Instituto Superior
de Engenharia
Politécnico de Coimbra

DEPARTMENT OF SYSTEMS AND COMPUTER ENGINEERING

The World of Anwin – Reinforcement Learning in Role-Playing Games

Project Report to fulfil the master's degree in Informatics Engineering
Specialization Area of Intelligent Data Analysis

Author

Gabriel Quintas Gomes

Co-Supervisor

Francisco José Baptista Pereira

Teresa Raquel Corga Teixeira da Rocha

Coimbra, July 2023

INSTITUTO POLITÉCNICO
DE COIMBRA

INSTITUTO SUPERIOR
DE ENGENHARIA
DE COIMBRA

ABSTRACT

The video game industry is vast and fast expanding, generating more revenue than the film and music industries combined. Likewise, the capabilities of machine learning algorithms keep improving and broadening, allowing them to recognize and understand increasingly complex patterns and, in some cases, respond accordingly. With this versatility and the prevalence of both areas, it is interesting to study different implementations of these algorithms and apply the findings to a new custom-built video game, specially made for this purpose. In this project, using the Unreal Engine 5, a video game was created with the goal of implementing a Reinforcement Learning model. The game will be a light Role-Playing Game with a story that can guide the player on a short journey. Many new tools were learned, and systems built to create an environment where the game could be built, all from the perspective of a novice video game developer. The combat system has a prominent role, with its agents all powered by the Reinforcement Learning that will be responsible for all decision the agent will make. The least developed side of current games being published is their Artificial Intelligence implementation, with this possibly being a viable alternative. In the end, it is important to achieve a good symbiosis between the game and the model built, to prove that an idea like this could be implemented on more mainstream games.

Keywords: Video game, Artificial Intelligence, Reinforcement Learning, Video Game Agent, Q-Learning, Unreal Engine 5.

RESUMO

A indústria dos videojogos é vasta e está em rápida expansão, gerando mais lucros do que as indústrias do cinema e da música juntas. Da mesma forma, as capacidades dos algoritmos de aprendizagem automática continuam a melhorar e a alargar-se, permitindo-lhes reconhecer e compreender padrões cada vez mais complexos e, em alguns casos, responder em conformidade. Com a versatilidade e prevalência de ambas as áreas, é interessante estudar diferentes implementações destes algoritmos e aplicar as descobertas a um novo videojogo especialmente criado com este propósito. Neste projeto, utilizando o Unreal Engine 5, foi criado um videojogo com o objetivo de implementar um modelo de Aprendizagem por Reforço. O jogo será um Role-Playing Game ligeiro com uma história que pode guiar o jogador numa curta aventura. Foram aprendidas muitas ferramentas novas e construídos sistemas para criar um ambiente onde o jogo pudesse ser construído, tudo na perspetiva de um programador de videojogos novato. O sistema de combate tem um papel proeminente, com os seus agentes todos controlados pelo modelo de Aprendizagem por Reforço, que será responsável por todas as decisões que o agente toma. O lado menos desenvolvido dos jogos atuais que estão a ser publicados é a sua implementação de Inteligência Artificial, sendo esta uma possível alternativa viável. No final, é importante conseguir uma boa simbiose entre o jogo e o modelo construído, para provar que uma ideia como esta pode ser implementada em mais jogos.

Palavras-Chave: Videojogo, Inteligência Artificial, Aprendizagem por Reforço, Agente de Videojogo, Q-Learning, Unreal Engine 5.

ACKNOWLEDGEMENTS

During the development of this project and during the master's degree, several people were instrumental for its completion, and I would like to thank them individually.

Thanks to my Parents for emotionally and financially supporting me and allowing me to pursue this opportunity.

Thanks to my Girlfriend for never letting me quit, always believing and making me believe in myself, and motivating me to always do better.

Thanks to my Friends for helping me with some of the ideas for the project.

Thanks to my two Supervisors and other Teachers that accompanied me, not only through the project, but also through the master's degree.

TABLE OF CONTENTS

FIGURE INDEX

TABLE INDEX

LIST OF ABBREVIATIONS, ACRONYMS AND SYMBOLS

AC	Armor Class
AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
CSV	Comma Separated Values
DDA	Dynamic Difficulty Adjustment
DFS	Depth-First Search
DL	Deep Learning
FPS	First Person Shooter
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HP	Health Points
HUD	Heads-Up Display
ID	Identifier
KPI	Key Performance Indicator
LOD	Level of Detail
ML	Machine Learning
NPC	Non-Player Character
RL	Reinforcement Learning
RPG	Role-Playing Games
RTS	Real Time Strategy
SL	Supervised Learning
TD	Temporal Difference
UE5	Unreal Engine 5
UL	Unsupervised Learning
UX	User Experience

1 INTRODUCTION

For years, Machine Learning (ML) applications have been increasing, expanding to domains like text, image, and sound. Although mostly used to train agents how to play video games, the examples in which Machine Learning algorithms are used to enhance a game's Artificial Intelligence (AI) are few. This is what this project is trying to tackle, exploring a possible AI implementation taking advantage of Reinforcement Learning (RL).

Most implementations focus on teaching agents rule systems of specific games to test how good they can become at playing them autonomously. This is an interesting concept, serving as a good benchmark for RL models, but ultimately does nothing to benefit the game. Ideally, a game would be created specifically with the intent of having the model aid in the experience, either from a purely AI training purpose, or actively making decisions in real time, augmenting the experience.

Even with several new models available, there are some types of games where it is hard to implement a RL model. Role Playing Games especially can have an incredible amount of depth and breadth, that makes adapting models to it harder. This is why most implementations are for simple games, shooters, or strategy games, where the technology can perhaps be best tested due to their number-based nature, where optimal strategies are rarely simple to determine.

The possibilities for these types of ML implementations are endless. Tools and systems can be built with these models that can aid developers to create amazing new game experiences. Procedural generation, automation in animation, creation of rulesets for agents, the possibilities are endless, and others will be explored later in this report. These tools in the hands of new developers can help proliferate and increase the number of small development teams, helping them realize their creative visions.

1.1 Motivation

In the scope of the final curricular unit of the master's degree in informatics engineering, the author decided to develop a simple video game in which a ML algorithm could be used to control the behaviour of adversaries within the game. This choice came about due to a passion to explore two areas of interest and see if they could be better combined to achieve something palpable for the average person playing a video game.

In theory, ML models are hard to implement real-time in a video game environment, but when that game has well defined states, these can be used to train an algorithm. If done correctly, a video game AI powered by an RL model, can pick what tactics to use and optimally determine its choice in every situation possible. An

implementation like this can also be taken further, with several different ways that it can change the way the mechanics of the game interact with the player.

This project is also a challenge to understand the complexities and intricacies of game development. Ever since its inception, gaming has garnered a massive following including the author, with great milestones still to cross. The medium is young and ripe for innovation and evolution. This together creates a crave for knowledge of how they are made and what tools are used to make them.

1.2 Objectives

The goal of this project is to create a video game and a model that's intrinsic to the playing experience. The game will be built from the ground up with this model in mind. It is meant to augment the AI of the game, granting it greater capacity to make decisions.

This game varies from the usual "learning how to play" approach that most ML implementations currently focus on, targeting instead an enhancement of the video game challenge and experience.

As previously mentioned, there is interest in understanding how video games are made at a professional level. In this project, the tools used to create the game are relevant and prolific in today's development environment. Gaining these skills and knowledge is something that's also considered to be an objective.

Without any previous knowledge of how video games are made, this project has a strong focus on learning new tools autonomously to create a video game environment where the RL algorithm can be implemented, trained, and tested. The tools created and the knowledge obtained can be used in the future to complete the video game and possibly publish it. Similarly, this project could be a steppingstone to a career in the video game industry, with some of the tools used being a part of many development studio's portfolio when creating a game.

In the end, the goal is to have at least part of a video game with a functioning ML model within that can optimally pick actions for the agent it is controlling, creating challenging scenarios for the player to confront. Although hard to compare, the model must be able to rival traditional rules-based systems that are normally present in today's video games. This will be measured using several types of custom-made agents.

1.3 Contributions

The creation of the project must contribute to something to be worth doing. Apart from the gathering of knowledge on the side of the developer, it also tries to create something that's, until now, rarely been attempted: Use RL models within a

video game. The project in this area is one of few to try this and may serve as inspiration for better models and implementations in the future. This area is still very unexplored, with good reason due to the computational demands of most RL models, but it is something worth exploring, even if the results are bad or the game is simplified to accommodate the model.

It also should prove that with time and perseverance, a game can be created by a single developer using a set of free tools. Although the game is not complete, it is feasible to do so in the foreseeable future, with a strong base and systems built here to support that future endeavour.

1.4 Structure

The rest of this document is structured as follows. Chapter 2 presents a background to concepts surrounding Machine Learning and Video Game Design. Chapter 3 describes the related work studied. Chapter 4 gives a broad overview of the game being developed while also explaining the fundamentals behind the engine being used to develop it. Chapter 5 is where the process of development will be explored in depth and in Chapter 6 the Machine Learning implementation used for this game will be explored. Chapter 7 will evaluate the results of the AI implementation versus conventional approaches. Finally, Chapter 8 will explore possible future work and the conclusion to the project.

2 BACKGROUND

During this chapter, several important concepts will be introduced and explained. These concepts are crucial to the development of this project and thus are required to properly explain and discuss the work.

The two major topics that this chapter will cover is Machine Learning and all its variations and intricacies, and Video Game design. Both research fields were important for the author to study so that the development process flowed better, making up for any knowledge gaps existent before the start of the project.

2.1 Machine Learning Overview

To describe the development process of the game's model and data structures, some concepts need to be explained. Most of these are essential for understand the development process, while others are just meant to grant the reader a broad overview of the subject matter.

Machine Learning, as stated in [1], is a branch of Artificial Intelligence and Computer Science which focuses on using large quantities of data to gradually train algorithms that can then perform complex tasks fast and efficiently. These can vary from Natural Language Processing (NLP), which is learning how to interpret text, to image classification or facial recognition algorithms. ML is commonly separated into 3 subcategories just like explained in **Error! Reference source not found.**.

Figure 1 – Visualization of the subcategories of Machine Learning (made by author).

2.1.1 Supervised and Unsupervised Learning

Supervised Learning is a subcategory of Machine Learning that uses a "labelled dataset", which is a set of data with a label attached to each instance that identifies what it is or what it represents. This dataset is then used to train algorithms that classify data or predict outcomes, with a human supervising the training methods ensuring that the results are accurate, and if not, making changes to correct them [2]. There are two major limitations with this approach. It requires a large amount of data to train and is very dependent on the quality of the data provided. This quality is determined by the relevancy and veracity of the data. If the quality is

bad, then the training process is intrinsically flawed and does not accurately represent the case study. This phenomenon is commonly referred to as "Garbage in, garbage out".

Although not as relevant to this project, there is also Unsupervised Learning (UL). This differs from SL in that it is used to analyse and cluster unlabelled datasets. These types of algorithms discover hidden patterns in the data without the need for human intervention (unsupervised). Their ability to discover similarities and differences between data make them the ideal solution for exploratory data analysis [3].

2.1.2 Reinforcement Learning

Another subcategory of Machine Learning is Reinforcement Learning. This is the more fascinating one, especially when applied in dynamic applications where the environment is not static.

RL is focused on learning through behavioural reinforcement. It does this by mimicking the way humans learn most things in life: trial and error, with positive and negative feedback along the way. In RL these are called rewards and they're at the core of how each model learns. The algorithm is trained on how to make the best sequence of decisions based on their attributed reward. With the goal of maximizing these rewards, the agent being trained learns what are the best choices in an uncertain and potentially complex environment [4].

RL models have a few key concepts that need to be explained before proceeding, since they will be used all throughout the report. These are:

- Agent – Represents the entity that is being trained.
- Environment – It is the surroundings with which the agent interacts.
- Policy – Determines how an agent behaves at a specific point in time. Thought process behind the decision-making.
- State – Abstraction of the situation the agent is currently on. Represents where and how they are in the environment.
- Reward – Represents the feedback the agent gets for picking a certain action in a certain state.
- Value Function – Measure of how great the cumulative reward granted to the agent for reaching a state is (state-value functions) or how beneficial it is to perform a certain action from a given state (action-value functions).
- Environment Model – Mechanism that mimics environmental behaviour and enables predictions of how the environment will respond.

In Figure 2 a visual representation of how some of these concepts interact can be seen.

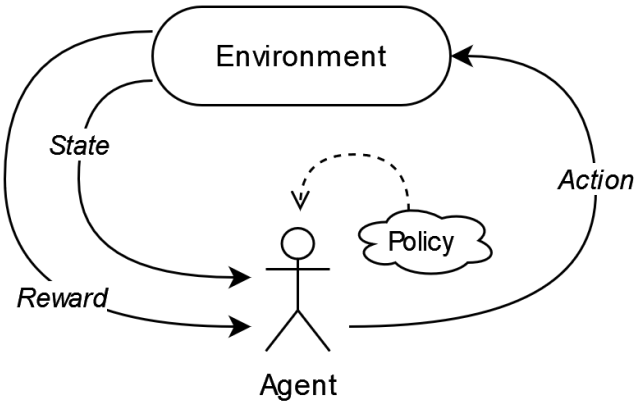


Figure 2 – Example of the Reinforcement Learning feedback logic (made by author).

These types of algorithms can learn without the need for large quantities of labelled data, removing one of the limitations of SL. Even then, there are still some limitations that can't always be worked around. The main one is that the reward and penalty system isn't always possible to be created. This is a recurring issue and happens when an environment does not allow for a system like this to be created due to its unknown properties. Not all actions in all spaces can be mapped to a reward or penalty, making the learning process impossible. Another limitation that can affect learning is the use of sparse reward or long-term rewards. These rewards are spread too far apart, and the model has a tough time understanding what's the best decisions to make to maximize the reward.

This is more applicable to real world scenarios rather than video games, given that the latter has pre-defined rules as well as win and lose states, but it still has problems. A game where the only outcome that matters is binary (either winning or losing) can cause the model to struggle with learning. In these games, if the only reward or penalty is at the end, it will (usually) take longer and be more difficult for the agent to learn how to play the game optimally.

Reinforcement Learning is becoming prolific in today's world. It is present in things like robot vacuum cleaners and autonomous driving cars. Both learn their own policy depending on their environment. For instance, an autonomous car's reward could be to keep the car within its respective lane, while the robot vacuum cleaner's reward could be the amount of dust it vacuums. These applications are ever increasing and diverging into new domains. The latest example of these models is the now famous GPT-4 from OpenAI [5], which uses RL as part of its training process to fine-tune the model to comply with its guardrails and the user's intent. The result is an extremely capable and context sensitive model capable of interacting with human written text prompts or questions.

Applied to games, the usual example is the AlphaGo project by DeepMind [6]. The team responsible taught a RL model to play Go by just making it play, knowing only the rules of the game. In the beginning it knew nothing but slowly it started to learn patterns and strategies, mastering the game, and defeating the best players in the world.

2.1.3 Markov Decision Process

With Q-Learning summed up, it is time to explore the concept of Markov Decision Processes (MDP). An MDP is a mathematical framework for modelling sequential decision making in situations where outcomes are partly under the control of the decision maker and partly unknown. Actions impact not only immediate rewards, but also subsequent states (and future delayed rewards). With these we can estimate the optimal value of each action in each state [7].

In MDPs, at each step, the agent selects an action available at the current state, and given the selected action, the system probabilistically moves to a new state and gives a reward to the agent. The agent's goal is to find a policy that maximizes its rewards over time. An example of an MDP can be seen in **Error! Reference source not found.**. As can be seen, when in a state, the agent can explore the environment using exploration policies, like random exploration, and then depending on the probability of each action, be rewarded accordingly. Using the MDP in the figure as reference, when the agent is in state S0, it can pick between 3 actions: A0, A1, and A2. If it picks action A0 it has a 70% probability of returning to state S0, receiving a reward of +15, and a 30% probability of going to state S1, receiving no reward.

