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

Gabriel Fernandes
Instituto Superior Técnico, Porto Salvo
gabriel.fernandes@tecnico.ulisboa.pt

ABSTRACT

We introduce an approach to automate part of the playtesting process in games made with Unreal Engine 4, 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 Unreal Engine in order to perform automated gameplay tests. We propose a Deep Q-Learning 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[1]. 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 constituents. The agents can also identify problems in the game environment while they explore it. From the results we confirm that Reinforcement Learning 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 different 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, Actor Component, TensorFlow, Plugin, Exploratory Testing

1 INTRODUCTION

Performing playtesting sessions is a very intricate process with numerous iterations where game developers 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.

The game industry is looking for ways to increase the level of maturity and efficiency of Quality Assurance 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. King[2], the creators of CCS[3], researched the advantages of artificial intelligence over human-based playtesting[4], 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, drives Funcom ZPX to search for ways to automate and improve their playtesting. When developing a 3D adventure and exploration video game, Funcom ZPX finds important to verify if the player can navigate between two points in the game environment. By automating this type of test, we may be able to ease the amount of work that humans perform in testing and quality assurance. Therefore, we are motivated in developing artificial agents capable of exploring the game environment and reach a chosen position in it. As a case study for this thesis, Funcom ZPX provides an early-stage prototype of a 3D adventure and exploration video game made in Unreal Engine 4 to test the solution provided during this thesis work.

With the objective to ease at least some costs such as time and human resources, 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 also find important to understand what problems the player may encounter while exploring the game environment to achieve the game objective (at destination). 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. Besides this, the machine learning approach must be capable of learning how to achieve the objective in the best possible way, and also find other solutions with interesting parametrizations scenarios, in order to provide information about distinct ways of achieving that goal.

We intend to use Reinforcement Learning, specifically Deep Q-Learning, 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 Deep Q-Learning 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

[1]Funcom ZPX: https://www.funcom.com/funcom-zpx/

[2]King Ltd: https://www.king.com/

[3]Candy Crush Saga Game: https://king.com/game/candycrush

Master's Degree Thesis,
2021.

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. We want to deliver 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.

2 BACKGROUND

Rare [1, 9] provides information about the type of tests used during the development phase of an AAA game, and therefore we understand where to 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. This work also shows that the Unreal Automation System is a very useful tool to create and perform automated tests for Unreal Engine game, with P.Negrão work [11] 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. 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, and Unreal Automation System recognizes it automatically when associated with a Map Level test.

Several studies such as [3, 8, 10, 13], show that deep reinforcement learning can be successfully used to create agents that explore complex game environments to achieve goals. [3, 8] demonstrate that it is possible to create a reinforcement learning algorithm that uses neural networks to train an agent into maximizing some score and win the game. Their results comprehensively demonstrate that a pure reinforcement learning 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 [8], shows that it is possible to use Deep Q-Networks 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 [3] 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 Deep Q-Learning method to explore a 3D game environment and achieve well defined goals.

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 [14] 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[12] and Novelty[6] 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 reinforcement learning, 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.

The Unreal Engine plugin for TensorFlow, tensorflow-ue4[7], 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 [2] 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[4] coded for TensorFlow and based on the implementation of Deep Q-learning with experience replay made for "Playing Atari with Deep Reinforcement Learnin"[10]. 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 [2] 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 (as shown during M. Bakhmadov work). 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 players to 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.

3 PROPOSED SOLUTION FOR THE DEEP QL-APF

The playtesting framework produced in the context of this thesis is a plugin that can be easily added to every Unreal Engine project. The solution is set around the tensorflow-ue4[7] plugin since it offers the connection between the two elements needed to deploy a reinforcement learning agent in the case study game environment.

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

3.1 Deep QL-APF Model

In figure 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.

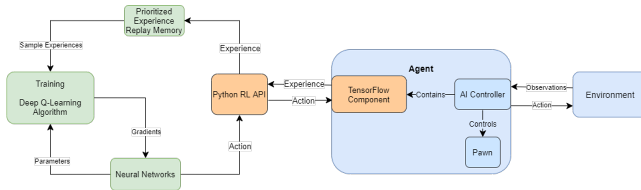

Figure 1: Model showing the Deep QL-APF main constituents and the information that is shared between them.

Following the Deep Q-Learning architecture by V. Mnih et al.[10], the green states represent the procedures a normal Deep Q-Network 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[5], which is the 3D character model that moves in the Unreal Engine 3D game environment. 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[7] 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 Deep Q-Learning algorithm. The TensorFlow component is added as a special sub-object to the AI Controller to allow information to flow to and from the reinforcement learning PythonAPI, allowing neural networks to execute the chosen action at each step in the environment. 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 Deep Q-Learning components are represented as green in figure 1. The PythonAPI is responsible for creating the python object that sets up the architecture for the Deep Q-Learning system. The Deep Q-Learning object is the main code file in the reinforcement learning 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, being also responsible for sending the best action to execute in the current state back to the PythonAPI.

