TÉCNICO LISBOA

Deep QL-APF: An Automated Playtesting Framework for Unreal Engine

Gabriel Salvador Fernandes

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Carlos António Roque Martinho
Prof. Rui Filipe Fernandes Prada

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. Carlos António Roque Martinho
Member of the Committee: Prof. Pedro Alexandre Simões dos Santos

November 2021

Acknowledgments

I want to thank my parents, girlfriend, brother and sister in law for the love and support they gave me over all these years and for always motivating me to achieve my goals. Special thanks to Maria Carolina for the help and courage she gave me during the thesis development.

I would also like to acknowledge my dissertation supervisors Prof. Carlos Antonio Roque Martinho and Prof. Rui Filipe Fernandes Prada for the insight and knowledge shared with me, which made this thesis possible. To the Funcom ZPX team, and specially Guilherme Santos and Susana Buinhas, for staying in touch with me during the whole process, helping and pushing me to deliver the best out of me.

Last but not least, to all my friends who stayed by my side during these years, helping me grow as a person. They were always there for me during the good and bad times in my life.

Thank you all.

Abstract

In this work we introduce an approach to automate part of the playtesting process in games made with Unreal Engine 4 (UE4), with the objective of speeding up and reducing the costs associated with manual playtesting. We use the Unreal Automation System to integrate the Deep QL-APF framework with the UE4 in order to perform automated gameplay tests. We propose a Deep Q-Learning (DQL) method for the agent to travel to a destination and achieve a well-defined game objective by trial and error, using feedback from its own actions and experiences. To validate the solution we use a single case study provided by Funcom ZPX.

Three experimental procedures were executed to assess the approach. We obtained results regarding two different agents learning performance and a visual representation of the path they performed. One agent is responsible for reaching the goal as quickly as possible while the other wants to reach the goal while moving close to the map environment walls. The agents can also identify problems in the game environment while they explore it. From the results we confirm that Reinforcement Learning (RL) agents are capable of learning how to achieve a game objective and find problematic areas in a Unreal Engine environment. We also found that the agents performed the behaviours we wanted them to, but, crafting agents to perform the same test and achieve the same game objective with different behaviours was complex and hard to come up with for us. We compared the problems found by agents with the ones found during manual playtesting and concluded that these automated agents can replace human testers when performing these type of exploratory test.

Keywords

Automated Playtesting, Artificial Agent, Reinforcement Learning, Deep Q-Learning, Neural Networks, Unreal Engine, Unreal Automation System, Functional Tests, Actor Component, TensorFlow, Plugin

Resumo

Neste trabalho introduzimos uma abordagem para automatizar parte do processo de testagem de um jogo desenvolvido usando a Unreal Engine 4 (UE4), com o objectivo de acelerar e reduzir os custos associados a realizar unicamente testes manuais.

Utilizamos o Sistema de Automação de Testes da UE4 para integrar a Deep QL-APF com a UE4, de forma a realizar automaticamente testes de jogabilidade. Propomos um método de Deep Q-Learning (DQL) para ensinar o agente a deslocar-se até um destino e alcançar um objectivo de jogo bem definido. Para validar a solução, utilizamos um caso de estudo oferecido pela Funcom ZPX.

Foram realizados três procedimentos experimentais para validar a abordagem. Obtivemos resultados relativamente à performance de aprendizagem de dois agentes diferentes e oferecemos uma representação visual do caminho percorrido por ambos. Um agente é responsável de atingir o objetivo da forma mais rapida possível enquanto que o outro quer atingir o objetivo deslocando-se o mais próximo possível das paredes do mapa. A partir dos resultados, confirmamos que os agentes Reinforcement Learning (RL) são capazes de aprender como alcançar um objectivo de jogo e encontrar áreas problemáticas em ambientes Unreal Engine. Também descobrimos que criar agentes diferentes para atingir um objetivo com comportamentos diferentes é complexo e difícil de realizar. Os agentes são capazes de identificar problemas no ambiente de jogo enquanto o exploram. Comparámos os problemas encontrados pelos agentes com os encontrados durante uma sessão de testes manuais. Concluímos que os agentes automatizados podem substituir pessoas na realização dete tipo de teste.

Palavras Chave

Playtesting Automatizado, Agente Artificial, Aprendizagem por reforço, Deep Q-Learning, Redes Neuronais, Unreal Engine, Sistema de Automação da Unreal Engine, Testes Funcionais, TensorFlow, Plugin

Contents

[1] Sea of Thieves Game: https://www.seaofthieves.com

List of Figures

Acronyms

AI	Artificial Inteligence
IT	Information Technology
QA	Quality Assurance
RL	Reinforcement Learning
SL	Supervised Learning
DQL	Deep Q-Learning
DQN	Deep Q-Networks
DRQN	Deep Recursive Q-Network
CCS	Candy Crush Saga
UE4	Unreal Engine 4
APF	Automated Playtest Framework
MCTS	Monte Carlo Tree Search
ALE	Arcade Learning Environment
FPS	First-Person Shooter
MOBA	Multiplayer Online Battle Arena
LSTM	Long-short Term Memory
ICM	Intrinsic Curiosity Module
GA	Genetic Algorithm
JSON	JavaScript Object Notation
CNN	Convolutional Neural Network

1

Introduction

Contents

1.1 Motivation

Performing playtesting sessions is an important and required step of game design and development. However, it is somewhat expensive in terms of resources and time. It is a very intricate process with numerous iterations where, game developers test their changes, expose their games to the target audience to obtain information about the current state of the game, and where level designers are capable of understanding if the environmental elements that were carefully combined are able to provide the game experience they were designed for, identifying potential design flaws and collecting feedback.

According to the Capgemini World Quality Report 2015, Information Technology (IT) spending on Quality Assurance (QA) and testing has risen to 35% and is expected to continue rising to support today's complex IT operations. The game industry is looking for ways to increase the level of maturity and efficiency of QA and testing while reducing the costs associated with it, using test automation as one possible approach.

The idea behind Automated Playtesting is to use artificial agents that can play the game and achieve the objectives in order to provide meaningful information about the game condition to the designers and developers. It can reduce the costs associated with performing playtesting sessions and contribute to a more stable product. King[1], the creators of Candy Crush Saga (CCS)[2], researched the advantages of Artificial Inteligence (AI) over human-based playtesting [1], using CCS as the case study. By allowing the designers to obtain feedback about the current state of the game before diving into playtesting with human players, they improve their content production pipeline by offering better quality content and more thorough and controlled playtesting.

The existence of success cases in the games industry regarding the usage of automation tests to increase the overall quality of playtesting sessions drove Funcom ZPX[3] to search for ways to automate and improve their playtesting. For this reason, we were motivated in working hand in hand to develop an automated playtesting framework that can be used to reduce the costs associated with the development process of new game content while increasing the overall QA and testing standards.

When developing a 3D adventure and exploration video game, Funcom ZPX identified that it is important to verify if the player can navigate between two points in the game environment. By automating this type of test, we found that it has potential for easing the amount of work that humans perform in testing and QA.

As a case study for this thesis, Funcom ZPX provided an early-stage prototype of a 3D adventure and exploration video game made in Unreal Engine 4 (UE4).

As shown in Figure 1.1, the initial phase of the level consists of a flat surface with a path for the player to climb and reach the top of the bridge. After climbing it, the only path available leads to a dead end

[1] King Ltd: https://www.king.com/
[2] Candy Crush Saga Game: https://king.com/game/candycrush
[3] Funcom ZPX: https://www.funcom.com/funcom-zpx/

Figure 1.1: Initial part of the game environment that is going to be used as a case study.

where the only solution is to climb the green cloth hanging on the left fortress wall. Finally, once inside the fortress, the player is expected to collect a key and open the gate to achieve the game objective and move forward.

1.2 Problem

The playtesting process is, as previously said, a time and resource consuming task that is expected to consume even more resources in the near future. With the objective to ease at least some costs such as time and human resources, we are interested in exploring if artificial intelligence can be used to complement playtesting sessions and provide meaningful feedback to game developers and level designers. Our intention is to use automation tests to verify if the player can navigate through each map module (game environment) and achieve the game objective. In addition to verify traversability, we find important to understand what problems the player may encounter while exploring the game environment to achieve the game objective (at destination).

We can state that this research has four main objectives, which can be divided into two different types of problems. From a **scientific** point of view, we want to deploy an agent in a game environment and make him capable of navigating between two points to achieve game objectives. Besides this, we aim to develop methods capable of detecting different results apart from the baseline solution, by us-

ing strategies that attempt to deviate from the procedure previously executed by the agent. We aim to develop methods capable of detecting if something wrong happened while exploring the game environment. These methods should detect if there are locations in the game environment where the player gets stuck or exits the map boundaries.

From an **engineering** perspective, we want to integrate the scientific contributions of our framework with the Unreal Engine. We will be validating the machine learning integration with a single Unreal project (case study) and explore the possibility of reusing the agents for different game environments. We are interested in facilitating the usage of the framework, giving designers and developers the possibility to adapt it for different contexts, i.e. different Unreal Engine projects.

1.3 Hypothesis

As an approach to the problem described above, we intend to create artificial players that can potentially provide meaningful information about the behaviours that human players might perform while going through a game environment.

Our main objective is to develop an agent capable of navigating between two points in the game environment and perform the actions needed to achieve a well-defined game objective. As an approach to this first problem, we intend to use Reinforcement Learning (RL), specifically Deep Q-Learning (DQL), a machine learning algorithm that employs trial and error to come up with a solution and achieve a specific objective. We hypothesize that by creating a DQL algorithm that can find a solution for the agents to proceed, reach the destination and complete the objective, we can contribute to automatically test a game level made in Unreal Engine. By handcrafting different types of agents that perform different tests on the game environment, we will understand if they can find different ways of achieving a well-defined game objective. The agents will differ from each other regarding the path performed to achieve the objective. By offering a DQL solution for automation tests we are opening new possibilities for testing games with RL.

As an approach to the engineering problems, we want to create an Unreal Engine plugin providing the scientific contributions of the Deep QL-APF Automated Playtesting Framework. It uses an open-source library to help the development and training of machine learning models. This plugin contains collections of code and data that developers can easily enable within the Editor on a per-project basis to perform automated tests. We will use Unreal Automation System to prepare the objectified automated playtesting tests for the case study previously introduced. To improve the framework usability in different contexts, our objective is to allow designers and developers to access and customize the functionalities of the framework through the Blueprint Visual Scripting system in UE4.

1.4 Contributions

Within the scope of the thesis, follows the contributions it offers:

- An Automated Playtesting Framework that can be integrated with Unreal Engine projects and automate part of the testing and QA process in games made with UE4.
- A state of art summary on relevant topics to automated playtesting and to the creation of agents capable of playing games, which have helped arrive at the current solution.
- Artificial intelligent agents that can be deployed in game environments to achieve game objectives using DQL. They are capable of navigating through the environment and verify if the path between two points is traversable, while presenting different behaviours between them.
- The integration with a case study provided by Funcom ZPX. This UE4 project was used to test if the hypothesized solution is viable and if it fits in with the type of content produced by the company.
- Two pre-made tests that developers and designers can use to verify if there is an available path to reach certain locations in the case study game environment, while looking for problematic locations in the environment where the player can get stuck or leave the game environment.
- The evaluation of the Deep QL-APF Automated Playtesting Framework results while considering this specific case study.

1.5 Document Outline

This thesis is organized as follows: In Chapter 1 we introduce the motivation behind the work, the problem raised by it, and the hypothesis based on the research done. The chapter ends with the document's delimitation. In Chapter 2 we give a bit of the background of the areas in which this thesis is operating. We provide an extensive background about the state of art, presenting what is being done to automate the playtesting process and create artificial players that can solve problems in a game environment. In the first section we guide the reader through different research papers that sought to improve the playtesting process in Unreal Engine projects by using automation techniques. This section is followed by an exhibition of research documents that present distinctive approaches to create an agent based on different player models. Through these agents, we can obtain information about players that have the same traits and behaviours. After this, we provide research that aim to deploy agents capable of achieving the game objectives in the best possible way. In the next section, we explore different methods that succeeded in making the agent explore novel states. Finally, we present studies that use machine learning algorithms in Unreal Engine projects. This chapter ends with a discussion about the different

methods, techniques and how the information provided by these studies can be used in our solution to achieve the objectives described in Chapter 1. In Chapter 3 we present our automated playtesting framework architecture, starting by presenting the Deep QL-APF components model, followed by the machine learning library integration with Unreal Engine 4. Finally, we present the Unreal Automation System and the architecture it has in place to execute automated tests in the UE4 editor or project build. In Chapter 4 we present the implementation of our automated playtesting framework. We start this chapter by presenting the algorithm specifications, followed by a preliminary assessment on the Deep Q-Learning algorithm, performed to understand if the algorithm we approached fits our needs and is suitable for achieving the expected contributions. An exposure on how Unreal Engine objects are implemented to establish a communication between the Engine and the Deep Q-Learning algorithm is detailed in the following section. Finally, we demonstrate how the automated tests are created and how their test objectives are defined and implemented. In Chapter 5 we present the results we obtained, the experimental procedures used to test the solution, and a deep analysis on the results. Finally, in Chapter 6 we close this thesis discussing the implications of the work results, presenting a conclusion and some routes in which the work can be continued and improved upon in the future.

2

Related Work

Contents

The video game industry has been showing interest in automating the testing and QA process as a way of reducing the costs associated with testing games. Automated playtesting applications must be implemented in accordance with the game engine to allow the information to flow from the engine to the framework in an understandable way. Since the objective of this research document is to provide an Automated Playtesting Framework for Unreal Engine, we start the state of art by considering one article that explains the playtesting process used in a commercial game made in UE4, as well as a research paper that introduces a framework for automating playtests in UE4, made together with Funcom ZPX.

2.1 Automated Playtesting Frameworks for Unreal Engine

2.1.1 Automated Playtesting in Sea of Thieves[1]

When Rare[2] started working on their latest project Sea of Thieves, they decided to completely change their testing approach. Instead of relying only on manual testing, they used automated testing to check every part of the code, including gameplay features [2, 3]. In Figure 2.1 we present their full QA and testing process.

Sea of Thieves is an open-world game where players can take the role of pirates. The openness of the game world meant that the scope for bugs was very high. There are so many interactions between gameplay features that need to be checked, that manual testing can become a mind-numbingly boring challenge. Instead of having humans spending so much time checking interactions between features, they let the game do it itself via automated tests. As you can see in Figure 2.1, Internal QA Playtesting with manual playtesting is only performed after the developer runs the changes through automated playtests and the game build undergoes an extensive check of automated tests. This way, human testers can focus on recognizing visual and auditory defects, performing exploratory testing to find new and unexpected problems, and checking game experience and assessing how the game feels.

Automated tests can interact with the game at different levels, testing individual code functions if necessary. Rare presented data on the advantages of using automated tests, ensuring that it increased confidence in each game build they launched to be tested by manual testers. With the use of automated tests, they were able to create and verify builds in a day and a half, in contrast with the sports video game developed by them, Kinect Sport Rivals[3], where they took 10 days. They also managed to reduce the number of full-time manual testers from 50 on Kinect Sport Rivals to 17 on Sea of Thieves. Another advantage was the reduction of bugs over the months of development. The maximum bug count that they achieved on Sea of Thieves was 214, whereas on Banjo Kazooie Nuts and Bolts[4] was nearly 3000,

[1] Sea of Thieves Game: https://www.seaofthieves.com
[2] Rare - Xbox Game Studios: https://www.rare.co.uk/
[3] Kinect Sport Rivals: https://www.rare.co.uk/games/kinect-sports-rivals
[4] Banjo Kazooie Nuts and Blots: Rare - Microsoft Game Studios, 2008

although one can always question the relevance of the comparison because they are totally different games. Due to the simple fact of having an automated test system, they knew immediately when a known problem had entered the build, managing to fix it early.

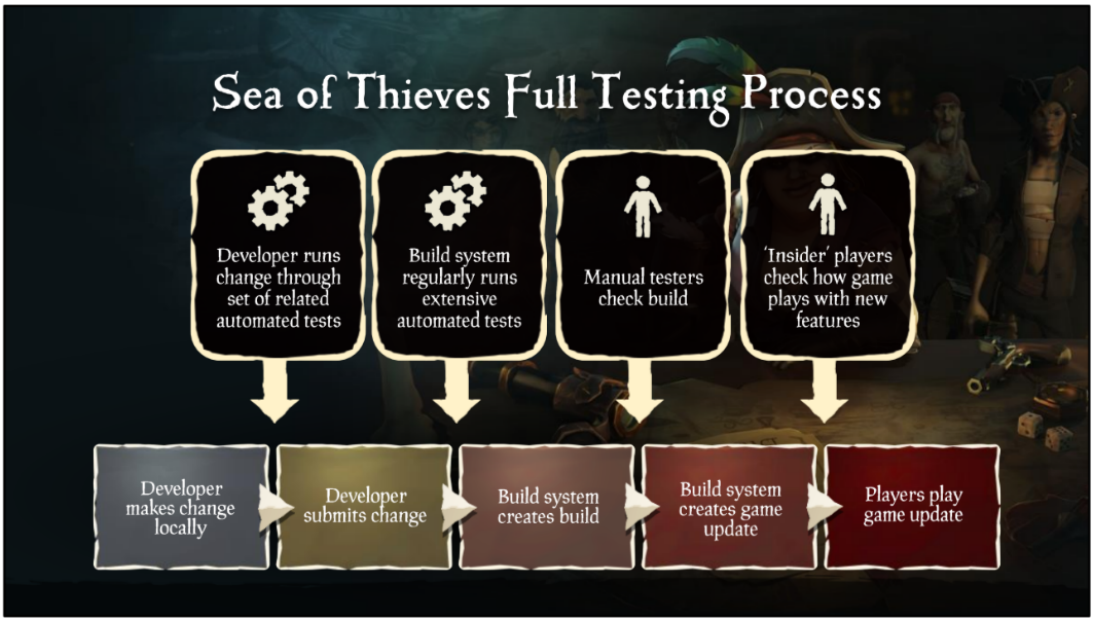

Figure 2.1: Full testing process in Sea of Thieves.

Figure 2.2 shows the amount of each type of test and its percentage in relation to the total of tests done. They only created a relatively small amount of performance[5], screenshot[6] and bootflow[7] tests. These were by far the slowest, so they used these as sparingly as possible.

The unit tests were used for the logic of the program, to check a specific operation on the smallest testable part of the code, which generally means testing at the code function level. In Appendix A, Figure A.1, Rare presents an example of a unit test that checks if a calculate distance function works as expected. First, they set up the data or objects that they want to test, next they run the operation that they're testing and finally assert if the results are what they expect. If the assertion fails, an error will be logged by the automation system and mark the test as failed. Unit tests can be used to cover every individual function of the game, and in theory build up testing that covers all the game. However, sometimes the way units interact can itself contain bugs, so alongside the unit tests they also created integration tests that generally cover a whole feature or action in the game.

[5] Performance tests collect data from the game to spot trends in load times, memory usage or frame rate.
[6] Screenshot tests output screenshots for use in a visual comparison against last good images, to check for unforeseen changes in the game visuals.
[7] Bootflow tests check the communications between client, server and game services.

