed for other games, despite the similarity between tests in other games.

In general, the problem consists of not being able to prove that automated tests and playtesting bring advantages and not only more labour. Another issue is having to implement the same tests for each new game development and not being able to reuse the previous tests since there can be several alterations according to each game.

[1] F. Team.We are Funcom.url:https://www.funcom.com/

1.4 Objectives

The main objective for this dissertation is to look for an alternative or an optimization for the automated playtesting used in ZPX Company[2] while trying to maintain the solution general enough to use in multiple games in development. Before the development of this dissertation, there was not any type of automated playtesting, and it would be beneficial for the company to start automating the playtesting in order to gain time and reduce laborious tasks. In order to determine whether the objectives were able to be achieved the results obtained while testing a level with both the implemented solution and the manual playtesting will be compared, the latter being the current method used to test the games in development. Comparing the results will allow to draw conclusions such as whether there was improvement in the amount of time spent in playtesting and whether it was able to make the playtesting task less laborious.

1.5 Document Structure

This document is structured in five major parts:

Introduction: Chapter one describes the motivation and the context behind this work, the problem that it brings and the objectives of this dissertation to try and fix the problem.

State of Art: The second chapter holds the misconceptions about automated testing, what not to automate, and different tests that can be automated (unit testing, integration testing, regression testing, playtesting). It presents some types of glitches that occur in a game with examples of games that were released without noticing them. It also introduces several ways to explore the problem in question and the way to use the tool provided by the Unreal Engine 4. It also gives an overview of the different techniques and solutions already in existence, theoretically, or even implemented.

Framework Architecture: In this section it is presented the purpose of the proposed framework and also the path taken to achieve the final framework. After the introduction of the framework and the decisions made to create it, the chapter proceeds to describe the framework using diagrams to show the architecture and some of the techniques used.

Framework Implementation: The fourth chapter describes the implementation of the framework. In the previous chapter, it is explained the components of the framework and the techniques used; in this chapter, it is presented the class diagram and it is detailed the implementation of the tests with code examples from the framework. It is also explained the chosen API (application program interface) with examples to show between the different APIs that were available to create the tests.

Results: The fifth chapter, the results of the framework are presented and the manual playtesting in graph form with an explanation for each of the graphs, and also the chosen metrics. These graphs detail the time and the performance of the framework and

[2] Z.Team.ZPX - Interactive Software.url:https://www.zpx.pt/

the manual playtesting. The results from the framework and the manual playtesting are evaluated in order to formulate an opinion and to see if the framework can achieve the objectives previously detailed.

Conclusions: The sixth chapter consists of the conclusions, and it's where a conclusion based on the results previously obtained and their evaluation in chapter four is formed. In this chapter, it is also presented the limitations of the work implemented and explained the future work. It is also presented any improvement or additional work that can be done to the proposed framework in the future.

Chapter 2

State of the Art

There are various ways to approach automated playtesting. By presenting some of these various ways, we get to understand a little better about automated playtesting and tests used in general. First its introduced the Engine Testing Tools incorporated in the Unreal Engine and Unity. They can be used to validate different areas of game development. Following these tools, a summary of some frameworks that were implemented and used internally to create two different games in different companies is shown. To finalize, some final thoughts are presented.

2.1 Misconceptions

The first misconception is that automated testing will give more free time to the development team. When doing manual testing, most of the time is devoted to exploratory and functional testing, where the testers would manually search for errors. Once the process is complete, if there is a new feature added, the tester must repeatedly go through the same steps over again to make sure there are no errors[9].

With automated testing, the time searching for the error, and repeating the same steps is cut drastically. The time is instead spent in coding the tests and making improvements to these tests repeatedly as adjustments are needed. Once a new feature is added, they just need to run the tests since automated testing allows for the recycled use of tests and see if the new feature broke something. After that they only need to write tests to include the new feature. Mainly, the time spent on the mundane tasks and repetition a manual tester would go through is instead spent focusing on more significant, more important issues involving the software.

The second misconception is that automated testing is better than manual testing. There are pros and cons with each one, and they are supposed to be used in different

stages of the development of the game. For example, automated testing should be used in the initials stages of the game; it should detect bugs before the game is built and released. The manual testing is used more to see if the game is fun and challenging to play, to see if it gives the expected performance, to compare the results to the expected behavior, and to see bugs that the machine cannot detect. Both of these tests should be used but in different contexts, because both of them bring significant advantages to the development team[8].

2.2 What Not To Automate

When creating tests, it should be taken into consideration what is expected. So if the player gets to a rock, they should go around it or not be able to pass this rock. It should not be created a test that tries all the possibilities for it to fail (so what if the player goes through the rock or gets beneath it, or can destroy it) because it would take us to an infinite number of responses. These negative/fail tests should not be automated. Negative tests are not straightforward to give a pass or fail result, which can help. The tests should be success tests, for example, a player was successfully not able to pass the rock (the only result that makes the test pass). If it goes wrong, so the player was able to pass the rock, the logs can be observed and the development team can see what happened[9].

Tests with an extensive pre-setup take too long and can make a negative difference. So if the test needs many assets, the loading time, and maybe even initializing a network connection in the case of a networked test, it would influence the time a game software developer needed to wait to proceed the rest of the development because it needs to wait for the result of the tests.

2.3 Types of Tests

There are various types of validation tests. These tests are made before the product release and can be made throughout the whole journey of creating a game or a product. These validations tests have different purposes, advantages, and disadvantages. Here it is presented some of the most popular ones used to test video games.

- **Unit tests** are typically automated tests written by software developers to ensure the intended behavior and design of a section of an application (known as the "unit") is met. The goal of unit testing is to show that the individual parts are correct by isolating each part of the program[22]. A unit test has several benefits since it provides a strict, written contract that the piece of code must satisfy. One can build up comprehensive tests for complex applications by first writing tests for the smallest testable units and then writing the compound behaviors between those.

To isolate issues that may arise, each test case should be tested independently. Substitutes such as method stubs, mock objects, fakes, and test harnesses can be used to assist testing a module in isolation.

Unit testing is great in the development cycle since it finds problems early. These problems include both bugs in the programmer's implementation and flaws of the specification for the unit. The author is forced to think through inputs, outputs, and error conditions in order to write a thorough set of test forces and, thus, more crisply define the unit's desired behavior. During the development, a software developer may code criteria or result that is known to be right into the test to verify the unit correctness. During test case execution, frameworks log test results and report them in summary if any of them fails, it will appear. The cost of finding a bug when the code is first written is considerably lower than the cost of uncovering, identifying, and correcting the bug later. Bugs may also cause costly problems for the end-users of the software since it can be viewed as a lack of care. Unit testing can encourage developers to structure functions and objects in more beneficial ways since the code can be very challenging to unit test if poorly written[6].

There are many benefits in using Unit Tests, such as increasing confidence in changing/ maintaining code. If useful unit tests are written, and if they are run every time any code is changed, it will be able to catch any defects introduced due to the change promptly. The development is also faster, seeing that there is a unit testing in place, the test is written, and the code is written, and then it runs the code. Writing tests takes time, but the time is compensated by the less amount of time it takes to run the tests instead of looking for errors with, for example, providing a few inputs that hopefully will hit the code.

- **Integration tests** come after the Unit tests and before system testing. It is where individual units are combined and tested as a group[30]. The goal of integration tests is to expose faults in the interaction between integrated units. So, for example, testing a full-length feature or action in the game, this means they provide coverage of multiple units and the communication between them.

  The individual modules (unit tests) are first tested in isolation. Once the modules are unit tested, they are integrated one by one, till all the modules are integrated, to check the combination behavior, and validate whether the requirements are implemented correctly or not.

  There are various reasons to use integration tests, if all the unit tests pass and an integration test fails, it means that it can generally discount the units themselves and look for other causes, perhaps to do with the communication between units. An integration test failure, due to the broader scope it covers, will take longer to investigate than a unit test that tests something particular, but they are still a useful high-level signal that something about a feature has broken. The logic implemented

by one developer is quite different from another developer, so it becomes essential to check whether the logic implemented by a developer is as per the expectations and rendering the correct value following the prescribed standards and integration tests provides some insight to see if the logic works. Modules also interact with some third-party tools or APIs which also need to be tested. The data accepted by that API / tool needs to be tested to see if it is correct and that the response generated is also as expected[28].

- **Regression Testing** is used to confirm that a new program or code change has not adversely affected existing features[10]. So in this type of software testing, the output of a component is stored, and when there are changes in the component, the new output is compared with the previous output. An expert can manually validate the output. Regression Testing is, in summary, a selection of already executed test cases which are re-executed to ensure existing features or functionalities work fine after the new code changes are done. It is also suitable for testing code stability[33]. But it is not suitable for developing evolving simulations where a large number of test-cases need to be re-evaluated often and manually by an expert. Regression testing is a manual subjective testing technique, while suitable for some areas like visualization, it does not work well for cases where high precision is needed, or many simultaneous events must be evaluated at the same time.

  Regression Testing is necessary because software maintenance is an activity that includes error corrections, optimization, enhancements, and deletion of existing features. These changes may cause the system to work incorrectly. Retest All is one of the methods for Regression Testing in which all the tests in the existing test case are re-executed, but this is very expensive as it requires an enormous amount of time and resources.

  If the software undergoes frequent changes, regression testing costs will escalate. The automation of regression test cases is a wise option in such cases. The extent of automation depends on the number of test cases that remain re-usable for successive regression cycles.

- **Playtesting** is a method of quality control that takes place at many points during the video game design process. A selected group of users plays unfinished versions of a game to work out flaws in gameplay, level design, and other necessary elements, as well as to discover and resolve bugs and glitches[34]. Also, the process mainly involves clarifying the vague points, adding fun elements or reducing boredom, balancing the victory situations, and so on. It is a process of iterative testing, applying feedback, and redesigning a game. It can involve gray-box testing (the designer tests as if a user), alpha testing (testing feedback by a peer), and beta testing (testing feedback by a potential user)[24]. There are various objectives when using playtesting, and some of them are possible to automate. Testing if the game

Figure 2.1: Zelda: Breath of the Wild DLC: Player glitches out of Trial of the Sword

is fun or challenging is not possible to automate (playability) because it depends on human interpretation. An algorithm cannot determine if a user will find the game too hard or too dull. Therefore, when automated playtesting is mentioned, it is about finding bugs, testing the game, analyzing the level to see if it is passable or if any holes transport the player to a non-playable area. This is the type of situations that can be automated since it is not dependent on the subjective thought of humans[35].

2.4 Types of Glitches in Video Games

A glitch is an action performed within the bounds of a game engine that was not intended by the game's developers. Unfortunately, glitches appear in the majority of games; sometimes, the fault is not from the testers. Certain glitches are almost impossible to detect because of the behavior for them to happen in particular. However, some of them are preventable. These are some of the common glitches:

- **Out of Boundaries:** It refers to going beyond the normal boundaries of an area on the game map, this can be done by passing through walls, jumping off objects in order to get out of the map. In figure 2.1, it can be seen a screenshot of the game Legend of Zelda: Breath of the Wild and that the player was able to pass to an area that was not the normal play area[23].

- **Interface:** It is a glitch performed through a game's menu system or another similar game interface. In figure 2.2, it can be seen that the developer was trying to create something in the episode when a glitch occurred, the developer cannot see the menu because of the glitch [25].

- **Loading:** When a player sees things such as flickering/missing/incorrect textures,

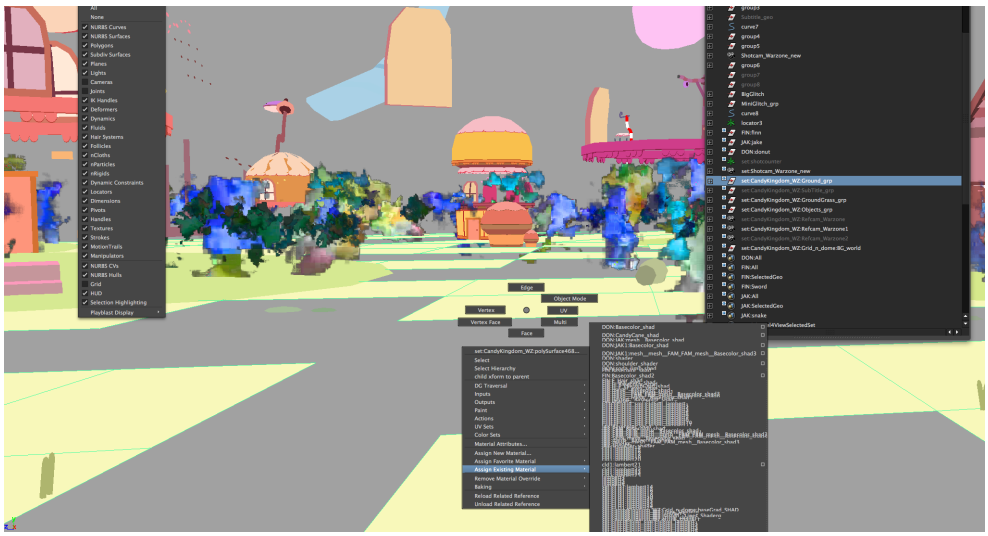

Figure 2.2: A glitch in the menu when creating an episode of the "TV series Adventure Time"

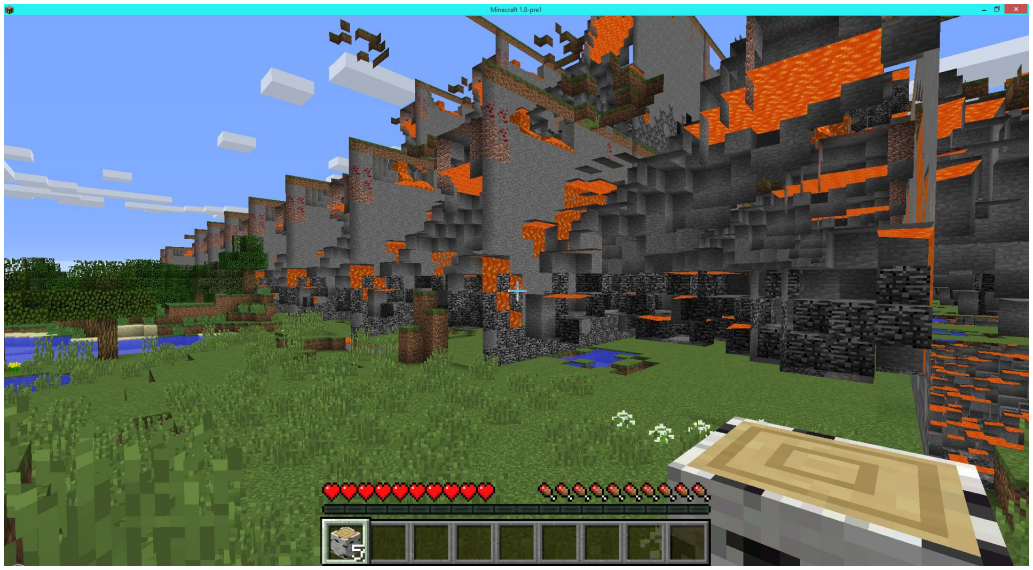

Figure 2.3: Loading Glitch on Minecraft

strange coloration, missing or disappearing geometry then it has encountered a loading glitch. This loading glitch in figure 2.3 is pretty normal to occur because sometimes it is the computer that cannot render the full image. In the game Minecraft this is pretty common because there is a lot of assets to load, and sometimes the player gets to an area where part of the geometry should be hidden by objects that have not been loaded at the time. In figure 2.3 it can be seen and example of this loading glitch since the magma and other parts of the rocks that are supposed to be hidden can be seen.