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 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 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. 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 Deep Q-Learning 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

The Unreal Automation System is a test framework that comes with Unreal Engine 4 as a plugin that developers can enable to perform automated tests. 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 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 Unreal Engine 4 offers an object named Gauntlet[6] 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.

Setting up a test is done by placing a Functional Test Actor 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. We recommend watching this video[7] demonstrating an example on how to create, execute

[5]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.

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

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

and check the results of a Functional Test. 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.

4 DEEP QL-APF IMPLEMENTATION

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 1, so that we can deploy reinforcement learning in Unreal Engine 4, as well as a specific implementation for the types of tests Funcom ZPX wants to execute on the game environment. 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 an 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.

4.1 Algorithm Specifications

The Deep QL-APF playtesting framework provides agents that use a reinforcement learning algorithm to test the game environment. Being able to create reinforcement learning agents is part of the general architecture for this thesis solution. The algorithm implementation is offered by A. Raghuvanshi[8] and it was implemented similarly to the model presented in V. Mnih et al. work[10]. 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[7] 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 networks described in the paper [10]. 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. Figure 2 presents a structure similar to the neural network used as models for the agents offered in the context of this thesis.

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

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 3: Deep Q-Learning algorithm pseudocode.

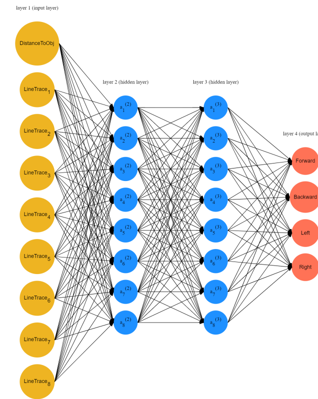

Figure 2: Architecture of the feed-forward neural network used in the Deep Q-Learning system.

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 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 3 below.

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.

4.2 Algorithm Preliminary Assessment

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

	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: 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.

and use it to perform tests, therefore the functionality of importing the module must work perfectly.

An overview of the neural networks architecture is presented in Figure 2. 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 Unreal Engine 4 Move to Location[9] function was used to move the character in each direction during the frames that are skipped. During the algorithm assessment, the rewards were calculated linearly based on the distance to the objective. 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 [2] 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 don't train the agents for the same tasks that M. Bakhmadov [2] did, but we take the structure and test it to perform exploratory tests on the game environment.

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. 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 8 frames between each algorithm iteration are the ones that yielded the best results because there are smaller fluctuations if compared with other frameskip assessment results. In Figure 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 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. 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

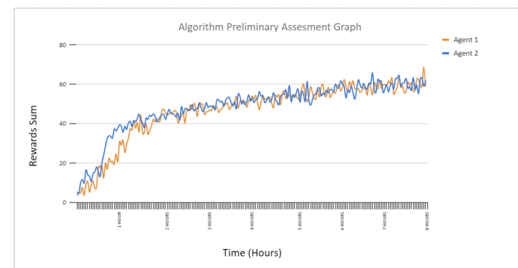

Figure 5: Graph showing the evolution of the rewards sum over time for 2 similar agents.

As a final remark, we can conclude that the algorithm is capable of training agents to achieve a well-defined objective in an Unreal Engine 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 reinforcement learning agents in the case study will take longer.

4.3 Unreal Engine Actor

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[7] 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 3. 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

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

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 reinforcement learning 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.

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**. Both of the derived AIControllers use the same actions for the different tests they perform, however, the reward system and the observations differentiate themselves. The purpose of Pathway Exploration 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 intention of the Wall Exploration 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. 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.

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 Unreal Engine 4. 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 Unreal Engine 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. 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. 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. 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 other Actor 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. These agents present 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.

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 6 demonstrate an example of this occurrence, where the player can jump off the bridge directly into the initial area.

Figure 6: 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[10]. When the agent starts colliding with this trigger, the objective and respawn point of the agent is updated through the Actor. 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.

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

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

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. The Gauntlet Automation System takes the build and runs the specified tests automatically. It can be used to verify certain features of the golden path, 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.

5 RESULTS AND ANALYSIS

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. 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 Deep Q-Learning 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[11] 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.

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

5.2 Experimental Scenarios

The first experimental scenario consists in the player character being 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 7 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 7: 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 Deep Q-Learning algorithm to find one available path to specific locations in the game 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.

The second experimental scnenario 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.

Last but not least, during the last experimental scenario we are going to compare the reinforcement learning 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 of the thesis document. 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.