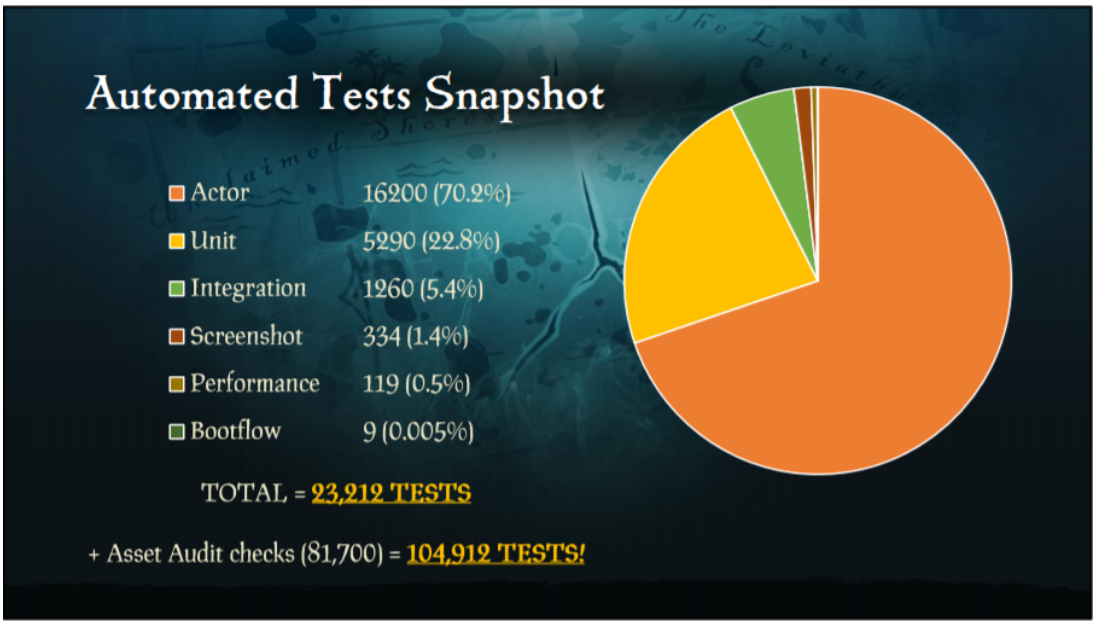

Figure 2.2: The amount and percentage of each type of test in relation to the total of tests done.

Integration tests in Sea of Thieves were created as maps in the Unreal editor. Each map would test a specific game behaviour in a fixed scenario and report back its results as a pass or fail, based on whether the behaviour happened as expected or not. To create the logic of what happens in these integration tests, they made use of the Unreal Blueprint system, a node based-scripting system within the Unreal engine.

Blueprints are very convenient for running integration tests since nodes can be latent and pause execution until a certain condition has occurred. As an example of an integration test for Sea of Thieves, they present a test for one of the most basic activities in-game, which is checking if when the player interacts and turns the wheel, its angle changes. Instead of loading a full client of Sea of Thieves, connect to a server, and place the player on a ship and let them turn a wheel, they narrow things down to test only the actual interaction between the player and the wheel. Therefore, they have a player, a platform for the player to stand on and a ship's wheel. As you can see in Figure 2.3 the logic for the test follows the three test stages in a similar way to the unit test: first they do the setup, then run the operation they want to test and finally check if the results are the expected.

Integration tests require a built version of the game or editor, which in turn relies on the whole Unreal engine. This inevitably means that they will be slower and less reliable. This was a particular problem for their gameplay code, which was built using common Unreal engine classes, like actors and components. When this started to become unsustainable they created a new type of test called 'actor tests', for this reason 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 fact a unit test for game

code, but one that tested Unreal engine concepts like actors and components.

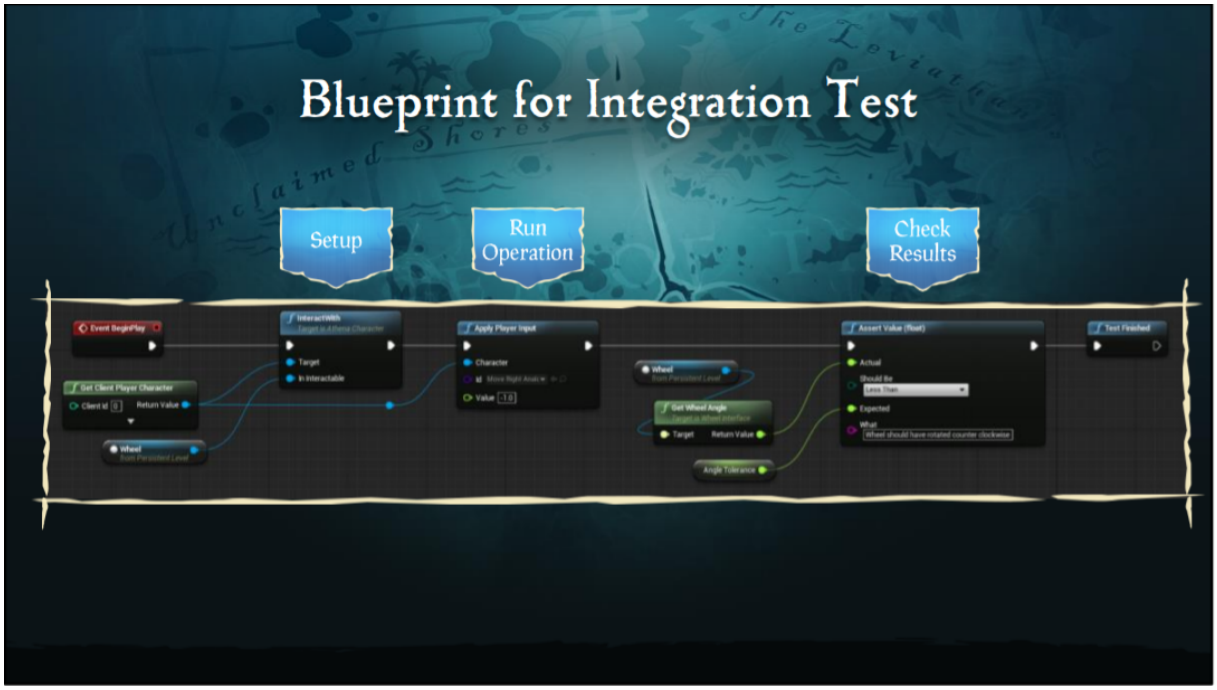

Figure 2.3: Integration test coded with Unreal Blueprint node based-scripting to check if an expected behaviour happened.

As an example of an actor test, see Figure A.2 in Appendix A, where they present the test for the shadow skeleton. The shadow skeleton has two states, dark and light, with different visuals. Its current state is based on whether it is currently exposed to light or not. When in darkness, it is virtually invulnerable, whereas in light, it has the same vulnerabilities as a normal skeleton. They created a test that will check that a shadow skeleton shifts from its dark state to its light state when the time of day changes. They could have done this with an integration test but doing it as an actor test is much more efficient.

As you can see in Figure 2.2, 70.2% of their tests are actor tests, which makes them the most common type by far. This was because it was the most convenient test type to provide coverage for their gameplay features. Only 5.4% of their tests were integration tests, but they provided vital high-level coverage of game features. This information is important for confirming that functional tests can be used in the context of this thesis to cover high-level features, such as validating that the game environment is in a good state for the player to play in, while checking if it presents an available path for the game objective.

2.1.2 Automated Playtesting In Videogames

P. Negrão [4] implemented a playtesting framework for UE4 that tests the integrity of the level and determines whether it is possible to exploit the game in some way. This dissertation was made in partnership with Funcom ZPX and the case study is the same as the one we will use to experiment and evaluate our playtesting framework.

The out of boundaries glitch was chosen as the focus for P. Negrão framework, making the designed and implemented tests specific for this issue. It refers to going beyond the normal boundaries of an area on the game map, by passing through walls or jumping off objects to get out of the map. 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, it could test if there were any holes or ways to exploit the game in order to get out of the playable area.

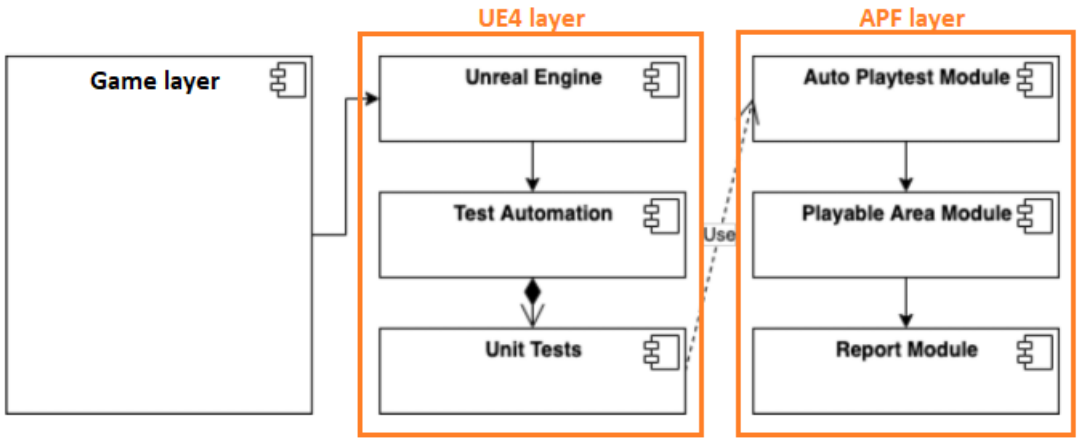

Figure 2.4: Components diagram of the implemented framework [4]. It is divided in 3 layers, the Game Layer, the UE4 layer and the APF layer, presenting how these layers interact with each other.

The architecture of the solution is composed by three main layers, the Game layer, the UE4 layer, and the Automated Playtest Framework (APF) layer. In Figure 2.4, we present a component diagram showing how these layers interact with each other. Firstly, the Game layer represents the game when running. 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 and integration tests. The APF is integrated with the UE4 layer at the level of the component “Unit Tests” which can be a unit test or an integration test. These tests define the game mechanics that the APF should execute in each section of the playable area. The way the APF was integrated made it independent of each project because the implementation of the game mechanics is made inside UE4 and therefore dissociated from the APF. This approach made it possible for the APF to be flexible and easier to integrate new forms of testing with the addition of new unit/integration tests.

The Auto Playtest Module controls the logic of the program, and all modules are connected to it. 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. The Playable Area Module is responsible for dividing the level area and check what locations are valid. The report module creates a report including the locations where the test failed. This way, the game level designer can use the information to test further or correct the problem.

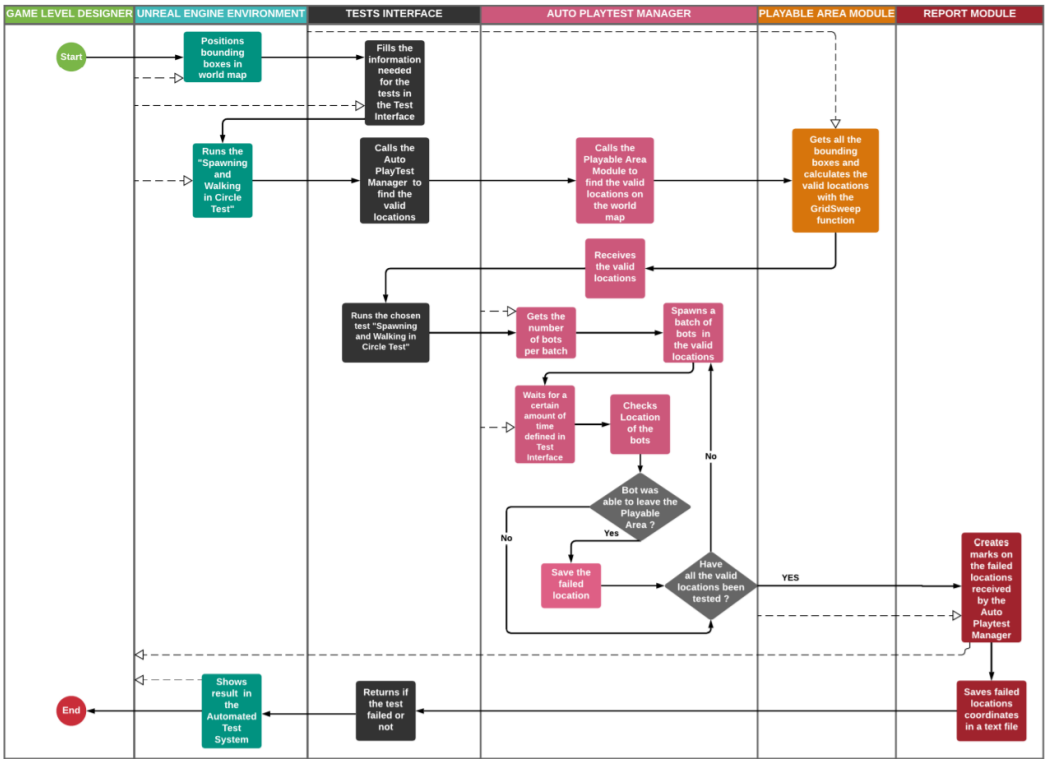

Figure 2.5: Activity diagram of the framework [4]. Demonstrates the activities in the framework when the game level designer chooses a test to perform.

The first step is for the game level designer to position the bounding boxes in a way that covers all the playable area. After, it is expected to configure the variables of the test in the Test Interface and run the test in the Unreal Automation System. Next, the manager class receives the valid locations, gets the type of test to run, and the variables needed. It spawns a batch of agents and waits for a determined period of time. Later, it checks the location of each agent and saves any failed locations. After testing all valid locations, it sends the arrays with the failed locations found by the test to the report module. After this, the Test Interface sends the result of the test. If a location with a functional problem is found, the test appears as failed in the automation system, ending the work of the framework.

After testing the framework with the case study and comparing it with manual playtesting, P.Negrão concluded that automating the playtesting process brings advantages for Funcom ZPX. The selected group of people that performed manual playtesting 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. The framework was able to run all the tests in sequence and create a report in approximately 4,9 minutes, a significant reduction of time. Although both human players and artificial agents were capable of finding the bugs in the case study, all the team felt improvements both on time and information analysis with the reports produced by the framework. From this work we got another confirmation that offering an automated playtesting framework to automatically execute a type of Functional Test displays positive results and is useful to the game development pipeline.

2.2 Using Player Modeling in Automated Playtesting

Our thesis intention is to contribute to playtesting sessions using automated tests. We are considering a 3D adventure and exploration game as case study, where it is interesting to understand what are the different ways to explore the environment. Therefore, during our state of art research, we also focus on research papers that use automated tests to understand how different types of players act when in contact with a game environment. Player modeling is the learning and use of computational models of player preference, experience and/or behaviour. The way the player interacts with the game depends on his **personality** and **gameplay skill**, and for that reason we present papers that intend to model the player personality using procedural personas and the player skill through sensory limitations.

2.2.1 Automated Playtesting with Procedural Personas

C. Holmgård et al. [5] present a method for modeling player decision making, using agents as AI-driven personas. The paper argues that artificial agents, as generative player models, have properties that allow them to be used as psychometrically valid, abstract simulations of a human player's internal decision making processes. They assume that players exhibit a particular decision making tendency or style when playing a particular level or game, and that this tendency can be captured and expressed by approximating a utility function that shapes their decisions in-game. The game environment, MiniDungeons 2 [6] is a turn-based puzzle game where a hero travels from the entrance of a dungeon level to the exit. They use RL, specifically Q-learning where the action-choosing module uses a utility function that maps states and actions to a numeric value (the utility). The small game world and limited number of hero moves in each level position permit the use of a lookup table for storing state action pairs. The reward function of the Q-learning agent is simply the model of the player's utility function. In order to produce multiple different personas for comparisons with players, a number of distinct agents were developed which had different playing styles. Two examples of agents that were crafted are the Runner and the Munster Hunter. The Runner has the primary objective of finding the exit in the fewest moves possible, while the Monster Hunter has the primary objective of killing as many monsters as possible with the secondary objective of finding the exit. Their conclusion was that the agents were useful as per-

sonas for characterizing and discriminating between the human players. While the Q-learning agents demonstrated to work well, the training of the agents is computationally demanding and hence time consuming. A better approach is using a generic trained agent, whose policy is not tied to a particular level. Possible approaches could include using agents based on Q-learning with neural networks, Monte Carlo Tree Search (MCTS) or evolutionary rule-based systems. They presented another paper [7] where they compare an evolutionary (evolutionary algorithm) persona representation with the previously devised method of Q-learning. They found that the evolutionary solution is better both at agreeing with human players and optimizing the rewards, while also being generalized to unseen levels. However, we will use RL in the context of this thesis because it is the machine learning technique we want to assess if it can be used for automated testing. This work provides really useful information on how to craft utility functions for RL in order to develop an agent that executes a specific behavior, such as the Runner and Monster Hunter. It was important for us to understand how these utility functions were crafted so that we can create agents with different behaviours for exploring the game environment.

Following this study, Holmgard et al. [8] presented another paper describing a method for generative player modeling and its application to the automatic testing of game content using procedural personas. They found that using archetypal generative player models (procedural personas) to playtest automatically with certain playstyles (Monster Killer, Runner, etc..) enables the understanding of how different players interact with different aspects of the game. They used the same case study (MiniDungeons 2), a game that is designed specifically to have a high decision density, meaning that every action matters, while requiring little to no manual skill. To control the personas, they used a variant of MCTS which is well-suited for building biased search trees in large search spaces. MCTS is a tree search algorithm that creates biased search trees for decision processes, focusing on exploiting the most promising moves to expand next, while balancing that by exploring more neglected branches of the tree. For their purposes, MCTS has several desirable properties which approximate how decision making occurs in humans: it evaluates the next best action based on a utility score for a predicted future state and operates under uncertainty of future outcomes. They found that the evolved personas differentiate their play styles, and in most maps, perform better than other personas with regard to their core priority. This work is relevant to confirm that automated testing using crafted agents is useful, even though it uses a totally different game environment and different algorithm from the one we use.

Mugrai, Holmgard et al. [9] also developed a method for performing automated playtesting with procedural personas for a match-3 tile game by evolving the utility function for the MCTS agent. By being able to model human players and play styles, they open the possibility of playtesting new levels, analyzing the approaches and how players play various levels for Match-3 games. Game designers gain further insights on various interaction patterns and study how various categories of players would respond within the Match-3 genre. Their objective was to develop four procedural personas, which model four different

types of play styles: trying to maximize score, trying to minimize score, trying to maximize the available number of moves and trying to minimize the available number of moves. By deploying these agents into real world Match-3 games, it opens up the ability to analyze level designs and the approaches taken to play levels by various player perspectives. Their score maximizing and score minimizing agents allowed them to evaluate and estimate the range of performance for human players. Also, comparing the performance of such agents across multiple boards aided in measuring what can be perceived as their difficulty levels. Just like the last work that was presented in this section, they confirm that procedural personas are really useful for performing automated playtesting, although they don't use a case study or algorithm similar to ours.

2.2.2 Modeling Sensorial and Actuation Limitations in Artificial Players

In this master thesis, A.Soares [10] took the developments in the deep RL field applied to Atari environments, mainly Deep Q-Networks (DQN), and modulated different types of player limitations. The way a player interacts with a game can in itself be different from how another one does it and that might prompt different players to develop different strategies on how to play the same game. There are different scenarios, for instance, delays, lag and action limitations. A metric of example can be the Actions per Minute. Strategies applied by professional players will be different from casual players, since they are able to act and react faster.