- **Game Breaking:** Game breaking glitches break fundamental mechanics of a game either temporarily or permanently. This glitch can include the save feature of a

Figure 2.4: Game-Breaking Glitch, The Pokémon Missingno

game, or the ability that is required to progress through the game, or the game can outright stop working. It can also be in the form of the game crashing on launch 100% of the time, thus making it unplayable and broken. The glitch shown in figure 2.4 is considered a game-breaking glitch because, when this pokémon that was not supposed to exist showed up, it could cause various effects in the game such as corrupting saved data or duplicating an item (master balls) and other effects[11], this means that some of these effects can make the game unplayable for the player.

This research shows some glitches that are common in video games. This kind of glitches reaches to the player because some of them are very hard to detect and reproduce. Unfortunately, there is always something that slips by the human eye.

2.5 Unreal Engine

The Unreal Engine[1] is a game engine developed by Epic Games. In 1998, Epic Games published the game Unreal a first-person shooter. Nowadays, it has been successfully used in a variety of other genres, like fighting games, MMORPGs, and other RPGs. The code is written in C++. Unreal Engine is an engine used by many game developers today, with it being open source. The Unreal Engine 4 is the most recent version, released in 2014 [26].

What a game engine does is that it provides various tools and programs to help to customize and build a game. Unreal Engine is very successful; it is even one of the most popular engines. Its popularity is mainly due to the multiplatform capabilities, and the ability to create high-quality AAA games with it. Other engines are used in game companies like Unity Engine, Game Maker or Construct2. The Unreal Engine was chosen because the majority of the games created in ZPX-Interactive Software were made using it. It is essential to communicate some components of the engine for the future framework and to understand more of the capabilities and resources of this engine.

[1] U.Team What is Unreal Engine 4.url:https://www.unrealengine.com/en-US/

The Unreal Engine is made up of a massive system of tools and editors that allows organizing assets around the engine: physics engine, graphics engine, input and the Gameplay framework, and online module[21].

The Gameplay framework contains the functionality to track game progress and control the rules of the game. The input system (that can be configured through the Gameplay framework)converts the buttons and keys pressed by the human player into actions that the in-game character performs. Gameplay classes such as GameState, GameMode, and PlayerState control and set the rules of the state of the game. The characters in-game are controlled either by players (using the PlayerController class) or by an Automated Intelligence (using the AIController class). The Pawn class is the base class that the in-game characters inherit from, whether controlled by the player or AI. The Pawn class has a subclass, the Character class, which is made explicitly for vertically-oriented player representation, for example, a human.

The access to Unreal Engine's source code gives users the freedom to create almost anything they can imagine. The base code in UE4 can be extended and customized to create whatever functionalities the game needs to have. Learning how the code of Unreal Engine works can unlock its full potential in-game creation since it can grant a lot more freedom.

Unreal Engine has also incorporated beneficial debugging features. One of these features is the Hot Reload function. This debugging feature enables changes in the C++ code to be reflected immediately in the game. Code View also facilitates quick changes in code. It is another feature that by clicking on the object code view category, it shows the relevant code directly in Visual Studio.

To understand a little more of this engine and not only its components, it is also important to talk about the different participants such as artists, the various designers, programmers and how they interact with certain tools from the Unreal Engine [16]. Some of the specialized groups are:

- **Artists** make the game look beautiful and visually appealing. They create all visible objects that appear in the game. This means it can go from trees to menu buttons in the game level. There are various specialized artists such as 3D modeling artists, animation artists, character artists, building artists, and some more. Artists normally create first using other software such as 3Ds Max, Maya, and MODO and then import the art created into UE4. They normally need to make use of Blueprints to create certain custom behaviors for the game.

- **Cinematic experts** are also trained artists. They have a unique eye and creative skills to create short movie scenes/cut scenes. The Matinee tool in UE4 is what they use most of the time.

- **Sound designers** work in the sound labs to create custom sounds/music for the game since they usually are musically trained and have an acute sense of hearing.

Their responsibility is also to import sound files into UE4 to be played at suitable instances in the game. They spent most of their time in the Sound Cue Editor on UE4.

- **Game Designers** determine what happens in the game, what goes on in the game, and what the game will be. In the pre-production, so the planning stage, most of the time is spent in discussions and documentation. In the next stage, the production stage, the designers often spend their time in the Unreal Editor to customize and fine-tune the level, and also, they ensure that the game level is created as planned by overseeing the game prototyping process.

- **Programmers** the group that creates the code that is needed to create the game. Before the production of the game, they look into the technology required and are also responsible for deciding which software programs will be needed and if they are capable of creating the game. Programmers also write the code to make the various objects created by the artist come true according to the plan that the designers came up with. They program the functionality and the rules of the game. Sometimes there is also a team dedicated to research and optimize graphics in games where the graphics are insane and extreme. This team, in order to support the needs of the game, which they are developing, modify and extend the features of UE4, by creating and modifying plugins or the source code in UE4.

2.6 Engine Testing Tools

2.6.1 Unreal Testing Tool

The UE4 has its own test framework. The Automation System is what eliminates the need for human interference to complete a task. In this case, it allows for the tests to be automated. This automation system is built on top of the Functional Testing Framework, which is the overall system in which the tests will be automated. The functional testing framework is designed to do gameplay level testing, which works by performing one or more automated tests[2]. Various tests can be made in this framework, such as:

- Unit tests - API level verification tests.

- Feature Tests - System-level verification tests such as verifying the program information change (PIE) or verifying in-game stats.

- Smoke Tests- Functional tests or unit tests. They are intended to be fast so they can run every time the editor, game, or commandlet starts

[2]U.E.Team, Automation System Overview.2019,url: https://docs.unrealengine.com/en-US/Programming/Automation/index.html

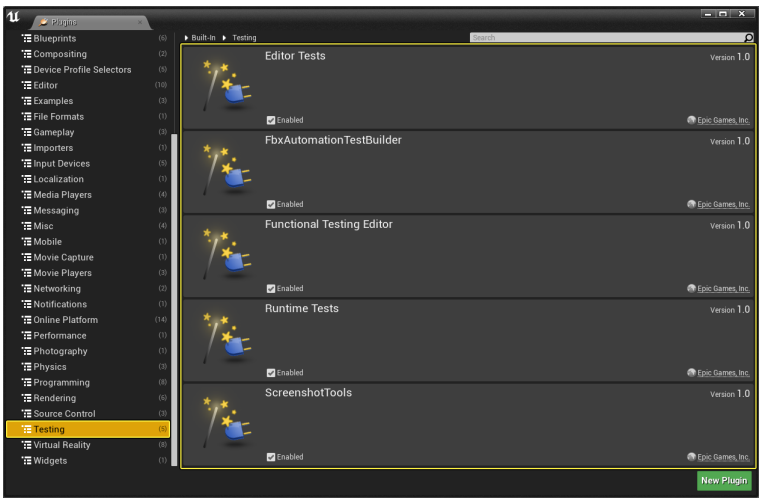

Figure 2.5: Plugin Browser in the Unreal Engine 4

- Content Stress - They are more thorough testing of a particular system to avoid crashes, such as loading all maps or loading and compiling all Blueprints.

- The Screenshot Comparison - Enables teams to compare screenshots that are captured in the editor quickly. The screenshots generated via an automation test can be viewed using the Unreal (Session) Frontend tool, where a history of screenshots is maintained (figure 2.6), enabling obvious rendering errors to be identified between builds.

- FBX Automation Test Builder - Creates a Test Plan for a single FBX file. This can be used to import or reimport and check against an array of expected results. The FBX (Filmbox) is a proprietary file format (.fbx) that is used to provide interoperability between digital content creation applications.

These tests are located in *plugins*, so there is a choice in the ones that will be used. In figure 2.5 the plugins and their organization can be seen.

While doing the research, it was found various mixed opinions for this framework in the unreal forums. Some of them said that the framework could be beneficial because it is integrated into the UE4, and the plugins are very organized. Other opinions said that although it is quite handy to use the test framework, the Unreal Test Framework is built on top of the engine; this means that the linking times will take much time[3]. To achieve a good workflow, it is essential to have quick feedback from new tests. If it takes 5 to 10 minutes for the tests to run every time they are modified it would likely not be feasible.

[3]https://ericlemes.com/2018/12/12/unit-tests-in-unreal-pt-1/

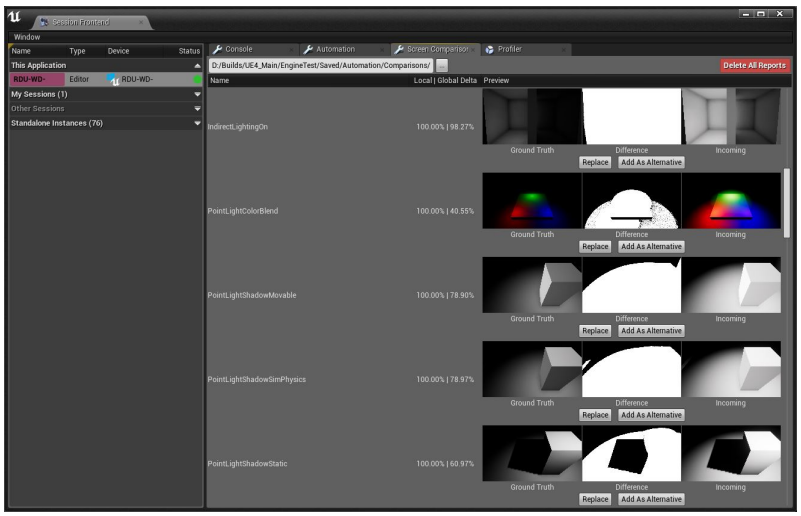

Figure 2.6: Example of the Screenshot Comparison in the Unreal Engine 4

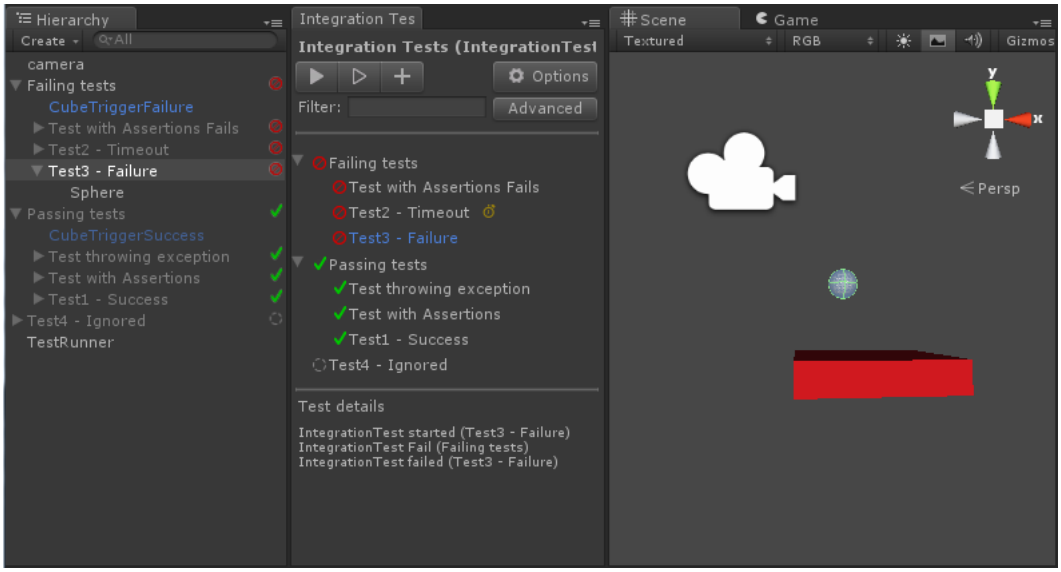

Figure 2.7: Test Framework- Exemplary Scene Shipped With The Framework

2.6.2 Unity Engine Testing Tools

The Unity Engine[4] also has a tool for automated tests. In this case, it has an Integration Test Framework (figure 2.7), a Unit Test Runner, and an Assertion Component[5] (figure 2.8). It is available in the Unity Asset Store, and it is free. The Unit framework also allows writing automated tests that drive the game in a way similar to how a user would, and it is based on the NUnit framework included with Unity and integrated with Test Runner (PlayMode). It also includes a lightweight dependency injection framework for object mocking. Still, the Unreal framework does have more options.

[4]U.E. Team.Unity Engine.url: https://unity.com/

[5]U.E.Team,Unity Test Tools Documentation.2015.url: https://bitbucket.org/Unity-Technologies/unitytesttools/wiki/Home

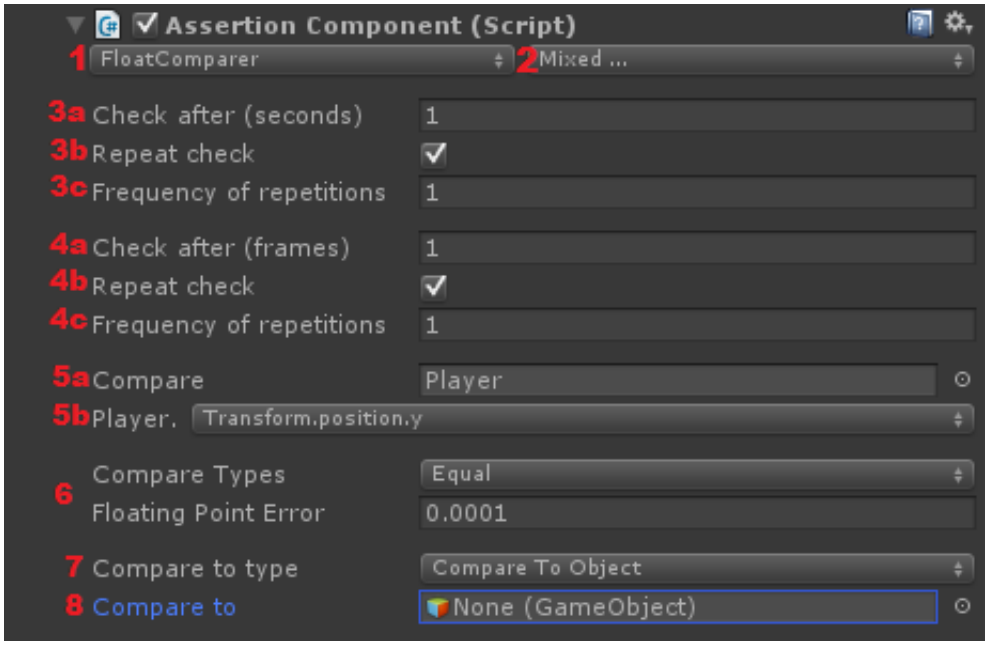

Figure 2.8: Assertion Component in Unity

2.7 Automated Testing Frameworks for Game Development

2.7.1 Automated Testing of Features in Sea of Thieves

In 2019, there were two talks about the current automated tests in the game Sea of Thieves. One was in the Game Developer Conference (GDC)[6] given by Robert Masella, a software engineer at Rare[17]. The other one was in the Unreal Fest Europe in the same year and was given by Jessica Baker, also a software engineer at Rare[1]. These two talks are about the same topic the new implementation for the Automated Tests and the effect that had in the game of Sea of Thieves.