5.3 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. 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, 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 8 and 9 we present the charts that display the variation of the rewards sum during the training of each agent.

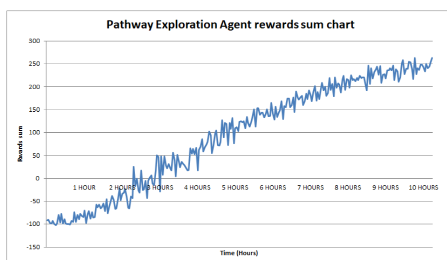

Figure 8: Chart that represents the variation of the reward sum during the training of Pathway Exploratory testing agent.

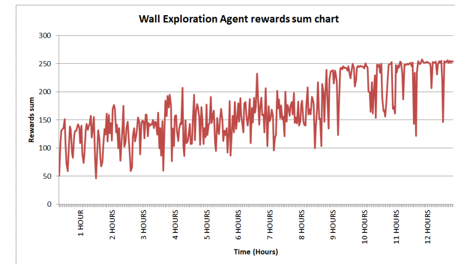

Figure 9: Chart that represents the variation of the reward sum during the training of Wall Exploration testing agent.

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 solution proposed. 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 reinforcement learning 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 10 and 11 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

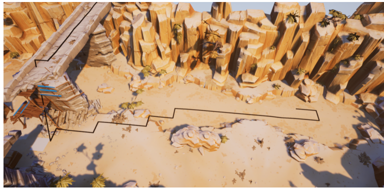

Figure 10: The black line trace in the figure shows the path performed by the Pathway Exploratory agent to achieve the first map module objective.

Figure 11: The black line trace in the figure shows the path performed by the Wall Exploration agent to achieve the first map module objective.

the previously trained reinforcement learning agents. In Appendix C of the thesis document, it is presented the path performed by both agents in all of the map modules.

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 together find all of the problematic locations shown in Appendix B of the thesis document. 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 12: 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 12.

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.

6 CONCLUSION

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 Unreal Engine. 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 efficiently. 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 reinforcement learning 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 have 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, reinforcement learning 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.

7 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 reinfrocement learning 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 reinforcement learning to perform functional tests in Unreal Engine 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. 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.

REFERENCES

[1] J. Baker. 2019. "Automated Testing of Gameplay Features in Sea of Thieves". https://www.youtube.com/watch?v=KmaGxprTUfI

[2] M. Bakhmadov. 2020. "IAP". url:https://github.com/magomedb/IAP

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Christopher Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *CoRR* abs/1912.06680 (2019). arXiv:1912.06680 http://arxiv.org/abs/1912.06680

[4] Stefan Freyr Gudmundsson, Philipp Eisen, Erik Poromaa, Alex Nodet, Sami Purmonen, Bartlomiej Kozakowski, Richard Meurling, and Lele Cao. 2018. Human-Like Playtesting with Deep Learning. In *2018 IEEE Conference on Computational Intelligence and Games, CIG 2018, Maastricht, The Netherlands, August 14-17, 2018*. IEEE, 1–8. https://doi.org/10.1109/CIG.2018.8490442

[5] Christoffer Holmgård, Antonios Liapis, Julian Togelius, and Georgios N. Yannakakis. 2014. Generative agents for player decision modeling in games. In *Proceedings of the 9th International Conference on the Foundations of Digital Games, FDG 2014, Liberty of the Seas, Caribbean, April 3-7, 2014*, Michael Mateas, Tiffany Barnes, and Ian Bogost (Eds.). Society for the Advancement of the Science of Digital Games. http://www.fdg2014.org/papers/fdg2014_poster_05.pdf

[6] Ethan C. Jackson and Mark Daley. 2019. 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, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, Manuel López-Ibáñez, Anne Auger, and Thomas Stützle (Eds.). ACM, 173–174. https://doi.org/10.1145/3319619.3321956

[7] J. Kaniewski. 2019. "tensorflow-ue4". https://github.com/getnamo/tensorflow-ue4

[8] Guillaume Lample and Devendra Singh Chaplot. 2017. Playing FPS Games with Deep Reinforcement Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 2140–2146. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14456

[9] R. Masella. 2018. "Automated Testing of Gameplay Features in Sea of Thieves". https://www.gdcvault.com/play/1026366/Automated-Testing-of-Gameplay-Features

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[11] P. Negrão. 2020. "Automated Playtesting In Videogames".

[12] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. 2017. Curiosity-driven Exploration by Self-supervised Prediction. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 2778–2787. http://proceedings.mlr.press/v70/pathak17a.html

[13] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nat.* 550, 7676 (2017), 354–359. https://doi.org/10.1038/nature24270

[14] A. Soares. 2019. "Modelling Human Player Sensorial and Actuation Limitations in Artificial Players".