They present various types of manipulations tested, with multiple interactions between themselves. By using Action Change, they are able of limiting implementations based on the principle of having an error probability associated with the execution of a certain type of action. For the Action Delay limitation, they replicate a delay in the execution of an action by the player. Finally, they use Miss Frame to simulate a visual type of limitation a player might have, or a simple visual miscue. The results seem to indicate the existence of different types of playing patterns for different limitations, although results are merely theoretical, no user testing was made. However, they point out that these strategies would be visible in real human players with real playing limitations. This work confirms that creating RL agents to simulate behaviours and getting results from them is useful and can contribute to understanding how players might interact with the game.

2.3 Developing artificial players to achieve game objectives

Automatically playtesting the game implies artificially controlling the player character to play the game. While the main goal of these research papers is not to provide playtesting feedback to the designers and developers, they can contribute to automated playtesting frameworks by presenting solutions on how to create agents that can learn how to act in-game and achieve game objectives.

V. Mnih et al. [11] offered the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning that combines stochastic minibatch updates with experience replay memory to ease the training of deep networks for RL. The network takes as input raw pixels and the output layer is a fully connected linear layer with a single output for each valid action. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network. They apply their method to seven Atari 2600 games from the Arcade Learning Environment (ALE) and demonstrated its ability to master difficult control policies for them. They use the same network architecture, learning algorithm and hyperparameters settings across all seven games, showing that their approach is robust enough to work on a variety of games without incorporating game-specific information. The network was not provided with any game-specific information or hand-designed visual features and was not privy to the internal state of the emulator; it learned from nothing but the video input, the reward and terminal signals, and the set of possible actions - just as a human player would. They show that their method achieves better performance than an expert human player on three games and achieves close to human performance on one. Three games are far from human performance, since they are more challenging and require the network to find a strategy that extends over long time scales. This work clearly demonstrates that RL can be used to train agents how to achieve a specific objective in a game environment and we used exactly the same algorithm presented in this work during the implementation of this thesis.

Silver et al. [12] introduced AlphaGo[8], a new approach to the game Go[9] that uses "value networks" to evaluate board positions and "policy networks" to select moves. The value network assigns value/score to the state of the game by calculating an expected cumulative score for the current state, and the policy network knows which actions should be performed at the current state to get maximum reward. These deep neural networks are trained by a combination of supervised learning from human expert games, and RL from games of self-play. Without any lookahead search, the neural networks play Go with MCTS programs that simulate thousands of random games of self-play. They pass in the board position as a 19x19 image and use convolutional layers to construct a representation of the position. They use these neural networks to reduce the depth and breadth of the search tree: evaluating positions using a value network, and sampling actions using a policy network. They train the neural networks using a pipeline consisting of several stages of machine learning beginning by training a Supervised Learning (SL) policy network directly from expert human moves. Next, they train a RL policy network that improves the SL policy network by optimizing the outcome of games of self-play. This adjusts the policy towards the correct goal of winning games, rather than maximizing predictive accuracy. Finally, they train a value network that predicts the winner of games played by the RL policy network against itself. Consequently,

[8] DeepMind AlphaGo: https://deepmind.com/research/case-studies/alphago-the-story-so-far
[9] Go is an abstract strategy board game for two players.

their program AlphaGo efficiently combines the policy and value networks with MCTS. Using this search algorithm, AlphaGo achieved a 99.8% winning rate against other Go programs and defeated the human European Go champion by 5 games to 0. Moreover, in [13] they introduce an algorithm based solely on RL, without human data, guidance, or domain knowledge beyond game rules. It uses only the black and white stones from the board as input features and uses a single neural network, rather than separate policy and value networks. It uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts. Their results comprehensively demonstrate that a pure RL approach is fully feasible. Even in the most challenging of domains it is possible to train to superhuman level, without human examples or guidance, given no knowledge of the domain beyond basic rules. Using this approach, AlphaGo Zero defeated the strongest previous versions of AlphaGo, which were trained from human data using handcrafted features, by a large margin, winning 100-0.

G.Lample and D.Chaplot [14] offer an architecture to tackle 3D environments in first-person shooter games that involve partially observable states. Doom[10] is a classical First-Person Shooter (FPS) game, and ViZDoom was presented as a testbed for deep RL. Agents learn from raw pixels and interact with the ViZDoom environment in a first-person perspective. They introduce a method for co-training a DQN with game features, which turned out to be critical in guiding the convolutional layers of the network to detect enemies. The agent divides the problem into two phases: navigation (exploring the map to collect items and find enemies) and action (fighting enemies when they are observed) and uses separate networks for each phase of the game. This makes the architecture modular and allows different models to be trained and tested independently for each phase. Both networks can be trained in parallel, which makes the training much faster as compared to training a single network for the whole task. Furthermore, the navigation phase only requires three actions (move forward, turn left and turn right), which dramatically reduces the number of state-action pairs required to learn the Q-function. They used a Deep Recursive Q-Network (DRQN) augmented with game features for the action network, and a simple DQN for the navigation network. Therefore, at each step, the network receives a frame, as well as a Boolean value for each entity, indicating whether this entity appears in the frame or not (an entity can be an enemy, a health pack, a weapon, ammo, etc). During the evaluation, the action network is called at each step. If no enemies are detected in the current frame, or if the agent does not have any ammo left, the navigation network is called to decide the next action. Otherwise, the decision is given to the action network. They conclude that the proposed model is able to outperform built-in bots as well as human players and demonstrated the generalizability of their model to unknown maps.

One of the most remarkable works using DQL in Multiplayer Online Battle Arena (MOBA) was proposed by OpenAI. Their results prove that deep RL method can be successful in a 5v5 Dota2 sce-

[10] Doom: id Software, 1993.

nario [15]. By defeating the Dota 2 world champion, OpenAI Five demonstrates that self-play RL can achieve superhuman performance on a difficult task. They define a policy as a function from the history of observations to a probability distribution over actions, which they parameterize as a recurrent neural network. The neural network consists primarily of a single-layer 4096-unit Long-short Term Memory (LSTM). Given a policy, they play games by repeatedly passing the current observation as input and sampling an action from the output distribution at each timestep. Their objective is to find a policy that maximizes the probability of winning the game against professional human experts. In practice, they maximize a reward function which includes signals such as characters dying, collecting resources, etc. This reward model is complex and well-structured, rewarding the agent for a set of actions which humans playing the game generally agree to be good. Although the Dota 2 engine runs at 30 frames per second, OpenAI Five acts on every 4th frame which they call a timestep. Each timestep, OpenAI Five receives an observation from the game engine encoding all the information a human player would see such as units' health, position, etc. OpenAI Five then returns a discrete action to the game engine, encoding a desired movement or attack. Certain game mechanics are controlled by hand-scripted logic rather than the policy. Instead of using the pixels on the screen, they approximate the information available to a human player in a set of data arrays. OpenAI Five uses this semantic observation space for two reasons: first, because its goal is to study strategic planning and high-level decision making rather than focus on visual processing. Second, it is infeasible to render each frame to pixels in all training games; this would multiply the computation resources required for the project many-fold. After ten months of training, it defeated the Dota 2 world champions in a best-of-three match and 99.4% of human players during a multi-day online showcase.

The works presented in this section confirm that we can use RL in a complex 3D game environment, while offering valid examples on how to craft a reward system and gather observations from a 3D game environment.

2.4 Curiosity/Novelty Search

The papers presented above are concerned in finding an optimal solution by obtaining a policy that maximizes the probability of winning a game. However, we are interested in obtaining several solutions and not only the best possible solution. We want to understand which are the possible paths for the agent to explore and achieve a game goal, and therefore we are looking for a method that explores the solutions space, noticing that Novelty and Curiosity-Learning is being used to encourage the agent to explore "novel" states. In addition to this, it is also used to encourage the agent to perform actions that reduce the error/uncertainty in the agent's ability to predict the consequence of its own actions. D. Pathak et al. [16], propose curiosity as an intrinsic reward signal to make the agent explore the environment and

learn skills that might be useful later in its life. They use this method to understand if curiosity-driven exploration can be used when rewards extrinsic to the agent are extremely sparse, or absent altogether. They formulate curiosity as the error in an agent's ability to predict the consequence of its own actions in a visual feature space learned by a self-supervised inverse dynamics model. Their agent is composed of two subsystems: a reward generator that outputs a curiosity-driven intrinsic reward signal, named Intrinsic Curiosity Module (ICM), and a policy that outputs a sequence of actions to maximize that reward signal. In addition to intrinsic rewards, the agent optionally may also receive some extrinsic reward from the environment. To evaluate their curiosity module on its ability to improve exploration and provide generalization to novel scenarios, the proposed approach is evaluated in two environments: VizDoom and Super Mario Bros. In VizDoom the agent explores a much larger state space and learns the exploration behaviour of moving along corridors and across rooms without any rewards from the environment. The intelligent walking behaviour learned by the curious VizDoom agent also transfers to completely new maps with new textures. In Mario the agent crosses a significant portion of Level-1 without any rewards from the game and helps the agent explore subsequent levels faster. E. Jackson and D. Daley [17] introduce and evaluate the use of novelty search over agent action sequences as means for promoting innovation. They also introduce a method for stagnation detection and population re-sampling inspired by recent developments in the RL community that uses the same mechanisms as novelty search to promote and develop innovative policies. Their methods extend a state-of-the-art method for deep neuroevolution using a simple-yet-effective Genetic Algorithm (GA) designed to efficiently learn deep RL policy network weights. "Can the history of actions performed by agents be used to promote innovative behaviour in benchmark RL problems?". Towards answering this, they implemented two novel methods for incorporating behavioural history in an evolutionary algorithm designed to effectively train deep RL networks. The first method is an implementation of novelty search in which, during training, the reward signal is completely substituted by a novelty score based on the Levenshtein distance [11] between sequences of game actions. The second method is not a novelty search, but rather a modification to the base GA that incorporates elements of novelty search to avoid population convergence to locally optimal behaviours. Using these two sets of experiments, they evaluated each method's effectiveness for learning RL policies for four Atari 2600 games. They found that while Method I is less effective than the Base GA for learning high-scoring policies, it returns policies that are behaviourally distinct. Method II was more effective than Method I for learning high-scoring policies. Method I experiment suggest that novelty search indeed creates selection pressure for innovation. Results demonstrate that novelty search over action sequences is an effective source of selection pressure that can be integrated into existing evolutionary algorithms for deep RL.

[11] The Levenshtein distance is a string metric for measuring the difference between two sequences, by counting the minimum number of operations required to transform one into the other.

2.5 Implementing machine learning algorithms for Unreal projects

The playtesting platform we want to deliver is dependent on a communication between the UE4 and a machine learning library. On one side, we need a library that allows us to implement a Deep Q-Learning algorithm that trains agents to achieve an objective based on feedback from his previous actions. On the other end, we need to execute the actions that were chosen on each step by the Deep Q-Learning algorithm in the Unreal Engine, so that the system controls the character during gameplay.

In this section we present a plugin that allows the usage of TensorFlow, an end-to-end open source platform for machine learning that enables training and implementing state of the art machine learning algorithms for Unreal Engine projects. Finally, we present a plugin that simplifies setting up a Deep Q-Learning system with agents in UE4.

2.5.1 Unreal Engine plugin for TensorFlow

Tensorflow-ue4 [19] is a Unreal Engine plugin for TensorFlow that enables training and implementing state of the art machine learning algorithms for Unreal Engine projects. It contains C++, Blueprint and python scripts that encapsulate TensorFlow operations as an Unreal Engine Actor Component, a special type of object that UE4 Actors[12] can attach to themselves as sub-objects. This plugin implies using the Tensorflow Actor Component as a Blueprint API to create a bridge between the Unreal Engine Blueprint System and a python code file (PythonAPI) that implements a machine learning algorithm using TensorFlow. It conveniently simplifies the communication between these two by making a component inside of the Blueprint scripting system that we can use to send and receive data from TensorFlow.

The plugin supports input/output from UE4 via JavaScript Object Notation (JSON) encoding. JSON is a lightweight data-interchange format that is easy for machines to parse and generate. An Actor using the TensorFlow Actor Component is in direct contact with the functions written in the python code file, changing information encoded in JSON back and forth. There is control on what type of data is forwarded to the python module. Figure 2.6 shows highlighted the parts of the architecture where should be handled how and what messages travel from the Deep Q-Learning system code to Blueprint code and vice-versa.

[12] An Actor is any object that can be placed into a level, such as a Camera, static mesh, or an AI Controller. They can be created (spawned) and destroyed through gameplay code (C++ or Blueprints).

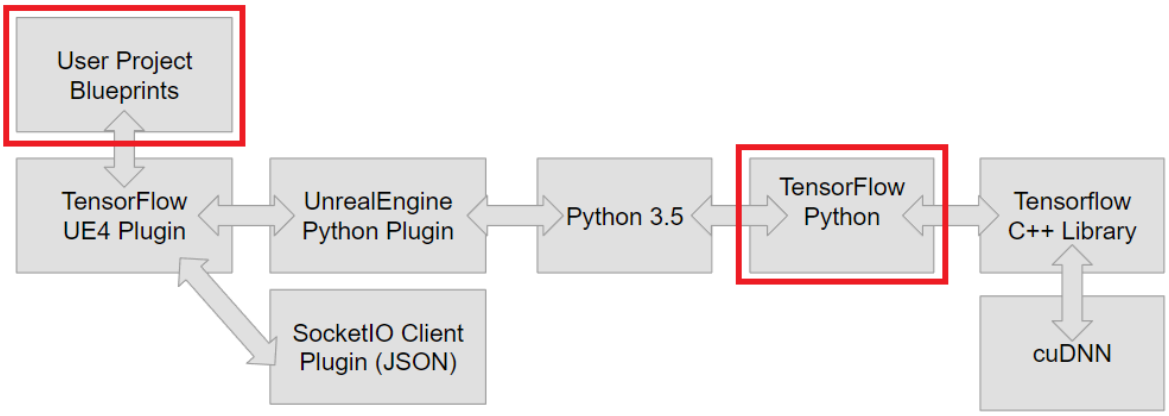

Figure 2.6: The Unreal Engine plugin for TensorFlow Architecture [19]

To show that the plugin work as intended, they present an example of machine learning usage inside of UE4. The example is a basic MNIST[13] softmax classifier trained on begin play with sample training inputs streamed to the editor during training. When fully trained, UTexture2D samples are tested for prediction. These are examples for general TensorFlow control and different MNIST classification methods with UE4 UTexture2D as input for prediction. The plugin enables control over training and also offers functionalities to save/load the model, so if there is already a trained model it is simple to setup model/load it from disk and omit the training function, forwarding the evaluation/input via the a callback. This work offers a connection between the Unreal Engine and a machine learning library that we can use to create RL algorithms, and we will use this plugin as a base for the development of this thesis.

2.5.2 Combining Deep Q-Learning with Unreal Engine 4

M. Bakhmadov [18] offers a system for UE4 that lets their users create controllers for characters using machine learning algorithms created using TensorFlow. The plugin that has a backend written in Python to handle all the machine learning calculations while the frontend is a Blueprint class that inherits from UE4's controller class. This way, it is possible to implement a controller that derives from UE4's AIController with added support for machine learning functions and variables that communicate with the Python backend. Since the system was created based on UE4's architecture, one can simply change the controller, which is the "brain" of an agent. Different input types as well as different environments were used to test what work best, as well as figuring out the best way to train a machine learning agent in UE4. All their research has been compiled into examples showing off different ways to use the system. Their results show that the agents are able to learn within different environments,

[13]The MNIST database is a large database of handwritten digits that is commonly used for training various image processing systems.

and that there's potential for even better results through further experimentation with the state representation, adjustments to the hyperparameters and additions to the DQN. The research questions focus on how different state representations will affect training and performance and how the agent performs in different environments. During the scientific results they expose their input on testing with Double DQN, Prioritized Experience Replay, Z-score. When comparing an agent using Double DQN and one using normal DQN, there isn't any notable difference in terms of learning rate or score. Double DQN also requires an extra network that needs to run predictions on, and therefore it uses roughly twice the computer power that DQN does. It is also shown that training using prioritized experience replay makes the agent learn much faster. Finally, M. Bakhmadov [18] presents that not using z-score normalization tends to deliver more fluctuations in the score compared to using z-score. These fluctuations are not major, but the training process becomes more stable with z-score than without. It is also worth noting that z-score normalization will give all the different observations the same importance when training.

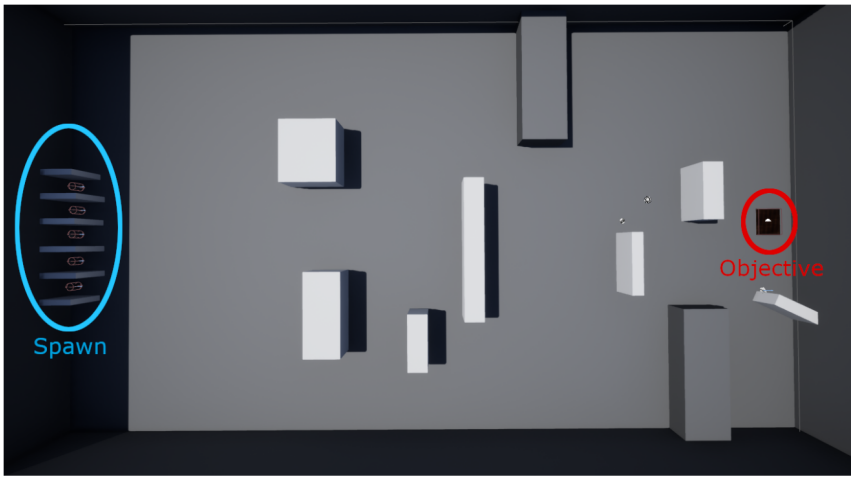

Figure 2.7: Overview of the race game mode map [18]

To test his solution, M. Bakhmadov [18] introduces the race game mode shown in Figure 2.7, where each agent has a goal and different obstacles on its way to the goal. All agents have a weapon that they can attack the other agents with and if one of them gets hit they are sent back to their start location (spawn). The first agent that reaches the goal will be the winner. The action space consists of 9 outputs: rotate left (10 degrees), rotate right (10 degrees), move forward (high speed), move right (medium speed), move back (medium speed), move left (slow speed), shoot closest target, jump or move towards the goal using the navmesh (slow speed). The system has handpicked values as the state that is being sent to the DQN. This handpicked state consists of the agent's distance to goal, rotation value, distances of what 8 line traces around the agent with 2000 in length can hit, the direction and rotation to the enemies the agent can see. The agent is rewarded 1 point if they reach the goal, and -0,25 if someone else reaches the goal. When an agent reaches the goal, everyone gets sent back to spawn. If

an agent kills an enemy, they are rewarded +0.05, and when an agent gets killed, they get -0,05. When we look at Figure 2.8, we can see that it takes 90 000 iterations to get 17 in score with handpicked inputs. Unfortunately, M. Bakhmadov [18] doesn't explain what consists an iteration of the algorithm nor how iterations translate to hours (time).