For starters Sea of Thieves[7] is an open-world game where the players are pirates that can sail ships (figure 2.9) attack islands, do missions, fight against skeletons, play with friends online or even just lots of tiny features like drinking and puking afterward. Sea of Thieves is a complex game, and with the complexity, it comes bugs and probably a lot of them. What they explain in these talks is how they made this the most bug-free game possible by utilizing the Unreal framework for automated tests and even their framework. The talk presents the reasons for changing their old way of testing because this was a game that would be continuously updated with new features or with new updates to correct eventual bugs. By adding new features if not correctly tested, the new updates could break the old features that were already in the game. So, for that reason, it was necessary to use automated testing to be able to test lots and lots of features and

[6] InformaTech, Game Developer Conference url: https://www.gdconf.com
[7] S. of Thieves Team.Sea of Thieves Game.url: https://www.seaofthieves.com/.

Figure 2.9: The gameplay of Sea of Thieves[32]

interactions that could break at any moment. They explained the problem when they used solely manual testing where lots of bugs were discovered by players and not by the manual test team. Following the problem with players finding many errors, they explain their framework, implemented on top of the Unreal framework. To accomplish their objective, they modified to a great extent the Unreal automation system. Next in the talk is an explanation of the types of tests that could be run in the framework, such as unit tests and integration tests. There were a lot more of them, but they detailed these two. The unit tests were used for the logic of the program, thus testing the code at the function level. Following the unit tests, they explain the integration tests. These tests are crucial because they test a full-length feature or action in the game; this means they provide coverage of multiple units and the communication between them. Therefore, when a unit test fails, they know which specific part of the code is failing. If all of the unit tests pass, but the integration test fails, at least they know that the problem is not the units themselves and look for other causes, perhaps to do with the communication between units. Integration tests in Sea of Thieves were created as maps in the Unreal editor. Each map would test a specific game behavior in a fixed scenario and report back its results as a pass or fail, based on whether the behavior happened as expected or it did not. After explaining more about the tests and how they made them and even how they test the multiplayer integration, they also present the way they use the new framework now (figure 2.10). In summary, here is the whole sea of thieves testing process that replaced the older, manual only process of previous games.

The only problem with this process was the time that some of the tests took to run, and they made a comparison. The unit tests took 0.1 seconds to run, and the integration tests took 20 seconds, and unfortunately, the majority of the tests were integration tests. Integration tests were slower since the integration tests had to load all the assets required and even had to initialize a network connection in the case of a network test. Also, the integration tests are very dependent, and that makes them slower and more unreliable.

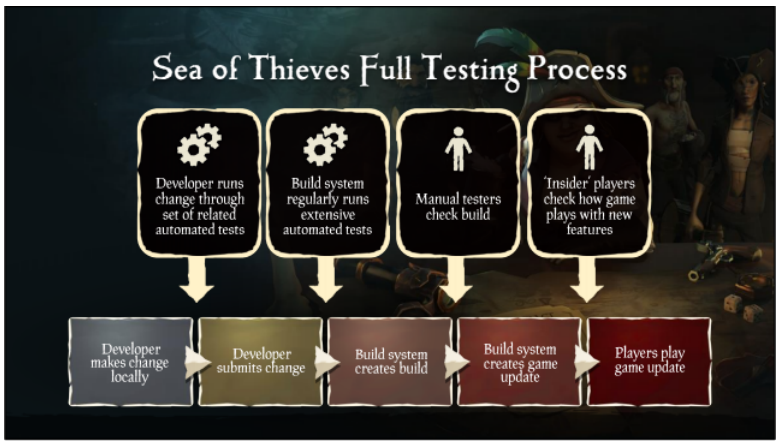

Figure 2.10: Testing Process in Sea of Thieves

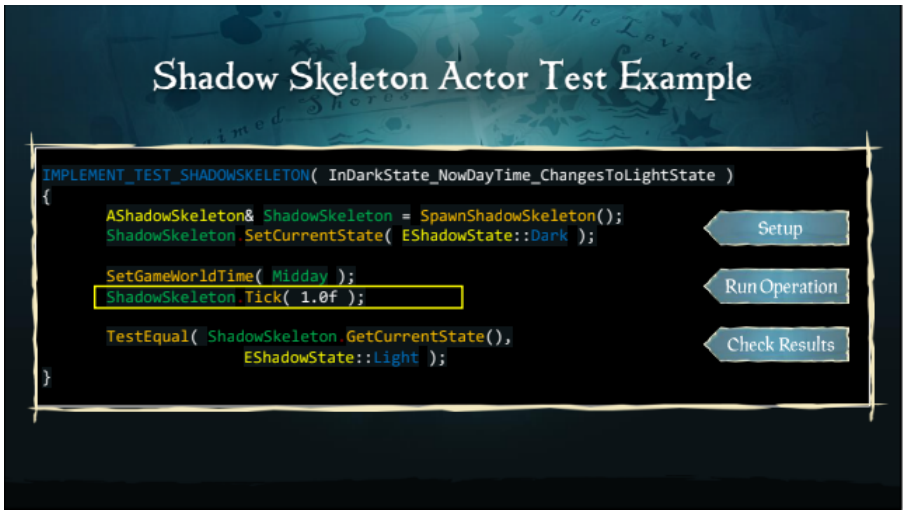

Figure 2.11: Testing Code For Shadow Skeleton

So they created a new type of test "actor test"so-called because they used the actor Unreal game object class. The actor test type gave them a useful middle ground between integration tests and unit tests. They were, in effect, a unit test for game code, but one that treated Unreal engine concepts like actors and components as first-class dependencies. So they gave an example to understand better this concept, a skeleton in their game changes when the day turns to night and vice versa, so what they made has they created a shadow skeleton and set its current state to dark. Then they run the behavior they want to test, which in this case is setting the world time to midday. Finally, in the check phase, they assert that the actor is now in the light state like expected.

In the code in figure 2.11, they tick the shadow skeleton explicitly for him to detect the new world time and change its state. Having to call a tick like this explicitly is the downside of actor tests, as they are testing the object outside of its normal environment. The major benefit they get is that this test runs incredibly quickly compared to running an integration test because there are no associated costs like in the integration test. So

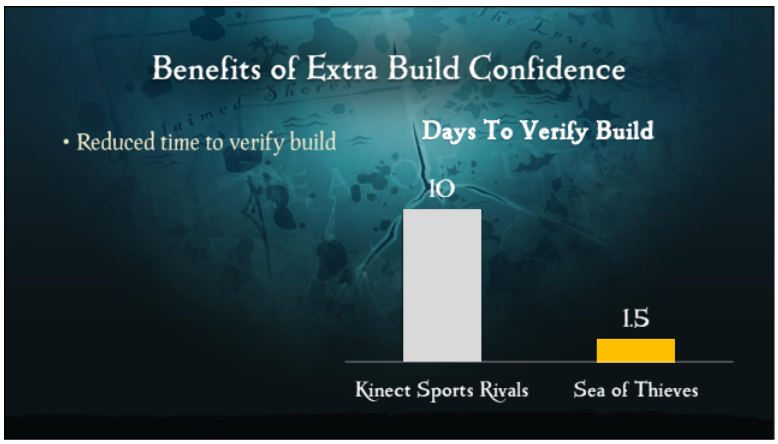

Figure 2.12: Benefits of the Automated Tests (Build Time)

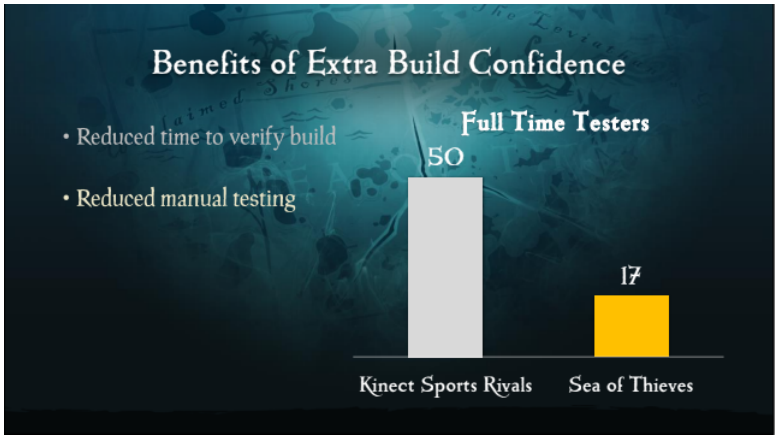

Figure 2.13: Benefits of the Automated Tests (Bug Count)

like this, they use the actor tests for the failure cases and the integration tests for the successful gameplay. To make a better improvement, they also started to test with the integration tests multiple related features, by doing this they only pay the initialization costs of creating a world, establishing network connections and loading all the skeleton game assets once, rather than incurring them multiple times.

Finally, in both talks, they speak about the benefits that these new methods brought to the game. The first one was that they were able to rapidly respond to players' feedback if they needed to because the time to verify a build was reduced(figure 2.12). There were also fewer testers(figure 2.13), and the bug count was consistently low(figure 2.14). The final positive aspect was that the developers said they worked less overtime comparing to when developing other games.

In conclusion, these talks were very eye-opening since they demonstrate that the automated test framework from the Unreal Engine is beneficial, and there is a great deal to gain from the benefits of using automated tests and implementing them in the process of creating a game.

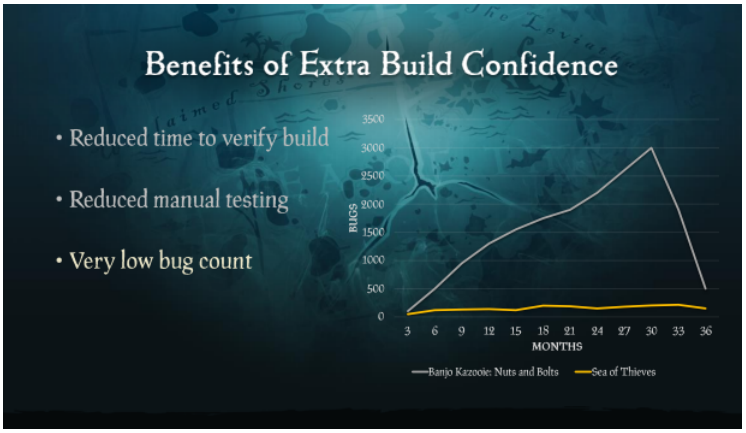

Figure 2.14: Benefits of the Automated Tests (Number of Testers)

2.7.2 Fast Iteration Tools in the Production of the Talos Principle

At the conference of the Independent Games Summit[12], speaker Alen Ladavac presented the tools that helped the production of "The Talos Principle"[18]. For context, "The Talos Principle"is a puzzle game designed by Croteam[8] and published in 2014. It was released for various platforms such as PlayStation 4, Nintendo Switch, Android, Xbox One, Microsoft Windows, iOS, Linux, macOS Classic and macOS. The game was successfully delivered by Croteam in many platforms, and the game itself is considered to be of good quality. At the conference, the speaker starts by explaining how they were able to achieve such quality while being able to not compromise their deadlines even with a small team (25 people)[14]. The first problem they tackle is reporting bugs; when a tester finds a bug, it writes a report about it, which includes the circumstances where the bug appears, its location, and how serious it should be considered. When a developer gets hold of the report, it runs the game and tries to reproduce the conditions for the bug to occur, and then it seeks to fix the error. The tester job has a hard task to accomplish and a boring one at that, to write the report, and then the work for the developer is also hard to be able to reproduce the exact conditions for the bug to show.

The way they found to improve this was by creating an automatic tool to report the various bugs inside the game (figure 2.15). If the tester finds a bug, it will mark it with the help of the new framework, and this mark contained the specific bug, a short description of its location, the person assigned to check the error, and the bug itself signalized. After marking it, the tester could continue to test the game, and the report of the bug would appear in the build monitor, which contains the various results of the automated tests, the detected bugs, and the result of the builds.

The developer would only need to check the build monitor(figure 2.16) go to the report with the bug, and then the instance with the bug would be loaded, and the developer could see the cause for the problem and fix it. Much easier for the tester and the developer. The speaker says that they saw vast improvements after using this new framework.

[8]Croteam.Official Website - Croteam.url:http://www.croteam.com/

Figure 2.15: "World Bugs System"Reporting inside the game.

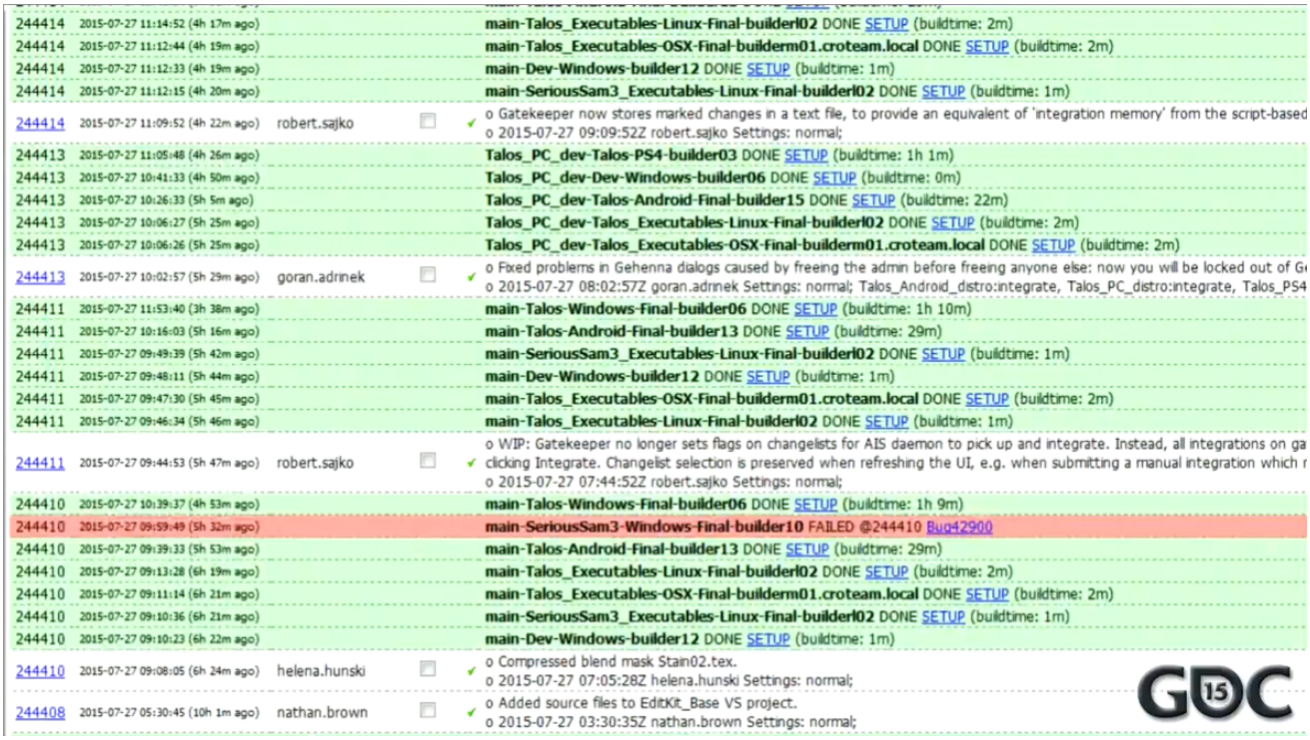

Figure 2.16: Example of the Build Monitor used by the developers.