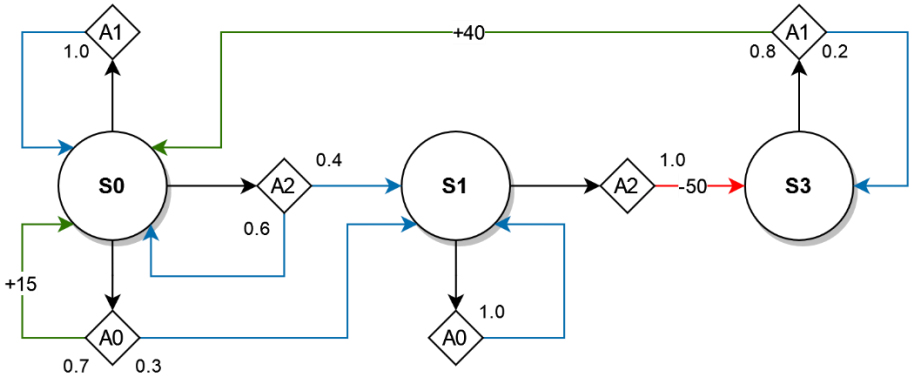


Figure 3 – Example of a generic MDP, with action probabilities and rewards (made by author).

2.1.4 Q-Learning

Now that the general Machine Learning categories have been established, there is the need to explore one of the variations inside the realm of Reinforcement Learning. This variation is called Q-Learning.

Q-Learning is a model free RL algorithm that is value-based. In a value-based approach, the random value function is selected initially, and then each value is calculated. This process repeats until it finds the optimal value function. Q-Learning is also an off-policy, temporal difference (TD) learning algorithm. The off-policy part means the policy being learned might be different from the one being executed by the agent. TD means that it learns from raw experience, only knowing the possible states and actions in each, and then stochastically exploring the environment. In other algorithms like Value Iteration, the agent knows all the

probabilities of each state transition, as well as the rewards associated with them. In a Q-Learning algorithm the agent does not know state transition probabilities or rewards. The agent only discovers the rewards after picking the action to be performed.

Value Iteration also has several limitations since the dynamics of the environment are usually unknown to the agent. There is also a problem with scalability for medium or large problems. The first one can be solved by Q-Learning since the agent does not need to know the environment to learn it but cannot totally solve the scalability issue. To solve that limitation, a Deep Learning algorithm is an alternative, but that's not relevant for this project.

A Q-Table is a data structure that's used to calculate the maximum expected rewards for action in each state, guiding the agent to the best action in every state. As seen in Figure 4, each Q-Table has a state-action pair that have corresponding Q-values. These values represent the sum of expected future rewards after selecting an action in a state.

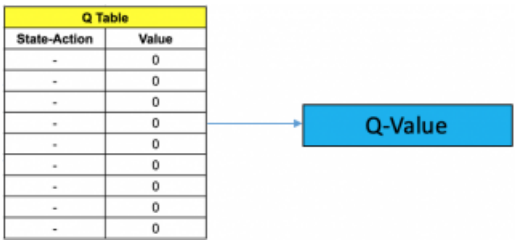


Figure 4 – Q-Learning with default Q-Table of state-action pairs and values [8].

To calculate Q-Values, the Q-Learning algorithm employs an "Update Rule". Before examining this rule, some other concepts need to be explained:

- Learning rate – Represents the size of steps taken towards the solution. The bigger the step the more it can learn new things, sacrificing the ability to learn smaller details.
- Discount factor – Represents the discount of the rewards. When it is close to 0 the agent is myopic and is only concerned with immediate rewards. Closer to 1, the agent will consider long term rewards.

With all Q-values starting as 0, the update rule seen in (1) is used to calculate the new Q-Value for a state-action pair by taking the old value (0 if it is the first calculation) and add the learning rate that is multiplied by the Temporal Difference error before. The TD error is computed by adding the next best estimate Q-Value, already multiplied by the discount factor, to the reward and then subtracting the old Q-Value.

$$Q^{new}(s_t, a_t) = \underbrace{Q(s_t, a_t)}_{old\ value} + \underbrace{\alpha}_{learning\ rate} * \overbrace{(\underbrace{r_t}_{reward} + \underbrace{\gamma}_{discount} * \underbrace{maxQ(s_{t+1}, a)}_{best\ future\ value\ estimate} - \underbrace{Q(s_t, a_t)}_{old\ value})}^{TD\ error} \quad (1)$$

The Q-Learning algorithm follows this sequence of events:

1. Initialize the Q-table.
2. Choose an Action.
3. Perform an Action.
4. Measure the Reward.
5. Update the Q-Table.
6. Return to step 2.

This process is repeated several iterations until the learning is stopped. During training, the Q-Table is updated, and the Q-Values maximized. Each state-action pair returns the expected future reward for that specific action, when in that state. When using Q-Tables to make the decisions, the Q-Values grant an easy way to determine which action is the optimal in each state. The highest value represents the optimal choice.

To sum Q-Learning up, it is an RL algorithm that "watches" an agent operate and gradually improves its Q-Value estimates that are then used to determine the optimal policy.

2.2 Video Game Design

This sub chapter will introduce some of the key concepts and difficulties that can arise when designing a video game as well as applying ML algorithms to it.

During the development there are several instances where an entity in the video game is referred to as an actor. This is another way of saying a video game agent except an actor does not need to have intelligence or make decisions. Instead, it only needs to exist and be something that can be spawned into a level able to have interactions with the player.

2.2.1 State Machines

There are several concepts that need to be explained. The first one will be what a game state is and why it is important to define it correctly. A state is something that all video games and applications have. Each has their own specific state machine since they are all different and have different behaviours. A state machine, also called finite state machine, is an abstraction used to design logic in applications. Based on a set of inputs or actions, the state machine tells what's the next state to transition to. A simple example of a state machine for an Android app fetching data can be seen in Figure 5.

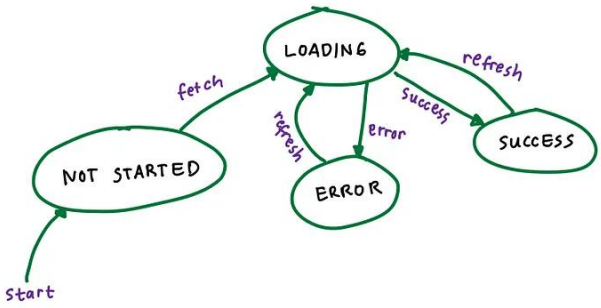


Figure 5 – Example of a simple state machine for an Android app that fetches data from an API [9].

The states that form the state machine are also an abstraction of a current situation the application is on. Focusing specifically on game states, they could be, for instance in a football game, "winning", "tied" or "losing", while in a racing game they could be the position of the player. The more complex a video game is, the harder it is to define a state. In RPGs for example, game states are hard to define with some concessions needed to be able to represent what is happening in the video game environment in state form. In these types of games, the number of states can also become very high, making state machines hard to construct and comprehend.

2.2.2 Meshes and Navigation Meshes

Another concept that will become important later is of meshes and navigation meshes. A mesh is the name used by developers and artist to describe a 3D object. It is a collection of vertices, edges and faces that are used to create the polygons that form the 3D object. This means that any 3D object seen in a video game level is composed of one or several meshes [10]. For example, a cube is a primitive mesh (primitive because it is one of the basic shapes), and a car is an amalgamation of meshes, derived from several shapes. This then leads into the concept of navigation meshes, also called "nav meshes" [11]. These are composed of a flat surface, with no collision with characters, that serves as an indicator to both the developer and the game's NPCs of where it is possible to navigate within the environment.

When it comes to animating meshes, it is common practice to use skeleton rigs. A skeleton is created on a model or mesh that requires animation. Like in the human body, this skeleton is composed by several "bones" that each control a different part of the mesh. In the case of a human body mesh there are bones for the arms, legs, spine, and head, among others. These can then be used to, with the help of animators and a concept of "keyframes", animate the in-game model of the human body. The keyframes are basically moments in time where the position of the skeleton is recorded. The animating software then calculates the transition between the current position of the skeleton and the one set in the keyframe using interpolation. With several keyframes it is possible to create complex animations that

are fluid and good looking. These concepts are all explained more in-depth in the following article [12].

2.2.3 Levels of Detail

Lastly, the concept of Levels of Detail (LODs) is also required to understand one of the major advantages of the graphics engine used to develop this game. Usually in most game engines, the assets are grouped into LODs, which are used to present information in a way that doesn't completely consume the memory resources of a graphics processing unit (GPU). When an asset is close to the camera of the player's controllable protagonist, the LOD is set to its highest quality, usually called "LOD-0". When that asset gets further away from the camera, the LOD will be "increased", and the assets changes to a less memory intensive asset, with lower number of polygons forming its 3D model. The distance at which the LODs change is determined either by the game engine or the developer and can range from LOD-0 to LOD-8. A visual representation of LOD changes can be seen in Figure 6.

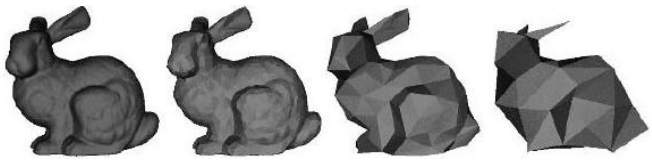


Figure 6 – Example of different LOD levels, from lowest (highest resolution and polygon count) on the left to highest (lower resolution and polygon count) on the right [13].

2.2.4 Environment Challenges

There are several elements that turn a video game environment into a challenging problem to tackle. In this subchapter, those challenges will be explored, both on the video game design side and on the integration with a ML algorithm.

Due to their real-time nature, video game agents are required to react quickly to any changes happening in the environment around them. In addition, any algorithm driving the in-game AI systems need to be high dimensional. Not only should they be able to account for a plethora of situations and potentially varying inputs at once, but they also must be able to perform at scale, possibly controlling large amounts of agents at any given moment.

The use of exhaustive search techniques when the AI is called for to decide on behalf of an agent, is not suitable for every game. A turned-based strategy game can afford to have an AI system that takes a bit longer to decide their next move, but in a racing game that is not the case, with inputs needed at every fraction of a second. It is also imperative that the proper research is done beforehand so that development effort is not allocated to the wrong algorithm.

One of the most important features of any video game is the fact that it is interactive and not static like a movie. This means that most of the enjoyment comes from interacting with the environment and non-playable characters provided by the

game, making the realism, dynamism, and consistency of all interactions a key performance indicator (KPI) for any video game's success.

The major concern and limitation with the use of ML in any sort of implementation is its high computational requirements, and this also applies to video games. Likewise, ML is also known for its training efforts taking a significant amount of time to complete. Any chosen learning technique should be able to perform within a specific period of time, especially if the algorithm in question is meant to be run real-time in a video game environment, as well as be able to perform with a limited set of resources, as defined by the game developer or game engine.

When it comes to teaching ML algorithms how to play a specific game it also has several difficulties to consider. Firstly, the variety of inputs can cause a problem, making the learning task more complex, as well as the setup process. Not only this, if the implementation is based on RL, there needs to be a reward system. To build a system of that type, it requires knowledge over the entire environment of the game, which is not always possible. Another major problem is that specific games have a reward system that is too sparse, meaning the agent does not receive a reward for a prolonged period. This usually requires a more complex incremental reward system, with intermediary rewards, to guide the algorithm controlling the PC on a good learning routine.

3 RELATED WORK

As technology advances and time passes, ML algorithms are becoming more common in everyday life's, with recent major strides in models like GPT-4 [5] and Dall-E 2 [14], both from OpenAI that have sparked major discussions and regulatory concerns about the evolution of this technology. In this chapter, a short overview of the different implementation of ML algorithms in all types of video games will be explored. The goal of this is to update the knowledge of the reader on what things have been experimented with, and what were the results of those experimentations.

For this chapter the author chose to focus on a qualitative research methodology. This choice derives from the fact that most of the information is presented in textual form, making it hard to condense. The data used to create the related work stems from websites like Google Scholar, IEEE and Scopus.

The use of ML algorithms in the domain of video games is nothing new. That being said, and like previously mentioned, the majority of current implementations are of algorithms learning how to play games. An example of this is when DeepMind, a subsidiary of Google, created the AlphaGo project [15] and later iterations to learn and defeat the best players in the world at the board game Go. Many companies have been using old games to test their machine learning algorithms, like Pong, Doom, Super Mario and Dota 2, as mentioned in [16]. This is achieved using RL algorithms where the machine learns how to play on its own, through trial and error, until it achieves success, being rewarded for good decisions at different stages in the training process.

The first implementations of ML in video games were SL. As the name implies, the Supervised Learning approach has the machine learn under the supervision of a human. This is a very effective way to teach a machine and usually results in great performance as long as there is a lot of data for the algorithm to learn. Herein lies the problem, the quantity of data. It is hard to provide this data *a priori* to the algorithm due to the ever-changing nature of a video game environment. In the rare scenario where that is possible, like exemplified in [17], the results can be good and sometimes even superior to a more conventional approach based on RL. In addition, document [18] also exemplifies a SL implementation that reiterates just how much the amount of data required limits the scenarios where it can be implemented.

Reinforcement Learning is the alternative to the earlier algorithms, and it is a much more versatile one, allowing machines to learn "on the fly" without the need for any pre-established set of data to pull from. This opens the door for numerous implementations of algorithms with the most varied end goals.

Creating a good overview of the different video game scenarios, documents [19] and [20] explain some of the challenges of different video game genres. Arcade games have it the easiest without any 3D movements, mostly used for AI

benchmarking. Racing games have continuous input (steering and pedals), requiring the RL algorithm to have at least a short-term reward and planning system, making the implementation challenging. First Person Shooters (FPS) have a large amount of visual input data that must be interpreted and understood. There are platforms like ViZDoom that allow machines to play the classic FPS shooter Doom. When it comes to Real Time Strategy (RTS) games, the challenge is significantly higher because multiple units can be moved at any time and the branching factor of decisions is typically enormous. This means that actions taken at the beginning of the game can have a major impact on the overall strategy, and it is extremely hard to create a reward system with these parameters. DeepMind and Blizzard partnered to create an API for RL to allow testing in games like StarCraft [21]. Lastly, the authors touch on two points that are very important to consider when it comes to RL in video games. First, the adoption right now is far more in the camp of teaching machines how to play like, or even better than humans, instead of being a part of the experience of the game. Second, there is a fine balance to be struck when it is implemented into the game's experience. A machine can, with time, play the game better than a human, making the process of playing against that machine not fun.

The authors of article [22] seem to have a possible solution for the problem mentioned in the previous paragraph. This solution is in the form of Dynamic Difficulty Adjustments (DDA). In it, the challenge is weighed against the player's perceived skill, dynamically adjusting the difficulty, preventing the game from falling into boredom for experienced players or frustration in the case of new players. This can also become a comprehensive accessibility feature for players with disabilities.

While exploring the Role-Playing Game (RPG) genre, the authors of [23] structured a very interesting model idea for controlling multiple agents in a grid. Using RL, they proposed an architecture where the several agents have a joint spatial representation, enamelling parallel exploration, assuring that experiences from one agent are immediately transferred to all others. This is an extremely interesting implementation of an algorithm that enhances the behaviour of the adversary and can improve the playing experience meaningfully.

The authors of article [24] propose an optimization of how the algorithms learn to control the agents. They suggest using Action Space Shaping to manipulate the reward given to the algorithm. With this, they can reduce the decision-making time, and optimize actions the agent can choose, thus resulting in a more efficient learning process. The basic idea is to give smaller intermediate rewards to the RL algorithm helping it converge faster and achieve an optimum policy [25]. There are 3 major categories of Action Space Shaping suggested but only one will be analysed here: Removing actions that are unnecessary or optional. This technique reduces the amount of training time the agent needs to learn a specific task. For this it must have domain knowledge of the environment as it may restrict the agent's abilities. This can happen in a situation where, for instance, the optimal action is optional, and thus removing it with Action Space Shaping being detrimental to the learning process. Shortcuts like these are meant to shorten the learning process of an agent when

teaching an algorithm how to play a specific game. The problem is that this does not scale to all games, due to their diversity. One specific Action Space Shaping is applicable in only one game. Also, if done poorly it can severely cripple the agent's ability to learn, especially when actions are removed from the option pool.

In [26], the authors challenged themselves to test the efficiency of conventional video game AI (planning agents) versus their RL counterparts (learning agents). The learning agents can be up to five times faster in certain scenarios than the planning agents. The problem with these learning agents is that they fail when the game is of a binary nature, having only win or lose options. This means they suffer due to long-time dependencies without any intermediary feedback to reward them for correct choices.