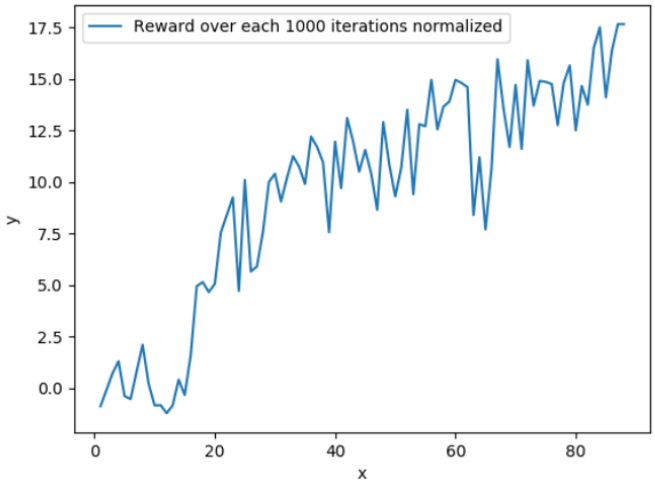

Figure 2.8: Plot of training session from 1 training agent in the race mode [18].

Based on the vast number of training sessions ran throughout the project, we conclude that combining RL algorithms with AI-tools in UE4 in the form of a plugin yield promising results both in usability as well as the behaviour the agents exhibit. This work offers a really good example on how to deploy RL agents in UE4 and was used as a guide during the development of the Deep QL-APF. It was also used directly to perform a preliminary assessment on the use of the V. Mnih et al. [11] RL algorithm use to perform the exploratory tests we want to. Results are shown in Chapter 4.

2.6 Discussion

The papers presented during the state of the art are all important sources of information to achieve the objectives of this thesis. Rare [2, 3] provides information about the type of tests used in a AAA game, and therefore we understand where we can contribute with automated tests. In addition, it shows that it is possible to perform automatic tests in Unreal Engine and that these automatic tests can reduce costs of playtesting sessions, such as the time to create/verify a build and the number of manual testers. Both works from section 2.1 show that the Unreal Automation System is a very useful tool to create and perform automated tests for Unreal Engine game, with P.Negrão work [4] ensuring us that it is possible to create a library and use its code logic to perform automated tests together with the Unreal Automation System. As presented during section 2.1, the Unreal Automation System allows the creation of a type of

test called Functional Test. These tests are created by spawning an Actor that can be scripted to perform a variety of verifications. This actor is called FunctionalTest Blueprint Actor, and Unreal Automation System recognizes it automatically when associated with a Map Level test.

Several studies such as [11, 13–15], show that deep RL can be successfully used to create agents that explore complex game environments to achieve goals. [14, 15] show that it is possible to create a RL algorithm that uses neural networks to train an agent into maximizing some score and win the game. Their results comprehensively demonstrate that a pure RL approach is fully feasible, without human examples or guidance, and given no knowledge of the domain beyond basic rules. The work that uses Doom as a case study [14], shows that it is possible to use DQN in 3D environments and that we can train the network to explore the game map. This study also demonstrated the potential generalizability of their neural networks to unknown maps. OpenAI [15] reveals how their reward model was implemented and how they defined the possible actions in a 3D game environment. Some of the game mechanics were controlled by hand-scripted logic rather than the network policy. The information described in these papers gives us possible directions for putting in practice a DQL method to explore a 3D game environment and achieve well defined goals. With the studies presented in section 2.2 and 2.3 we can realize that it is possible to model the personality and skill of a player and perform tests with different agents that use the different player models. C. Holmgård et al. [5] demonstrate that Q-learning is capable of incorporating the concept of a utility function through the reward model, and A.Soares work [10] reveals that we are able to create limitations at the level of observations and actions used in Deep Q-Learning. With this information we know that we can model the observations, actions and rewards passed to the network and therefore we can perform different types of tests depending on the information we want to obtain.

To cover the space of solutions, we present papers on Curiosity and Novelty search that allow the agent to explore novel states in an automated way. Results demonstrate that novelty search over action sequences is an effective source of selection pressure for innovation that can be integrated into existing evolutionary algorithms for deep RL, and that curiosity helps an agent to explore its environment in the quest for new knowledge. As we first need to integrate the framework, train artificial neural networks for the agent to move in the environment from a starting point to an end point and create tests that use these networks, we leave these methods open, being possible directions for the future.

Finally, in section 2.5 we presented solutions for implementing machine learning algorithms in Unreal Projects. The Unreal Engine plugin for TensorFlow is a precious tool that we can use to establish a communication channel between the Deep Q-Learning system and the Unreal Engine Actor that controls the agent in game, because TensorFlow provides the library needed to deliver neural networks and the Q-Learning Algorithm. M. Bakhmadov [18] shows that it is indeed possible to make this connection using the Unreal Engine plugin for TensorFlow and use outsource neural networks and Q-Learning

algorithm[14] coded for TensorFlow and based on the implementation of Deep Q-learning with experience replay made for “Playing Atari with Deep Reinforcement Learnin” [11]. This last work marked a starting point in the implementation of the work reported in this dissertation, being a role model that we followed during it. We want the Deep Q-Learning system to control the agent in a Unreal Engine 3D game and M. Bakhmadov [18] showed us how to do it. However, our system is focused in performing tests on the traversability and environmental issues of the map, rather than a race mode where agents compete to reach an objective. The actions of our agent must be similar to the actions players can use in game, because our objective is to explore the environment and find issues that we are not expecting and players may randomly find. There is a deep necessity of crafting a reward system, per Unreal Project, that matches the game and the tests we want to perform in it. There's enough information to assume that using a plugin to introduce a Deep Q-Learning testing platform to a Unreal Project is feasible and contributes to a modular architecture with the advantage that users can use it in different projects.

[14] Arushir implementation of Deep Q-learning with experience replay: https://github.com/arushir/dqn

3

Proposed Solution for the Deep QL-APF

Contents

In the following sections we explain the architecture of this framework and each of the distinct components that compose it. The playtesting framework produced in the context of this thesis is a plugin that can be easily added to every UE4 project. The solution is set around the tensorflow-ue4 [19] plugin since it offers the connection between the two elements needed to deploy a RL agent in the case study game environment.

We start this chapter by presenting a model that shows an overview of each component of the Deep QL-APF and how they communicate to provide a solution in the context of this thesis. The section that follows provides an overview of how the tensorflow-ue4 [19] plugin architecture is used in the context of this thesis. Finally, we detail the Unreal Automation System structure for performing Functional Tests and in the final section it is explained the system integration with the Functional Testing Framework.

3.1 Deep QL-APF Model

In Figure 3.1 we present an overview of the Deep QL-APF playtesting framework main modules and the information that flows between them. The framework main modules are distinguished by colors.

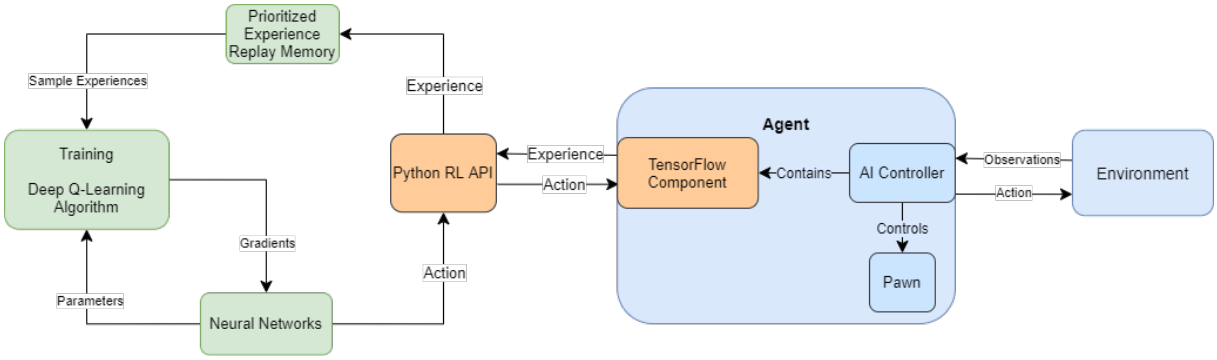

Figure 3.1: Model showing the Deep QL-APF main constituents and the information that is shared between them.

Following the DQN architecture by V. Mnih et al. [11], the green states represent the procedures a normal DQN implementation with handpicked observations and actions would take. A training experience containing the reward and observation for the chosen action is passed to a memory that regulates the storage of experiences, limiting the set of experiences the agent can save and sample through its experience replay. Multiple experiences are compiled together in a sample and used to tweak the gradients of the neural network during the training session. The blue nodes represent the constituents that Unreal Engine offers to perform actions in the game environment. The agent is composed by the AI Controller and the Pawn[1], which is the 3D character model that moves in the Unreal Engine 3D game environment.

[1] Pawn is the base class of all actors that can be possessed by players or AI. They are the physical representations of players and creatures in a level.

It executes actions in-game and receives observations from a handpicked values that characterize the surrounding environment.

At the center of the model, and identified by the orange color, we represent the constituents that tensorflow-ue4 plugin [19] offers to setup a flow control with the machine learning library, receiving actions from the neural networks and performing them on the environment. The PythonAPI forwards to the Engine an action the neural network chose to be executed in the environment. It is also responsible for making the agent experiences flow from the Engine to the DQL algorithm. The TensorFlow component is added as a special sub-object to the AI Controller to allow information to flow to and from the RL PythonAPI, allowing neural networks to execute the chosen action at each step in the environment. As shown during Chapter 1, this solution lets us train the agents and then use them to achieve a well-defined objective.

3.2 Machine Learning Integration with Unreal Engine 4

The tensorflow-ue4 plugin [19] is at the center of this thesis architecture. As previously presented, it is an Unreal Engine plugin that enables the creation of this thesis Deep QL-APF playtesting framework for UE4. As shown in Figure 2.6, it depends on two very important plugins, the UnrealEnginePython[2] plugin and SocketIO Client[3] plugin.

UnrealEnginePython enables multi-threading, python script plugin encapsulation and automatic dependency resolution via pip, a package-management system used to install and manage software packages. Simply specifying tensorflow as a python Module dependency makes the editor auto-resolve the dependency on first run. The multi-threading support contains a callback system allowing long duration operations to happen on a background thread (e.g. training) and then receiving callbacks on the game-thread. This enables TensorFlow to work without noticeably impacting the game thread. SocketIO Client is used for easy conversion between native engine types (Blueprint or C++ structs and variables) and python objects via JSON. Every message going back and forward is coded in JSON. Both of these plugin and an embedded python build are included in every release of tensorflow-ue4 [19] so you don't need to manually include anything, just drag and drop the plugin folder into the project.

The DQL components are represented as green in Figure 3.1. Each one of this states corresponds to one Python object with its own properties and methods. The PythonAPI is responsible for creating the python object that sets up the architecture for the Deep Q-Learning system. The DQL object is the main code file in the RL architecture. It contains the Q-Learning algorithm and creates the neural networks and the prioritized experience replay memory needed for the Deep Q-Learning system. It uses both of these objects to train the agent based on its experiences. It is also responsible for sending the best

[2]UnrealEnginePython: https://github.com/getnamo/UnrealEnginePython
[3]SocketIO Client: https://github.com/getnamo/socketio-client-ue4

action to execute in the current state back to the PythonAPI. Each one of these objects have a particular task that must be performed in the RL system, and will be detailed in the following Chapter 4.

The Blueprint API is set in the form of an Actor Component, a special type of object that can be added as a sub-object of an Actor. Described in Figure 3.1 as an orange state, the TensorFlow Component can be used to load the PythonAPI module presented above. Represented as a blue state in Figure 3.1 we present the playtesting AIController, a special type of actor that Unreal Engine offers to control non-playable characters in-game. Controllers are non-physical actors that can be attached to a pawn to control its actions, managing its artificial intelligence. In Figure 3.2 we present an AIController using the Tensorflow Component as a sub-object.

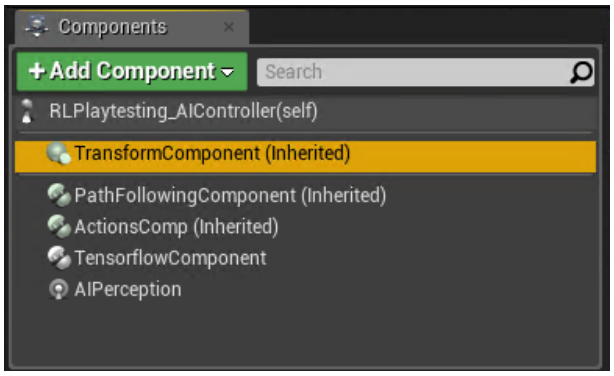

Figure 3.2: TensorFlow Component as a sub-object of an AIController in Actor Blueprint.

This AIController is responsible for using the features offered by the Tensorflow Component and send at the game start all the information needed by the PythonAPI to start up the DQL system. The Tensorflow Component allows the AIController to receive callbacks on the game-thread in order to execute the action selected by the algorithm. It also offers the possibility to run functions in the PythonAPI, for instance, running a iteration of the algorithm or save the neural networks model.

3.3 Unreal Automation System

As shown in Figure 3.3, The Unreal Automation System is a test framework that comes with UE4 as a plugin that developers can enable to perform automated tests.

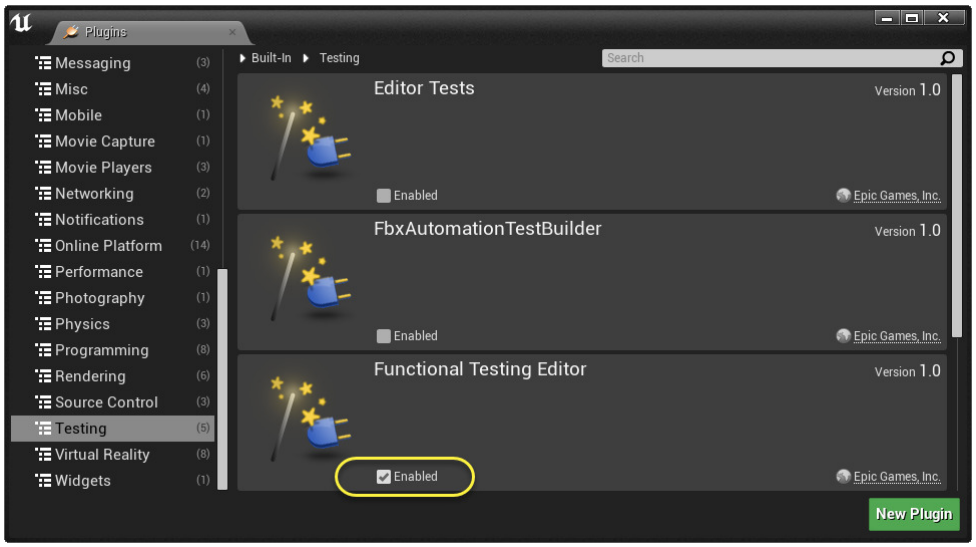

Figure 3.3: Plugins Browser to enable Automation Tests.

The Automation system eliminates the need for human interference to complete a task and allows 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. These tests that are written in code can be broken down into the following categories depending on their purpose or function:

- **Unit tests:** API level verification tests. By definition, unit tests check a specific operation on the smallest testable part of the code, which generally means testing at the code function level. See Figure A.1 in Appendix A for examples of these.
- **Functional/Feature Tests:** These tests generally evaluate larger pieces of software such as gameplay mechanics or specific actions in game systems, for example, AI behavior or dropping an inventory item.
- **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 focus of this thesis is Functional Tests, a test that is attached to a Unreal Engine Map.
- **Screenshot Comparison:** Enables QA testing to quickly compare screenshots to identify potential rendering issues between versions or builds.

The Automation tab is part of the Unreal Session Frontend, as shown in Figure 3.4.

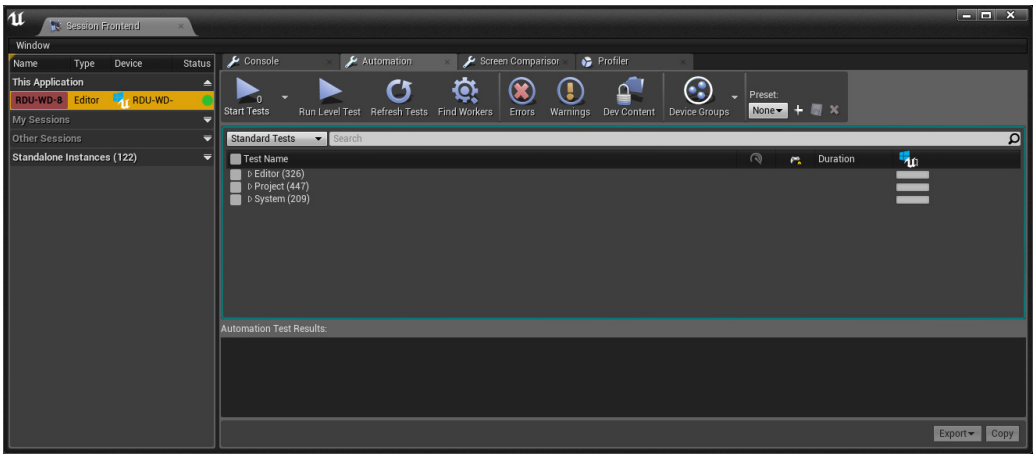

Figure 3.4: Session Frontend with the Automation tab in focus.

It enables developers to run automation tests on any other devices that are connected to their machine or are on their local network. Running tests in the editor is as simple as going to the automation tab and selecting the tests to execute. In addition, the tests can be set to run on built executables or remotely by a build system. A standalone tool can be created to allow running the tests from outside the editor. This means that developers often don't need to run the game at all to see if their latest code iteration had broken anything, as the tests run automatically and give them fast feedback on their changes. This point is very important because it enables agents to be trained automatically in a recent build, erasing the need for developers to interact at all with the Unreal Automation System to train or test with the trained agent. Tests are implemented in the game/engine code for each project and UE4 offers an object named Gauntlet[4] which is an automation tool in Unreal Engine. It is a C# program which can install and run game builds on devices. Gauntlet will also gather any test artifacts (logs, crash reports, etc), and package it nicely for the user. This framework allows developers to run tests outside the editor, erasing the developers need of training the neural networks and running the tests by themselves.

[4]Gauntlet Automation Framework: https://docs.unrealengine.com/4.27/en-US/TestingAndOptimization/Automation/Gauntlet/

3.4 Functional Testing Framework

Setting up a test is done by placing a Functional Test Actor in a Level, as shown in Figure 3.5.

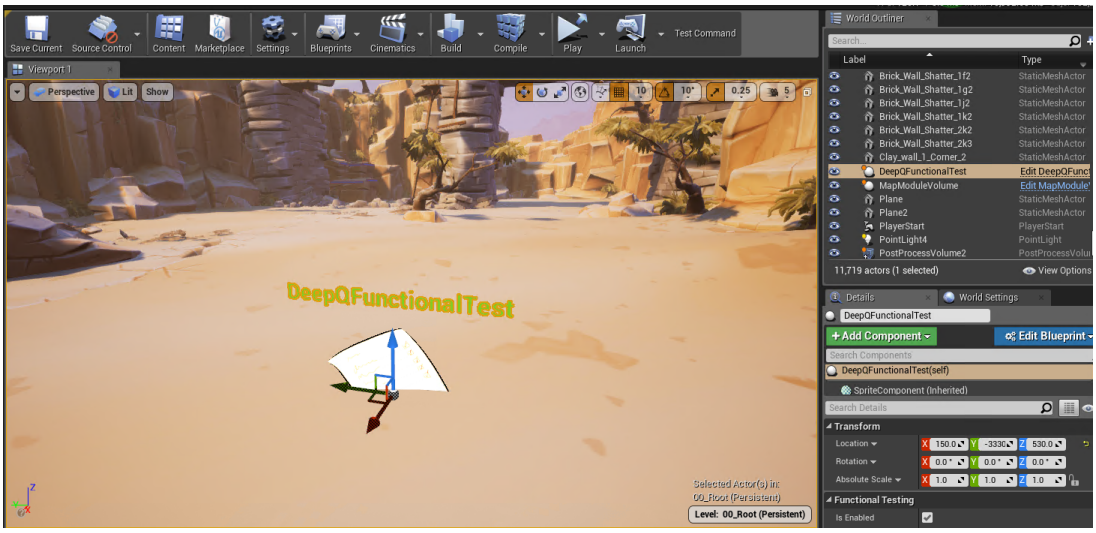

Figure 3.5: Functional Test Actor placed in a Map level.