The second tool introduced in this talk was a bot that playtested the game. The developers created a bot and trained it to be able to play "Talos Principle."They trained it by playing the game first and making the bot "watch"the footage and learn with the gameplay from the developers. After that, the results were astounding. The bot was able to complete the game, and when it was optimized, the bot could complete the entire game in thirty minutes because the bot played in fast-forward mode meaning it did not respect real-time synchronizations but played as fast as the CPU could calculate physics and A.I. By having the bot completing the game, the developer knew that the features changed or added did not break the game because the bot was still able to complete it, and they knew the game was not impossible to complete. A human tester would take hours to do the same. The bot changed the process by making it a lot more time-efficient and saving a huge part of the budget since the bot was pretty much doing the work of eight people. As they said, *"It clocked over 15000 hours of testing just at the end of the production. If we compared this to the same amount of human testing, it would mean that one person would have to play for over eight years, full day."*[2].

The only downsize, or disadvantage of these two new tools was the time invested in designing and implementing them. The speaker said that this did not happen overnight, and they had been years in the making. He added that they started around 2001 and only finished them by the time the production of the game started in 2012/2013. However, the efforts did bear fruits, as it can be seen from the success of the game, which shows that by automating tests and playtesting, they gained huge benefits even if it took a lot of time to create a great tool.

2.8 Manual Playtesting

Studios use manual playtesting in order to test their games. The difference is the way they arrange playtesters. Playtesters can be ordinary developers or employees from the company that take the time of their day and test the game and create reports containing important feedback. However, appointing employees for playtesting is not always feasible due to the amount of time it may take to test the game. The longer it takes to test a game, the more precious time the studios are taking away from an employee that could be accomplishing tasks they are actually qualified to do. Playtesting is not as easy as playing and determine if a player is able to clear the game. Playtesting involves a lot of metrics and time put into testing. Regular employees like artists or developers are not trained to test the game with the metrics in mind. These metrics consist of measurable qualities such as efficiency, performance, difficulty in some actions or levels, level of boredom, and excitement[5]. With those thoughts in mind, some studios, usually bigger studios, hire companies to test their games. These companies provide competent and qualified playtesters who are capable of testing a game in various ways and modes in order to conceive a detailed report on the issues found while also providing important

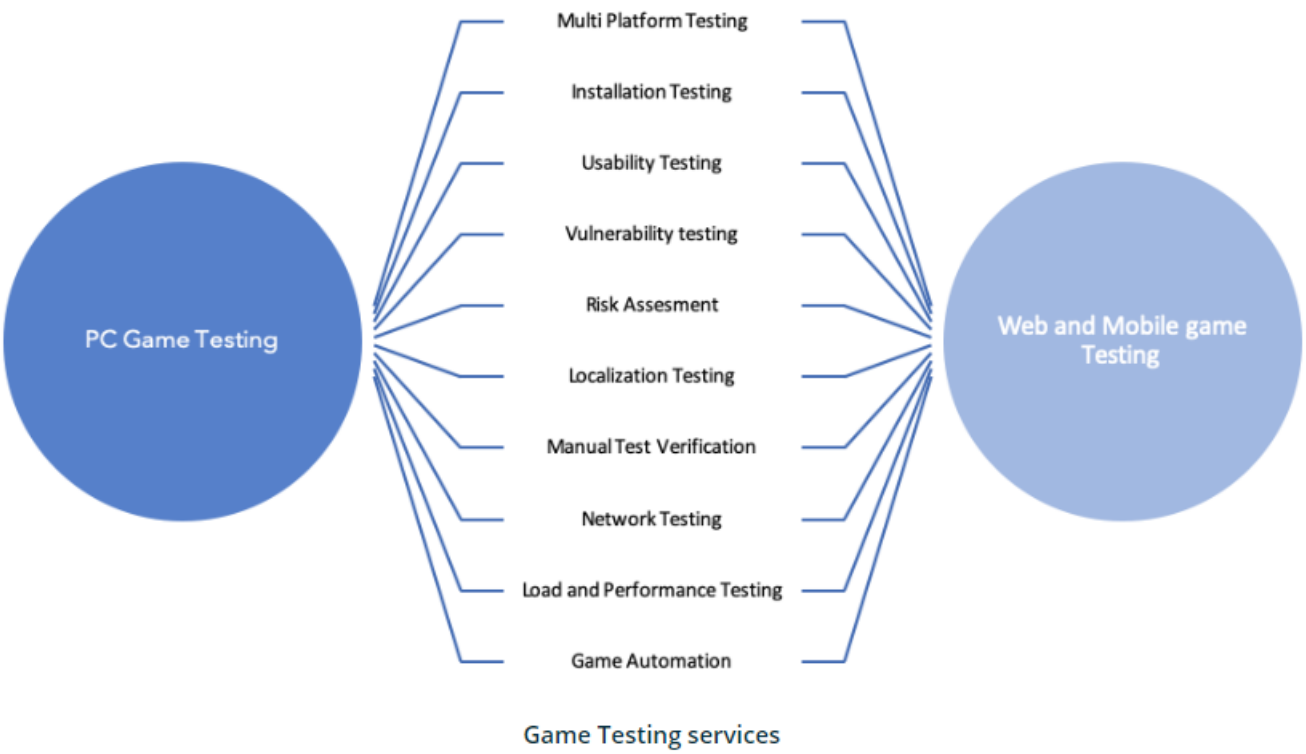

Figure 2.17: Example of the tests made by the company KiwiQA.

feedback for future improvement. An example of such a company is KiwiQA[9], a company that provides "a team of experienced game testers and QA personnel, delivering unmatched games and mobile app testing services."[10]. This company is an example of many others that are experienced in playtesting games. On their website, they present a number of different ways used to test the games, and assure that an experienced number of professionals in the area of playtesting run those tests. As it can be seen in figure 2.17, the number of tests conceived are quite varied and test a lot of different areas of game development. It can be understood why studios would pay companies such as KiwiQA considering they demonstrate a high level of professionalism and variety.

Unfortunately, this type of companies are not accessible to every studio, especially indie studios where the budget is limited[19]. Hiring a company for the playtesting costs a substantial amount of money since it generally consists of a group of playtesters, and playtesting takes a considerable amount of time. Therefore, the studios end up paying for these hours of work. This means that many indie developers attempt to recruit people online to gain as much feedback as possible. They can create a detailed guide to help the playtesters test their game since an unexperience playtester might have difficulties in playtesting[31]. They usually give a prototype to a group of selected people, and in the end each person only needs to fill a smart sheet. A smart sheet is a questionnaire with

[9]https://www.kiwiqa.com/game-testing.html
[10]https://www.kiwiqa.com/game-testing.html

a certain amount of questions defined by the studio that can be filled by the playtester. Typically, these playtesters work for free. The problem that arises with these playtesters is that the game can sometimes get leaked, and even if the entire game is not leaked, some details might be, which can be gruesome to deal with in some cases.

A way to keep the cost of playtesting low is to let the developer test the prototype in its initial stages. So only the mechanics and physics get tested in order to ensure that everything is going smoothly. And then, when the game is more robust, they hire a company, as mentioned before, to test the game more thoroughly. Sometimes this feedback delays the release date of the game since creating feedback can take a long time, and after receiving it the studio needs to correct the errors pointed out by the company.

Playtesting is always a necessity since there is always a need to have people with no background on the game give their unbiased opinion. The unbiased opinion of a group of people is crucial to every studio, indie or not. If hiring a company can be part of the budget, then that is excellent for the studio since its employees can focus on other parts of the project, and since the company is bound to a non-disclosure agreement, it cannot disclose any content from the game. And so, studios typically believe playtesting to be a very important part of game development despite its possible costs[4][15].

2.9 State of the Art Summary

In summary, this chapter demonstrated that there is a growing preoccupation in using automated tests. The engines are, little by little, incorporating their own automated test frameworks and trying to improve them. The Unity Engine was not focused on since the dissertation was made in partnership with the ZPX-Interactive Software, and they use at their core the Unreal Engine, but the effort put into the Unity Engine can still be appreciated. An analysis of two different game development studios that implemented their own frameworks for automated playtesting was presented, showing the excellent results obtained that way. The advantages that it brought to these studios can be seen, such as the way the developers gain time, which can then be used for more important things (improving features and gameplay) and the superior quality when delivering the end-product. By the end of the chapter it is presented the ways studios typically arrange playtesting, by hiring companies or gathering people from the internet to playtest for them. Even if manually executed, it can be recognized how important playtesting is in game development.

Chapter 3

Framework Architecture

For this dissertation, the objective was to create a framework for automating playtests. As explained before, various general and recurrent glitches happen in various games. The out of boundaries glitch was chosen as the focus for this framework, making the designed and implemented tests specific for this issue. These tests were created to test the integrity of the playable area, area that a player is allowed to play in. By doing a set of movements, such as walking, jumping and pulling objects (depending on the game), it could test if there where any holes or ways to exploit the game in order to get out of the playable area.

The expectation for this framework was to make life easier for testers and game level designers. For testers, since by taking away this tedious and repetitive task from them, they can focus on other parts of testing, such as playability or the fun and challenging parts of the game. The game level designer's work also becomes more comfortable and convenient. Typically, testers would find errors, write a report and then deliver it to the game level designer, so it could then reproduce the errors and attempt to fix them. With the framework a game level designer can just run it and quickly detect any problems with the level that he is working on. Thus, this framework takes away the middle man for the troubleshooting of the level.

Another goal was for the framework to be able to integrate and add new features with the upcoming future projects, and make it flexible enough so that any necessary improvements can be made without much struggle. Another requirement was to be an actual framework meaning to be general enough for any game to have fewer things changed from game to game.

The tests considered for this framework were playtesting tests. Therefore, they test the game in the game thread. The bots can be observed walking, jumping, and executing an assortment of different activities at the specified level.

The implementation of the framework was chosen to be on top of the Unreal Engine.

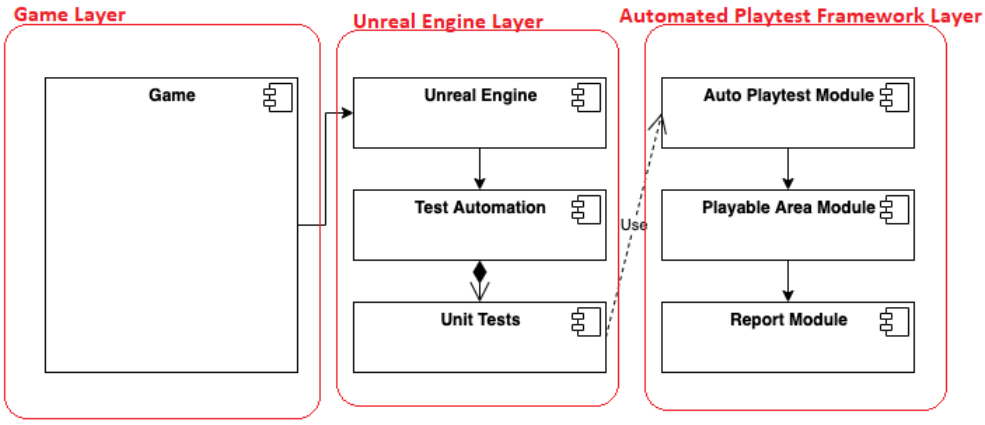

Figure 3.1: Components diagram of the implemented framework with the layers.

As stated in previous chapters, ZPX-Interactive Software uses the Unreal Engine to create the majority of their games. So it came as evident to use the same engine as the base for the framework. This way, since both the game and the framework originated from the same engine, it was possible to have access to the same tools and components; thus, making testing the game easier. By using the Unreal Engine, the Automated Test Framework already implemented could be used and could also be used as a reference and foundation for the framework. This allows for possible future improvements and usage as a base for future frameworks in other engines. The framework uses the language C++ since it is the language used to code in this particular engine. It was also used Visual Studio as the editor.

In the following sections, it is explained the architecture of this framework and each of the distinct components that compose it. At the end of the chapter, the scope of this framework is overviewed since it is still in a prototype phase, and it is worth mentioning what type of environments and tests this framework includes at this moment in time.

3.1 Architecture

The architecture is composed of three main layers, the Game layer, the UE4 layer, and the Automated Playtest Framework (APF) layer, figure 3.1. All of these three are interconnected with each other. In figure 3.2 the component diagram can be seen and how these layers interact with each other. Firstly, the Game layer represents the implementation of the mechanics of a game. The UE4 layer is the platform between the support systems and the implementation of the game mechanics, offering a system named "Test Automation"that allows the creation and execution of unit tests and integration of personalized tests. In the diagram of the components in figure 3.2 the corresponding component for the personalized tests is identified by the name of "Unit Tests". The third and final layer is the APF, created for this dissertation and composed of three main components:

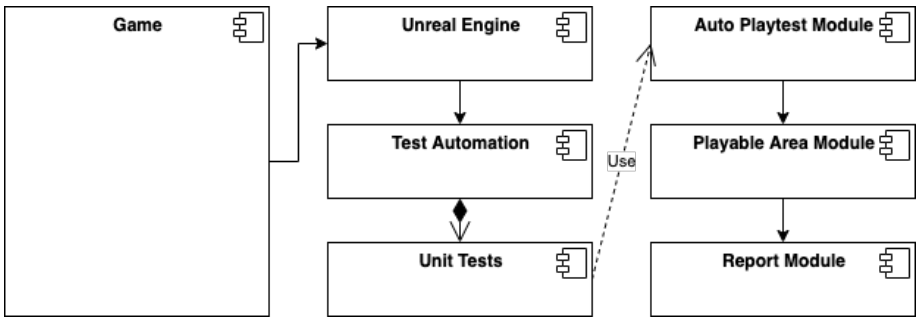

Figure 3.2: Components diagram of the implemented framework.

- **Auto Playtest Module:** This module controls the logic of the program, and all modules are connected to it. It is important for the APF since it takes all modules and creates the logic of the test chosen by the game level designer.
- **Playable Area Module:** This is where the level area is divided and where the locations that are valid to be tested are saved. This module is crucial because if there was an infinite number of valid locations the test could run forever while attempting to test every location possible (even the ones already outside of the level area). This would take a major toll on the time and performance of the test.
- **Report Module:** This is the module that creates a report, as the name indicates. This report includes the locations where the test failed which the game level designer can use to test further or correct these locations.