Further techniques like the one presented in [27] where the authors propose a way of using the actor's voice to create facial animations for their characters. In theory, these types of algorithms can detect emotion and intent in an audio performance, made by the voice actor ahead of time, and create an accurate facial animation for that dialogue. Likewise, the article [28] mentions that current developers are already using Machine Learning algorithms to create voice performances to use in their video games. Both implementations could be a great way of dealing with expansive video games, like Role-Playing Games (RPG), where there is a lot of interactions between the player and non-playable characters. This goes a long way in reducing the development time needed long term to populate the in-game world.

Another interesting implementation is based around the use of ML and video games to create a video game mechanism by which patients affected by multiple sclerosis can use their cognition to stimulate and exercise their motor ability, as exemplified in [29] and [30]. The participants in these studies agreed that the video games "increased their motivation and retention compared to their experiences with conventional therapies".

With the broader applications of ML in video games covered, there are also other important techniques to explore, within the RL subcategory.

The authors in [31] created a comprehensive classification and applications of Q-Learning algorithms. They analysed several Q-Learning applications from single and multi-agent scenarios. The most prominent were the single agent scenarios that incorporate the regular Q-Learning and Deep Q-Learning. There are also other methods like Double Q-Learning that strive the solve the over estimation of the next best value the regular algorithm has, and Hierarchical Q-Learning that tries to solve the issues that arise when the number of states becomes too high for the regular model, without needing a Neural Network like in Deep Q-Learning.

One of the major issues for developers when creating video game AI is that they are using routines that are pre-programmed to play a single game with a single set of rules. This, as determined by the authors of [32] consumes too much human resources as well as time, for a system that ends up being rather lacking. They say

that these agents usually present characteristics like "lack of planning", "lack of consistency" or "extreme accuracy" among others. These are all behaviours that deter from the experience of a player when witnessed. The solution proposed is to use Q-Learning in the video game Counter Strike, simulating the environment in a 2D map and using the normal Q-Table and Q-Learning update rule to train the agent to perform better than their counterpart built into the game.

Another set of authors in [33] proposed the use of a Reinforcement Q-Learning Deep Neural Network to create a model that was able to outperform human players and other models in the same domain at creating strategies to play the game. Their input method, although automatic, is a bit peculiar, with them using stills of the game (basically screenshots) to train the model. This was an interesting idea that has limited practical utility due to the demands imposed by a Deep Learning based algorithm.

This project will try to create a game where a RL model can be implemented. Unlike the majority of models analysed, this model needs to run within the game. This means that while operating at high resource efficiency, the model needs to remain powerful and able to determine the optimal choice for the agent its controlling. Some parts studied for the related work will aid in the conceptualization and creation of the required model, avoiding some problems other authors encountered. Lastly, this project aims to demonstrate that the concept suggested can be applied in real time scenario.

4 DESCRIPTION OF THE GAME – THE WORLD OF ANWIN

In this chapter the game will be described in detail, explaining all the research performed and exemplifying some of the inspiration for the initial conceptualization of the game's systems. All systems that were deemed necessary follow a specific formula and have a specific objective to achieve, allowing a developer to create a full video game. Lastly the objectives also need to be established, this serving as a future metric for the success of development and these are described in the chapter as well.

4.1 Conceptualization

To be able to start creating a video game there needs to be a concept from where to draw inspiration. With little experience there were several things to consider when conceptualizing the game, such as the complexity of systems and the amount of bespoke 3D assets and animations. To prove the viability of the concept proposed for this project there must exist a combat system, in which the RL model would run, that is easy to control, and that can be described well in a game state. There is also the opportunity to create a small and interesting fantasy story to propel the gameplay forward and justify the combat system's existence. To achieve this, systems would have to be built to present and track that story as the game is played.

With that established, the game will be a story-based Role-Playing Game (RPG) with dialogue options and a main short story to follow. During the game, players will have to fight a mysterious hive-mind like enemy that has the ability to learn how to best play in any situation. These would be the adversaries and they would be controlled by a RL model that learns the best way to play in any given situation.

The main inspirations for the type of game being made are the franchises "Divinity Original Sin" and "Pillars of Eternity", both in Figure 7.

a) b)

Figure 7 – Examples of games that served as inspiration: a) Pillars of Eternity [34], b) Divinity: Original Sin 2 [35].

In terms of art style, the game needed to be something simple and without much detail. This stems from the fact that there was no previous 3D modelling experience that could support the creation of fully custom-made assets for the game. This is one of the reasons for the choice of the game engine. As it will be explained later in the chapter, the game engine has a library of high-quality free assets and materials for any developer to use.

The game would have classes for the player to choose from, each with their special abilities and specificities. This would also be enhanced by a levelling system that allowed player to customize their character further.

For the combat system, a simple turn-based experience was the initial idea. This system would work on top of a map that would be custom built for each combat encounter. The entities engaged in combat would move in the environment either freely or in a grid, with a specific movement speed and actions per turn. There would also be abilities that the player can use in combat, allowing for a versatile combat experience. What and how many these would be is something that needs to be determined later in development.

Lastly the game needed a name to be referred as. This is not strictly necessary, but every game has a development name that is used to refer to it. The development name of this game was "The World of Anwin". This name stems from the fictional world it takes place, created by the developer, where the god that created the world is named Anwin.

4.2 Objectives

When embarking on the development of a big project like this one, it is crucial to establish what objectives are meant to be achieved. These objectives are meant to be a guide in all moments of the development process, serving as a constant reminder of what was set out to do in the beginning.

It is normal for the objectives of a project to shift slightly during development, after all it is impossible to predict every detail that will inevitably make things harder and challenge the possibility of reaching the determined goals. The approach to take when dealing with problems like this is to return to the original objectives and understand how a compromise can be made while still reaching the overall goals originally established.

Realistically with games taking years to develop with the efforts of hundreds if not thousands of developers, it is unrealistic to assume that the game could be finished during a year of solo development. This was already a difficult task on its own but there is also the need to learn how to use the Unreal Engine 5 and learn how to properly develop a video game. With these expectations in mind, it is certain that the game won't be finished by the end of the project's expected delivery date.

When it came to establishing the general goals for this project, they are defined as:

1. Understand the complexities of Reinforcement Learning techniques applied to video games.
2. Create a model that could be implemented within a custom-made video game using those techniques.

The complexity of these tasks is not to be underestimated, without extensive experience in both domains the likelihood of not reaching the full extent of the goals established is great, but the challenge is intriguing enough to be worth it. The second goal can then be separated into smaller, more palpable milestones to follow during development, already mentioned in chapter 1:

1. Create a short video game.
2. Create a combat system where a RL model can be implemented and tested.
3. Create the RL model and incorporate it in the combat system.

Part of the interest of this project is to understand how video games are designed and made from a "behind the scenes" point of view. The video game industry is a fast growing one, with even more room for expansion, so a career focused on the concepts that come into play within this project is a good possibility, provided the right opportunity were to arrive.

In terms of the implementation of the RL model, there will need to be a lot of research made in the area. As seen in the Related Work chapter, most of the work of Machine Learning algorithms in the domain of video games is in the "learning how to play" type of approach. This is either done with Supervised Learning, which is under human supervision and under constant tweaking of the models, or with Reinforcement Learning where the model is given the rules of the game and discovers the best actions for a specific agent within the game. The least explored area is the direct incorporation of a model in the design and architecture of the video game, and this is what this project's model will be aimed towards, understand how a video game environment can support a real-time model running its "Artificial Intelligence" algorithms.

Lastly it is important to clarify that the game being built for this project, if it all goes well and the quality of the product lives up to expectations, it could be a candidate for a publication down the development road. It is important to reiterate that this is not a goal for the project within the development timeline established, but it should be considered a baseline to allow for future work to improve and evolve the concept forward, possibly into publication.

When reaching the end or even during the development of the project, there is the unlikely conclusion that what is being attempted is not feasible. In this case it is important to recognize why the current implementation is not working and if there was another way of approaching the challenge that could have resulted in a better

outcome. Basically, the result of the project can be that an implementation of a RL model like this in a game isn't doable. This then must be properly documented and described in the conclusion and future work chapter to allow for any author that comes after to know that this approach is not feasible, and what could be done to make it work.

4.3 Required Systems

In this subchapter there will be a thorough explanation and description of the systems that were required to create the game. Some of these had to be custom made, others were already provided by the game engine or required some form of adaptation to work as needed. As a broad overview, these were the functional requirements that the systems had to fulfil:

- Mission tracking and progression indicators for the player.
- A mechanism through which the story could be delivered to the player. Being an RPG there was the need to have ways that the player could use to affect the story.
- An interface that could deliver the essential information to the player without being too cluttered.
- A way for the game to be saved and loaded so that multiple play sessions were possible without the loss of progress.
- A combat system that allowed the implementation of a RL model.

Each of the previous bullet points formed an idea of what systems would be needed to create during development. There are countless ways of implementing any one of those systems, so research had to be performed to discover what ideas from other video games could serve as inspiration for what was about to be constructed, serving as a reference point during development for what the system should be performing or looking like.

4.3.1 Quest System

The Quest System is a system that is meant to control and structure all the quests (missions) in the game. All RPGs have a system where the players can view and track their missions, being able to see the ones that have been completed. This type of system interacts with the player in two parts: The first one is the journal or quest log system where all quests are listed and detailed. The second one is the UI element that presents the currently active quest and the object to follow to progress it. Both systems have small examples in Figure 8. These two parts are the most important link between the game's quest system and the player, allowing players to

always check objectives and previous missions, removing the mental toll of, for instance, having to remember the story after a few days without playing.

a) b)

Figure 8 – Examples of quest systems. a) The current mission selected in the game "The Witcher 3: Wild Hunt" [36], b) The quest journal in the game "Dragon Age: Origins" [37]. (Images taken by the author).

Apart from the front end of the system there also needs to be a back end that can store and track the progress of missions. This is a system that can easily cause problems if it fails, and even many big budget games have problems with quest tracking, preventing progress from happening. The system that will be developed needs to be robust but simple to avoid problems when tracking quests. The less moving parts the system has, the less points exist where problems can arise.

4.3.2 Dialogue Boxes

One of the defining aspects of most RPGs is the ability to engage with Non-Player Characters (NPCs) during the story and having dialogue interactions with them. This is present in several games like the previously shown "The Witcher 3: Wild Hunt" and "Dragon Age: Origins". Once again, the inspiration for how to create an interaction system comes from these games.

Interactivity is something that the video games genre has over all the other entertainment genres, allowing for a deeper and more personal connection between a player and the character they are playing, going way beyond just the regular fun that's usually associated with playing a video game. RPG-like games prove that the interactivity factor can be something that doesn't get in the way of delivering an interesting or compelling narrative to the player. Instead, they can significantly enhance it allowing player to put their own mark on the story and in-game world through their actions. These actions are usually represented by choices made during gameplay, but mainly during dialogue options. There can also be decisions that do not affect the overall outcome of a story, and that exist merely to grant the player a sense of ownership of the character they are playing as. These choices allow them to

be smart, cunning, or even oblivious, with this freedom of choice being one of the crucial tenets of what a Role-Playing Game is supposed to be.

In Figure 9, there is an example of a dialogue system in the game "Dragon Age: Origins". Relatively old by today's standards, this game has a robust and easy to understand system that gives players several dialogue options to choose from, while allowing them to still see the last sentence that was said either by their character or NPC. This is a smart system allowing for quick reference to the subject of the conversation or the last things that were said before the dialogue choice appeared, meaning that even if they get distracted, they can pick an option without being totally clueless while doing it. The game previously mentioned is from 2010 and is a good reference point for how the dialogue system for the game in development should look and feel like to the player.

Figure 9 – Example of the dialogue and choice boxes in the game "Dragon Age: Origins". Dialogue sentences at the top, dialogue choices at the bottom. (Images taken by the author).

The major problem that can arise while developing a system like this is where the dialogue will be structured and how. Unlike quests and quest objectives, the amount of information that needs to be stored can easily escalate into uncontrollable numbers as seen in games like Starfield, yet to be released, which has over a quarter of a million lines of dialogue as reported by VG247 [38].

4.3.3 Graphical User Interface

Any game needs a good user interface to function properly. A bad interface makes the experience of playing the game a chore, increasing the level of frustration for the player, usually leading to complete disengagement from the video game or, in the cases where it does not reach that level, the avoidance of some game mechanics due to the cumbersome nature of navigating the user interface. This area of study is commonly referred to as User Experience (UX) and it has become more and more important as technology and games evolve.

Just like intuitive controls are required for a pleasant gaming experience so is a good user interface. In some games there are toggles to allow players to customize the amount of information displayed on the screen, since some of it can be quite intrusive and distracting. It is a fine balance to get it right, with the good games usually having a mix between a clean and a stylized interface of whatever intellectual property they belong to, without losing the practicality that should be intrinsic to every information displayed to the players.

Once again, "The Witcher 3: Wild Hunt" is a game that gets referenced and its user interface can be seen in Figure 10. It managed to strike a balance between practical and pretty without obstructing big portions of the screen.

As seen in Figure 10, the game has the player character locked to the bottom centre of the screen, allowing for a good view of the path that lies ahead for the player. The bottom corners are assigned to the control instructions, and it is the only thing that could be considered obtrusive while not being completely practical. For certain players it is necessary to always have the key prompts present on screen as a reminder but for others, they become intuitive and muscle memory after a few hours of player. Thankfully there is an option to remove these two prompts in the game's settings. Without those the only elements left on screen are the truly crucial ones: the health bar and the map and objectives.

The health bar as seen in the figure is a truly great achievement in UI design. Within the left corner of the screen the developers managed to cram so much information in an organized and visually pleasing way. The red bar signifies the health of the main protagonist while the white bar above it marks the progress the character has in the current level, so in that specific case, the character is halfway a certain level. The yellow bar is representative of the character's stamina and the black bar a way for the player to keep track of their toxicity level, something that is a part of the games' mechanics that aren't worth mentioning here.

On the right top side of the screen, another set of information is presented to the player, and it is all crucial for the play experience. While some player and designers argue that the use of a mini map detracts from the exploration experience, drawing the player's attention instead of what is in front of the main protagonist. This is a valid argument, but in a game as expansive as "The Witcher 3: Wild Hunt" it was necessary to guide the player. The same widget in the corner presents the weather and time of day, their current quest and quest objective as well as a waypoint for where they need to go. This is a very organized system, allowing for large quantities of information to be displayed to the player without much difficulty.

The UI that's going to be created for this project and this game will have massive inspirations from the interface seen in Figure 10, given that it is an RPG as

well, and will likely emulate the same health bar and quest trackers, maintaining their positions on the screen.

Figure 10 – Example of the complete UI for the game "The Witcher 3: Wild Hunt" [39].

4.3.4 Other Systems

In this subsection, a few other systems that are being considered will be explained, presenting a case for their existence and how they could be implemented. These systems are not crucial, and some are not even considered necessary, they are just part of a bigger vision to create an RPG that could eventually be taken to market after more development time.

Another factor of an RPG is their character creation and levelling systems. These systems are usually supplements of class systems, where each class has its own behaviour and place in the game's mechanical core, and it falls to the players to choose which one they want to play as, adding "replayability" (term used in the gaming industry to describe a game's ability to keep people playing it [40]) and depth to the video game. They can create a sense of ownership and customization to the character the player is controlling, allowing them to pick what their character can do and then actively being rewarded for it. This is called a positive feedback loop, when an action takes a long time to achieve, like for instance gaining enough Experience Points (XP) to unlock a new skill, the game then rewards the player by granting them a new fulfilling skill to use.

Though the game in development is not going to be complete, the game should have a small levelling system, that could then be built upon to create a more robust one.

Another system that should be created is a way to save and load the progress of the game. In other games there are a lot of implementations of a system like this, with some having an "auto save" feature where the game saves itself automatically in a certain period, for example every 10 minutes.

For this game, a simpler system would be enough. A way to pause the game and enter a menu to save the game. This leads to another necessity of the game which is menus and a way to enter them in the middle of the game, commonly referred to as "pausing the game". This would bring up a menu where player could save or load the game, but also exit it if he so chooses.