This Actor is scripted to run a set of tests that can be built into the Functional Test itself (as a child code class or Blueprint), or assembled directly in the Level Blueprint. If a Functional Test requires a more complex setup, or is intended to run multiple times (either in a single Level, or on multiple Levels), overriding the Unreal Engine class *AFunctionalTest* is the recommended method. Figure 3.6 shows the creation of a Functional Test blueprint that it is offered by this framework to perform automated tests on the game environment. We also recommend watching this video[5] demonstrating an example on how to create, execute and check the results of a Functional Test.

Extending the base Functional Test class in code or Blueprints grants the ability to use PrepareTest, RunTest and IsReady functions, considered important for running tests with setup times longer than one frame and more complex or inter-dependent tests. However, in the context of this thesis we will only use the functional test to code functions that mostly log the information we want to obtain with the RL agents during the training and testing performed in the game environment. Functional tests are created as maps in the Unreal editor. In the context of this thesis, each map test uses specific handcrafted agents in a fixed scenario and the test reports its results as a pass or fail, based on whether the behavior is the one expected or not. Constructive and valuable feedback is sent to the message log during the execution of the functional test, such as time spent on each map module, time to achieve the game objective and number of times that the agent got stuck or left the map boundaries.

[5] Functional Tests setup video: https://www.youtube.com/watch?v=HscEt4As0_g

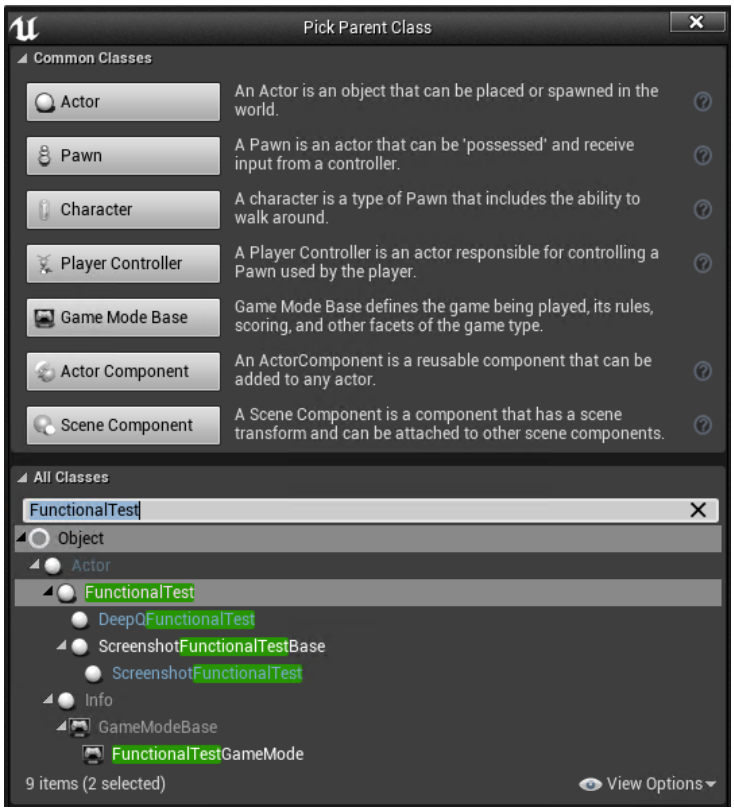

Figure 3.6: Creating Functional test blueprint extending the base Functional Test class

In this chapter we presented the architecture of the Deep QL-APF, which is built in a modular way separating the functionality of each framework component into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality. The DQL algorithm is detached from the tensorflow-ue4 [19] and any other neural networks and Q-Learning algorithm can be used, as long as it is implemented using the TensorFlow machine learning library. This is a strong point of this thesis because changes can be made without heavily impacting how the framework is structured to perform test with RL agents in UE4. The Unreal Automation System is a precious tool that we will use to create and perform different tests automatically, while the Gauntlet Automation Framework allows us to execute the training and perform the tests outside the editor in recently created builds, erasing the developers need of training the neural networks and running the tests by themselves. In the next chapter it is detailed the steps that were executed during the implementation of the Deep QL-APF playtesting framework.

4

Deep QL-APF Implementation

Contents

During the development of this thesis there was a special focus on implementing two different solutions: a method for connecting the three modules presented in the Deep QL-APF model presented in Figure 3.1, so that we can deploy RL in UE4, as well as a specific implementation for the types of tests Funcom ZPX wants to execute on the game environment. In order to grant the tests proposed during Chapter 1, different Unreal Engine Actors must be crafted depending on the type of test the developer wants to execute and the information that should be obtained during that particular test. The agent rewards system and observations must be specific on how the agent should behave and it must be rewarded to achieve a well-defined objective. Besides this, tests must be created to automatically collect any feedback that may be valuable to the designer during the development of the game.

In this chapter we start by presenting the RL algorithm that was used to train the agents, explaining how outsource TensorFlow neural networks and Q-Learning algorithm can be adapted to run in this thesis context. The following subsection presents the Unreal Engine Actors needed to use the RL algorithm and also control the character in game. Subsequently, we present preliminary results achieved by training and testing a neural network on a simple environment. Last but not least, the automated tests that this framework offers are detailed, describing how they are created and how to use them to receive feedback from the agent behaviour.

4.1 Algorithm Specifications

The Deep QL-APF playtesting framework provides agents that use a RL algorithm to test the game environment. Being able to create RL agents is part of the general architecture for this thesis solution. The algorithm implementation is offered by A. Raghuvanshi[1] and it was implemented similarly to the model presented in V. Mnih et al. work [11]. This code implements state-of-the art deep reinforcement learning algorithms in Python and is integrated with the TensorFlow machine learning library. This solution is convenient since tensorflow-ue4 [19] Plugin uses this machine learning library in specific.

A. Raghuvanshi uses a simple fully-connected network with 2 hidden layers and an output layer instead of the Convolutional Neural Network (CNN) described in the paper [11]. A feed-forward neural network consists of a number of simple neuron-like processing units, organized in layers and every unit in a layer is connected with all the units in the previous layer. Not all of these connections are equal, as each connection may have a different strength or weight. The weights on these connections encode the knowledge of a network. For the sake of simplicity, a feed-forward neural network that takes handpicked values as input is welcomed since a CNN requires to transform an image representation of a 3D game environment into something the neural network can take as input. Figure 4.1 presents a structure similar to the neural network used as models for the agents offered in the context of this thesis.

[1] A. Raghuvanshi RL alogrithm implementation: https://gist.github.com/arushir/a955f15ab8c5d641f45d8a32bba4f931

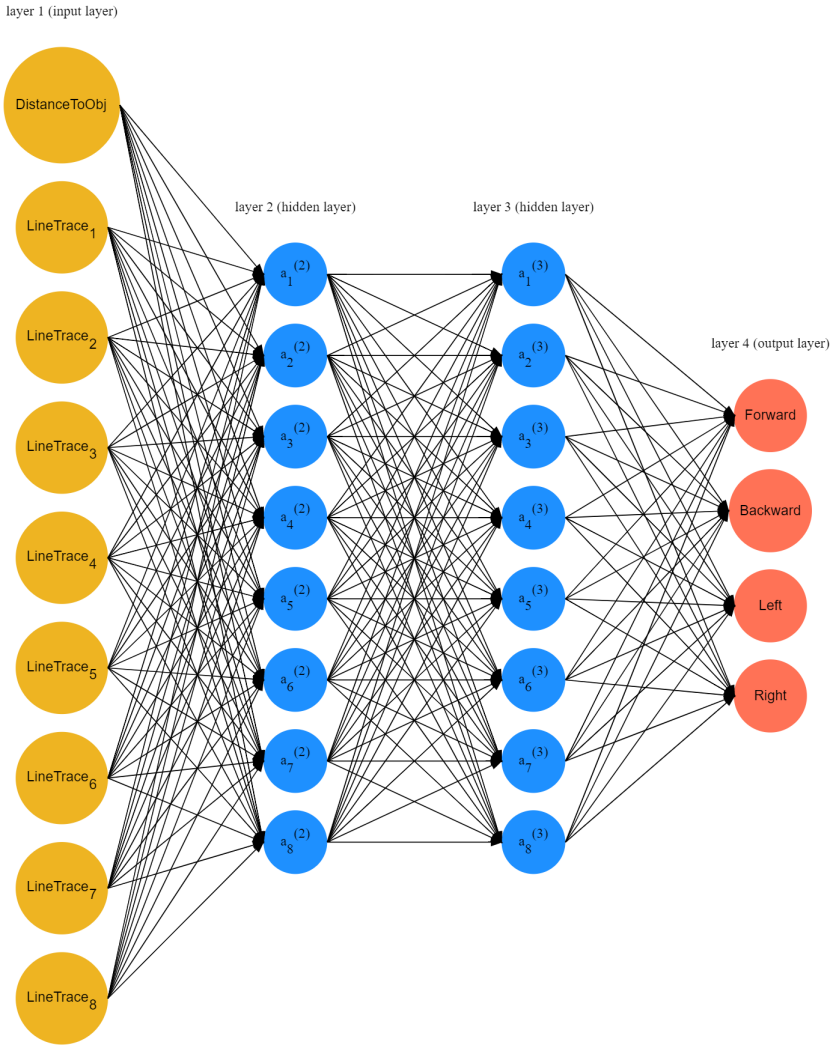

Figure 4.1: Architecture of the feed-forward neural network used in the Deep Q-Learning system.

The Deep Q-Learning algorithm updates the parameters of the neural network that estimate the value function, while this function objective is to maximize the sum of rewards over time. It happens through samples of experiences drawn from the algorithm's interactions with the environment. This algorithm utilizes a technique known as experience replay [20] where the agents experiences are stored at each time-step in a data-set, pooled over many episodes into a replay memory. During the inner loop of the algorithm, Q-learning updates, or mini-batch updates, are applied to samples of experience drawn at random from a pool of stored samples. After performing experience replay, the agent selects and executes an action according to an E-greedy policy. The full algorithm, which is called Deep Q-learning, is presented in Figure 4.2 below.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Figure 4.2: Deep Q-Learning algorithm pseudocode.

During section 4.3 and Chapter 5, we present results and the hyperparameters used in this thesis DQL algorithm. It is straightforward to follow the algorithm logic, correlate it with how it is implemented by A. Raghuvanshi and adapt it to work in the context of this thesis. We found out that it wasn't possible to estimate a number of episodes needed for the agent to achieve the objective in-game and therefore we employed continuous training over time, and not over only some episodes.

4.2 Algorithm Preliminary Assessment

It was made a preliminary assessment of the agent behaviour while using the Deep Q-Learning algorithm presented above to control the Character in UE4. Before scaling it to the case study and use the Deep Q-Learning system to perform tests on the game environment, the general solution must be validated to understand its feasibility in the context of this thesis. The agent is trained and the sum of rewards is logged over the period of time it is running. After training the neural networks, the developer should be able to import the neural networks model and use it to perform tests, therefore the functionality of importing the module must work perfectly.

It was used a simple and controlled Unreal Engine game environment to test the agent performance, as you can see in Figure 4.3 below.

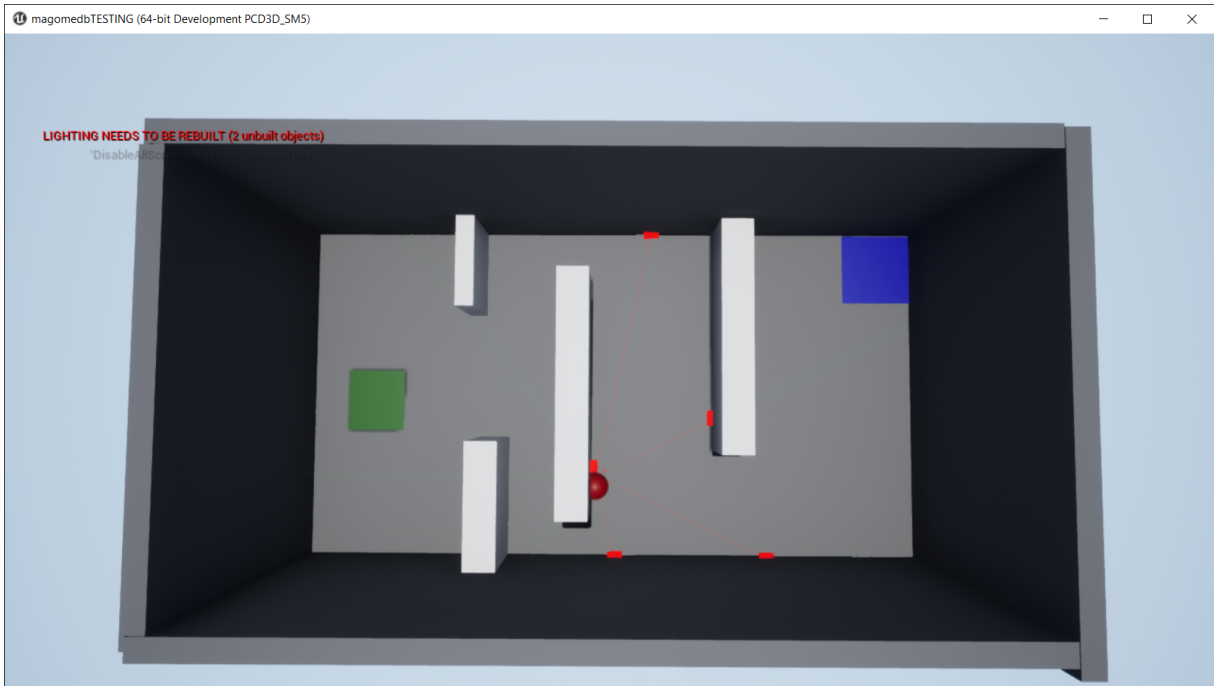

Figure 4.3: The Unreal Engine map used to evaluate the agents performance over time. The green square is the objective location while the blue square is the starting location. The agent is represented as a red sphere and the red squares on the walls are the points where the line traces intercept the map constituents.

An overview of the neural networks architecture is presented in Figure 4.1. The input layer is the agent observation at each step and consists of nine different inputs, one providing the pathfinding distance from the current position to the objective location and 8 other line traces cast around the character to get the distance from the agent current position to the other actors that the line traces collide with. The actions are 4 and correspond to the basic general movement: forward, backwards, left and right. Since we use a frame skipping technique, the neural network isn't continuously sending inputs to the agent, and the UE4 Move to Location[2] function was used to move the character in each direction during the frames that are skipped. When the agent is closer to the initial position, the reward it gets is smaller but when closer to the objective the reward is more significant. M. Bakhmadov [18] implemented perception of the game environment is similar to the one we hypothesized in our approach and use in the algorithm assessment, but the reward system is totally different. We want to perform exploratory tests on the game environment and in the algorithm assessment we use a reward system and define possible actions more well suited for exploratory testing. We don't train the agents for the same tasks that M. Bakhmadov [18] did, but we take the structure and test it to perform exploratory tests on the game environment.

[2]Makes AI go towards specified destination location, providing the possibility of using pathfinding and projecting the destination to navmesh.

	Agent 1	Agent 2
Hidden Layers	64 - 64	
Mini-Batch Size	32	
Frame-skip	8	
Target Network	Yes	No
Learning Rate	0.0001	
Epsilon	Decreases from 1.0 to 0.1	
Gamma	0.99	
Regularization	0.001	

Figure 4.4: Table presenting the hyperparameters used by similar agents that executed the algorithm preliminary assessment. One agent is using a target network and the other is not.

While testing the algorithm, we let some hyperparameters stay constant while varying others during the different runs made on the game environment. Hyperparameters are used to control the learning process. The hyperparameters that remain unchanged during the different runs are the learning rate, the gamma and regularization. Learning rate controls the rate or speed at which the model learns, while the gamma quantifies how important are future rewards. Regularization provides an approach to reduce the over fitting of a deep learning neural network model on the training data and improve the performance of the model on new data. The epsilon is a hyperparameter that defines the probability of selecting a random action at each step. During the run, the value decreases from 1.0 to 0.1 to oblige the agent to explore as many new states as possible during the training. We conducted an assessment prior to the results shown in this subsection to find the amount of nodes the neural network must have in each hidden layer, the optimal mini-batch size and the number of frames that should be skipped. During the training of these different agents it was found that 64 nodes for each neural network hidden layer wielded the best results in this game environment, while the mini-batch size with best results was 32. To find out how many frames the algorithm should skip we compare the results obtained by agents skipping 4, 8 and 16 frames, finding out that agents that skip eight frames between each algorithm iteration are the ones that yielded the best results. In Figure 4.4 we present the hyperparameters used in the results of the two different agents presented as algorithm assessment for this thesis.

The results presented in Figure 4.5 display the rewards sum over time for the agents that got the best results during all of the preliminary assessments previously executed. The results demonstrate that during the agents training they were able to reach the objective location regularly. They also manage to reach the goal regularly when the trained neural network model is imported. We also use a simple frame-skipping technique where the agent obtains observations from the game environment and selects actions on every 8 frames. Without frameskip, the agent is incapable of learning how to achieve the

objective efficiently. The agents that used frameskip of 4 and 16 don't demonstrate a stable learning, presenting inumerous fluctuations in the rewards sum value over time. Frameskip value of 8 frames present smaller fluctuations if compared with other frameskip assessment results. According to the state of art, a target network is used to make the learning more stable, however, by comparing both agents results we can't observe a big improvement regarding learning stability.