The APF is integrated with the other two layers at the level of the component "Unit Tests"that is represented in the component diagram 3.2. A unit test or an integration test uses the generic components in the APF to define the game mechanics that it intends to execute to validate each section of the playable area.

The framework was conceived to be scalable and flexible. The way the APF was integrated made it independent of each project since the implementation of the game mechanics is dissociated from the APF and the use of the APF is made at the level of the unit tests which are also independent of the APF. This approach made it possible for the APF to be flexible and to be easier to integrate new forms of testing the playable area with the addition of new unit tests. This demonstrates the expansion capacity of the framework.

In the next sections the described components are explained in more detail. In the end, the logic of the framework is explained with an activity diagram.

3.1.1 Playable Area Module

The objective for the Playable Area Module is to identify the playable area, perceived as the area that a player is allowed to play in, and split it into numerous valid locations in order for them to be tested. The split is achieved through the use of algorithms and

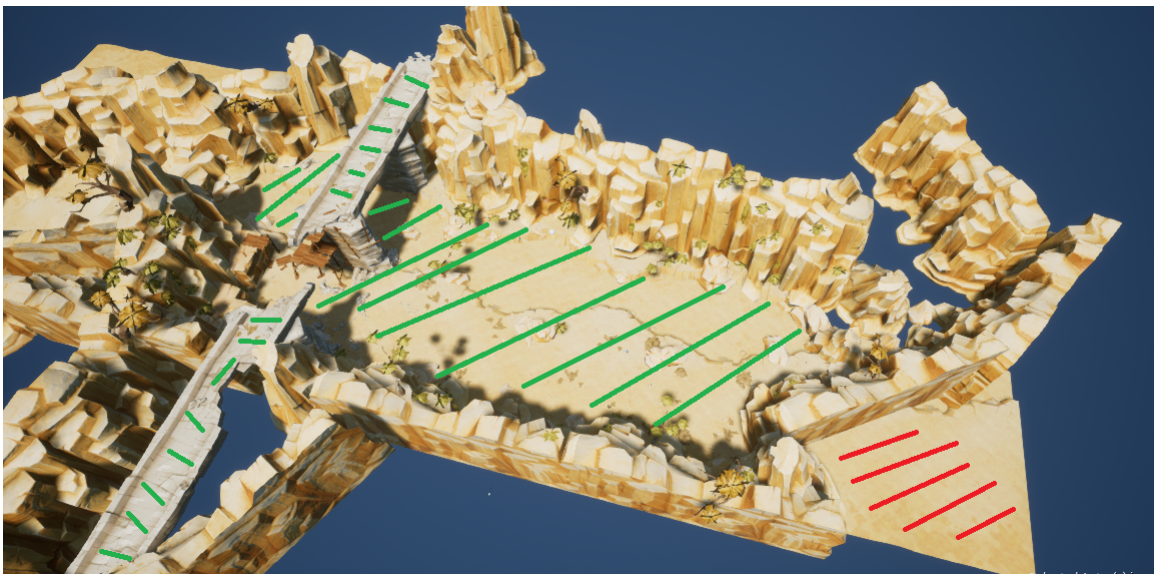

Figure 3.3: Map example with the playable area in green and non playable area in red.

in the context of this dissertation the implemented algorithm sweeps the level area and splits it into a grid. The size of each cell in the grid is configurable.

The playable area, as said before, is the area that a player is allowed to play in; therefore, everything that the player can step on. Areas such as the standard floor, climbable boxes, and other types of platforms are considered inside the playable area. However, defining the playable area is a challenging task because the path where a player can walk is not entirely defined or regular as it can be seen in figure 3.3. In this figure the playable area is marked in green while the non playable area is marked in red. When creating a map, the limits are usually created by introducing mountains or something tall enough that the player cannot pass, but behind the mountains there is still the rest of the platform that the map stands on, as it can be seen in the same figure 3.3. The rest of the platform is not used since the player is supposed to stay within the established limits and not exploit the entirety of the area available for map creation.

Typically, platforms for map building purposes are shaped like squares and rectangles since it is easier to use default objects, such as cubes, and expand them to meet the game level designer needs. With the platform built, objects such as mountains, trees and walls can be added in order to limit the player to the desired playable area. This is a much easier process than re-configuring the platform and adapting it to fit the map since the game level designer would have had to use customised objects to reconfigure it in some way and still add the chosen limiters in order to limit the player and prevent it from falling of the edge of the map. Thus, the most common practice is to use the default objects to create the platforms.

With this knowledge, it had to be decided how to limit the playable area. The platform cannot be classified as valid in its entirety since the game should be limited by the margins established by the game level designer. The margins cannot be defined

Figure 3.4: Map example with the failed locations marked.

either since they depend on the objects used, such as different types mountains, walls, rivers and trees. Being able to use several different objects to define margins means automatic margins categorization was not possible. As such, it was figured that it would take months of research and implementation to achieve something automatic due to the large number of variables that it takes to categorize margins and what is considered to be valid. So taking into consideration all of the work involved in margin categorization, it was decided to implement something manual to limit the playable area, something that could limit the playable area as necessary but was still flexible enough to be used in different playable areas besides the platforms and standard floors referred to previously, such as caves or under bridges. This is important since these also represent playable areas but are not typically at the same level as the standard floor and not as detectable as the standard path. Hence, it was decided to create a box that could be placed by the game level designer. The area inside this box would be considered playable area and its flexibility would allow the game level designer to size it accordingly and place as many boxes as necessary to limit the playable area. By having these boxes placed in the level, the problem with the margins was resolved. When the test runs the boxes visibly disappear but they are technically still there for the logic of the test although invisible to the human eye.

3.1.2 Report Module

The Report Module objective is to generate reports or, key performance indicators, based on the data obtained from the tests execution. The report results can be presented in different formats: text, graphics or images. In this dissertation it was developed a

report that identifies the zones in the playable area where the test failed, allowing to identify the zones where the player was able to escape. This report also consists of a text file with coordinates of the locations where the test failed, and in order to visually aide the game level designer, it was marked in blue the failed locations found by the raycast and in red the location that the tests failed. Both blue and red failed locations found are marked on the map opened by the automation system from UE4, one example of these marks can be seen in figure 3.4. This map always opens when running the tests from the automation tests window, and the game level designer can watch as the test runs and determine whether the test failed by looking at the automation results panel. If the test failed, the game level designer can look at the open map and identify the marks and their locations. This kind of report is more straightforward than the report that testers present because, in this one, the game level designer can observe the test running and understand precisely what happened. It can identify what test failed and what reasons led to its failure. Therefore it understands the set of actions that resulted in the glitch. It also has access to the saved marks and respective coordinates for posterior analysis.

3.1.3 Auto Playtest Module

The Auto Playtest Module takes all modules and creates the logic of the tests chosen by the game level designers. It is integrated with the UE4 by the unit tests which use the modules retrieved and joined by the Auto Playtest Module. By getting the Playable Area Module, spawning the bots and calling the Report Module, the Auto Playtest Module can orchestrate the logic of the unit tests.

With the implementation of this framework, extra care was taken with its generality and scalability, as previously mentioned. It is easy to add more modules or substitutes since the logic is in the Auto Playtest Module, but the actual implementation of the functionalities is in the modules. So, for example, if a different algorithm was to be used for the playable area, it would only need to replace the module, and everything would work out as intended.

3.2 Framework Logic

In order to explain the logic of this framework, an activity diagram, in figure 3.5, was created that demonstrates the activities in the framework when the game level designer chooses a test. The first step is for the game level designer to position the bounding boxes in a way that covers all the playable area, after that it configures the variables of the test in the TestInterface and runs the test in the automation system of the Unreal Engine. Following these steps, the framework presents the variables as public for the AutoPlaytestManager, which runs the logic of the test. The first step is to create all of the valid locations, and for that, the manager class calls the playable area module that calculates all the valid locations. Next, the manager class receives the valid locations, gets

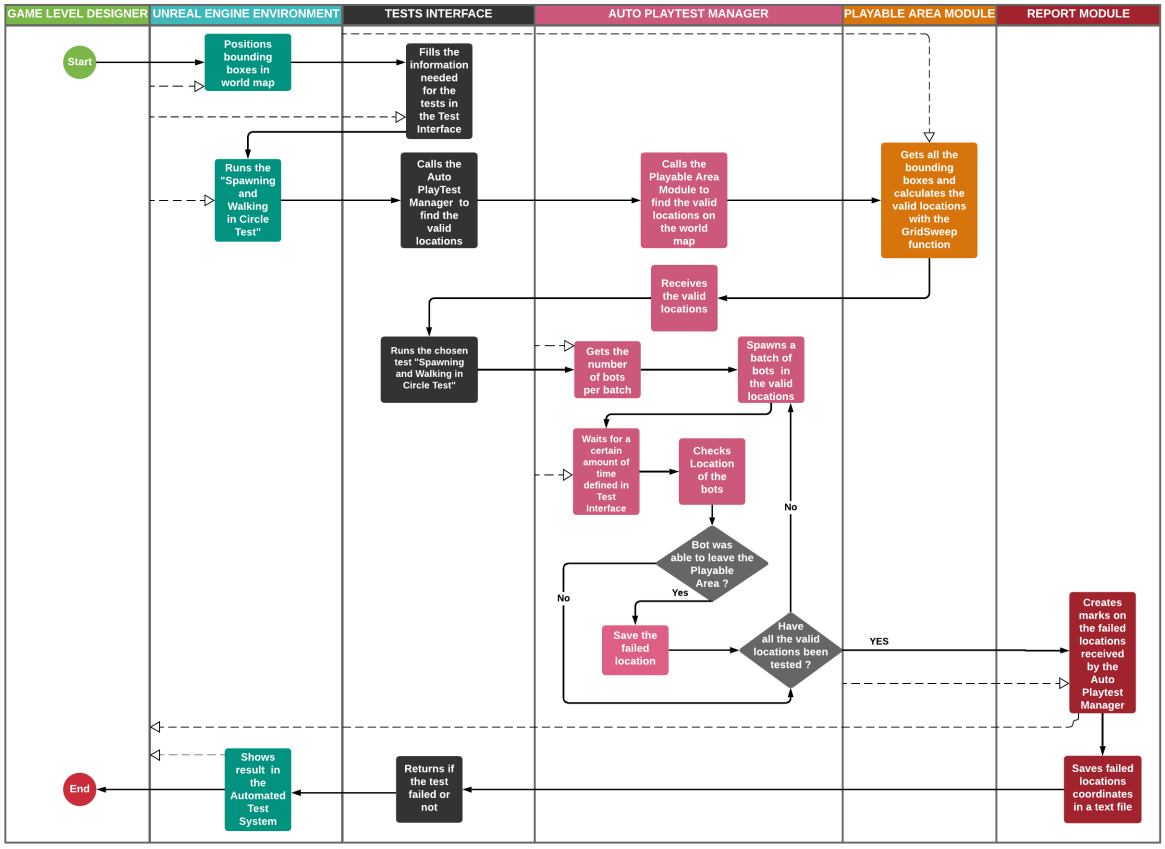

Figure 3.5: Activity diagram of the framework.

the type of test to run, and the variables needed. It spawns a batch of bots and waits for a determined period of time. Later, it checks the location of each bot and saves any failed locations. After testing all valid locations, it sends the arrays with the failed locations found by the test plus the raycast to the report module. This report class marks each failed location with its corresponding color in the map opened by the automation system and saves the coordinates of each failed location in a text file. After this, the TestInterface sends the result of the test. If a failed location was found, then the test comes back false, and the test appears as failed in the automation system, ending the work of the framework.

3.3 Scope Definition

Conceptually, the framework can be applied to test 3D, 2D and 2.5D games. It is not limited to the type of game since the delineation of the limits of the playable area is transversal to any type of game. However, for limit purposes of the dissertation scope, it was decided the focus of the framework would be 3D games. Also related to the dissertation scope, the definition of the playable area is manually achieved by the game level designer. However, by expanding the Playable Area Module from the framework it would be possible to integrate other mechanisms that could automate at least a part of the

process when it comes to defining the playable area. As such, the playable area is defined by a set of volumes. Although the concepts are transportable to other game engines, it was decided the UE4 should be used for the implementation of the framework since it is the tool that ZPX makes use of and also due to the fact that UE4 offers a very interesting Test Automation platform. The unit tests are the point of integration of the framework by defining and integrating the game mechanics that validate the playable area. All of the tests created are considered to be playtests. The playtests act in accordance with the following aspects:

- Some unit tests are included in their composition before running the test itself; these perform simple tasks such as checking beforehand whether the world was opened correctly.
- The playtests run in the level and the game thread, so it is the same environment as when it is pressed play since due to the UE4 characteristics the tests that resort to API of the physics engine need to be executed in the game thread.
- The playtests use the physics of the game and any limitation that the game imposes.

In the framework, three playtests were implemented. The first test is the most simple test created in order to determine whether the framework was working correctly and to serve as a base for the rest of the tests. This test is called "Spawning Bots Test"and spawns bots in the valid locations of the playable area and checks if their location changed after a few seconds. The second test is more complex, it is called "Bots Moving In a Spiral Test"and it has the same logic as the base test, except in this one the bots walk in a spiral in order to determine whether they can get out of the playable area by walking. The third test has a configurable jump and it is named "Configurable Jump Test"with the same logic as the second test but the bot jumps the number of times that the game level designer decides. By creating these three tests the playable area is tested with different sets of movements and is introduced two different ways to use the base test.

In order to test the framework, an adventure and exploration videogame was used with more simplified game mechanics in order to validate the concepts of the framework. However, it could be easily applied to another type of videogame namely survival games that the company is currently creating. Once more, in order to concentrate the effort on the concept tests the framework intends to evaluate, it was decided to use a section of the level of the game that occurs in an outside exterior. Although the framework can test interior environments, we chose an exterior environment for the scope limitation and also due to the fact that the playable area is defined manually.

Chapter 4

Framework Implementation

In this chapter the implementation of the framework is explained. It is presented a class diagram which is analyzed and described. The implemented tests are further detailed and the API used to create them is also mentioned and compared with the other API from Unreal Engine.

4.1 Class Diagram

In figure 4.1 we can observe the class diagram which contains the main methods and variables of the implementation of the framework. As it can also be observed, all of the interfaces are connected to the Auto Playtest class, acting like a manager for the classes. This framework is connected with the Unreal Engine by the class "Test Interface"which is where the playtests subsist.