Another menu the game being developed should have is a main menu. This is the menu that would be displayed to the player upon opening the game, usually granting them the opportunity to continue their game by loading a save, creating a new one if allowed and change the options of the game like graphic quality and resolution. The latter one is unlikely to be required for this project, but the other menu options are important.

4.3.5 Combat System

The creation of the combat system is one of the most important steps to properly realize and successfully achieve the objectives previously set for this project. Like the other systems being built there are several ways to tackle the challenge, but the main requirement to keep in mind while developing the combat it that is must be able or even facilitate the use and implementation of the RL model.

There are several possible ways to go about creating a combat system for an RPG and it can be even daunting evaluating all the options available. The main purpose of the system being developed is not to be flashy or super animation heavy since there is no previous animation experience. If that was one of the objectives the goals as well as the time schedule intended for the completion of the project would need a total restructure.

The combat system of an RPG is very number, and statistics (stats) based. This means that there is a disconnect between the player's actions and what happens, since there are many background calculations between pressing the attack button and the enemy health going down. By default, it isn't fully transparent what is happening in the background and how the numbers are turning out, making failed attacks and abnormal damage numbers feel like the game is cheating the player. For these situations, and like it is shown in Figure 11 marked in blue, the game usually creates display areas that show the stats being rolled in the background, explaining why everything is happening and how the numbers are affecting the events in the combat. This is a crucial element to implement in the video game being developed, due to the possible lack of animations demonstrating what is happening.

Figure 11 also shows an interesting UI concept for the combat system that centres the main actions that the player can take in the lower centre of the screen and uses places like the top centre to display the current turn order, allowing for a clean way to display who's turn it is.

Figure 11 – Example of the combat system of the game "Pillars of Eternity 2: Deadfire" [41].

When it comes to the implementation of the combat and its mechanics there are a lot of ways to go about it. This is especially true with the movement system and how it is going to work with the Reinforcement Learning implementation. There are two main ways to create the player character movement. One is a free movement system, which is the norm for most video games, where the player can move freely through the game level without any restrictions. This implementation is aimed more to action-oriented games, that place the main character in a complex scenario and the player must use the mechanics to win the combat. This might cause some issues when it comes the RL implementation since it is likely to cause trouble with the representation of the player's location in the game world, but it is still a worthy option to try out. An example of a system like this is the game previously shown "Pillars of Eternity", although that game is not action oriented, since it works with a turn-based combat system. The alternative to the free movement system is a grid-based one. This works like many old tabletop RPGs like "Dungeons and Dragons" where the player character must move on a grid with square tiles, being limited on its movement per turn by a square limit. This has its advantages, namely the reduction in complexity of implementation and possibly an easier interconnection with a RL model. The disadvantages are that it is less intuitive and reduces the player's freedom of movement, something that some don't like. An example of a combat system like this is shown in Figure 12 with the game called "Into the Breach". When developing, both techniques should be tested to see which one works better for the goals established.

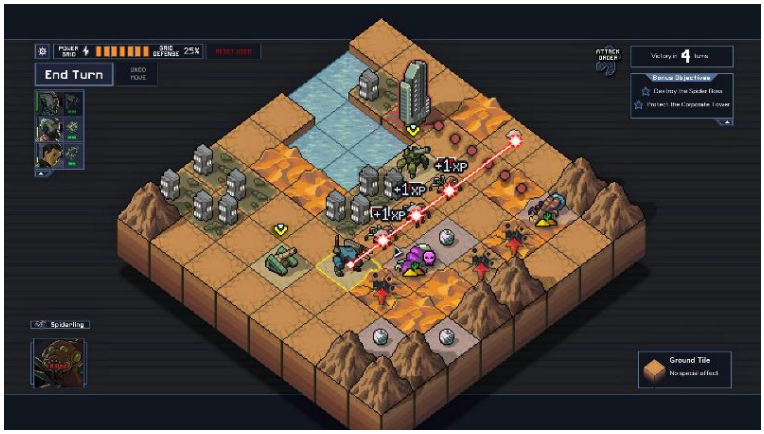


Figure 12 - Example of the grid-based combat system from the game "Into the Breach" [42].

4.4 Unreal Engine 5

The choice of game engine needs to be explained, as some of features possessed by the engine are crucial to the development of the video game by a novice developer.

There are several features that need to be considered when choosing the engine. For instance, the game engine needs to be open source to avoid the necessity of purchasing a license. Fortunately, there are a lot of options in this regard, with many companies making their products available for free to incentivise new developers. Other features like the ease of learning must also be considered.

Since it has a free license model and possesses developer friendly features that allow for even novices to start creating games, the Unreal Engine 5 was chosen to be the game engine for the game "The World of Anwin".

4.4.1 Background

Developed by Epic Games, the Unreal Engine has been around since 1998. Back then it was created to power the then brand-new title "Unreal". Since then, it has been used in thousands of games like Borderlands, Mass Effect, Bioshock, Gears of War and Fortnite [43].

The game engine is built in the C++ programming language allowing it to be powerful but also very portable. From an Unreal Engine 5 project a game can be built for Windows, MacOS, Linux, Consoles as well as mobile platforms [44].

In 2020 the 5th version of the Unreal Engine was announced and later released in April 2022. This new version was headlined by two main new features: Nanite and Lumen.

Lastly, the engine is also free for anyone to try, making one of the best options for up-and-coming developers to develop their games on. Epic Games only begins collecting royalties on games made with UE5 after one million US dollars [45].

4.4.2 Level Editor

To create the levels of the video game, UE5 like many other game engines, provides a level editor as seen in Figure 13. This level editor is where all the visual parts of the game come together to create the playable level.

In the editor it is possible to add new geometry to a scene like trees, houses, and other random objects to populate the game world. Similarly, it is also possible to sculpt the terrain under all these objects. Although the engine does not possess a modelling tool, it does have a terrain sculpting tool. This allows the developer to create interesting maps, with different elevation features.

The editor is also where the collision limits and navigation areas or meshes for the characters are established. This concept has already been explained previously but as a reminder, it is the area all actors in the game require to move.

Within the editor itself the developer can also control all the lighting in a scene, with Unreal Engine having several types of lights built in by default, including a full global illumination system that allows the developer to place, by hand, the sun in the sky and watch in real-time the effect it has on the world and its shadows.

All the systems that will be created to interact with the player character will have a physical object that is then dragged from the content browser to the level, just like a file from a folder to another, and placed where the developer wants.

In summary, this is where the video game's virtual world is created and where it can be play tested in real time to determine how development is going, allowing for easy debugging of mechanics and assets.

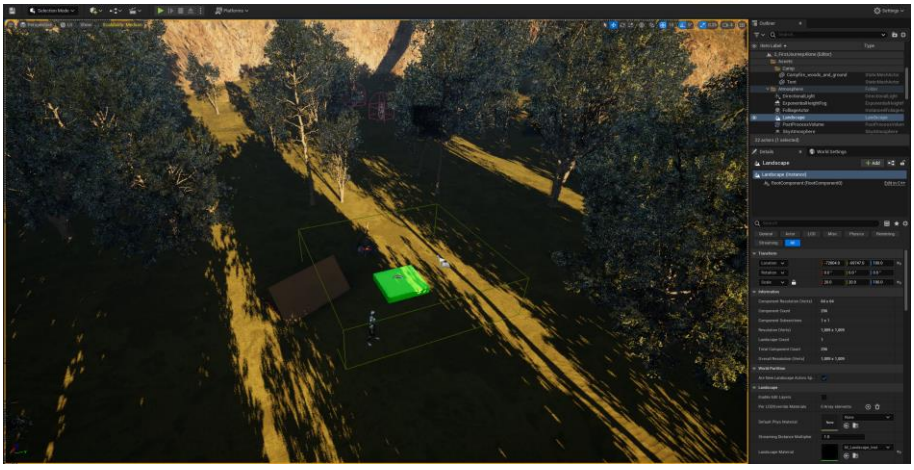


Figure 13 – Example of the interface of the Unreal Engine 5 level editor.

4.4.3 Programming and Blueprints

When it comes to coding the game logic there are two possibilities: Since UE5 is built on the C++ programming language the logic can be programmed in C++ as well. Alternatively, UE5 provides a proprietary visual programming language called "Blueprints" as shown in Figure 14.

C++ is the native language on which the game engine was build and thus provides tremendous optimization and flexibility to create whatever feature the developer wants or needs. This allows for better overall performance, given that it is a lower-level programming language compared to Blueprints, meaning a language that is much closer to what a machine would understand natively.

Blueprints, meanwhile, were created to help new developers come to grips with game programming, with a much lower skill requirement and being very forgiving with errors. They are also required to create textures and materials.

To quickly understand how Blueprints are created and executed, the example from Figure 14 will be used. The most important concept of UE5 Blueprints is flow control. In a normal programming language, code is read from the top down, computing everything along the way. The same principle is applied to Blueprints, with the difference being that the developer can guide the flow. Nodes can have an input, output, or both for the flow to pass through. This flow is represented by the white line connecting the nodes. These white lines control the flow and tell the compiler which nodes to trigger next. As explained in [46] we can "Think of this as electricity, and the white lines are powering nodes along its path". Flows can start either in a red node if it is an event, or a purple node if it is a function. Some nodes also have these coloured connectors on the left and/or right. These represent input (left) and output (right) parameters of that event or function. Their colours are associated with the class they belong to. In the case of the figure, a blue wire means that it is a non-primitive Class, and the green wire means that it is an Integer (primitive class). Alternatively, when not connected, like in the node "Destroy Actor", the input will target the class of the object itself, showing "self" to indicate that association.

In the example provided, the event "Actor Begin Overlap" was triggered. This means that an actor, overlapped with the current one, with the output "Other Actor" being a reference to the actor class that overlapped. The flow guides the execution to the casting of the event output "Other Actor" to the class Player. If the cast was successful the flow continues and using "As Player" (this is the variable that was created during the casting process), the developer can call functions from the class Player. In this case it called the function "Add Score", sending the local variable "Score to Give" as an input. After this is done, the current actor destroys itself with the "Destroy Actor" function. This example belongs to a game in which the player can collect pickups that grant score by collecting them. When they hit the pickup, this code is executed, and the score is added to the Player class and the pickup disappears (is destroyed).

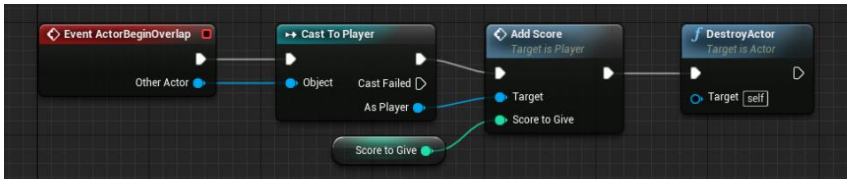


Figure 14 – Example of the Unreal Engine's Blueprints [46].

Each blueprint follows a normal programming logic with the execution starting from left to right and following the white lines. This is an event-driven type of programming language, meaning that actions can be caught and controlled by the developer with the use of events represented as the red header nodes. The turquoise nodes represent casts to another class, the grey ones represent "branches" or most known as "if" conditions. Green nodes are getters for main classes and finally the blue nodes are functions that are being called.

For these reasons, and due to the fact, that without consummated experience in C++ it was easy to make language-based mistakes that would break the game, that then required fixing, consuming the already relatively short development time, the choice was made to develop the game fully using the Blueprints visual programming language. The same programming principles apply to a visual programming language as in a regular one, like for instance "If" conditions and "For" loops among others.

4.4.4 AI Behaviour

In this subchapter, the tool for AI systems present in Unreal Engine 5 will be explored. It can be used to create robust algorithms to run the in-game entities, while still being friendly to new and less experienced developers.

As seen in Figure 15, the engine allows for the creation of complex and very heavily customizable AI Behaviour Trees. These can be created for pretty much any entity that requires an AI and can be controlled by the developer with several event triggers. Since Behaviour Trees are event-driven, they are always passively listening to triggers and allow for the control of its several functionalities with Blueprints or C++. Using the tool itself is like the Blueprint system, with a visual programming language approach. Each purple node seen in the Figure 15 is a task that will be triggered if the conditional "decorator" nodes in blue are met.

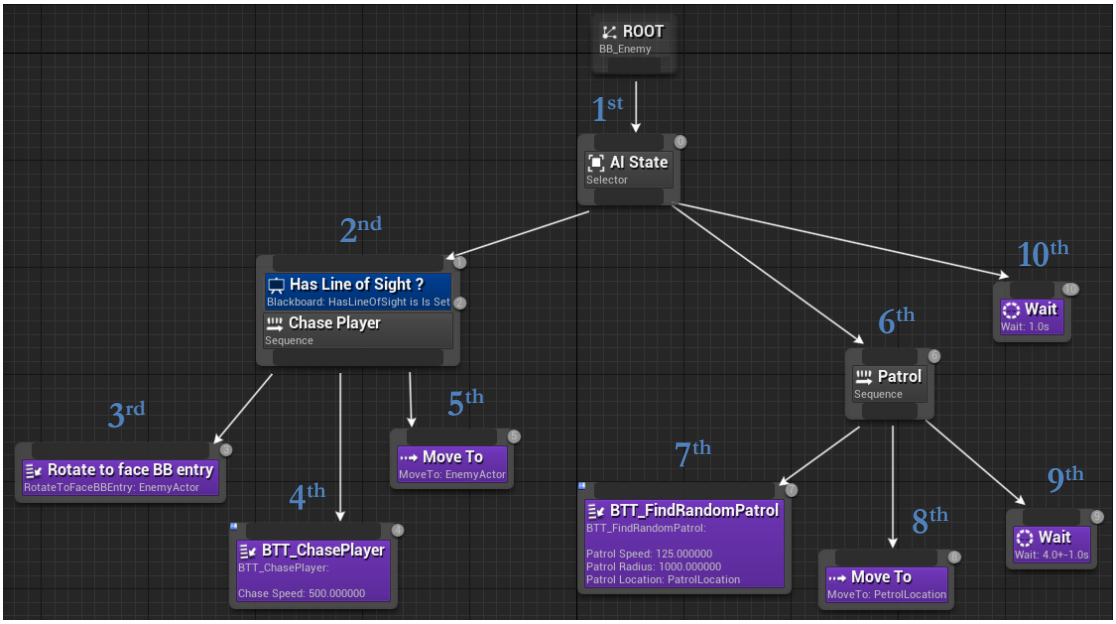


Figure 15 – Example of an Unreal Engine 5 AI Behaviour Tree [47].

The grey nodes are called "composite" nodes and they can be sequences, selectors, and parallel executors, all starting from a previous node (doesn't have to be the root node), allowing for different tree or branch behaviours. The tree, no matter the nodes, works using a Depth-First Search (DFS) traversal, executing nodes from left to right. If looked closely at the nodes and conditions of the tree, its noticeable that each has a number in their top right corner that represents their run order. Furthermore, numbers were added in blue to aid the perception of the executed order. All the information about to be summarized can be accessed in the Unreal Engine 5 documentation [47].

A sequence works by executing its children from left to right cycling until one of them fails. When a child fails, the sequence fails as well, stopping the execution. If all children succeed, then the sequence succeeds and continues to cycle. To prevent a sequence from ending when a child fails, there is a conditional decorator node called "Force Success" that makes it so that if it succeeds it goes on to perform the task and if it doesn't it just moves on to the next tree branch.

Another composite node is the selectors. They work differently than sequences, stopping the execution when one of the children succeeds. If a selector's child succeeds, the selector also succeeds. If all the selector's children fail, the selector fails.

Lastly, there are simple parallel nodes which allow a single main task node to be executed alongside a full tree. These are the only node that change the normal execution order of trees, allowing multiple tasks to be executed at once. When the main child finishes, there is a setting called finish mode that dictates what happens to the tree, and it can be immediate or delayed. Immediate means that the background tree will be aborted as soon as the main task finishes. Delayed means that the background tree will be permitted to finish once the main task has finished.