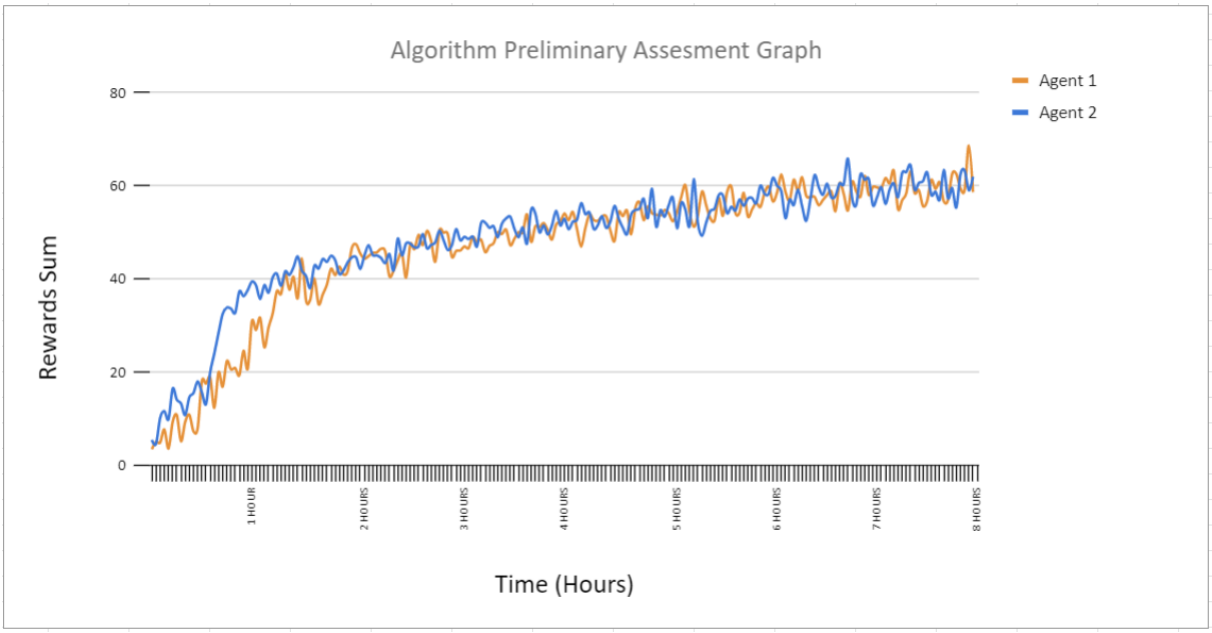

Figure 4.5: Graph showing the evolution of the rewards sum over time for 2 similar agents.

As a final remark, we can conclude that the algorithm is capable of training agents to achieve a well-defined objective in an UE4 game environment, therefore it delivers what we need in our solution and can be used in the context of this thesis. The algorithm needs 6 hours of training in this simple game environment to reach a state where the agents achieve the best results and maintain them maximized. It is expected that training RL agents in the case study will take longer.

4.3 Unreal Engine Actors

In the context of this thesis we want to automatically obtain information about what may happen when the player is walking through an Unreal Engine game environment. For that, we need to craft a specific AI Controller that controls the Character in-game and instructs it how to behave using RL. The agent behaviour must be crafted in a way that offers meaningful information during the development of the game.

During this thesis a base AIController was implemented which will be called from now on **Deep QL-APF Playtesting AIController** and integrates the TensorFlow Component as a sub-object. The

tensorflow-ue4 [19] offers to the base AIController features to initialize everything needed to start running the Deep Q-Learning algorithm. These features are passed down to the derived AIControllers that need to be implemented per each type of test performed in the environment. All the logic implemented for the AIControllers was done using the Blueprint visual scripting system.

The first mission of the Deep QL-APF Playtesting AIController is to use the TensorFlow Component to call an initialize function in the PythonAPI, in order to setup the Deep Q-Learning algorithm using the TensorFlow library. It sends at the game start a tuple containing ordered variables needed to create the main loop of the algorithm presented in Figure 4.2. It is also responsible for triggering two different looping events in the Deep QL-APF Playtesting AIController to control the algorithm loop. There's an event for saving the neural network model each minute of the Engine runtime through a TensorFlow library method, and another that retrieves the experience from the derived AIController that is controlling the agent every 8 frames. This experience is sent to the function that runs the RL algorithm epoch, returning an action to the derived AIController that's attached to the Character in-game. The actions, rewards and observations must be crafted depending on the type of test the developer wants the agent to perform in the game environment. In the context of this thesis, the tests that should be performed are exploratory tests using the general character movement to navigate thought the game environment, meaning that the actions are specific to this type of test. The climbing mechanic should be checked using functional tests or by creating another neural network and AI Controller for climbing, which is presented in Chapter 6 as Future Work.

The observations and rewards system must be set depending on the objective the agent should achieve and how the developer wants it to behave during the algorithm runs. This implementation must be created in the classes that derive from the Base Deep QL-APF Playtesting AIController. We offer two different AIControllers that can be used to train and test the case study game environment, which are named **Pathway Exploration AI Controller** and **Wall Exploration AI Controller**, as shown in Figure 4.6.

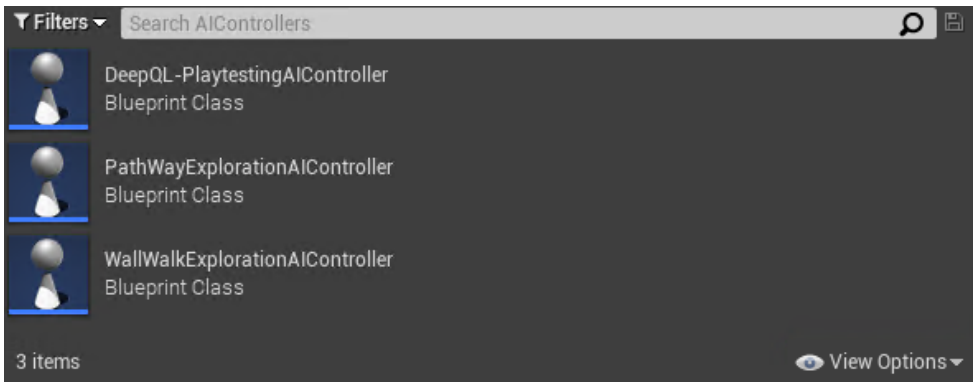

Figure 4.6: Editor view of the three AIControllers objects produced during the thesis.

As it was previously explained, each derived AIController receives and executes the action in the game environment. Both of the derived AIControllers use the same actions for the different tests they perform, however, the reward system and the observations differentiate themselves. For the Pathway Exploration AI Controller, the observations consist on the distance from the character to the objective using the pathfinding distance. It is calculated using the navigation mesh and the A* pathfinding algorithm offered by UE4. It also uses 32 line traces that calculate the distance to the other map constituents it collides with. The reward system is crafted in a way that rewards the agent depending on if the action executed got the UE4 character closer to the objective or not. This means that we must verify the previous distance to the objective and check if it is higher than the current distance. The Wall Exploration AI Controller introduces one new input to the neural network that informs the agent if there are any line traces colliding with other map constituents, meaning that the agent is closer to an environmental wall. The line traces are shorter than the Pathway Exploration AIController and the reward system was modified to push the agent to move closer to the environmental walls. The reward given at each timestep depends on the same reward system created for the Pathway Exploration agent, but comprehended between -0.7 and 0.7, depending on if the agent is moving towards the objective or not. It also incorporates another reward signal calculated from the line trace that found the shorter distance to the actor it collides with in the game environment. This reward value varies from -0.1 to 0.4. The negative reward is offered when there are not any line traces hitting an environmental wall. We hypothesize that this way the agent will tend to move closer to the boundaries of the map since the rewards are higher there. By using this test, the developer can verify what may happen to the player when trying to achieve the objective the best way possible as well as achieving the objective while walking near the map walls. These are two different behaviours that players may perform while playing the game, and therefore, there may be valuable feedback to be obtained from these two agents procedures.

While training the agents in the case study game environment we spotted different types of problems. The main problem was the agent getting stuck using only the directional movement, which was solved by simulating the jump action to release the character, a behaviour that players tend to execute while trying to free themselves from this type of situations. It worked effectively for most of the cases, but sometimes the agent stayed stuck. On those cases, the location where the agent got stuck was saved in a log file and the training was restarted. All of these places ended up being confirmed as problematic locations for the player. The agent was also capable of discovering a place where it could leave the playable area, which is a problem in the game environment and precious feedback to the developer. Therefore when the character leaves the playable area and falls out of the game world, these locations are also saved on the testing log.

By taking a look at the game environment, we can spot three different map modules that are separated by the climbing mechanic. This is very common in exploration and adventure games, where

players finds obstacles that need to be solved using a specific game mechanic. This means that we need neural networks trained for executing those mechanics in order to get past the obstacles. In the context of this thesis, we are focusing on exploring the game environment with the directional movement, using a single neural network for that purpose, which means that different sub objectives must be placed in the environment for each map module where the agent can move before encountering the climbing obstacle.

When the agent is exploring the environment, it may fall directly into other map modules, meaning that it can't achieve the objective that was set to be attained in the last map module. Figure 4.7 demonstrate an example of this occurrence, where the player can jump off the bridge directly into the initial area.

Figure 4.7: Visual Representation of player going from one map module to another. The red arrow represents the player movement.

While taking this into consideration, we crafted a way to detect when that happens to reset the current objective and set the next respawn point. It was created a Modular system for testing the agent, separating each map module. The developer is obliged to place them in the environment and explicitly tell the module order for the path that the agent must execute. The modular system consists of an Unreal Engine Actor that contains as a sub-object a Trigger Collider[3]. When the agent starts colliding with this trigger, the objective and respawn point of the agent is updated through the Actor. In Figure 4.8 we present the proprieties that must be set in the Map Module Actor.

[3]Triggers are Actors that are used to cause an event to occur when they are interacted with by some other object in the level

Figure 4.8: Editor view of the Map Module Actor proprieties.

The developer must identify the objective and respawn point of that map module, as well as the order in which the agent should travel to achieve the main objective of the case study, which is to reach the fortress gate. The starting module and terminal module must also be identified. All of these systems were really important in order to execute the tests presented in the next subsection. This means that developers can reuse the Deep QL Automated Playtesting Framework to create RL agents, more precisely the Deep QL-APF Playtesting AIController, but they still must implement different solutions for problems that may appear from using the framework to perform specific types of tests in different game environments.

4.4 Automated Playtesting Tests

The focus of this framework is to find different ways of exploring the game environment, making the designed and implemented Actors and Functional tests specific for solving this issue. To create the logic of what happens in these Functional Tests, we made use of the Unreal Blueprint system. During the exploration of the environment, it is possible to craft methods for gathering precious feedback and report it through a Functional Test. The agent must be trained in the game environment first, so that it can then be used to test the environment. In the course of the implementation of this thesis, it was obvious that the agent could find issues during training, and therefore the agent training also became a possible way to obtain feedback. The training of the agent must be ran jointly with the Functional Tests that are placed in the map, and therefore a derived AI controller must be associated with one specific Functional Test. When the Functional Test is used for training the agent, the Test always passes without a time limit, and prints out the feedback the test is expected to deliver in the message log. This way, while the agent is learning to explore the environment and reach an objective it is also looking for problems in the game environment. The idea is that the training can occur off working hours, i.e night time or weekends, and the trained agents are always available for testing the environment during work days. If the test doesn't pass within the time limit, it means that the agent is incapable of achieving the game objective. Figure 4.9 presents the Session Front End Automation Tab containing the tests that were created in the context of this thesis.

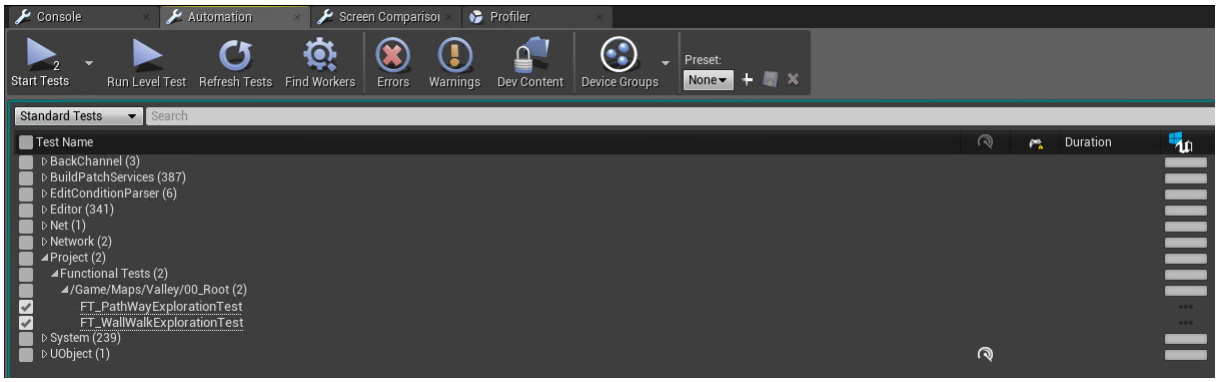

Figure 4.9: Session Frontend Automation Tab containing the tests implemented using the Deep QL Automated Playtesting Framework.

We want to deploy functional tests that can be run in the builds that are created after the developers submit something new to the file management system. The Gauntlet Automation System takes the build and runs the specified tests automatically. It can be used to verify certain features, for instance, if the path is traversable and the player doesn't get stuck in any environment constituents (for instance, rocks and vegetation). This way, agents can be trained automatically without humans preparing and running the test. The file system is capable of training the agents from time to time. If there are any problems, test engineers can take a look at it and solve the problems in order to get it back working automatically again.

Agents don't have time constraints and therefore we can speed up the world timer in order to tick faster. The time dilation for both agents is set to 5, which means that the world is ticking 5 times faster than usual. We found that this value speeds up the training while giving us enough control over the amount of frames we wish to skip using the frame skipping technique. Another concern was that raising this value ends up reducing the machine performance.

4.4.1 Pathway Exploration Test

The purpose of this test is to find the best route to the objective location. It is one simple test that guarantees that there is an available path for the player to advance in a linear game level.

The Pathway Exploration AIController contains the implementation for executing the training and the actual test with the trained agent. At each timestep, it is responsible for: checking if the agent is stuck or fell from the world boundaries, executing the pretended action using a movement function, gathering observations from the environment, calculating a reward based on those observations and also send back to the functional test log the feedback the developer wants to obtain with this test.

The Unreal Engine Pathfinding System offers a method to find the distance from the current position to the objective position using the nav mesh. The remaining observations are provided by the line traces

cast around the agent, in order to create a perception of its surroundings. When the line traces hit an Actor, they return the distance to that actor. The agent is controlled with the UE4 Move to Location function that move the agent up, left and right in straight line, during the specified algorithm frame-skip. The reward system compares the previous distance to the objective with the current one, giving a positive reward if the agent gets closer to the objective, and a negative reward if it goes further away. The reward value is set depending on the distance it moved during the last iteration. There's a divisible value that controls the amount of reward that is calculated at each step. The agent always receives -0.1 reward to prevent it from getting stuck and avoid those places again. The reward value is normalized between -1 and 1.

By running this agent within a functional test, we can check if the player has an available path to progress trough the environment and reach a certain position on the map. The functional test is used to start the tests automatically from the file system and record messages in the test log for the developer to read the information afterwards. If any, this feedback is composed by the location where the agent got stuck, the location where the agent left the world bounds and the time that it took to reach the objective.

4.4.2 Wall Exploration Test

The intention of this test is to find an available path to the objective while moving close to the environment walls. This test was made to demonstrate, in the context of this thesis, that it is indeed possible to create various tests with agents that behave differently while executing them. The agents behaviours must be set by crafting different reward systems and handpick observations, while maintaining the same actions. We also hypothesized that this test will be able to find more problems related with getting stuck, leaving the map boundaries and actor mesh problems. The Wall Exploration test has the same objective as the test presented in the section above, which is to move to a specific location in the game environment. The main difference is how the observations and reward system work together to create a different behaviour for the agent. The line traces cast around the character are shorter and the reward system delivers positive rewards when the line traces touch a wall in-game. Since the rewards will be higher when this happens, and there is a input informing the agent when the line traces hit something, it is expected that the agent will try to maximize the actions by walking close to the walls. The reward system was inspired by the work of C. Holmgård et al. [5], where it is presented the Runner agent which receives a positive reward when reaching the objective location, and a negative reward when not moving. In comparison, for instance, the Treasure Collector agent receives the same rewards as the Runner agent, with the addition of receiving a positive reward when it collects treasures. Inspired by these agents, we created the Wall Exploration agent that wants to move to the objective while walking close to environmental walls, being a derived agent from the Pathway Exploration agent because it is tasked to achieve the same game objective but in a different way, just like the agents presented in Holmgård et al. [5] work. The

distance from the agent current position to the objective position is given by getting the linear distance between them, instead of using the pathfinding distance. We do this because we don't want to find the best available path with this agent, and it is interesting to understand how the agent behaves with this slightly different observation value.

In this chapter we presented results confirming that the RL algorithm is a promising solution for teaching agents to achieve a specific objective in an UE4 game environment. Subsequently we presented the Unreal Engine Actors we had to introduce to the project in order to perform the tests detailed in this last section, stating that every test that uses RL agents need a specific implementation regarding the Engine objects needed to perform them. Developers who use this framework on their Unreal Engine project do not need to change the Deep QL-APF AIController, which is used to create a connection between the machine learning library and the UE4, but are compelled to create a Functional Test to automatically run the test and receive useful information in the test log, as well as creating a derived AIController in order to obtain observations, calculate rewards and execute the actions relevant to the test that the developer wants to execute. This thesis offers 2 implemented agents that can be used to perform exploratory tests in any other UE4 project with minimal changes. They can also be used as a guideline for the implementation of similar tests with RL agents in other UE4 projects. In the next chapter we present the experimental procedures executed for both testing agents, detailing the results obtained and performing a deep analysis on what those results mean. We will assess if the DQL solution can be used to perform the type of tests we want to deploy in the case study game environment.

5

Results and Analysis

Contents

In the previous chapter we demonstrated that the algorithm preliminary assessment was promising because the agent learned how to go through a simple maze and reach the objective consecutively in a controlled environment. We want to validate the DQL solution with the case study that was provided, and therefore we are going to present results on the agents learning behaviour during this chapter. Besides evaluating the learning capacity of DQL agents, we also want to assess if they can be used to perform exploratory tests. By handcrafting different types of agents to execute different tests on the game environment, we hypothesize that this framework can be used to provide meaningful information about different procedures that players might take to explore the environment and reach a well-defined objective. This chapter provides results on the two different agents presented in the previous chapter. It will be discussed if the agents behaviour is different from each other and if the feedback they provide while exploring is useful to validate the case study game environment. Besides looking for different paths to reach the objective, it is interesting that agents find problems in the game environment, such as places where the player might get stuck or leave the map boundaries. The case study already presents places where the character can get stuck, but, as an experimental procedure, the case study will also be modified to contain another 2 places where the agent can leave the environment and three specific locations where the character can get stuck and can't leave the place. The results regarding the problems found by the agents will be compared with real players that playtested the game environment manually to find an available path to the game objective while reporting any problem they find. If the playtesting framework agents find the same amount of problems as real players, then we can confirm that the Deep QL-APF has potential to be used to automated tests and reduce valuable resources such as people availability and time. During Chapter 3 we already concluded that developers and designers don't need to spend time executing the functional tests offered in the context of this thesis because they can run automatically to train the agent in a submitted build while logging any problem they find.

5.1 Experimental Procedures

The experimental procedures will be conducted in the case study offered by Funcom ZPX to test the Deep QL-APF playtesting framework. As summarily detailed above, three different procedures were done in the context of this thesis. We want to use the Pathway Exploratory Functional Test and Wall Exploration Functional Test to assess if the agents learn how to maximize the sum of rewards in the long run and achieve a certain position on the map (objective) consecutively. This procedure will let us know if the DQL algorithm is working as expected and can be used in the context of this thesis to control a character in-game, train the agent to achieve an objective and perform a specific test using that same agent.

The second experimental procedure will be performed by comparing the behaviours of the agent

trained during the Pathway Exploratory test with the agent trained during the Wall Exploration test. We argue that by changing the agent perception and reward system, we can craft agents that present different ways of exploring the environment. The assessment will be done by visually comparing both agents after they are trained and their neural network model is imported. The agent will leave a line trace while moving in the game environment so that its possible to compare the paths they perform to reach the gate fortress (game objective).