In the class **Playable Area** seen in the class diagram 4.1, two functions can be identified, the GetBoundindBoxesfromWorld and the GeneratePointsinMap. The first function mentioned was implemented in order for the framework to get the bounding boxes from the level and this way detect the playable area. The next function uses the bounding boxes function since it was implemented to generate the valid locations (coordinates x, y, z) of the playable area. It gets the maximum and minimum points of each bounding box and runs the grid sweep algorithm across the bounding boxes. A physical *raycast* function was used. A physical raycast is a function from the engine wherein sum, an invisible line starts at a start point A and goes in some direction infinitely or to an end point B. The idea is that this line collides with anything that has got physics. By having the raycast in a direction from top to down, holes could be detected (because they do not have anything physical associated) and save the locations that the raycast hit, and were not null, as valid locations. The raycast followed a grid pattern to test the majority of the area chosen. If it

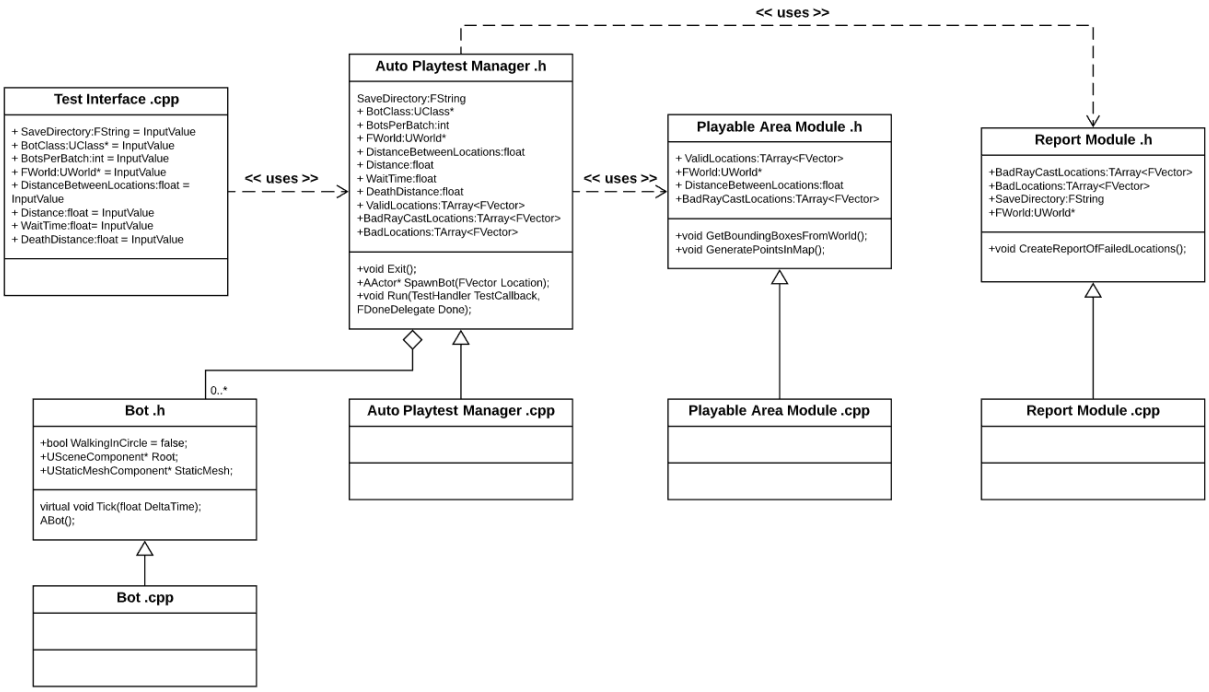

Figure 4.1: Class diagram of the implemented framework.

did not follow this pattern, there was the possibility of having large patches of the area heavily tested, and some thoroughly not verified. The distance between each point was configurable in the TestInterface. In sum, the raycast was able to get the valid locations by guarantying it to be something physical such as a floor, a ramp, or a platform. When colliding with it. Holes detected by the raycast could then be ruled out, with no need to test those locations further and were saved in the BadRaycastLocations array to later mark those locations in the level.

In the **Report Module** class the function CreateReportOfFailedLocations can be seen. This is the function responsible for marking on the map the failed locations and saving the coordinates of these failed locations in a text file. It gets the information of the bad locations from the AutoPlaytestManager and with those it draws the marks on the map and generates the text file. This way the game level designer can see where on the map it failed and if it wants to know the exact coordinates it has that information in the file.

In the class diagram, figure 4.1, it can be observed that the **AutoPlayTestManager** class receives all the information needed to the other modules, and even for itself, it also includes three methods in which two of them are truly relevant. The Exit method is just a method to close the map in case the game level designer does not want to observe the marks on the map and only determine whether the test failed in the automation system interface on the Unreal Engine. The SpawnBot function is a function that spawns bots in a given location, usually the valid locations produced by the playable area module. These bots can be any bot the game level designer desires to observe, and the function is general

enough to accept any type of bots. The Run function is where the logic of the test is implemented, and thus the modules are called in here. This way, the game level designer only has to look at the Test Interface to change the parameters and at the automation system from the Unreal engine to choose the test.

In the same class diagram, figure 4.1, it can be observed a class for a **bot**. This bot was created to help test the framework, and it is a very straightforward bot where its representation is a simple sphere. This bot could have been a human or a robot to represent more of the standard concept of a bot, but unfortunately the computer where the framework was tested could not handle a high degree of complexity for the visual representation.

The boolean WalkingInCircle can be seen in the bot class in figure 4.1. This boolean drives the bot to start walking when it is created; this way, when the bot is spawned, it automatically starts walking in a circle, but only if the playtest changed the boolean value, since it starts with a false. The logic for the bot, meaning the actions that it can or will execute, is implemented in its class. This way, there is an access to an assortment of actions for all the bots. It is only need to call them in the TestInterface since it is within the representation of the tests to call the type of action that the game level designer would want the bot to perform.

4.2 Unreal Engine Automation Test API

The Unreal Engine has two APIs for automated tests. In this chapter, both of them are explained since the implementation started by using one and then it was decided the other one should be used instead. By explaining each one, the decision for the switch can be justified as well as the downsides of using the other API.

4.2.1 Old Automation Test API

The first test was created using the API from the Unreal Engine. While using this API, it was discussed that the API was overly mediocre, and it could not achieve the intended results. In order to explain the problems with this API and why, in the end, it had to be switched, it is explained the elements used to create the automated tests. There are two automation tests: simple and complex. Both types are implemented as derived classes based on the FAutomationTestBase class. Automation Tests are declared by macros, and implemented by overriding virtual functions from the FAutomationTestBase class. Simple Tests are declared using the IMPLEMENT_SIMPLE_AUTOMATION_TEST macro, while Complex Tests require the IMPLEMENT_COMPLEX_AUTOMATION_TEST macro.

One thing that can be observed in figure 4.2 is that there are various sub-tests in one full test. For example, it starts by testing the world; however, the final test checks if it had three cubes in the world, this was the main part to be tested. The problem with this process is that if another test was to be created, it would need to open the map again

```cpp
IMPLEMENT_SIMPLE_AUTOMATION_TEST(FExampleTest, "ExampleTest", FMyTestUtils::TestsFlags)
bool FExampleTest::RunTest(const FString& Parameters)
{
    // 1. Before we do anything the game map must be opened.
    AutomationOpenMap("/Game/StarterContent/Maps/StarterMap.StarterMap");

    // 2. Check if the game World is valid. If not - end this test immediately.
    UWorld* World = FMyTestUtils::GetWorld();
    TestNotNull("Check if World is properly created", World);
    if (!World) return false;

    // 3. Count the number of ACubes.
    int32 CubesCount = 0;
    for (TActorIterator<AActor> It(World, ACube::StaticClass()); It; It++)
    {
        CubesCount++;
    }

    // 4. Test if there are 3 enemy characters.
    TestTrue("Check if there are 3 cubes on the level", CubesCount == 3);

    // 5. Exit this map.
    ADD_LATENT_AUTOMATION_COMMAND(FExitGameCommand);

    // 6. The test has been finished with success.
    return true;
}
```

Figure 4.2: An example of a test using the API from Unreal Engine.

and, once more, test if the world was valid, so it would be taking away precious time by repeating these tasks. If there were more advanced and sophisticated tests then the number of repeated sub-tests would quickly scale up. Another thing is that this API returns the RunTest function as true by default which is not specific enough and does not provide enough information about the results of the various tests inside the RunTest.

In the last part of the code in figure 4.2, it is introduced the latent commands. The latent commands can be queued up during RunTest, causing sections of the code to run across multiple frames. The problem in this API is that they queue at the end of the code, which is not a problem in the shown test, but in a more complicated test such as the ones created for the framework, the latent commands did not execute their job as it was required. In the shown test it is used a latent command to exit the game, the latent command that was needed was one that would force the code to wait a few seconds. A simple test was to spawn a few bots, wait a few seconds before running the next function, and then checking whether any of the bots fell from the world by checking their location; the code for this test ran up sequentially until its last function, and the latent command only waited at the end. Ideally, the latent command "ADD_LATENT_AUTOMATION_COMMAND (FEngineWaitLatentCommand(1.f));"would wait one second to run the next function in the sequence so, after the spawn and before the function that checks the location of the bots. In order for this latent command to work, everything that has to wait for a latent command to finish must be a latent command itself. If every one of the functions had to be changed to a latent command, the test would become extremely complex. Moreover, changing them all to latent commands would take away one of the primary objectives which were to make the framework as general and straightforward as possible for the

```
BEGIN_DEFINE_SPEC(FNewEnemyCountTest, "ExampleNewAPITest", FMyTestUtils::TestsFlags)

UWorld* World;

END_DEFINE_SPEC(FNewEnemyCountTest)

void FNewEnemyCountTest::Define()
{
    BeforeEach([this]()
    {
        AutomationOpenMap("/Game/StarterContent/Maps/StarterMap.StarterMap");
        World = FMyTestUtils::GetWorld();
        TestNotNull("Check if World is properly created", World);
    });

    It("Test Cubes Count", [this]()
    {
        int32 CubesCount = 0;
        for (TActorIterator<AActor> It(World, ACube::StaticClass()); It; ++It)
        {
            CubesCount++;
        }
        TestTrue("Check if there are 3 cubes on the level", CubesCount == 3);
    });

    AfterEach([this]()
    {
        FMyTestUtils::Exit();
    });
}
```

Figure 4.3: An example of the same test but using the new API from Unreal Engine.

game level designer.

4.2.2 Automation Spec API

On the 25th of July of 2019, the Epic team published an official documentation for Automation Spec, the new API that follows the Behaviour Driven Design[29].

In figure 4.3 it can be seen a few new macros that need to be taken into consideration, the BEGIN_DEFINE_SPEC and END_DEFINE_SPEC macros. This is somewhat the same as in the other API, but in between these macros it can be defined important variables common to the tests. These macros allow defining variables as members of the test. Therefore, for every test that is going to be created, they have some common specifications. In the example from figure 4.3, the common thing is the world. Previously in the other API, it had to be ensured for every test that the World variable or any other important variable was available and had to perform that extra check to determine whether it could continue the test or not. Now the game World is obtained before each defined test and is checked using the "TestNotNull"function. If the "BeforeEach"phase fails, the tests will not be performed.

The expectations within the "Define"macro are defined through the use of a few macros such as BeforeEach(), It() and AfterEach(). It() is the code that defines a real expectation for this new API. It() can be called from the root Define() method. In summary, It() can be an individual test to be executed. In the example from figure 4.3, it only has one It(), but it could have had more, and thus these tests could be executed in isolation

```
BEGIN_DEFINE_SPEC(FTestsInterface, "FrameworkTests", AutoPlaytestManager::TestsFlags)

UWorld* FWorld;
AutoPlaytestManager TestRunner;

//Everything that it is common to the test
END_DEFINE_SPEC(FTestsInterface)

void FTestsInterface::Define() {
    Describe("", [this]() {
        BeforeEach([this]()
        {
            AutomationOpenMap("/Game/ElBorak/Maps/Valley/00_Root_2.00_Root_2");
            FWorld = AutoPlaytestManager::GetWorld();
            TestNotNull("Check if World is properly created", FWorld);

            TestRunner.SaveDirectory = FString("C:/UE_Projects/Trial_Test/Saved");
            TestRunner.BotClass = ASimpleBot::StaticClass();
            TestRunner.BotsPerBatch = 60;
            TestRunner.FWorld = FWorld;
            TestRunner.DistanceBetweenLocations = 400.f; //distance between each bot
            TestRunner.Distance = 500.f;
            TestRunner.WaitTime = 10.f; //period of time for each batch

        });
```

Figure 4.4: Framework parameters.

if one of the tests fails while others do not. The way the test are written makes them self-documenting, which is an enormous help in the future for everyone who is picking up this framework.

As mentioned before, there is also the BeforeEach() and AfterEach() macros, both rather self-explanatory. They represent core functions that enable us to run code before the subsequent It() code runs and run the code after the It() code runs, respectively. It can be seen these two functions at work in the same example as before (figure 4.3) where BeforeEach() tests the world for all of the tests (even if in this case there is only one) and AfterEach() closes the world in order to end the test and avoid memory leaks, but only after the test is over.

In the implemented framework, it is also used the latent completion feature. It is used when a test that performs an action that takes multiple frames needs to be written. It can be used the overloaded LatentBeforeEach(), LatentIt(), and LatentAfterEach() members. Each of these members is identical to the non-latent variations, except their lambdas take a simple delegate called Done. When using the latent variations, the Spec test type will not continue execution to the next code block in the test sequence until the actively running latent code block invokes the Done delegate.

4.3 Implemented Tests

Every test implemented has a set of parameters that are common to all tests. The game level designer can configure the parameters for every test since they were implemented using the automation spec API.

In figure 4.4 it can be observed the several parameters that are configurable for the

tests. The game level designer only has to look in this section to configure the test as it wants. This is common to every test. These tests can run from the Console Command Line, either in the Editor or with Command-Line Parameters from the operating system.

4.3.1 Spawning Bots Test

Firstly it was decided to create the most straightforward test that it could be imagined: a test that spawns a batch of bots, waits a certain amount of time and checks if any of the bots fell out of the playable area. After checking the locations the failed positions are saved and the current batch is removed before spawning a new one in order to preserve the performance. It continues with the spawning, waiting, and checking each location until all of the previously saved valid locations are tested. It spawned a batch of bots to optimize the test. If it was only tested one location at a time, it would not be productive. By doing batches,the time and the test can be optimized. The batch can also be adjusted since the performance varies from computer to computer, making the test optimal for each computer used and for every size and scale of the map. The basic test such as this one was created in order to check whether the implemented framework was working correctly before getting into the more complicated and advanced tests. Quality and durability/consistency were chosen over quantity and complexity since starting with the complicated tests would potentially lead to an excessive amount of bugs which would not ensure legibility and scalability.

4.3.2 Bots Moving in a Spiral Test

The spawning bots test is the base test. With this test, it was possible to create more sophisticated tests based on its structure. So when creating other tests, they all followed the spawning bots test. Firstly, they spawn, and in more complex tests, they do a set of movements; the tests then waits for those movements and finally checks the location of the bots. The second test that was implemented followed this basis, and after spawning a batch of bots, they started walking in spirals, getting further away from the original location as time got by. With this test, it could be verified if the bots would fall from the world on their own.

The actions that the test wants the bot to perform are implemented in the bot class. This way, when switching bots for another game, the test can use the bot's actions that are specific for the game since for some games there is usually a set of actions that can be specific to it. For example, in a certain game the player can pull objects, but in another game it cannot, and instead it can run. By having the actions in the player class, the can be used as a bot and test the game with the particular actions for the game.