The other main important factor in the AI Behaviour Trees is the Blackboard. This Blackboard contains several developer-defined "keys" each with their own class that hold information used by the Behaviour Tree to make decisions. For example, you could have a Boolean key called "Is Light On" which the Behaviour Tree can reference to see if the value has changed. If the value is true, it could execute a branch that causes a fly (the entity being controlled by the Behaviour Tree) to be attracted to the light and if it is false, if could execute a different branch where the fly moves randomly around the environment.

Following the behaviour tree shown, a small explanation of how these trees are processed will be formed. This behaviour tree is used in the patrol system of an NPC. The first branch is rooted on a selector, meaning that, executing from left to right, the tree checks which branch can run (select behaviour):

1. Chase Player Branch (2nd node): it has the conditional node of "Has Line of Sight". If this condition is met, the NPC will begin chasing its target, entering the "Chase Player" sequence. If the condition is not met, the selector node will move on to the next selector child.

   a. Sequence: the NPC rotates to face the player (3rd node), then it sets is chase speed (4th node), and finally moves to the location of the player (5th node), beginning the chase. These tasks are repeated until one of them fails, failing the sequence. Once again, nodes execute from left to right.

2. Patrol Branch (6th node): it does not have any conditional nodes, so it automatically enters the sequence if the "Chase Player" branch condition failed. If the sequence fails, the selector will move on to the next selector child.
   a. Sequence: the NPC finds a random patrol location within the navigation mesh (7th node), moves to it (8th node) and when reached, waits for 4 seconds, plus or minus 1 second (9th node). These tasks are repeated until one of them fails, failing the sequence.
3. Wait Branch (10th node): When this succeeds the selector node will also succeed and so the tree finished running successfully.

When the tree finished running, the AI component stops, waiting for the combat to restart over again.

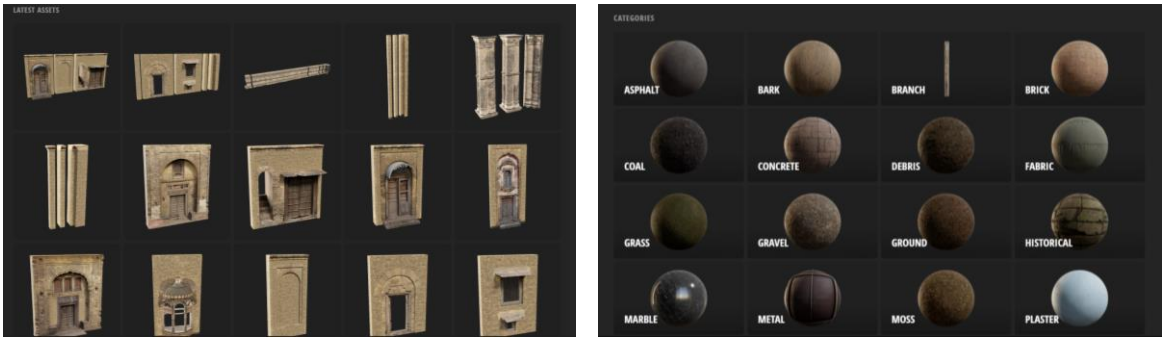
In general, Behaviour Trees offer a tremendous amount of customization, supporting examples as simple as a fly being attracted to a light source or as complex as simulating another human player in a multiplayer game that finds cover, shoots at players, and looks for item pickups.

4.4.5 Quixel and Nanite

In this subchapter, an overview of the tools called Quixel, Quixel Bridge and Nanite will be performed, explaining why they're essential for the creation of a video game by a novice game developer and how they broadly work.

Quixel is a library of 3D assets that has become famous due to its incredible amount of free "Megascans". These are content captured using techniques such as photogrammetry [48] to create extremely high-resolution assets. It has a library of thousands of 3D assets, surfaces, imperfections, vegetation, and decals, with new ones added daily.

During the creation of Unreal Engine 5, Epic Games bought Quixel and were able to create a gateway between the Unreal Engine 5 and the Quixel library. This was dubbed "Quixel Bridge" and allows a developer to open the Megascans library right within the level editor and after choosing what they want to use, just drag the subject into the level with the bridge doing all the importing work. In seconds this feature adds an extreme high-resolution asset to any level and can do so at any time. The only disadvantage with these Megascans is that at their highest resolution they tend to become very large in terms of file size. Small examples of these assets can be seen in Figure 16.



a) b)

Figure 16 – Example of two pages of Quixel Megascans a) assets and b) materials.

All the scans are created to fit Unreal Engine 5 perfectly but can also work in other 3D modelling programs. The direct integration means that other UE5 features are baked into the assets. One of these features is "Nanite" which is a replacement for traditional assets management in video game levels. Assts are usually grouped into Levels of Detail (LODs), which are used to present information in a way that doesn't completely consume the memory resources of the GPU. This previously explained technique creates a phenomenon called "pop-in" in which the player can see objects changing LODs right in front of their eyes, with this is especially common in open-world games due to the large environments. This means that developers must create several LODs for their game's assets and know how to manage and transition between them properly, which is a process that consumes time and resources that a novice developer may not have. With the introduction of Nanite, the whole paradigm shifts. Nanite is Unreal Engine 5's virtualized geometry system that intelligently computes only the detail that can be perceived and no more. This basically means that an object with Nanite geometry has automatic LODs.

4.4.6 GUI Editor

When it comes to developing a game, a major part of the interaction with the player is the User Interface (UI). As already explored previously in this chapter the UI is key to delivering focused and well-structured information to the player so they can manage and decide their actions.

The Unreal Engine 5 provides a built in Graphical User Interface (GUI) Editor. Within this editor, the developer can create a complete interface, filled with text boxes, progress bars among other widgets. All these widgets are then connected to events and can be updated in real time by the developer's code when the situation calls for it.

Apart from updating all widgets on the UI, the developer is also able to control their properties like placement, how long it is shown for, and even if it is visible or not.

As seen in Figure 17, the editor directly represents the layout of the screen the player will be seeing, allowing for precise placement and organization. Each of

the objects can be scaled and modified. To create another, it is as simple as dragging it from the menu on the top left corner that houses all the different available widgets. Several user interfaces can be created for the game, that the developer can then choose to change during runtime with a small piece of code.

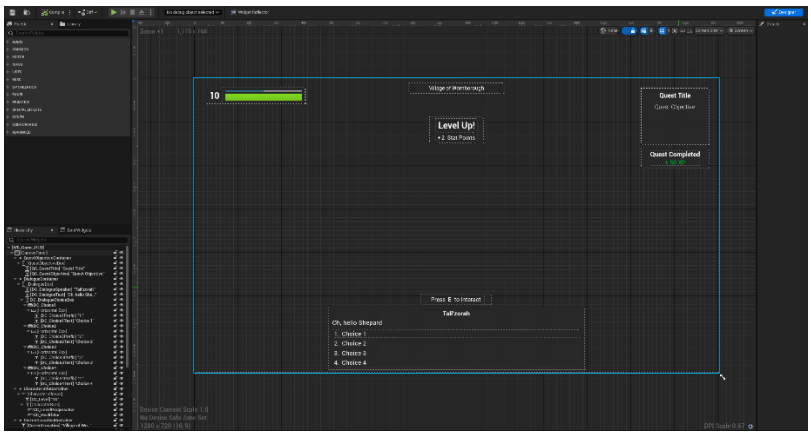


Figure 17 – Example of GUI Editor in Unreal Engine 5.

This is also where floating widgets can be created to display floating information that can then be pinned on an entity, to show for instance a health bar of an enemy. This bar can be attached to the 3D model and follow it around, while still being able to be updated in real time.

This is a very useful tool that can create great UIs even without custom models, allowing for novice and unexperienced developers to create a sleek and practical designs on their own.

5 DEVELOPMENT OF THE VIDEO GAME

In this chapter, all the details of the development of this video game will be described and explained. This is one of the most important chapters because it explains in as much detail as possible the entire process of creating the video game, with the only exception being the implementation of the RL model, that has its own chapter dedicated to it.

Much of the work required during the development of a video game like this is research. Until the start of development, the experience of the developer was limited and thus a lot self-taught work needed to be done. While some conventions can be found online during research, others only come through experience and experimentation. Part of the excitement of developing a product like this is understanding the complexity and nuances behind these pieces of software so many people spend countless hours on.

One thing that must be made clear is that the Unreal Engine 5 is a very powerful engine, requiring a good computer to run the game and an even better one to properly develop it in. With an "AMD Radeon RX5600 XT" GPU installed in the development machine, which has 6 GB of video memory, the process of development was filled with complications. It is still a very capable video card even though it is starting to run low on the video memory. When developing this game, the number of crashes witnessed were substantial and most of them were due to the lack of video memory. This improved as development went on and the stability of the 5th version of the engine increased, but it was still and issue with more complex environments.

5.1 Development Structure

To begin the development of the video game, there was the need to establish a structure that could be followed during the length of the project, trying to prevent future backtracking that would waste time. If done wrong, the developer could reach a point in development where they had worked themselves into a corner, and where the only way to progress was to redo some of the previously completed work. If done correctly though, the development progresses seamlessly with little to no development time spent on reworking or fixing previously created tools or code.

For this game, there was a necessity to establish an order that made the development simpler and organized. As mentioned in chapter 4, there was specific systems that were required for the development. The order established was the following:

- Create the tools needed for the game to track and present a story to the player.
- Create a way to store information and a save and load feature.

- Create the combat system with the mentality that it needs to be designed for a future implementation of a RL model.
- Lastly, implement the RL model in the previously developed combat system, making the necessary modifications for it to work.

With the choice of engine, some systems that are necessary already came built-in to the project. When creating a new UE5 project, there are several template options that allow for specific styles of games to be created like First-Person Shooters (FPS) or Racing games. For this project the "Top-Down" template was used. This immediately created the ability to walk by right clicking on a position of the visible map. This was later supplemented with the creation of movement based on keyboard inputs: W – up, A – left, S – back, D – right. This was all created by simply inserting some options into the project settings.

5.2 Building the Necessary Tools

To be able to construct the game's parts a set of tools needed to be built. With these tools established, the game could be procedurally built in the level editor. Some of these tools are objects that can be placed within the game environment to perform a specific task. These objects, will have parameterized fields that will inform the game of their functionality, allowing the developer to quickly create several different types of functionalities and event sequences, using the same object, similar to polymorphism used in object-oriented programming.

For most of the things that happen in the game, the "Update State" event is called. Most of these tools will call this event, sending as parameters the things that need to happen, and the game will update itself automatically, processing, according to the parameters, what kind of actions need to occur.

For some tools or systems built, there was the necessity to store information outside the game given the increased quantity of data. Systems like the quest and dialogue had to have a file where their information could be stored and easily edited when change was required. This led to the creation of a file storing logic, that would store the information in coma separated values (CSV) files and be read by the game engine and properly processed by the object logic implemented, creating the respective information classes. This type of information storing cannot be used in real time, so this way of saving files only works effectively for information that does not need to be updated in real time while the game runs. All of this will be explained in greater detail later in this chapter. Every system that requires a CSV file for information storing will be specified in their respective subchapters.

5.2.1 Quest System

When the game was initially conceptualized, one of the goals was to create an RPG with a story, set in a fantasy world, where the player could embark on missions and progress through them. For this an all-encompassing system needs to be created to store and manage the missions that create the story of the game. This system is called the "Quest System".

It starts by establishing concepts and defining what functionalities are required to achieve the goals. With a specific type of story and mission structure in mind, it is clear that quests need to be separated into several sub objectives. This means that quests, either main or secondary, have a multi-step structure making them more interesting and allowing for more variation. With all of this in mind, several concepts were created:

- Story – all-encompassing container or class that keeps track of everything within the story, like main quests and secondary quests.
- Quest – Singular mission that can have a series of objectives and parameters attached to it.
- Quest Objectives – The lowest level of quest, the quest objectives usually come in groups and are part of a quest's progression.

Each of these concepts has a class representing it in-game. These classes store information relative their respective concept. The Quest System class diagram can be visualized in Figure 18, serving as a visual representation of the structure and logic of all classes.

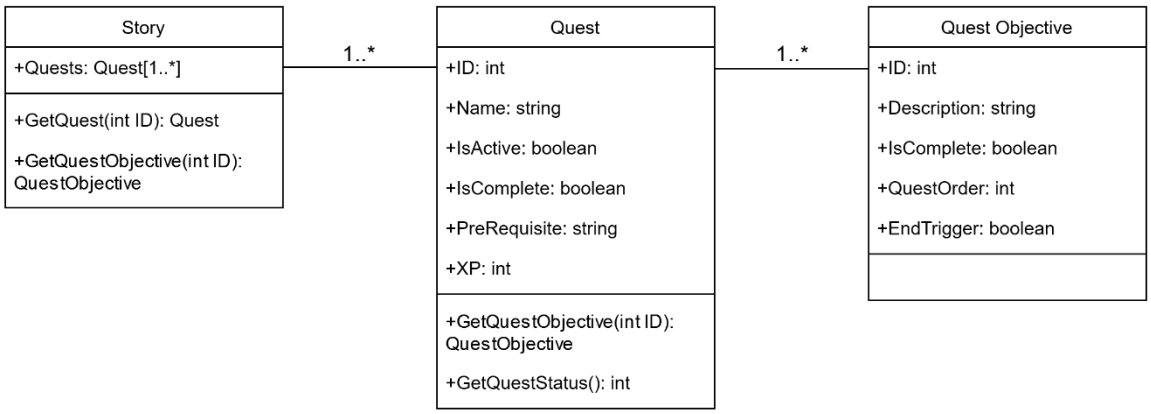


Figure 18 – Simplified version of the Quest System class diagram.

The Story class holds all the quests of the game, stored in an array, by order of how they were created. This order is irrelevant since each quest object has a unique identifier (ID). This identifier is in string form and follows the format: "q" – for quest followed by the number of the respective quest. For example: "q1" – quest number 1. When it comes to the quest objectives, they all also have IDs that follow a specific format. In this case the format would be q1 – quest 1 followed by "o" –

objective, and the number of the corresponding objective: For example: "q1_o2" – quest 1, objective 2. Although not required, for ease of use and identification on the side of the developer, the IDs of the quest objective are also unique, meaning no two quest objectives are the same even if they have the same name and description. This story class also has all story flags that represent specific story facts that have happened. After some research this method is frequently used by game developers to create scenarios that can easily be checked further on, possibly in other missions and quickly assess if the requirements are met for a specific quest or even game ending. An example of this is for instance if a player has found a specific item on the first mission. With a flag set, this fact can easily be checked for future reference when the developer requires.

The Quest class has the variables that define an instance of a quest. Each quest is defined by their unique ID and has a few obligatory variables. Each quest must have a title defined when creating the object, as well as Experience Points (XP) that will be awarded to the player once he finishes the quest. If necessary, the quest will also have an array of quest objectives objects. Optionally the quest can have a pre-requisite, serving as a filter or safeguard from triggering a specific quest accidentally when it should not be possible to do so. Lastly there are also Boolean flags to represent the quests current state. These are the "Active" and the "Is Complete" flags. As their names suggest, they keep track of facts like, is the quest finished and if it has begun, meaning if the player has discovered this quest. These last flags are important to check pre-requisites of other quest as well as to display the quest log. The quest log is a subsystem of the quest system that the player can access at any time outside combat or dialogue. When opened by pressing the "J" key, for "Journal", the player is greeted with a full screen information of the quests active and the quests already completed, as seen in Figure 19.

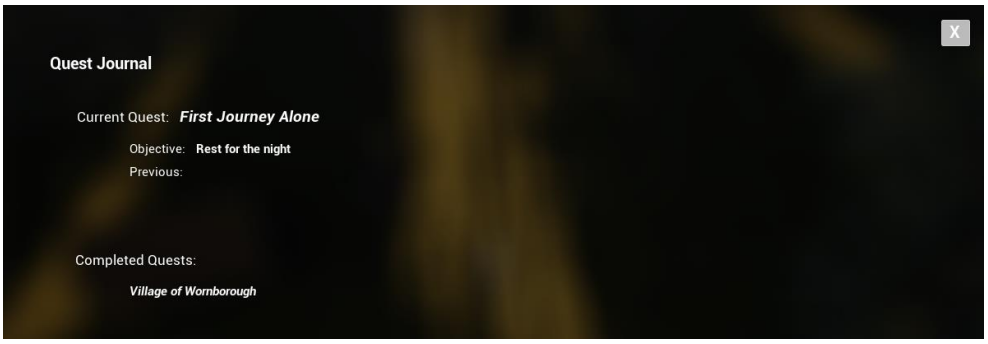


Figure 19 – Example of the in-game quest journal at the start of the second mission.