Last but not least, the final procedure is focused on understanding if the Deep QL-APF playtesting framework is capable of reducing human and time resources needed to perform this type of exploratory tests manually. When the agent is training and testing, their main task is to find an available path to the objective location while looking for problems such as getting stuck and leaving the playable area. We want to compare the problems that the agents find in the case study with the problems that human players performing an exploratory playtesting in the same game environment find. We draw conclusions by analysing the playtesting questionnaire[1] given after the playtesting session and comparing the results relative to the problems players found in the game environment. During the following subsections the testing scenarios are explained in detail.

5.1.1 Experimental Scenario 1: RL Agents learning how to reach a location in the game environment

The environment where the experiments where performed is the case study detailed in Chapter 1. The player character is possessed by one of the derived AI Controllers created for the available tests and the reinforcement learning agent is placed in the game environment. We start the game by running the functional tests in the session front end and leave the agents training during 10 hours. In figure 5.1 it is presented the hyperparameters for both agents.

	Deep QL-APF playtesting framework Agents
Hidden Layers	1024 - 512
Mini-Batch Size	128
Frame-skip	8
Target Network	No
Learning Rate	0.00001
Epsilon	Decreases from 1.0 to 0.1
Gamma	0.99
Regularization	0.001

Figure 5.1: Hyperparameters used for the agents that perform the Pathway and Wall Exploration tests.

With this experimental scenario we want to understand if the crafted agents are both capable of being trained by the proposed DQL algorithm to find one available path to specific locations in the game

[1] Questionnaire that was handed to human players after the manual playtesting: https://forms.gle/P4QLkci5ogTBDKZFA

environment. The rewards sum should reach an high value and maintain it when the agent is constantly reaching the objective. We will analyze the reward sum line chart and draw conclusions for both agents learning performance.

5.1.2 Experimental Scenario 2: Agents with different handcrafted behaviours

This experience consists in using the two agents trained during the previous experiment and comparing the path they took to reach the objective position in the game environment. The Pathway Exploratory testing agent is trying to find the best path to the objective, while the Wall Exploration testing agent has its rewards system modified to maximize the rewards when the agent is near an environmental constituent while moving towards the objective. The two agents behaviours are compared by a visual representation of the path done by both agents to the objective location. Both agents neural network model is imported after the training session and the agents are set to run until they reach the final objective (fortress gate location). With this experiment we can understand if it is possible to create different behaviours for reinforcement learning agents that want to achieve the same game objective. We will also assess the number of times each agent gets stuck or leaves the playable area, in order to compare both agents ability to find problems in the game environment.

5.1.3 Experimental Scenario 3: Comparing manual and automated playtesting for exploratory tests

During this experience we are going to compare the RL agents ability to find problems in the game environment with the problems that human testers performing manual playtesting find. The human testers are asked to play the game for 30 minutes and their objective is to check if there's an available path to the fortress gate (objective) while looking for problems in the game environment constituents. The manual testers are asked to go from the beginning of the level until the end, repeatedly, during 30 minutes and point out the problems they find. They are asked to find problems in the environment constituents that may ruin the players experience while exploring the environment. We don't explicitly tell the problems they are expected to find, such as getting stuck or leaving the playable area. The actions available are the same as the ones the agents use, which are the directional movement and the jump action. The case study was modified to introduce the 5 problematic locations displayed in Appendix B. We want to check if the manual testers or the agents are capable of finding these problems. The results obtained by the manual testers are then compared with the agents results so that we can conclude if the agents are suitable for performing exploratory testing. If we can observe similar behaviours between the agents and the human testers, and if the number of problems found is similar, then we can confirm that this framework can replace the human testers and therefore reduce the resources needed to perform

this types of exploratory tests. The human playtesting will be observed so that it is possible to compare the human behaviour with the agents behaviour. We will deliver the questionnaire after playtesting in order to assess the problems players found and how they felt while performing this type of exploratory test.

5.2 Automated Playtesting Framework Test Results and Analysis

The agents were trained in a single run that took around 10 hours. The system specifications are the same for all the agents that were trained in the context of this thesis. The CPU is a AMD Ryzen 7 3800X 8-Core Processor with base clock of 3.89 GHz and the GPU is a NVIDIA GeForce RTX 2060. The installed RAM has a memory size of 64GB.

During the first experiment we found out that the Pathway Exploratory agent is indeed capable of learning how to reach a specific location in the game environment consistently with the parametrization scenarios presented in Chapter 4, but the Wall Exploration agent struggles with learning how to achieve the objective repeatedly while walking close to the environment walls. It takes way longer for this last agent to achieve the objective and its movement is irregular, moving randomly while close to the environment walls until it achieves the objective. In Figure 5.2 and 5.3 we present the charts that display the variation of the rewards sum during the training of each agent.

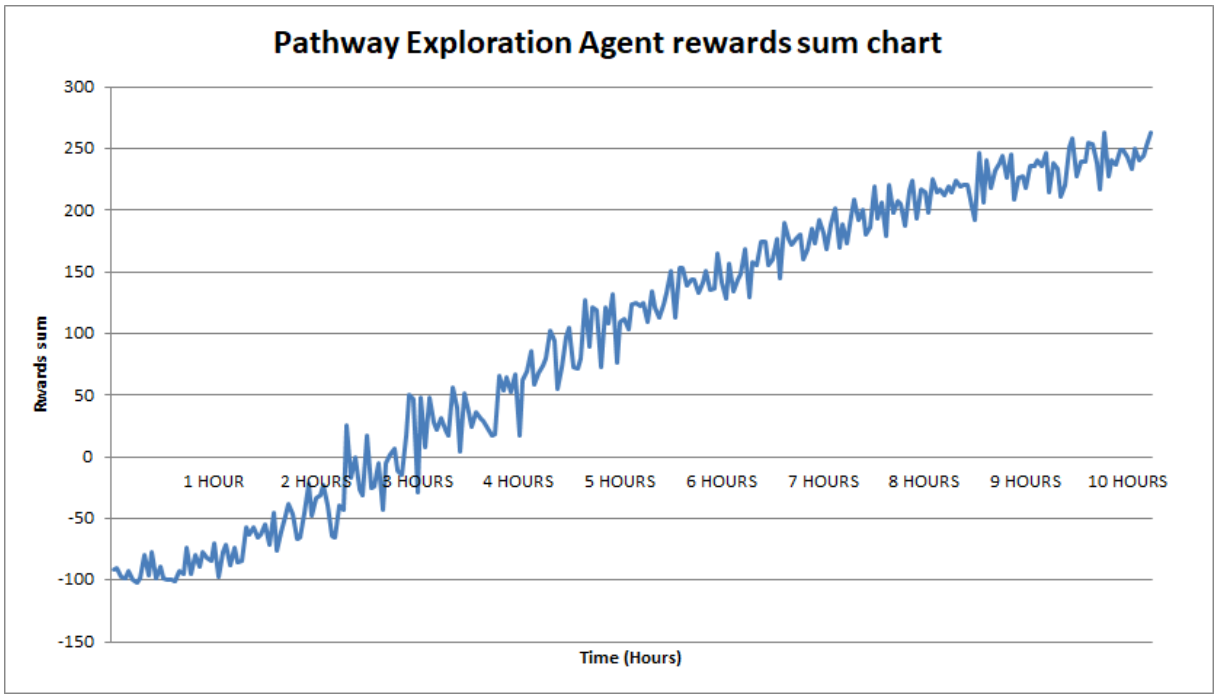

Figure 5.2: Chart that represents the variation of the reward sum during the training of Pathway Exploratory testing agent.

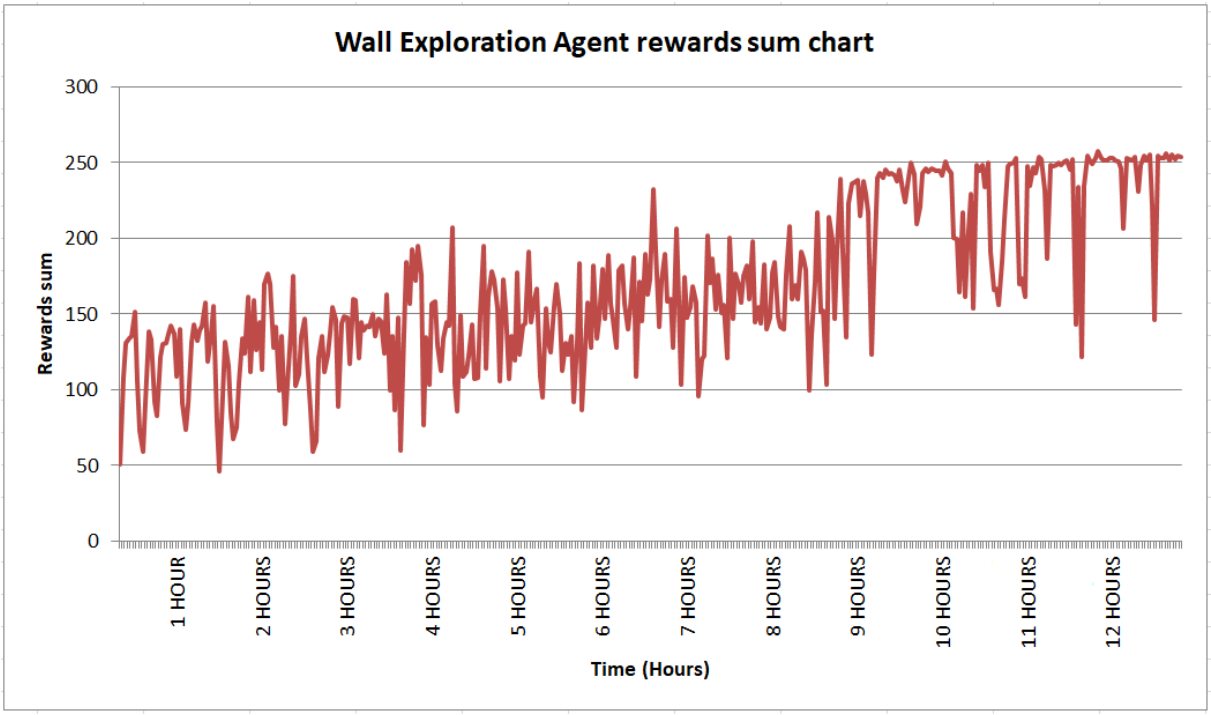

Figure 5.3: Chart that represents the variation of the reward sum during the training of Wall Exploration testing agent.

From observing the results we can easily see that the Pathway Exploratory agent rewards sum is growing steadily during the 10 hours of training and it seems to start stabilizing after that period of time, maintaining the same reward sum for some time. This means that the agent is learning what actions maximize the reward value at each time step and therefore we can conclude that it is learning how to achieve a specific objective in the game environment by trial and error, using the algorithm presented in Chapter 4. Although the Wall Exploration agent is capable of learning the behaviour we want it to execute, which is to walk close to the environment walls and eventually reach the objective, the learning chart represents an irregular reward sum line that grows slightly over time. We also found that this agent took over 10 hours to stabilize the rewards sum, meaning that its training is not efficient. For us, it is obvious that the Wall Exploration agent is capable of reaching the objective, but not in an efficient way, meaning that this agent doesn't find the objective location repeatedly in each map module while moving close to the environment walls. Taking into account the results, we state that the Pathway Exploratory agent utility function is well crafted and that the agent is capable of learning the exact behaviour we want it to perform. Comparing the two agents reward sum growth, we can confirm that the Wall Exploration agent doesn't show the positive results the Pathway Exploration agent does, since its chart doesn't grow over time to a point where it stays relatively constant. This means that the RL policy is not getting well defined by the agent and therefore we conclude that it is not easy to craft a very specific behaviour for the agent to execute that complements two different objectives, walking close to the environment walls

and reach the objective. After importing the trained agents neural networks models and execute both agents in the environment, we found out that the Pathway Exploratory agent takes around 7 seconds to reach the objective, while the Wall Exploration agent time to reach to objective is very irregular, taking between 1 minute to an undetermined amount of time to reach the fortress gate.

During the experiment described in experimental scenario 2 we were focused on understanding if it's possible to handcraft agents that execute different behaviours while performing tests in the game environment. We focused on tests that explore the game environment and find an available path to each map module objective location while checking for problematic areas in the case study. In Figure 5.4 and 5.5 it is visually described the path that the Pathway Exploratory agent and the Wall Exploration agent perform in the first map module while executing the functional test with the previously trained RL agents. In Appendix C it is presented the path performed by both agents in all of the map modules.

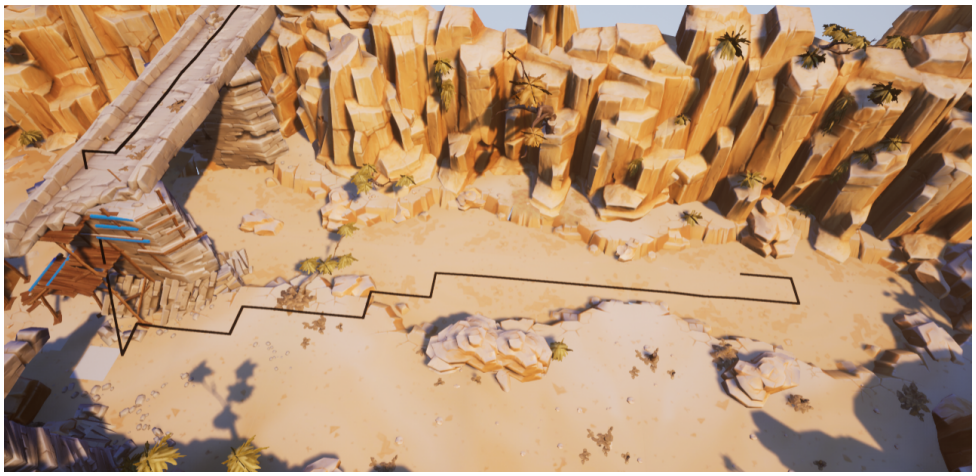

Figure 5.4: The black line trace in the figure shows the path performed by the Pathway Exploratory agent to achieve the first map module objective.

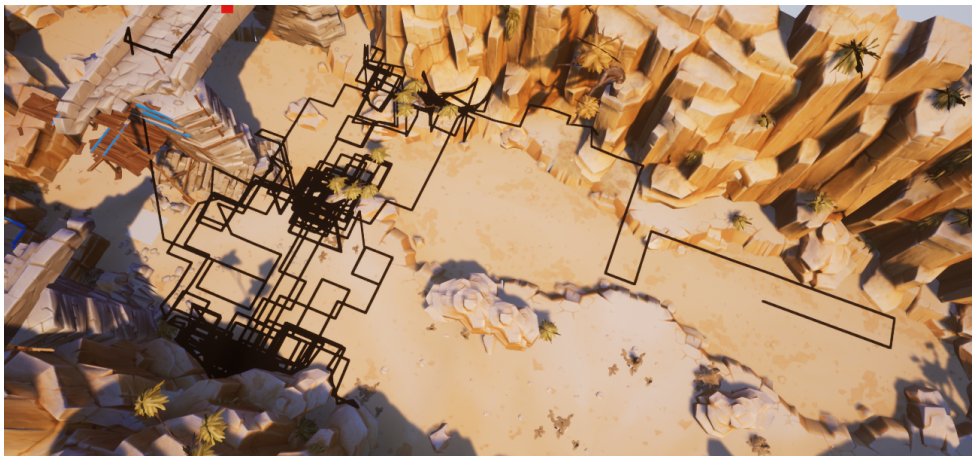

Figure 5.5: The black line trace in the figure shows the path performed by the Wall Exploration agent to achieve the first map module objective.

From observing both pictures, we can see that both agents path is totally different. The Wall Exploration agent behaviour is erratic and struggles to find an available path from the starting location to the fortress gate, while the Pathway Exploratory agent moves directly to the objective in the best possible way. However, the Wall Exploration agent still tries to move closer to the Actors that the line traces hit, while the Pathway Exploratory agent doesn't. This results prove that it is possible to handcraft agents that produce different behaviours, but we consider to be difficult to craft two agents that achieve the same objective efficiently while behaving differently. The reward system we prepared might not be the best for this type of agent, since it seemed confused about what actions to execute, not understanding how to maximize the rewards sum over time consistently. During this experiment we were also interested in assessing which agent found most of the problems introduced in the environment, while training to achieve the game objective. Normally, agents don't find these problematic locations after they are trained to achieve the objective, since they are not trained to find issues in the environment. The Wall Exploration agent is the only one that can find problems after being trained, however, it always finds the same problems as it did during training. The agents found most of the introduced problematic areas because they are exploring the environment during a considerable amount of time while training. During training, both agents find the problematic locations shown in Appendix B in Figure B.1, B.3, B.5, while the Wall Exploration Agent also found the second place where the player can leave the game environment boundaries, represented in Figure B.2 and the Pathway agent finds the problematic area presented in Figure B.4. After this experiment, we can conclude that we are capable of creating two agents that behave differently from each other and both are capable of finding almost all of the problems introduced in the game environment. However, only the Pathway Exploratory agent can efficiently find a path to the objective after being trained and their neural network model loaded to perform the test. We conclude that both agents are suitable for performing exploratory tests to find the type of problems introduced in the game environment while training, while the Pathway Exploratory agent can be used to find an available path for the objective.

	Times stuck	Times they exited the playable area
Player 1	2	1
Player 2	1	0
Player 3	1	2
Player 4	1	1
Player 5	2	2

Figure 5.6: Table demonstrating the number and type of tests found by each player that performed the playtesting session.

During the 3rd experimental procedure we want to compare the problems found by performing tests with RL agents, with the issues that humans found in the game environment during manual playtesting. Since we already collected information about the number and type of problems found by each agent,

we had to perform a playtesting session with human players to collect results for comparison. I inquired multiple people, at random, to perform this manual playtesting session and 5 people that regularly play videogames were submitted to a 30 minutes playtesting session that we attended and observed closely, asking the players to explore the game environment and report any problems found on it. They are asked to progress on the level until they reach the objective location. Results show that every player is capable of reaching the fortress gate and the problems each one found is presented in Figure 5.6.

The questionnaire results show that players found most of the problems introduced in the game environment but can't find all of them by themselves. The 5 playtesting sessions ended up being enough to find all of the problems in the game environment, finding the same problems that agents did. Although human testers have shown good results regarding finding these problems, they felt really frustrated in performing this type of test. Most of them, after 10 minutes of gameplay felt that there wasn't anything more to do in the game environment and wanted to stop the playtesting session. Only one human tester (Player 5) performed the playtesting during the 30 minutes and, not surprisingly, was the one that found most problems. When asked how they felt while performing the playtesting, their answer was mostly that they were pretty bored due to the fact that there aren't any game features besides the climbing mechanic. Their opinion is that finding problems in the environment is not something they are willing to do because it doesn't imply exploiting a game feature to find bugs, a task they say that would be more fun for them. While observing the playtesting, we found that players moved closer to the environment walls in order to find the environmental problems. That was a great discovery, since we tried to create a agent with similar behaviour. Last but not least, we found that people are not effective nor efficient to perform this type of test because it is a repetitive procedure without any meaningful reward. They show that their attention span while performing the playtesting is short, and most of them decided to stop playtesting.

In Chapter 6 we present the conclusions that arise from the analysis on the results shown in this chapter. Taking into account the results and the research done, future work is presented as possible upgrades to the Deep QL-APF test framework.