4.3.3 Configurable Jump Test

This test follows the same principle as the spiral test but introduces a new aspect to the tests, a configuration for the actions performed in the test. Previously, the game level designer could only manipulate parameters that were common to all tests, such as the number of bots in a batch or the size of each cell. This new configuration only affects the jump test where the game level designer chooses the number of left and right jumps for the bot to perform. By creating this test, the framework demonstrates more freedom and flexibility for the game level designer within the tests and show a broader view of the tests.

Chapter 5

Results of the Framework

5.1 Framework Test Results and Analysis

In order to demonstrate the value of this framework its generality and the scalability cannot be the only focus; satisfactory results needs to be shown to determine whether it is an improvement when comparing with manual playtesting. For this reason, two tests were performed where different features were varied in order to understand which features granted the best results. Acquiring the best features allowed it to be generalized to other maps in different scales while having those values in the back of the game level designer's mind. The game level designer also obtains a basis from where they can choose the number of bots to spawn on that level; thus avoiding doing it randomly since that would take away precious time from the tests. Finally, the obtained results can be compared with the manual playtesting and determine whether there was an improvement or if something was gained from this framework.

The first test that was implemented to test the playable area was, as mentioned in the previous section and not counting with the base yest, a bot that walks in a spiral. The playable area is divided, and each bot has a respective area to test. In this test the bots starts at the center of the area, that is called a cell, and starts walking in a circle with increasing radius over time while testing the rest of its cell.

The second test was created to demonstrate some other types of actions that can be used to test the playable area and also to show more complexity in the configuration of a test. The second test is called the "Configurable Jump Test". In summary, the bots jumps to the left or to the right and the game level designer configures how many jumps the bot should perform and to which side the jump should be executed. For example, the game level designer can require the bot to jump two times to the right and three to the left.

These two tests have a common ground, both use bots and test the same playable

Figure 5.1: Player view of the map that is going to be used in testing.

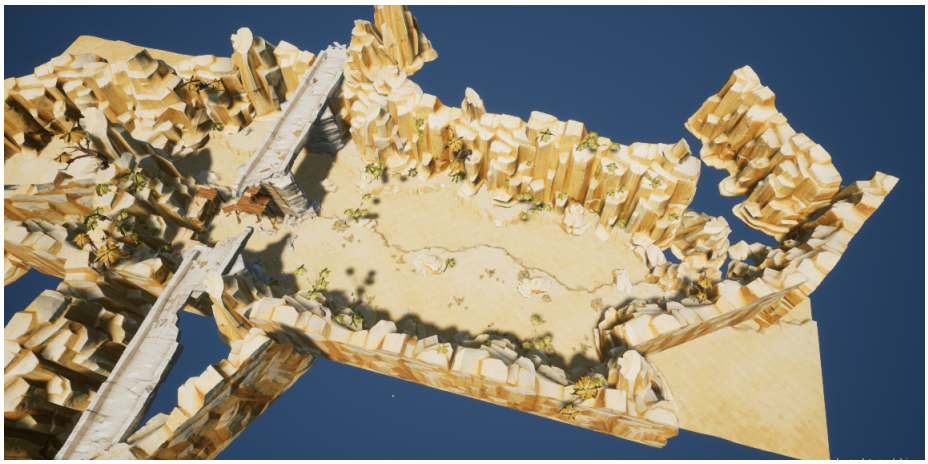

Figure 5.2: Top view of the map that is going to be used in testing.

area. A significant advantage of this framework is the large number of variables available to configure the tests such as the number of bots to be spawned in a batch, the time that these bots have to test their areas, and the distance between the bots. There are other configurable variables, but the ones specified were the ones considered for the tests. The problem arises when the game level designer does not know which value is the best for each variable. In the tests, various values were tested in order to find the best values for these variables. By getting the best result it could then be used to compare with the results of the manual playtesting and have a basis for future maps. The values that were chosen for the default values were spawning 30 bots in each batch, with a distance from each other of 300, and a test period of 5 seconds. These values were chosen by looking at the map and subjectively determining what could work, and they were used as default to determine how the configurable variables would influence the test. Afterwards, it was established the individual best for each of them and joined them in order to determine which combination of the best values brings the most desirable result.

In figure 5.1 and in figure 5.2 it can be observed the map that was tested. This map

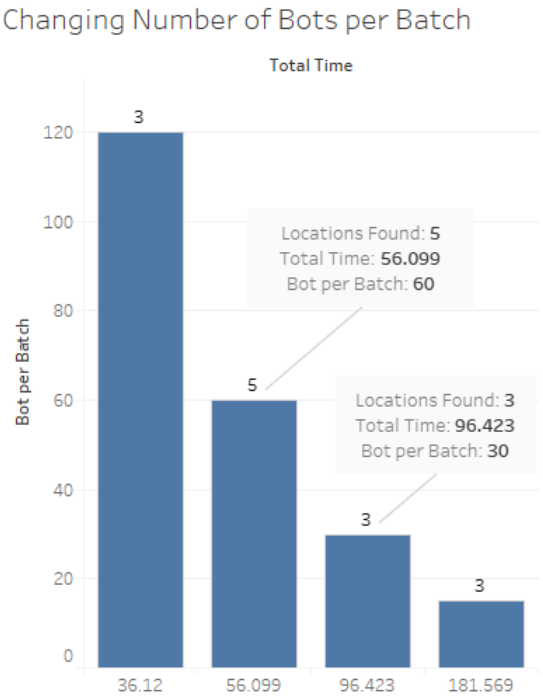

Figure 5.3: Results when running the test and altering the number of bots per batch.

was from a game that was being developed by ZPX. They created a hole in this map to see how the framework would behave and if it would discover the hole that cannot be seen by the human eye.

Before proceeding with the test results, the machine used to run the proposed tests needs to be described. Some tests can have a visually better performance in comparison to other machines. Therefore, these tests were run in a portable computer, with the following specs:

- RAM(Random-access memory): 8GB.
- CPU(central processing unit): Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHZ.
- GPU(graphics processing unit): NVIDIA GeForce GTX 950M.
- VRAM(Video RAM): 2GB GDDR3.

5.1.1 Bots Moving in a Spiral Test Results

As mentioned before, this test consists of a bot moving in circles and testing if it can get out of the playable area. There are different parameters that can be changed, so the number of bots spawned in each batch was the first parameter to be tested. As it can be seen in figure 5.3, it was tested with 15, 30, 60, and 120 bots in a batch, and determined how long it took and how many failed locations it obtained. Firstly, there is only one hole introduced on purpose in this map, so when the results exhibit more than one location

found, it means that there are other locations on the map where the bot cannot only stand but also manage to get out of the playable area. The reason for this is that the bot's initial location was saved instead of the hole's location. It was decided to save the initial location and not the location of the hole since the objective is to understand whether the bot can leave the playable area. Since the test determines the movements of the bot they can be replicated and it can be determined how the bot left even if there is no hole. However, if there is a hole in the map where the bot just falls through, the game level designer can also determine where it came from and the location of the hole. Therefore, in these results, the number of locations failed does not necessarily mean there are three holes, as it can be seen in figure 5.3, it means there were three bots that managed to leave the playable area.

In figure 5.3 it can be observed that by altering the number of bots, the time it took for the test to finish was changed and a slight alteration in the number of locations found. The bar with 30 and 60 bots was marked because the time they took was considered to be a reasonable amount, and both included bots that found the hole. The bar with 120 bots in a batch was not marked even thought the time it took was the lowest one. The reason for this is that when the test is running, it could be seen the bots moving and testing the game, but with 120 bots, it got awfully confusing, with some bots falling from the bridge, or the small cliffs, and colliding with each other. These collisions made the test extremely cluttered, and it could not be observed anything. In the test with 60 bots there were also some collisions which was probably the reason as to why five bots found the hole instead of three like with the other results. However, it was possible to smoothly observe what the bots were doing, and there was a lot less colliding. The reason for the bar with 15 bots to not be marked was because of the time it took. That time for a map of this size is terrible, and the test needs to be the fastest as it can be.

In figure 5.4, can be observed the difference it makes by changing the cell size for each bot. The numbers considered were 150, 300, 400 and 600. The 400 and 300 were marked for the exact reason as the bots per batch test. Both presented results with favorable times and found the hole. It was also possible to understand what happened during the test, and the bots were not excessively colliding with each other. The 800 and 600 cells were not considered since it was a considerably large space for the bots to test, and it was possible to notice they were not testing everything. The 150 cell test was also not marked because it took an exceptionally long time, and the bots were intensely colliding with each other since they did not have space to test.

The last test was devised to experiment with the parameter of time for each batch of bots. The numbers 5, 7, 10, and 14 seconds were the ones chosen. As it can be seen in figure 5.5 the 7 and 10 seconds were marked since it was not considered the time to be awfully large. It was obvious that by increasing the time for each batch the test would naturally take longer. However, when the test was running it was viable to observe that these times seemed to be a quite satisfactory time for the bots to test their cells. The 14 was not marked because it was considered to be an excessive time for the bots; after

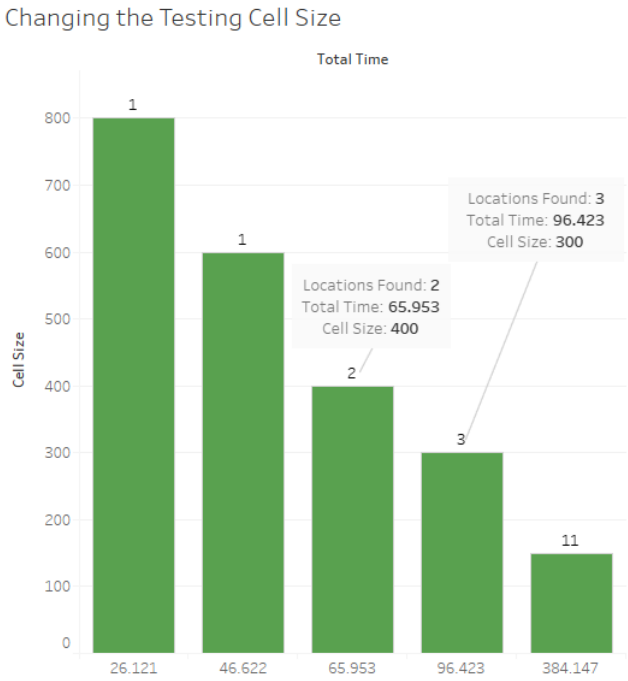

Figure 5.4: Results when running the test and altering the length for the area of each bot.

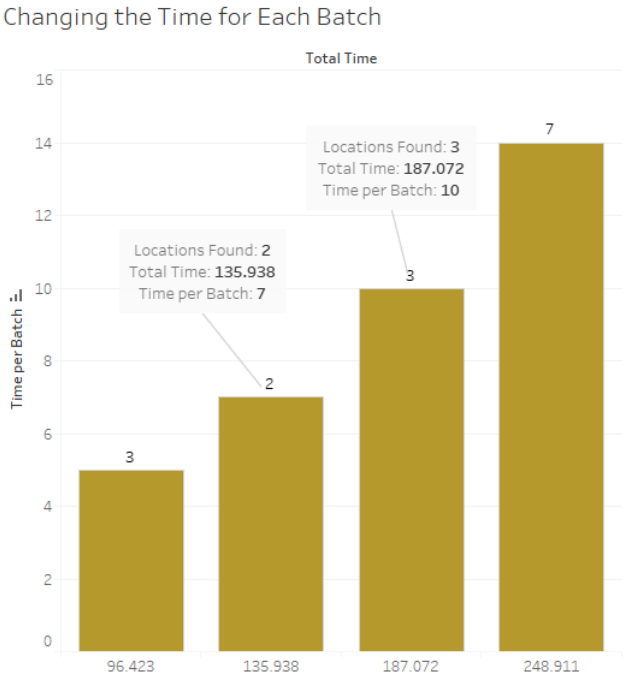

Figure 5.5: Results when running the test and altering the time for the bot to test their area.

Locations Found	Total Time	Bot per Batch	Cell Size	Time per Batch
1	51.28	60,0	400,0	7,0
3	68.407	60,0	400,0	10,0
	128.368	30,0	300,0	7,0
4	90.386	30,0	400,0	7,0
	118.861	30,0	400,0	10,0

Figure 5.6: Results from the test with the best parameters.

14 seconds, the bots were already getting out of their cells and moving to other random places, which was perceived as a waste of time.

In the end, these parameters were tested together to see which set of parameters was the best.

In figure 5.6 can be observed, the several results produced by joining the best parameter of each previous test. In theory, there were supposed to be nine results, but when doing the tests, it was observed that by combining the cell size of 300 with the rest of the parameters it was not getting the expected results. The bots were colliding too much into each other, which was not the intention. Therefore, one of the best results consisted in the one highlighted in figure 5.6 with 30 bots per batch, a cell of size 400 and a period of seven seconds. It was feasible to observe the bots testing the maps without much problem, and in the end, four bots found the hole. There was one more combination of certain parameters that produced satisfactory results while also achieving the best time. The same parameters were considered as the previous combination but altered the number of bots per batch. In this specific map figure 5.1, the bots kept consistently falling off the cliffs or the bridge and colliding with the bots that happened to be on the ground. With a more open map, it should be possible to add more bots per batch.

5.1.2 Configurable Jump Test Results

This test allows for a configurable jump. The game level designer configures how many jumps it requires the bot to perform and to which side. In this case, it was decided to perform two tests. In the first test the bot jumps twice to the right and twice to the left while in the second test the bot jumps twice to the right and once to the left. This way, it can be observed what varies in between tests. It was used the best parameters obtained in the previous section; therefore, thirty bots per batch with a test time for each batch of seven seconds and a cell size of 400.

In figure 5.7 can be observed that a test time of 10 seconds was also considered. It was observed that seven seconds was not enough for the chosen set of actions. Therefore, a more extensive period for the batch had to be considered for the bots to be able to perform their actions. Because of that, there was a significantly higher total time, but since the bots were able to perform all of the actions, they were able to find two failed

Two jumps right and two jumps left

Locations Found	Total Time	Bot per Batch	Cell Size	Time per Batch
1	86.136	30,0	400,0	7,0
2	118.696	30,0	400,0	10,0

Figure 5.7: Results from the configurable jump test with the configuration for two jumps right and two jumps left.

Two jumps right and one jump left

Locations Found	Total Time	Bot per Batch	Cell Size	Time per Batch
2	86.815	30,0	400,0	7,0

Figure 5.8: Results from the configurable jump test with the configuration for two jumps right and one jumps left.

locations.

In figure 5.8 it is presented the results of the test with a configuration of two right jumps and one left jump. It was first tested with seven seconds, and by analyzing the bots it was possible to observe that they had enough time to complete the set of actions and found two failed locations within an acceptable total time per the rest of the results in the spiral and jump tests.