All these classes are stored withing the Game Mode class which controls all access to this information following the same principles as encapsulation in object-oriented programming. This class is also responsible for the creation of the quests and can tell, using the flags in each quest, if it has or not been complete, making quest requirement checks easy to perform.

The creation process for each quest starts with the structuring of the information. This starts with the information required for each quest and quest

objective. As already explained previously, each quest has Quest ID, Quest Name and XP to award its completion. Likewise, the quest objectives need to be well defined as well, each with its own ID and description.

Like in other situations throughout the project, there needs to be a system to store information in an organized way and at a large scale. For this purpose, the choice was clear to go with a CSV file to store all the quest related information, like it can be examined in Figure 20.

When importing a CSV into Unreal Engine 5, it is required to parse data into a Data Structure. This is a special pre-built class where variables can be created, each with their specific type, requiring that the headers of the CSV file match with the names of the variables, allowing for organized importing at the press of a button. Like all other Data Structures that were used during the development of this project, the first column of the CSV must have an unnamed header. This serves as a point of reference for the engine to name its internal Data Structure lines, and by just using simple Integers in a sequence it also makes it simpler for fetching the information once it needs to be used in the creation of the quest classes.

To avoid having multiple files for quest and quest objectives, there is only one file that stores all the information. Each line of the file represents a quest objective, with a field marking what quest it belongs too. This is an easy way to do things because when creating the classes, a quest objective that does not have a created main quest associated with it, triggers its creation. If a quest objective already has their quest created, it skips the creation part and just adds a new objective to the array already created within the quest with that specific Quest ID.

Each of the CSV lines also has a few other columns representing other information, both for the quest and its respective objectives. When a quest objective is the last in a quest, there is a Boolean flag that marks that ending, called "End Trigger". During creation when this end trigger is added to the quest objective, the system knows the next CSV line, if there is any, is related to another quest. The XP reward of each quest is also provided in the final line of each quest. The Quest Order column is a redundancy, representing a second way of keeping the quest objectives organized. This is until the introduction of optional quest objectives. Like many other games, some objectives are not obligatory, giving extra information to the player, perhaps contextualizing a certain mission better, but that ultimately are not relevant to the overall quest structure. For this the quest order becomes important because it allows the developer to set two objectives with the same order making them optional. When two objectives have the same quest order, the system when checking for order integrity, will aways see the expected number, but the player can have a different experience. Like seen in Figure 20 in line 6 and 7, the "Hide" and "Investigate" quest objectives have the same order. This means that the player can either hide or go investigate whatever it is, that the quest system will recognize both actions as the correct one and move the quest progression forward.

	A	B	C	D	E	F	G	H	I
1		QuestID	QuestName	QuestObjectiveID	ObjectiveDescription	QuestOrder	EndTrigger	QuestXP	PreRequisite
2	1	q1	The Village of Wornborough	q1_o1	Discover what is happening	1	FALSE	50	
3	2	q1	The Village of Wornborough	q1_o2	Witness the events unfolding	2	FALSE	0	
4	3	q1	The Village of Wornborough	q1_o3	Escape the village	3	TRUE	0	
5	4	q2	First Journey Alone	q2_o1	Rest for the night	1	FALSE	50	q1
6	5	q2	First Journey Alone	q2_o2	Investigate	2	FALSE	0	
7	6	q2	First Journey Alone	q2_o3	Hide	2	TRUE	0	

Figure 20 – CSV file structure for quest related data storing.

Finally, when creating a new quest objective that does not have an already created quest, it is indicated if the quest has any pre-requisites that need to be met before it can be triggered. Like previously explained, this is a safety and control feature for the developer to control the triggering of quests in a specific order.

All of what was explained about the creation process of quests can be visualized in Figure 21.

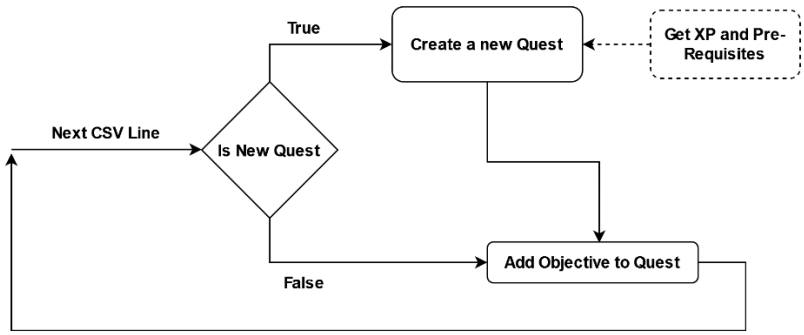


Figure 21 – Visual representation of the creation of quest and quest objectives.

All the quest and quest objectives are created before the game starts to run. Since it is all text information, the game can afford to load it all at once at the start without worrying of performance impact. This information is all stored on the Game Mode class, which is responsible for storing all the loaded data of the game like quests, combat encounters and dialogues.

The visual manifestation of the quest system is its implementation with the UI system. It works by having a quest box in the right upper hand corner of the screen, showing the active quest and the current quest objective needed to progress. The whole UI implementation will be explained in greater detail later in this chapter, but a simple visualization of the system can be seen in Figure 22.

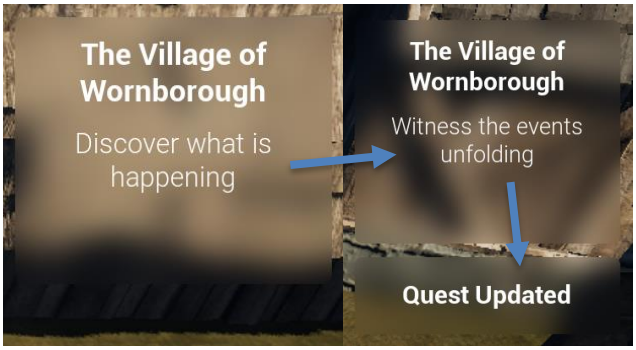


Figure 22 – Visualization of progression marked by the quest indicator during play.

5.2.2 Stage Trigger

Keeping the story related goals in mind, systems needed to be built to support the quest system. These systems' intent is to, through their parameters, control the progression of the story and quests throughout the game. For this goal to be achieved, a trigger of some sorts needed to be created, so that the player character movement through the world could set off events in a mission. One example of this is when the player character enters a room where something is supposed to happen. With a trigger placed at the entrance of this hypothetical room, it creates a stage (hence the name) for an event to be triggered in-game called "Update State". This then informs the quest system of what happened and to what mission that event corresponds to, allowing the game to update itself, consequently updating the UI as well. For this purpose, the stage triggers as presented in Figure 23 were created.

With this trigger being an object in the level editor, it means that it can be edited and morphed. The size is completely modifiable at the developer's wishes, allowing for a smaller or bigger triggering stage. It is important to note that the green cube seen in Figure 23 is there just so that the developer can easily detect and select it. The object itself has no collision with player or non-player characters, meaning that they can pass through it. The entirety of the trigger is hidden when the game is running. When a stage trigger is stepped on by the player character, which happens when the PC enters the wired box seen in the figure, the event "On Component Begin Overlap" is fired. After this, using the developer defined trigger parameters, the Update State event is called, updating the game as necessary.

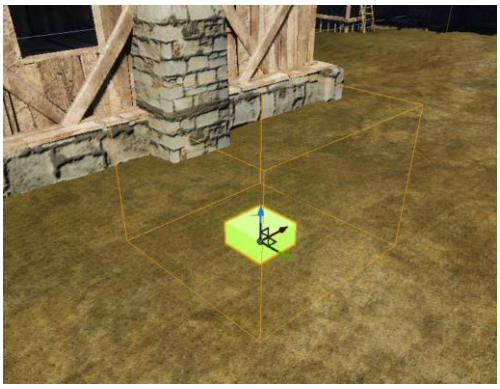


Figure 23 – Example of default stage trigger object spawned in a level.

Each trigger also has several parameters that need to be introduced, some being optional and others obligatory. These are composed by:

- Quest ID – String.
- Quest Objective ID – String.
- Quest Order – Integer.
- Quest End – Boolean
- Dialogue ID – Integer.

- Load Level – String.
- Direct Load – Boolean.
- Pre-Requisites – String.
- Limited Trigger - Boolean

The quest related parameters are used to manage the Quest System. The Quest ID indicates the unique identifier of the quest being triggered. This quest identifier has a pre-determined format, already explain in the previous Quest System subchapter.

The second parameter is the Quest Objective ID that similarly to the Quest ID, has a pre-defined format established in the Quest System subchapter, and is used to track the current objective and what needs to be done to trigger the next.

The next one is Quest Order, that serves as a way to keep track of the order of each quest objective. This is important especially in cases where there are optional objectives. Quest Order then works in tandem with the Quest Objective ID to differentiate the two objectives that have the same order but different quest objective IDs.

Dialogue ID is used to connect the stage trigger to the dialogue system. When a stage is triggered, there is the option to either call the quest system or the dialogue system (or both). This makes it so that there can be dialogues started by the stage trigger. One example of a possible utility for this is when a player character enters a room, and the narrator needs to pause the game and start describing in detail what is happening.

Load level is a string parameter that when entered, depending on the value of Direct Load, loads a level specified. If Direct Load is true, the "Update State" event will immediately load the new level, if not, usually when there is a dialogue before the load is required, the load will only happen after the event being processed is completed.

The Pre-Requisite parameter is another safety feature that was introduced to make sure that a quest could not be triggered if a specific other quest was not yet complete. When set, the stage trigger will always check in the Game Mode class to see if the Quest ID inserted has a pre-requisite and if it has been completed. The class Game Mode holds all the story facts, as well as the quests that were completed, so checking if a quest was or not completed is very simple.

Lastly, there is the Limited Trigger Boolean flag which serves a simple purpose. There can be certain scenarios where the stage can be triggered multiple times, and there are times, usually for story reasons, where it cannot. With this flag the stage trigger can easily filter out unwanted triggers.

This system alone could be used to create a game in a limited fashion. It allows for the triggering of mission on a stage basis with varying sizes, meaning that any

location-based triggering can work as a mission progress marker. It can load new levels making it essential as the main way to transition between levels. Lastly, it can trigger dialogue sequences. This makes viable the use of this type of trigger as a story telling mechanism, triggering dialogue sequences.

Though the use would be limited, it could also be used to trigger an interaction with a stationary NPC. In this hypothetical scenario the trigger would be small, close to the NPC and when the player approached an interaction would trigger. This is a reasonable way of doing things, but it is very limited in broad implementation when it comes to interacting with NPCs. It also has the downside of making these interactions automatic, just requiring the player to move the character to a specific position, taking away the player's choice on what to do. This requires another type of trigger, the Object and NPC Interaction Trigger.

A different concept of stage trigger was also implemented called the "Location Trigger". This trigger works in the same way as the Stage Trigger does, just with a different purpose and a much simpler parameterization, just requiring the location name. Since there is no map in the game to tell players where they are in the world, it was required to have a way to indicate where this level was located within the grander world. For this the location indicator was created, as can be seen in Figure 24. This could be useful when a mission said to look for something on a specific place. With a location indicator, the player would know that they're on the correct location to look for that specific something. The Location Trigger then works by creating a much larger stage encompassing all the level, or merely parts of it. An example of this is when the player is in a town, the location trigger is placed to encompass the whole town and with the name of that town set as a parameter. Then, when leaving town, the player would enter the outskirts and there would be another location trigger, bordering the previous one, that would make the location indicator change when entered.

Figure 24 – Location indicator on the top centre of the screen.

5.2.3 Object and NPC Interaction Trigger

To supplement the previously detailed system, there was the need to create something that could encompass interaction with entities. Within the game world these entities could take many forms like any type of object or an NPC. This system would require an interact action on the side of the player, unlike the Stage Triggers where it happens as you walk on the pre-determined stage.

With these requirements set in place, it was time to create the new system. Like other systems, it requires some advanced parameterization to allow it to be both robust and versatile in its implementation.

Early on the implementation, it was determined that these would need to be two separate objects in the engine, a "Object Interaction Trigger" and a "NPC Interaction Trigger". This necessity came from the fact that an object behaves differently than an NPC. While an object is just a 3D model with a mesh attached and some textures to give it the appropriate colour, an NPC is a model with a skeleton attached to it. As previously explained, a skeleton is a set of "bones" that allow the developer to control the position of that mesh and animate it.

Now that it is established that two different objects need to be created, it is required to understand how they work. When the player approaches one of these types of triggers, there is an area around them dubbed the "Interaction Bubble" that then triggers an event called "On Component Begin Overlap". Unlike the stage triggers, this does not have an immediate effect, instead it tells the Player Controller (class that controls the player character's functionalities) that there is a possible interaction available, and with what this interaction is, saving that information. This then begins a chain of events that makes the interact key, usually E, available to press, and it also updates the UI so that the player gets a prompt on the screen saying, "Press E to Interact". Both the interaction bubble and the UI prompt can be seen in Figure 25. If a player leaves the interaction bubble, the event "On Component End Overlap" is triggered, and the interact prompt goes away with the interaction information within the player controller being erased. This is meant to allow for follow up interactions without the confusion of what is currently being interacted with.

a) b)

Figure 25 – Example of NPC Interaction Triggers. a) Example of the Interaction Bubble, b) Example of the interact prompt.

When inside the interaction bubble the player can press a key to interact with that Object or NPC trigger. This now toggles the player character to enter "Interact Mode". This is done by telling the player controller with help of a key event listener, that several things withing the character need to change. First, the character no

longer can move, either by mouse click on a place of the map or by using the directional input keys, since it would not make sense for the character to move away once an interaction like dialogue as started. Second, the option to open submenus like the quest log is disabled. This is done so that the UI doesn't get jumbled or confused, fetching information to update a UI element that is no longer there. Thirdly, the whole UI is updated, opening a dialogue box for instance, and replacing the interact prompt with the corresponding new one. When the interaction is over, all returns to normal. The UI resets back hiding the dialogue box, freeing the lower centre of the screen. The interaction prompt returns as well, showing the player that the NPC or Object can be interacted with again.

When it comes to parameterization both the Object Interaction Trigger and NPC Interaction Trigger have these in common:

- Dialogue ID – Integer.
- Other Dialogue ID – Integer.
- Limited Interaction – Boolean.
- Free Roam – Boolean – Exclusive to NPC Interaction Trigger.
- Can Interact – Boolean.
- Is Player Companion – Boolean – Exclusive to NPC Interaction Trigger.
- Pre-Requisite – String.

When it comes to the use of the Dialogue ID parameter, it works just like it did on the Stage Trigger. A pre-determined identifier must be inserted, allowing the Player Controller to request the corresponding dialogue sequence from the Game Mode class, which stores all the loaded information of the game, to interact with.

After the first dialogue interaction, a certain object or NPC might change their behaviour. This is represented by the Limited Interaction and Other Dialogue ID parameters. When an interaction is only supposed to happen once, the limited interaction flag is set to true and after the first interaction where the Dialogue ID is presented to the player, the system then checks if there is another dialogue sequence to present. This other dialogue sequence is represented by the Other Dialogue ID parameter, working the same way as the original dialogue request, fetching information from the Game Mode class. For any further interaction with the object or NPC after the first, the Other Dialogue ID sequence will be presented to the player.

Every level has the presence of a navigation mesh. This is a volume that is calculated by the game engine with the dimensions set by the developer, allowing for entities to navigate on. This is not a perfect system with floor irregularities having a likelihood of causing the navigation mesh to not recognize all the traversable areas, but still is widely used by developers due to its simplicity and relative effectiveness.

An example of a navigation mesh is shown in the Figure 26, represented by the dark green area.

This definition was important to make because of the concept of free roam. When an NPC Interaction Trigger has the free roam flag set to true, the Interaction Trigger now is a mobile one. It calculates a possible movement position, always set within its current navigation mesh, and moves to it, with the animations provided by the default Unreal Engine skeleton rigging system. As a result, the NPC moves naturally through the levels environment, making the work feel a bit more interactive and livelier. With the NPC Interaction Triggers having this free roam capability, it allows for them to be placed as random, non-interactable NPC as well. For this reason, the flag Can Interact was created, making these types of NPC creatable just by toggling a flag in the editor. When a moving NPC is interacted with, both the NPC and PC stop moving and the conversation starts.