6

Conclusion

Contents

In this chapter there is a summary of the work produced during this thesis and the conclusions obtained from the results that were achieved. To finish off, we present some possibilities on how to improve the proposed playtesting framework.

6.1 Conclusions

This thesis work began with the intent of understanding if it was possible to offer an automated playtesting framework capable of automating certain types of tests that could reduce resources, such as the time and human resources needed to assess the quality of the game environment during the development of games made in UE4. Our intention is to use automation tests to verify if the player can navigate through each map module and achieve a certain location in the game environment (game objective). Besides the importance of verifying traversability, it is also worth to understand what problems the player may encounter while exploring the game environment to achieve the game objective. We want to deploy agents in an UE4 game environment and make them capable of navigating between two points to achieve certain objective on the map. Besides this, we aim to develop methods capable of detecting different paths to the game objective. During the research we did, we found different works that intend to reduce this type of resources, providing different solutions on how to do it. We hypothesized the use of RL agents to perform tests that can offer relevant information to developers and designers about the game environment quality, assessing if they could be used as a solution for performing exploratory tests that find a way of achieving the game objective while presenting different ways of doing it. We found that the Unreal Automation System and the Gauntlet Automation Framework could be used to automatically execute this type of tests in the UE4 without the need of developers doing it.

In order to test the solution, we created two different agents that execute different behaviours when trying to find a path between the initial location and the game objective (fortress gate). The Pathway Exploration agent is tasked to find one of the most efficient paths to the game environment, while the Wall Exploration agent is tasked to find an available path through each game environment while moving close to the environmental walls.

During the presentation of results, we show that Pathway Exploratory agent is able to learn how to find one of the most efficient paths to reach the game objective. We also provide results showing that both agents can find problems in the environment while they are training. Although the Wall Exploration agent is capable of performing a different behaviour apart from the Pathway Exploration agent, we found that it isn't capable of achieving the game environment consistently. However, it is clear that it can still explore the environment while trying to achieve the game objective, and results show that they find a similar amount of problems in the game environment constituents. Comparing the results of the problems found by manual testers with the problems found by agents, we can conclude that RL agents

can be used to replace humans performing this type of tests. Also, by assessing the questionnaire done in the context of this thesis, we conclude that human players are not interested in performing this type of test on the game environment, explicitly saying that automated tests should be used in these cases.

In light of what has been said above, we have achieved the contributions proposed during this thesis proposal. We can also state that the platform needs an update before being used in a game environment, including the improvements suggested in the future work. However, taking into account the results, we can say that reinforcement learning has the potential of being used to test game environments. Given the state of the art, RL proves to be versatile enough to perform different types of testing, such as a test that checks how many resources a player can obtain in a specific game environment. Our test automation platform demonstrates this potential, but, it was difficult to create different behaviors to achieve a specific goal. In the next section we offer possible improvements to the playtesting framework.

6.2 Future Work

We believe that there are still some aspects that need to be addressed, and for that reason we intend to explore them on future work.

One of the topics to address is the generalization of neural networks to similar game environments. Agents can be trained without overfitting to the game environment in which they were trained, and this makes them available for solving multiple RL problems as 1 test for several similar game environments.

Curiosity/Novelty Search was not used in the context of this thesis because the idea of this thesis is to lay down the foundations for using RL to perform functional tests in UE4 games. However, it is something interesting to deepen and explore in the future. It opens the opportunity for generating different behavior policies automatically while removing the necessity of handcrafting agents, a task that was proven to be difficult for complex tasks.

As a way of adding value to the framework capacity of creating different types of RL tests, there should be an assessment on how new mechanics could also be trained and added to the agent behaviour. The climbing mechanic is crucial for progressing through the environment, and should be developed further in time since it would make the test automation platform more complete. Using agents trained to perform the different mechanics needed for the case study requires the identification of zones where the agent must perform each mechanic. We recognize it would be advantageous for the developer to have access to a neural network model generally trained for climbing that could find not only different ways to climb but also problems regarding climbing. It was not implemented in the context of this thesis, since we decided to focus on the directional movement (WASD) of the character to explore the game environment. Another challenge I had during this thesis was to transform the discrete actions chosen by the neural networks into exactly the same input the player has access to, that is, continuous

input on the analog sticks to move the character around the game environment.

At the moment, the framework saves the locations where there is an environmental problem, but, for future work, the path the agent took to reach the problematic location should be saved and a visual representation of that path should be drawn in the game environment automatically. This way, the developer can easily identify the problem and how to reproduce it.

In conclusion, we believe it is possible to improve the framework in conjunction with a tests automation team in order to deploy a deep reinforcement learning playtesting framework for the production of a game such as the one Funcom ZPX is currently producing.

Bibliography

[1] S. F. Gudmundsson, P. Eisen, E. Poromaa, A. Nodet, S. Purmonen,B. Kozakowski, R. Meurling, and L. Cao, “Human-like playtesting with deep learning” in 2018 IEEE Conference on Computational Intelligence and Games (CIG). IEEE, pp. 1–8.

[2] 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 (visited on 06/11/2020).

[3] J. Baker, “Automated Testing at Scale in Sea of Thieves”, Unreal Fest Europe 2019, Unreal Engine. url: https://www.youtube.com/watch?v=KmaGxprTUfI (visited on 06/11/2020).

[4] P.Negrão, “Automated Playtesting In Videogames”, Master thesis in Computer Science and Engineering, FCT NOVA University of Lisbon, 2020.

[5] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis, “Generative Agents for Player Decision Modeling in Games”, Proceedings of the 9th International Conference on Foundations of Digital Games, 2014, pp. 1-8.

[6] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis, “MiniDungeons 2: An Experimental Game for Capturing and Modeling Player Decisions” in Proceedings of the Foundations of Digital Games Conference, 2015, pp. 1–3.

[7] C. Holmgård, A. Liapis, J. Togelius, and G. N. Yannakakis, “Evolving personas for player decision modeling” in 2014 IEEE Conference on Computational Intelligence and Games. IEEE, pp. 1–8.

[8] C. Holmgard, M. C. Green, A. Liapis, and J. Togelius, “Automated playtesting with procedural personas through MCTS with evolved heuristics”, in 2018 IEEE Transactions on Games (ToG), IEEE, pp. 1–10.

[9] L. Mugrai, F. Silva, C. Holmgåard, and J. Togelius, “Automated playtesting of matching tile games” in 2019 IEEE Conference on Games (CoG). IEEE, pp. 1–7.

[10] A. Soares, “Modelling Human Player Sensorial and Actuation Limitations in Artificial Players”, Master thesis in Information Systems and Computer Engineering, Instituto Superior Técnico (IST), 2019.

[11] Mnih, V., Kavukcuoglu, K., Silver, D. et al. “Human-level control through deep reinforcement learning”, Nature 518, 529–533 (2015). https://doi.org/10.1038/nature14236

[12] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of go with deep neural networks and tree search” *nature*, vol. 529, no. 7587, 2016, pp. 484–489.

[13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang,A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the game of go without human knowledge” *nature*, vol. 550, no. 7676, 2017, pp. 354–359.

[14] G. Lample, D. Chaplot, “Playing FPS Games with Deep Reinforcement Learning” *in Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2018, pp. 1-7.

[15] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, “Dota 2 with large scale deep reinforcement learning”*arXiv preprint arXiv:1912.06680*, 2019, pp. 1-66.

[16] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, “Curiosity-driven exploration by self-supervised prediction” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 1–12.

[17] E. C. Jackson and M. Daley, “Novelty search for deep reinforcement learning policy network weights by action sequence edit metric distance” *in Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2019, pp. 1-10.

[18] M. Bakhmadov, “IAP”, Bachelors thesis in Machine Learning, Norwegian University of Science and Technology (NTNU) , 2020, url: https://github.com/magomedb/IAP

[19] J. Kaniewski, “tensorflow-ue4”: TensorFlow plugin for UE4. 2019, url: https://github.com/getnamo/tensorflow-ue4

[20] L. Lin. “Reinforcement learning for robots using neural networks. Technical report”, DTIC Document, 1993.

Appendix

A.1 Automated Playtesting in Sea of Thieves

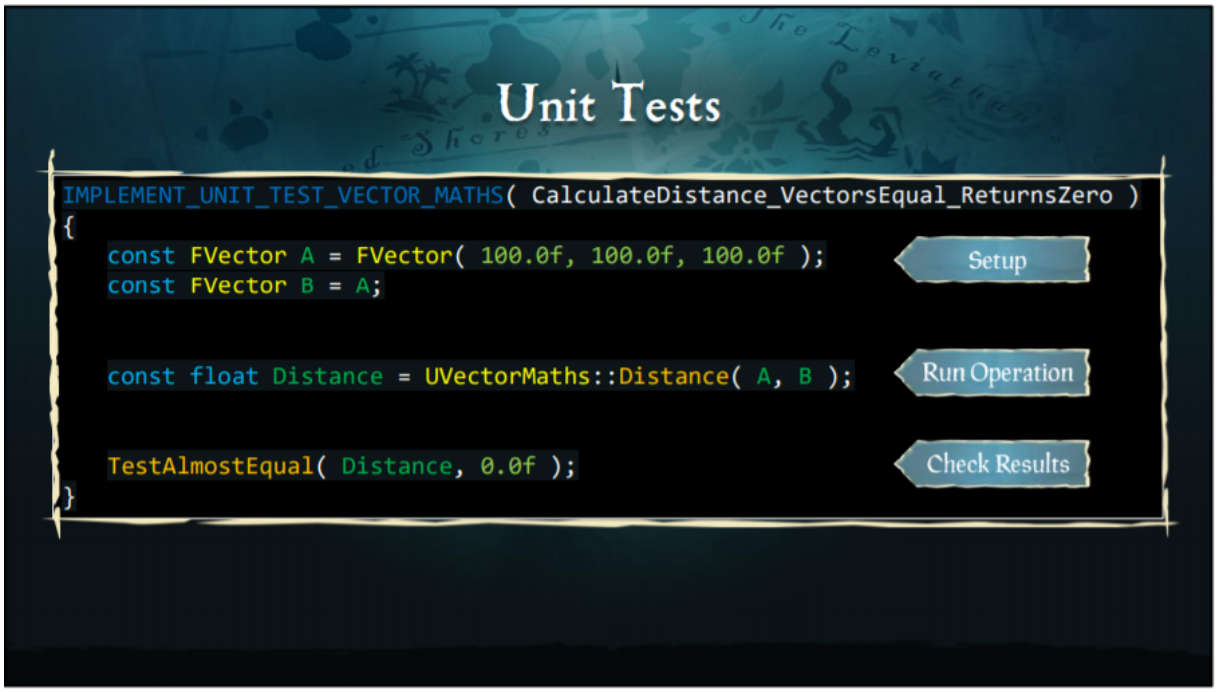

Figure A.1: Test that checks if a calculate distance function works corretly.

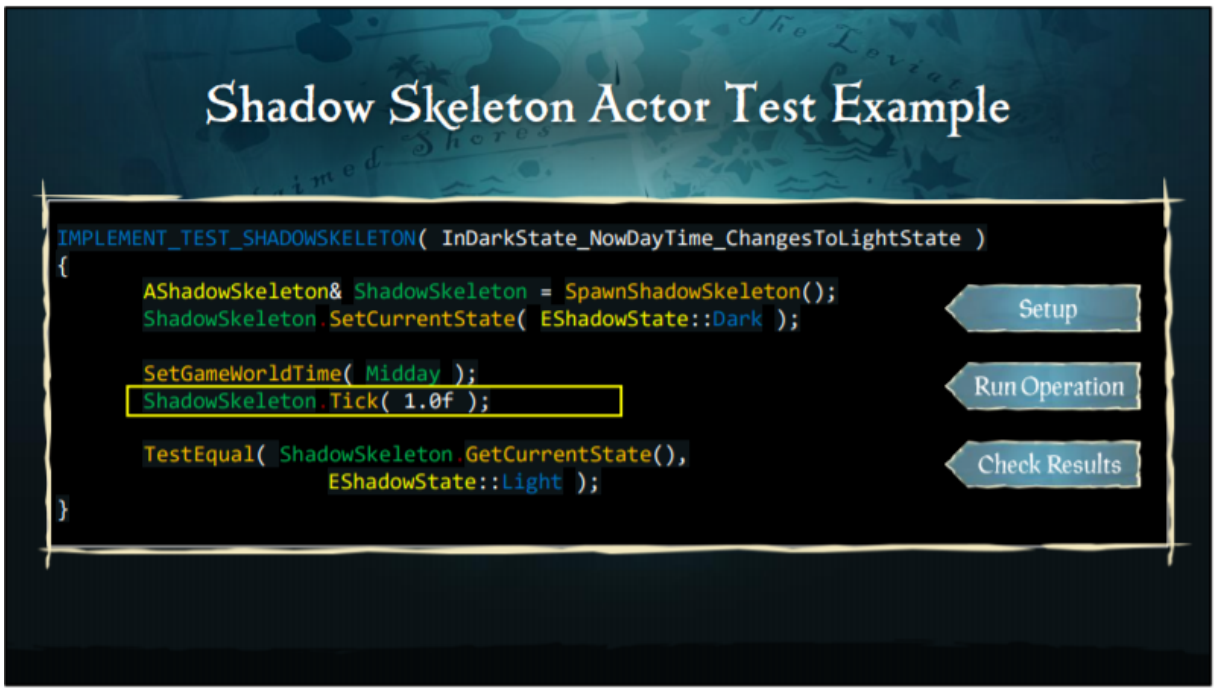

Figure A.2: Actor test.

B

Appendix B

B.1 Problems in the game environment

Figure B.1: First location where the character can leave the map. A well timed jump can move the character out of the map boundaries. It shows the agent line traces outside of the playable area.

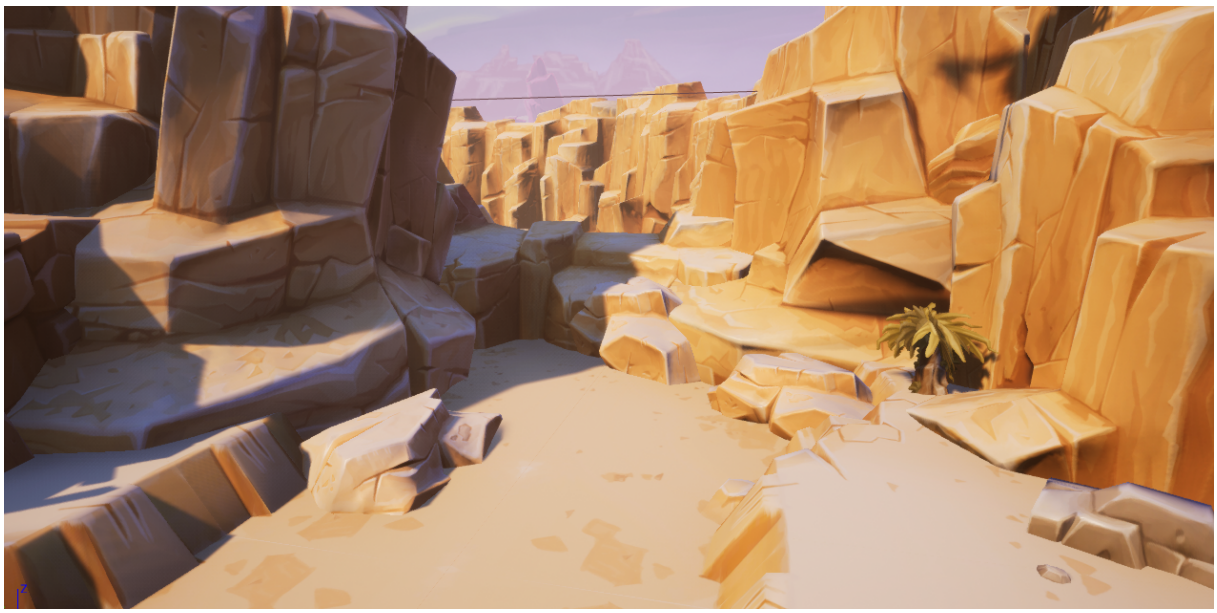

Figure B.2: Second location where the character can leave the map. The character can move from the right side and leave the map boundaries.

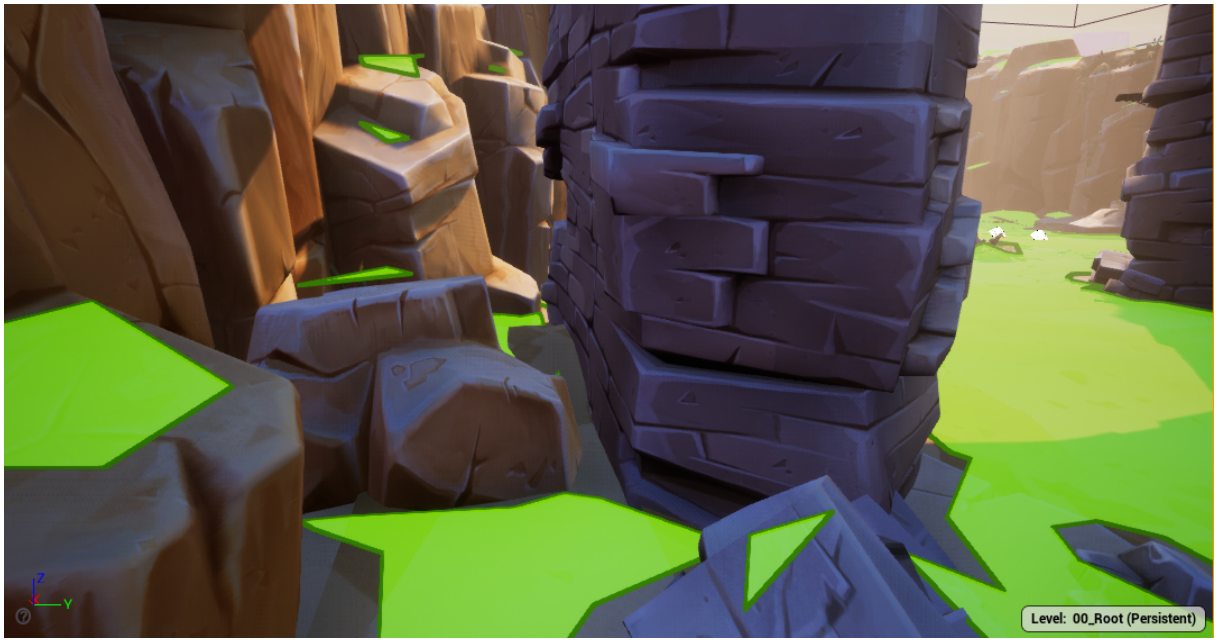

Figure B.3: Location where the player can enter but can't move because there's a rock blocking the passage.

Figure B.4: Location where the character can get stuck between the cactus and the wall when falling from the platform at the top of the figure.

Figure B.5: Location between the bridge pillar and an rock placed in the context of this thesis. It causes some animations glitches and the player can leave with some difficulty, while the agent cant.

C

Appendix C

C.1 Path executed by the crafted agents after being trained

Figure C.1: The black line trace in the figure shows the path performed by the Pathway Exploratory agent to achieve the second map module objective.

Figure C.2: The black line trace in the figure shows the path performed by the Pathway Exploratory agent to achieve the second map module objective.

Figure C.3: The black line trace in the figure shows the path performed by the Wall Exploratory agent to achieve the second map module objective.

Figure C.4: The black line trace in the figure shows the path performed by the Wall Exploratory agent to achieve the third map module objective.