5.2 Manual Playtesting Results and Analysis

For the manual playtesting a group of six volunteers was asked to test the developed level. This group of people consisted of individuals between the ages of 23 and 27 with experience in playtesting within games in the alpha state. The alpha state corresponds to the state when a game is released but can still have a lot of errors since it has not been thoroughly tested by the developer. These games are released to a few selected players in order for them to test it and give feedback to the developers.[7]. This was important because they already have some experience with testing games and trying to exploit games to give feedback to the developers

The group of volunteers was asked to test the level that was provided thoroughly and fill out a form in a report form afterwards. This form delineates a series of questions that start by asking whether the tester found any errors. If any errors were indeed found then they filled out the rest of the form which included a description of each error found, a screenshot of the error itself in the level, an explanation for the place the error occurred at and how it occurred, how long it took them to playtest the level and how long did they take in total which includes playtesting and filling the report. The total time was needed

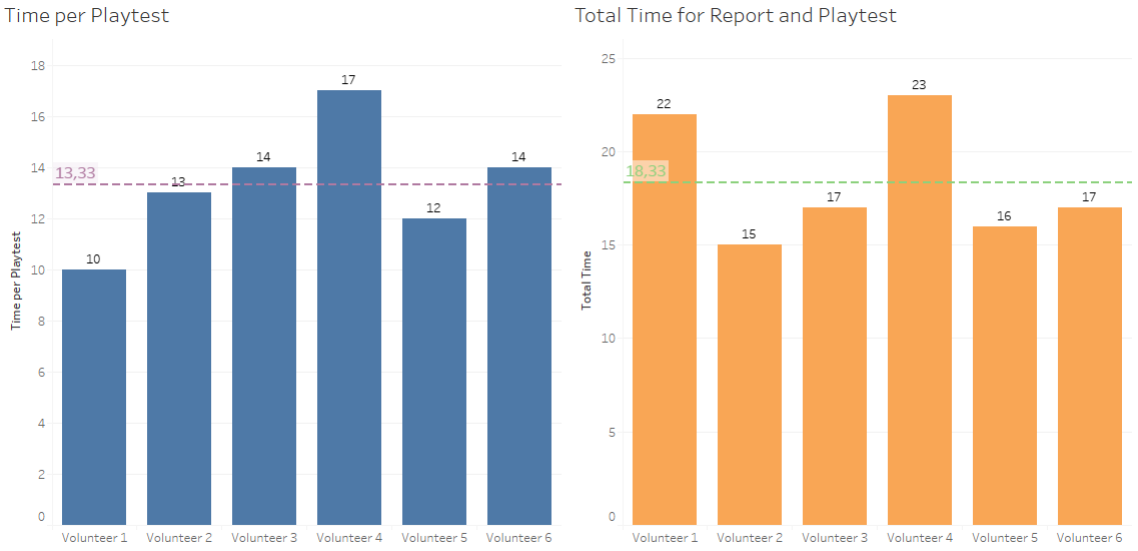

Figure 5.9: Results from the form about how much each person took per playtest and in total.

since it is important to determine how long it takes them to fill the form which can give some insight on how long a game level designer would have to wait for the feedback.

In figure 5.9 it can be seen that the selected group of people took on average 13,33 minutes to complete the playtesting and when added the time taken to fill out the report they took on average 18,33 minutes to complete the process. It can also be observed on 5.9 that some of the volunteers took longer to fill the report, these were the ones who filled the report with more details. All of the volunteers in the group found some sort of error as it can be seen in the answer given in the form in figure 5.13. All of them found the hole hiding in the ground as they wrote in the form *"Holes in-floor"* or *"Mesh collision error (falling through the floor)"*. Four testers found an error with the movement of the player *"The movement error happens in any ledge while holding onto it and making the rolling movement, first occurred on the first ledge towards the bridge."* and three testers found an error with the meshes *"The mesh alignment error occurs in some objects along the floor where the player starts and causes the objects to not align properly, exposing holes in the map."*. Then when describing where these errors occurred the group explained with *"The mesh collision errors happens on the floor and causes the player to fall through it on the right side of the map along the wall."*, *"Right side when you start the game"* or *"On the ground floor at the other side of the map."*. It can also be observed, in figure 5.10, the movement bug they referred to, in figure 5.11 how they were able to screenshot where the hided hole was and in figure 5.12 the mesh error.

It can also be noticed that some answers were more detailed than others such as the ones that only presented the answer as "hole in floor". The detailed ones took longer which means that if the game developer wanted more detail the whole process would take even longer.

Figure 5.10: Screenshot taken by a volunteer to show the movement error

Figure 5.11: Screenshot taken by a volunteer to show the hidden hole

Figure 5.12: Screenshot taken by a volunteer to show the mesh error

Figure 5.13: Results from the form in the question if a person found an error on the level.

5.3 Comparison Between Framework and Manual Playtesting

It was possible to discover the hidden hole in the level both through the use of the framework and through manual playtesting. The difference was that in manual playtesting some testers were also able to discover a bug in the player behavior as well as an error in the meshes which created a little hole in the map. This hole was only noticeable to a person since it was too small for the player to fall of it. The bug exhibited by the player's behavior could have been discovered by the framework if it had been created more tests with a bigger range of movements but the error in the junction of the meshes would unfortunately not have been discovered. The framework was able to run the first test in 90 seconds and the other two ran in 118 seconds and 86 seconds. To run all of the tests in sequence it would be a total of 294 seconds which is approximately 4,9 minutes to run the tests. This time corresponds to the total amount of time taken to run all the tests, receive the report with the locations marked in the level and a file with the coordinates for each error. The manual playtesting took on average 18,3 minutes to playtest and fill a form. The problem with manual testing is that it is not taking into account the professionalism of the testers, if a tester is not attentive enough and does not fill the report with detailed information then the game level designer will also take longer to evaluate and reproduce the errors.

Thus, it can be concluded that the manual playtesting found a bug that the framework was not able to find otherwise, but on the other end the framework took less time and got a report in the exact instant the test finished. This way, it is possible to identify advantages and disadvantages for each way of testing.

Chapter 6

Conclusions

6.1 Conclusions from Results

After testing the framework implemented and comparing it with the manual playtesting it can be safely concluded that the automation of the playtesting brings advantages for the game developer studio as it had improvements both on time and with the report. When creating more tests for the framework the total time will increase however even if it takes longer than the playtesting the features covered by the framework will not need further manual playtesting since it already has an automatic test associated. So it improves the future time for the playesting. Unfortunately, it could not be witnessed the difference and impact the framework could potentially have in the production of an actual game since it takes a lot of time to develop one and there was not enough time to experience the usage of the framework in a real game production. It was tested in a level that the studio developed and it could be observed that the framework had room for improvement and that it already brought advantages to the studio, making it a solid foundation for future work.

Studios can benefit from automated playtesting since they can cut early use of playtesters by implementing their own framework just as this dissertation demonstrated. By generalizing the framework the studio does not need to implement the tests for each new game, they can simply reuse it. Unfortunately, the dissertation does not prove whether the framework would be able to cut the costs since the framework was not used from the beginning to the end of the development of a game, it can only be assumed that by cutting the number of uses of playtesters that there would be less money spent in general but this is only speculation. It can be also said that the use of the framework can take away the repetitive task of testing the level manually over and over again or even the laborious task of implementing the tests for each new game. However, there is still

a need for manual playtesting since there were certain errors that were only found by manual playtesting, though this can be introduced in a later phase.

In the end, the framework was able to achieve the initial objectives which consisted on the optimization of the automated playtesting used in the ZPX Company while maintaining the solution general enough to be used in the different games to be developed. This framework was also modular which makes future additions a lot easier.

6.1.1 Framework Limitation

In the end, this framework still holds some limitations that could not be taken care of with this dissertation. For example, the existence of holes that are too small for the bot to pass through but still visible to the player. These holes are generally made by not connecting the meshes right. This way, the framework cannot detect all of the holes, only the ones that the bot can pass through. Another thing is that these tests only evaluate the physical part of the playable area, but they cannot test the qualitative part of the playable area such as playability, difficulty or even if a player is able to finish the level.

The playable area is limited by the game level designer, and that process would be a lot easier if it was automatic and not by giving another task to the game level designer.

These tests can mark the locations from which a player is able to leave the playable area. However, they do not show if a bot can reach certain areas that are supposed to be accessible.

The last point is that the test fails on the borders of the map, since it tests chunks of the map which correspond to several mini-maps, as it can be seen in the image presented in chapter 5, figure 5.2. The borders for the mini-maps are considered holes since the bots fall through. However, it is supposed to not consider the test to have failed because the bots are supposed to fall through, since those borders are connections to other mini-maps which are just not bound together.

6.2 Future Work

In the future there is some work planned for this framework. A dissertation for the continuation of this framework is already in progress. This one will test some qualitative parts of the game such as the difficulty of a certain part of the level, for example, by analyzing the number of movements that need to be made in sequence.

There will be more tests proposed for the framework and it will be added other types of bots. Automatic detection of the playable area will also be implemented in order to make the game level designer's job even easier.

Bibliography

[1] J. Baker. *Automated Testing at Scale in Sea of Thieves | Unreal Fest Europe 2019 | Unreal Engine*. URL: https://www.youtube.com/watch?v=KmaGxprTUfI.

[2] J. Bycer. *Conversing with Croteam*. 2015. URL: http://game-wisdom.com/guest/croteam.

[3] A. J. Chek Tien Tan. "Towards a Non-Disruptive, Practical and Objective Automated Playtesting Process." In: (2011).

[4] L. Eckardt and S. Robra-bissantz. "Playtesting for a Better Gaming Experience : Importance of an Iterative Design Process for Educational Games." In: Hoblitz 2015 (2018).

[5] M. A. Federoff. "HEURISTICS AND USABILITY GUIDELINES FOR THE CREATION AND EVALUATION OF FUN IN VIDEO GAMES." In: (2002).

[6] S. T. Fundamentals. *Unit Testing*. URL: http://softwaretestingfundamentals.com/unit-testing/.

[7] M. Futter. "What early access can do for you (and what it can't)." In: *The GameDev Business Handbook*. 2017.

[8] Guru99. *Automation Testing Vs. Manual Testing: What's the Difference?* URL: https://www.guru99.com/difference-automated-vs-manual-testing.html.

[9] Guru99. *AUTOMATION TESTING Tutorial: What is, Process, Benefits & Tools*. 2019. URL: https://www.guru99.com/automation-testing.html.

[10] guru99. *What is Regression Testing? Definition, Test Cases (Example)*. 2019. URL: https://www.guru99.com/regression-testing.html.

[11] M. Hawthorne. *TOP 7 GAME BREAKING GLITCHES IN VIDEO GAME HISTORY*. 2017. URL: https://www.gamebyte.com/top-7-game-breaking-glitches-in-video-game-history/.

[12] InformaTech. *GDC-Independent Games Summit*. URL: https://www.gdconf.com/conference/independent-games.

[13] S. Jia and C. Yang. "Teaching software testing based on CDIO." In: *World Transactions on Engineering and Technology Education* 11.4 (2013), pp. 476–479. ISSN: 14462257.

[14] A. Ladavac. *Fast Iteration Tools in the Production of the Talos Principle*. URL: https://www.gdcvault.com/play/1022784/Fast-Iteration-Tools-in-the.

[15] S. Laitinen. "Do usability expert evaluation and test provide novel and useful data for game development?" In: *Journal of Usability Studies* 1.2 (2006), pp. 64–75. ISSN: 1931-3357.

[16] J. Lee. *Learning Unreal Engine Game Development*. Vol. 2016-February. 2016. ISBN: 9781784398156.

[17] R. Masella. "Automated Testing of Gameplay Features in Sea of Thieves." In: (2019). URL: https://www.gdcvault.com/play/1026366/Automated-Testing-of-Gameplay-Features.

[18] J. Matulef. *Serious Sam dev Croteam details PS4 puzzler The Talos Principle*. URL: https://www.eurogamer.net/articles/2014-07-08-serious-sam-dev-croteam-details-ps4-puzzler-the-talos-principle.

[19] P. Mirza-Babaei, N. Moosajee, and B. Drenikow. "Playtesting for indie studios." In: Oct. 2016, pp. 366–374. DOI: 10.1145/2994310.2994364.

[20] L. Offir. "Monopolistic Sleeper: How the Video Gaming Industry Awoke to Realize That Electronic Arts Was Already in Charge." In: (2006).

[21] Packt. *An overview of Unreal Engine*. 2015. URL: https://hub.packtpub.com/overview-unreal-engine/.

[22] R. Parkin. "Software Unit Testing." In: *IV & V Australia, The independent software testing specialists* (1997), pp. 1–4. URL: http://www.ivvaust.com.au/downloads/UnitTesting.pdf.

[23] A. Perry. *Zelda: Breath of the Wild' DLC Glitch: Players escapes the Trials of Sword to find a hidden area*. 2017. URL: https://www.mic.com/articles/181373/zelda-breath-of-the-wild-dlc-glitch-players-escapes-the-trials-of-sword-to-find-a-hidden-area.

[24] "Playtesting 1." In: (), pp. 1–27. URL: http://www.in.tum.de/fileadmin/w00bws/cg/Teaching/WS1819/ComputerGamesLaboratory/playtesting.pdf.

[25] D. Rourke. *Datamoshing the Land of Ooo A Conversation with David OReilly*. 2013. URL: https://rhizome.org/editorial/2013/apr/25/datamoshing-land-ooo-conversation-david-oreilly/.

[26] A. Sanders. *An Introduction to Unreal Engine 4*. 2016. ISBN: 9781315382555.

[27] P. Smyth. *7 Reasons Why Software Development Is So Hard*. 2012. URL: https://www.finextra.com/blogposting/6836/7-reasons-why-software-development-is-so-hard.

[28] SOFTWARETESTINGHELP. *What Is Integration Testing (Tutorial With Integration Testing Example)*. 2019. URL: https://www.softwaretestinghelp.com/what-is-integration-testing/.

[29] C. Solis and X. Wang. "A Study of the Characteristics of Behaviour Driven Development." In: *37th EUROMICRO Conference on Software Engineering and Advanced Applications* (2011), pp. 383–387.

[30] S. T. F. (STF). *Integration Testing*. URL: http://softwaretestingfundamentals.com/integration-testing/.

[31] S. D. Team. "GodTear, Playtest Guide." In: (2018), pp. 1–4.

[32] B. Tyrrel. *Sea of Thieves Review*. ACCESSED= 01/02/2020. URL: https://www.ign.com/articles/2018/03/28/sea-of-thieves-review.

[33] J. Watkins, S. Mills, J. Watkins, and S. Mills. "Regression Testing." In: *Testing It* (2011), pp. 91–98. DOI: 10.1017/cbo9780511997310.013.

[34] Y. Zhao, I. Borovikov, A. Beirami, J. Rupert, C. Somers, J. Harder, F. D. M. Silva, J. Kolen, J. Pinto, R. Pourabolghasem, H. Chaput, J. Pestrak, M. Sardari, L. Lin, N. Aghdaie, K. Zaman, and A. I. Mar. "Winning Isn ' t Everything : Training Human-Like Agents for Playtesting and Game AI." In: (), pp. 1–12. arXiv: arXiv:1903.10545v1.

[35] A. Zook, E. Fruchter, and M. O. Riedl. "Automatic Playtesting for Game Parameter Tuning via Active Learning." In: *Foundations of Digital Games* (2014).