Figure 26 – Example of the navigation mesh implemented in the first level of the game. Dark green layout on the floor represents its bounds. Bright green blocks are Stage Trigger. Purple spheres are NPC Interaction Triggers.

The exclusive flag Is Player Companion is meant to allow for a player companion to be created using the same object as an NPC Interaction Trigger. In all aspects they should be the same, with the only difference being that instead of free roaming, they follow the player character around the map. This feature could be more useful later in the game's development where player companions become a prominent addition to the gameplay.

Lastly, the Pre-Requisite is a condition that, just like in the Stage Triggers, prevents in this case an interaction from being presented to the player if the pre-requisite, which is always a quest, is not yet known or completed. This makes it so that information can be filtered by the developer to the player, establishing a story telling order that's crucial to telling a story in an RPG.

5.2.4 Dialogue and Choice System

Like the Stage Trigger explained previously, the game was idealized as an RPG with a small story within. This requires a mechanism to deliver the story to the player since there won't be any voice acting or facial animations. The alternative option is to have a dialogue box where contextual information appears when the game needs to inform the player of something, or a dialogue is triggered.

The creation of the dialogue system was something that required a lot of research and experimentation. More even than the quest system, the dialogue system was required to hold a lot of structured text information. When a dialogue starts, there is a sequence of sentences with a specific order that need to be displayed, each with a specific speaker attached to them. In a smaller scale this problem could have easily been hard coded into each scene making it easier to implement but destroying the possible versatility and scalability of this system. Therefore, something else was required, something that could hold the dialogue sentence and the respective speaker among other important parameters that are story progression related.

This problem led to the creation of two concepts that became crucial in implementing the dialogue system:

- Dialogue Tree – all-encompassing concept and class that stores, organizes, and provides structured pieces of dialogue. Composed of several Dialogue Instances.
- Dialogue Instance – single instance of dialogue, composed by a sentence, the speaker, and other optional parameters.

Each time a conversation needs to be created for the game, the base structure is always a dialogue tree. Each tree has a unique identifier that allows for easy fetching of the structured information. Each Dialogue Instances is composed of all the information a dialogue requires to be presented to the player in the UI. Within each tree, each instance also has a unique identifier, called "master", but this only works when paired with the jumping system. Instead of being procured based purely on order of masters, the dialogue tree checks every time the parameter called "jump". The jump parameter came into play later in the development of the dialogue system, being used to guide the dialogue tree to the next master to present. When the parameter jump exists in a dialogue instance, the tree knows that the next master to fetch is stored in that jump parameter. The creation of the jump wasn't necessary for the regular dialogue system at the beginning, but since this game is meant to be an RPG, there should also be dialogue choices that could alter the fate and outcome of the game. From this a new concept was created, that being a "Dialogue Choice", and this in turn made the jump parameter a necessity.

With the 3 concepts introduced, the Dialogue System class diagram can be structured as seen in Figure 27.

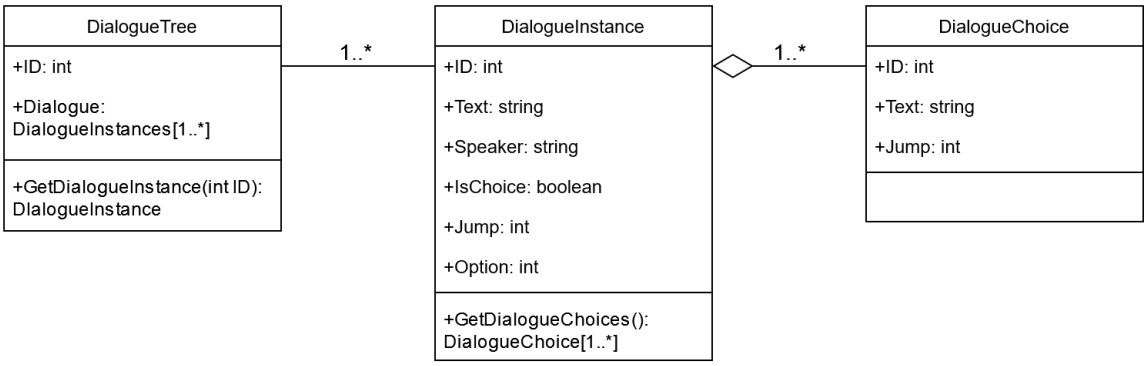


Figure 27 – Simplified version of the Dialogue System class diagram.

The most complex part of implementing a dialogue choice system was to incorporate it with the way the dialogue was stored outside the game. As it can be seen in Figure 28, the dialogue is stored in CSV files, making them easier to read and to structure the information in. With the Unreal Engine's native CSV import into data structures, the information could be created outside of the engine, in an application like Excel or Google Sheets and then imported into the game engine. With each file separated per level, the dialogue instances and choices were all stored there. As its possible to confirm in Figure 28, the master is set without a header name, serving as a row name for the UE5's data structures, followed by the Dialogue Tree ID. There are specific parameters like Quest IDs that are present to allow a quest to progress triggered by a specific instance of dialogue, which is an important variation to the quest system.

Apart from those parameters, the most important for the way the class logic works are the following:

- Choice – Boolean.
- Option – Integer.
- Jump – Integer.

	A	B	C	D	E	F	G	H	I	J	K	L
1		DialogueID	Speaker	RGB	Sentence	Choice	Option	Jump	QuestID	QuestObjectiveID	QuestOrder	QuestEnd
2	1	1	Evelyn	(1,0.23,0)	What is happening here?	FALSE	0	0	null	null	0	FALSE
3	2	1	Woman	(0.23,0,0.3)	(Screaming) By the godess, t	FALSE	0	0	null	null	0	FALSE
4	3	1	Evelyn	(1,0.23,0)	What? Who?	FALSE	0	0	null	null	0	FALSE
5	4	1	Woman	(0.23,0,0.3)	(Screaming) Someone stop t	FALSE	0	0	null	null	0	FALSE
6	5	2	Narrator	(0.13,0.3,0)	A man protects his children,	FALSE	0	0	null	null	0	FALSE
7	6	2	Evelyn	(1,0.23,0)	Is everything ok?	FALSE	0	0	null	null	0	FALSE
8	7	2	Man	(0.05,0.12,0.3)	Let's go kids, we have to go i	FALSE	0	0	null	null	0	FALSE
9	8	3	Narrator	(0.13,0.3,0)	As Evelyn approaches the en	FALSE	0	0	q1	q1_o2	2	FALSE
10	9	3	Narrator	(0.13,0.3,0)	Her eyes start to water as sh	FALSE	0	0	null	null	0	FALSE
11	10	3	Narrator	(0.13,0.3,0)	The family is surronded by ru	FALSE	0	0	null	null	0	FALSE
12	11	3	Narrator	(0.13,0.3,0)	There is not a single sound c	FALSE	0	0	null	null	0	FALSE
13	12	3	Evelyn	(1,0.23,0)	I can't believe this, they've g	FALSE	0	0	null	null	0	FALSE
14	13	3	Evelyn	(1,0.23,0)	I must leave this place!	FALSE	0	0	q1	q1_o3	3	TRUE
15	14	4	Narrator	(0.13,0.3,0)	Still in shock, Evelyn is prepa	FALSE	0	0	q2	q2_o1	1	FALSE
16	15	4	Narrator	(0.13,0.3,0)	From deep in the forest, she	TRUE	0	0	null	null	0	FALSE
17	16	4	Evelyn	(1,0.23,0)	Is someone there?	TRUE	1	18	null	null	0	FALSE
18	17	4	Evelyn	(1,0.23,0)	(Remain Silent)	TRUE	1	21	null	null	0	FALSE

Figure 28 – CSV file structure for dialogue related data storing.

When reading the CSV all the instances are organized by dialogue (tree) ID. By checking each line for the presence of the choice Boolean flag, the Game Mode

class was able to decern what instances were followed by dialogue choices. When found to be true, the next few lines in the CSV files were no longer being considered dialogue instances, but now dialogue choices. This then led to the possibility of optional dialogue choices, meaning that pressing them would grant the player with information, without needing to decide yet to progress the dialogue. The player can distinguish between them by looking at their colour in the UI. Marking these was simpler, with the option parameter being 0 for non-choice instances, 1 when the dialogue choice was not optional and 2 when the choice was optional. When the player picked the dialogue option by clicking on it with the mouse or by pressing its respective number presented in the UI, the dialogue tree would know to look at the jump parameter. To create all of what has been explained, the engine fetched line by line the CSV file and updates the classes according to each instance variables. If choice is false in a specific line, then a new dialogue instance will be created within the same tree. When it is true, the system will instead get the last dialogue instance created and add to it a new dialogue choice that has a jump parameter to what master that choice will lead if picked. This means that an instance knows if it has choices within, making it easy to organize and present the information to the player. This process is represented visually in Figure 29.

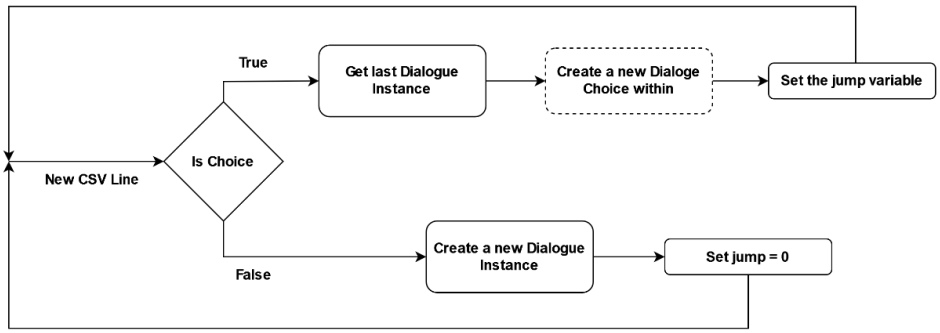


Figure 29 – Visual representation of the creation of dialogue instances.

Now with the dialogue tree completely structured and created the access to it becomes easy. When a dialogue tree ID is called for, the Game Mode class searches for the right tree and requests from it the first instance of dialogue. This request repeats once the player skips to the next instance of dialogue. Each time a request is made, a new dialogue instance is retrieved and checked to see if it is a dialogue choice instance, meaning it has dialogue choices within. If false, then the new dialogue instance is fetched by incrementing the previous master and prints it to the screen, with the speaker and corresponding sentence. If it is false, then all dialogue choices present within are fetched and the UI prints both the speaker, sentence, and all dialogue options to the screen. This sequence is visually represented in Figure 30. When the dialogue choice is chosen, the dialogue tree uses the jump parameter of that choice to discover the correct master to jump to and fetched that dialogue instance instead.

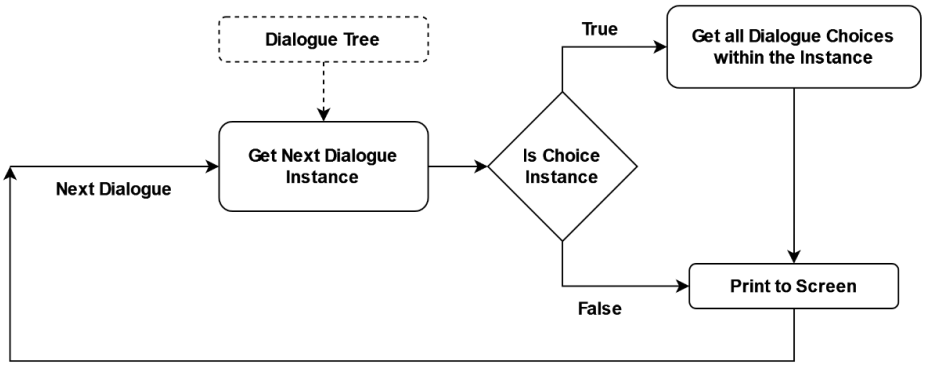


Figure 30 – Visual representation of the Dialogue fetching system.

Each dialogue instance can also have a pre-requisite, usually a quest, to allow for filtering of the dialogue options. This means that if a player has not completed a quest that would grant them relevant information for the current one, the player will have less dialogue options to choose from since they don't meet the pre-requisite.

This system is the most complex built specifically for the game. Not all the functionalities are crucial to properly present a story to the player as originally intended, but together all the parts make the system more robust and grant the developer a better way to tell the story, with the opportunity to make it interactable via the dialogue options. It is also important to mention that if the developer chooses to, the dialogue option system is completely facultative. If it is not being used, every choice parameter is set to false in the CSV and the dialogue tree will keep incrementing masters, presenting an entire dialogue tree with no problems. An example of the dialogue system with choices can be seen in Figure 31.

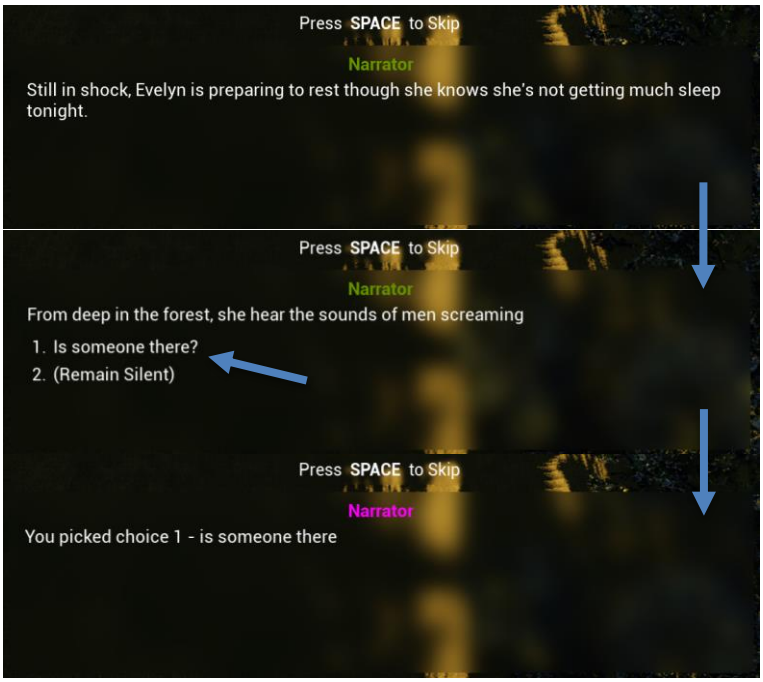


Figure 31 – Example the way the dialogue system functions with choices.

5.2.5 Combat Trigger

Since this game is an RPG, there is a combat system that will be explained in detail later in this chapter. To begin each combat encounter there needs to be a system that can trigger it, and like the stage triggers, the "Combat Trigger" was created for this purpose. A stage can be created and set in a level just like the stage trigger used for story, except this one has a slightly different parameterization due to its different use case.

In addition to the normal quest parameters of Quest ID, Quest Objective ID, and Quest Order, that allow the progression of quests even when entering a combat encounter, there is also the same Dialogue ID parameter allowing for pre combat conversations that can even be used to avoid the encounter. This grants the player a choice that allows him to avoid the combat encounter altogether. This is ideal for RPG since part of the interest of these games is the choice making aspect. Lastly there is also a Load Level parameter that tells the game what level is going to be loaded if the combat starts.

The other new parameters are as follows:

- Combat Encounter ID – String.
- Auto Trigger – Boolean.

The Combat Encounter ID parameter is used to determine what combat encounter will be loaded. At the beginning of the game, all the different combat encounter scenarios are loaded into the Game Mode class, each of them having a unique identifier that can be used to fetch their information.

The Auto Trigger parameter is simply to quickly define if the trigger should immediately load the combat level, or if it is worth checking the Dialogue ID to spawn the dialogue box before the combat begins. If the parameter is true, the trigger immediately loads the level specified in the Load Level parameter and begins the combat, triggering the in-game events that change the UI and load all the combat assets.

In Figure 32, the combat object can be seen spawned into the level. Just like the stage trigger, it can be edited and morphed into any direction, allowing for versatility for the developer to make it the shape they require. It is also important to note, the orange cylinder in the centre of the stage is only meant to allow the developer to see and select the object better. It is also colour coded so that the developer knows the difference between a stage trigger and a combat trigger without having to select the object and checking its properties. Like the stage trigger, the object itself has no collision with player or non-player characters and is hidden when the game is running.

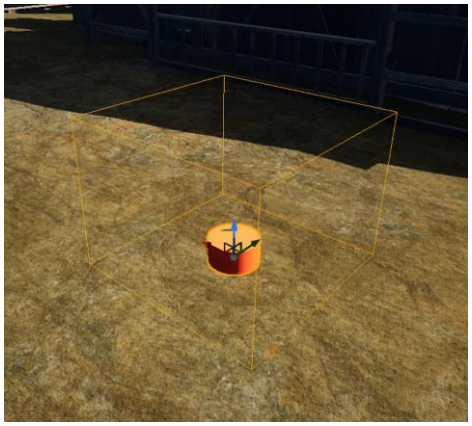


Figure 32 – Example of default stage trigger object spawned in a level.

5.2.6 User Interface

Just like most computer programs, the game being developed requires a user interface to be designed.

Since the game isn't very complex, the interface should reflect that, showing only the essential elements, at each moment, keeping the UI easy to read and clean for the player.

Using the already showcased GUI Editor provided by the Unreal Engine 5, the learning process was now separated into two phases. First, the phase where research and testing would be done to learn how to use the editor. This would grant some comfort and knowledge of what was and wasn't possible to achieve with the current experience level and default tools. Second, it was important to define what the interface should look like for this game.

Researching how the GUI level editor works took some time. Initially some features appear to be more approachable than they are. For instance, creating lists of things in the editor is not a simple task. The editor in real time cannot create elements of a list. This means that if a list element has for instance one text parameter, it cannot be changed by simply calling its "Set Text" function. This is unintuitive for novice developers and when thinking strictly from an object-oriented programming point of view, which most of this game is designed around. The solution is to create listeners in the UI linked to variables and then update the variables. This makes it so that the list is created and instead of using set functions, the parameters just automatically update because of the listeners.

Similarly, adding and removing widgets to the screen is more complicated than initially expected. Adding widgets to the screen requires good management on the part of the developer. All the event triggers remain operational even if they are not at the forefront of the UI. This can cause problems where, for instance, the player consults the Quest Journal and within it calls for an interaction or a dialogue skip. If not controlled, this will call the event listeners that are not active causing errors and might cause the game to even crash.

Following the research, the understanding was enough to start building something in-engine as a proof of concept. First, a new HUD class needs to be created. This class will be the main Heads-Up Display (HUD) of the whole game. This class is derived from the default HUD class and will serve as a canvas for widgets to populate. Each widget can hold as much of the screen as the developer so chooses, but as previously mentioned, there must be some discipline when accessing and changing their information.

Now it is time to analyse the UI created for the game as seen in Figure 33. When creating the interface, several containers were formed (marked in blue), each with a specific purpose. A container is an object that can be created in the Unreal Engine graphical user interface editor to wrap smaller objects. Within each of container, there were several text fields that can be modified in real time by other systems. The advantage of an approach like this is that the whole container can be manipulated at once as well as in parts. This becomes useful when the information present in the container is no longer needed and can be hidden. When this happens, the interface will trigger a hide function and the whole container becomes invisible in the UI until it is required again to display updated information.

Figure 33 – User Interface of the game in exploration mode. Information containers are marked in blue.

This interface is directly connected to the Dialogue and Quest System, being updated when those systems are triggered. Those systems can modify the content of each container at will, being directly called by the functions that process and update the information. For example, when a player triggers a stage trigger object, the event "Update State" is called and the quest progression is confirmed by the quest system, then the quest log in the top right of the screen is updated, displaying the new quest or quest objective. Similarly, the dialogue box is connected to the dialogue system, displaying the correct sequence of each dialogue tree. When a player presses the next dialogue key (by default the spacebar), the dialogue system is called upon to fetch the next instance of that tree and automatically updates the screen. In the specific

case where the dialogue instance is also a choice instance the UI creates the respective buttons for each dialogue option.

For the occasions when a dialogue option is pressed, the UI becomes the triggering system, feeding the option back to the dialogue system. It processes the right continuation of the dialogue and feeds back to the UI the correct following dialogue instance. This is unconventional, since all other key inputs are processed by either the game or player controller, but it was the easiest and most intuitive way to implement a system of this kind.

When entering combat, the UI needed to change, showing different information to the player to allow for a focused and streamlined interface. This happens by creating a whole new widget with the new layout and setting it as the new overlay. As seen in Figure 34, most of the UI from the exploration mode was removed, adding new widgets to display the relevant combat information (marked in blue). The only widget that remained was the health and level widget in the top left corner of the screen, due to its relevance both in and out of combat.

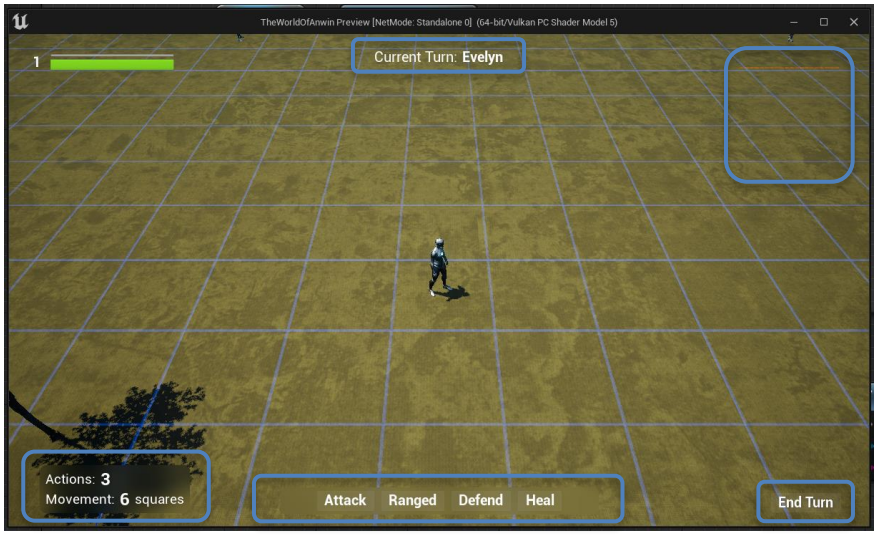


Figure 34 – Combat User Interface developed for the game.

Like many of the examples shown in chapter 4, the UI is heavily inspired by some of the best RPGs ever developed. To keep the screen as clean as possible, only the bottom widgets have delimited backgrounds allowing them to be easily seen. By contrast, the top centre widget that displays the current turn order and the top right stats page are completely transparent. The stats page in particular was required to be transparent due to it easily filling up the screen, impeding the player perception of the battlefield. At the bottom, the attack information is placed in the centre allowing for easy reach from wherever the mouse is on screen while the "End Turn" button is set aside preventing it from being accidentally pressed by the player, ruining the turn. On the opposite side, the current turn information most relevant to the player is displayed. More information could have been shown in this area, but it would cause the corner of the screen to get too bloated with information and it was decided against.

The moment the combat ends, the UI switches back to the exploration mode. This is done by setting the previously created widget as the current widget being overlayed on the HUD.

To interact, the player in its turn can use the centre bottom bar to select the option they want to use, and then pick a target. Although the game does not have a visually represented attack range for each attack, it will inform the player if what they are trying to do is impossible. This would happen in a situation where, for example, the player is trying to perform a melee attack at a distance. After picking the correct attack, they will then enter a "Choose Target Mode", being prompted by the UI to pick the target. They can then choose the target by left clicking it and after doing so all the information is processed by the combat system, the stats are rolled and printed to the screen, and the actions, health and respective widgets are updated. Alternatively, the player can also move with the character, allowing them to get a better position.

5.2.7 Save, Load and Menus

In this subchapter, the save and load systems will be explored and explained along with its accompanying main menu.

The majority of video games has a main menu. In it the player can control several options of the game including starting a new game, loading a previously saved one, configuring the options of the game and exiting it. This meant that there was the need for the creation of another system to incorporate the main menu and all its features.

Thankfully, since saving and loading games is such common practice among the video games of today, UE5 has system in place to facilitate the creation of these systems. To start creating a Save and Load system it is first required the creation of a Save class. This is done by deriving the already built in Blueprint for saves. With this class the custom object and classes that are required to be saved can be setup inside the it. For this game, there are 3 container classes that store all information within the game, and these are:

- Game Mode – class that stores all the static data of the game, as well as all the quest system related information like missions, flags, and level information.
- Game Controller – class that controls most of the logic of the game, thus storing some temporary information that needs to be saved to properly load.
- Player Controller – class that stores all the player character related information like their stats and position.

With these object references present within the Save class, the game runs as normal until the player decided to save a game. To do this, they must press the P (pause) key that opens the Side Menu seen in Figure 35, and the options are

presented. This menu pauses the game, and the player can then choose from all the options available in the menu:

- Continue – continues the game closing the side menu and resetting all other widgets.
- Save Game – opens the save menu where the game can be saved.
- Load Game – opens the load menu where a game save can be loaded.
- Exit to Menu – returns the game to the main menu.

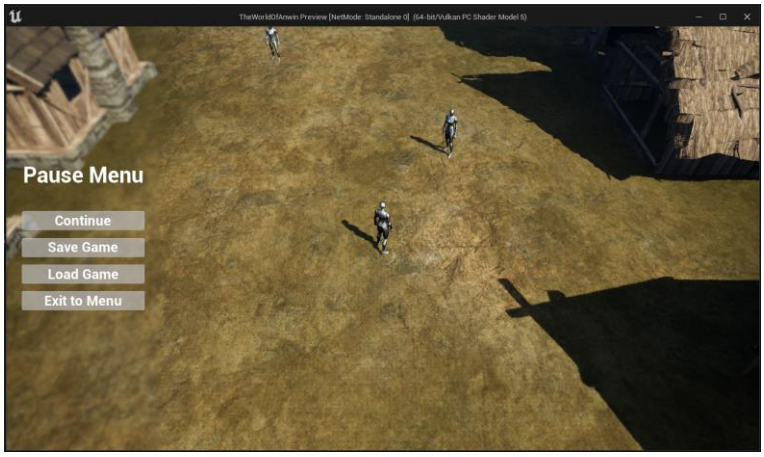


Figure 35 – Side or Pause Menu of the game.

When the player wants to save the game, they pick the save game option, choose a name and the game is saved. This name from now on become the unique ID of that save file. When loading the game, the name of each available save is presented to the player. Loading a game will cause the game progress to be lost, and the game will be loaded in the new location and with the stored game state. It is important to reiterate that the game can only be saved when not interacting with an object or NPC and when not in combat. If the player tries to do this and presses the P key to open the Side Menu, the save game option will not be enabled.

When it comes to the Main Menu of the game, as seen in Figure 36, it is shown and loaded at the start of the game. From there, it is possible continue the latest saved game if there is one, load other saves of even start a new one. It is from here that the game can be closed, by pressing exit.

For the game in development, there was no immediate need to create a way of managing the options and settings of the game, so the options button in the main menu is only there are a placeholder for a possible future implementation of an options menu.

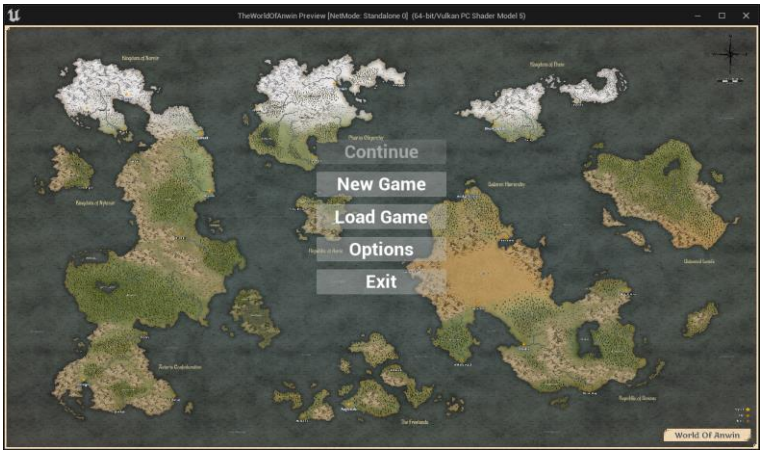


Figure 36 – Initial Main Menu of the game. The background image of the menu is the map of the fictional world the game is set it, created by the author.

5.3 Building the Combat System

Building the combat system was an interactive and demanding process. The requirements that were established in chapter 4 were that it needed to facilitate the implementation of the RL model with its features, allowing it to control the adversary agents during an encounter. During this subchapter, the decisions and process of coming up with the game's combat system will be explained.

5.3.1 Combat System Development and Logic

Initially the combat was pitched as a turn-based system, with a controlled movement range for each character and these specifications are quite broad. There are easy and complicated ways to achieve this. Initially, test and prototypes were created to see if there was some way to create a system that didn't function in a turn-based manner. This stemmed from the potential of the game engine being used, coupled with exciting possibilities and demos that were researched at the time. The prototype yielded interesting results but even then, the idea of how the AI implementation of the enemies could have been connected to a RL model was hard to turn into a concrete proof of concept implemented in-engine. Due to this, the idea was abandoned, and the combat pivoted back to the initially conceptualized turn-based type of combat.

Following from the attempt at the real-time combat system, there were two options to develop the new combat system, having the turn-based style in mind. This time there was a special focus on how the system could use the RL model and how they could communicate without much friction. For this it was important to remember a few concepts about video game environments and how they can be represented. Like previously explained, the idea of a video game state is important in this scenario specifically, because the better a video game state can be portrayed or calculated, the better the integration with the RL model will be.

With a simple system, like one without a map and just a text interface, the combat system can easily be described in a state, without having to worry about the position of all the actors in the environment, which is always a major difficulty. During the development of another prototype, it quickly became apparent that although this system was better for state representation, it lost a major visual component, and the combat experience was stale and unengaging to play. Another idea was required.

The final prototype was the idea of a simple system, grounded in the idea of a grid-based system. This would take direct inspiration from the game displayed in the conceptualization phase "Into the Breach", where the characters would move in this grid, with a limited movement speed, allowing for better definition of the position related part of the game state definition which will be crucial when creating and linking the model to the combat system. Still the game state could be represented in several different ways. For instance, the state could be represented by the position of one actor on the grid, which means that for a grid of 20x20 squares there would be, assuming no more stats are included in state definition, 400 states to consider, one for each place an actor, friendly or not, could be in. This is not optimal and after some consideration it also became clear that it wasn't necessary. There is no need to know where an actor is on the grid, removing most of the complexity of the previous method. Instead, the only relevant information was just how far from its target an actor is. In this new scenario there would be only 3 possible "position" states to consider also called "Range Increments":

- Has target in melee reach – the melee ability can reach the target.
- In ranged attack reach – can reach the target using the ranged attack ability if it has one.
- Out of reach – cannot reach the target with an attack.

At this point, there are 3 states in the game, but this isn't enough to make a comprehensive analysis of the current state of all actors. But first, to establish other possible states, the combat rules needed to be established. Like previously described during chapter 4, the combat would give each actor in the combat a specific number of actions that could be performed and a maximum move distance. Now that the system has been confirmed to be a grid-based one, the movement needs to be counted in squares, with each character only being able to move a fixed number of squares per turn. When it comes to the actions that each actor can perform per turn, a few systems were considered. Instead of creating systems from scratch, rules from other systems could be adapted. Many tabletop RPGs follow a similar grid-based combat, and each has a slightly different way of controlling a character's actions per turn. In this case, a system inspired by Pathfinder Second Edition by Paizo [49] was used, where each turn, a character has 3 actions and can do whatever it wishes with them. At the start of their turn, the number of actions would be reset and 3 more would be available.

5.3.2 Combat Stats

Now with the 3-action economy per turn defined, the statistics used for in-game entities need to be established, as well as what types of that can be performed.

In terms of determining statistics, once again the inspiration was the tabletop RPG Pathfinder Second Edition. In Pathfinder, the characters statistics are defined by among others, their Health Points (HP), their Armor Class (AC) which is how hard they are to hit, and their attack and damage bonuses. Each time an attack is performed by an entity in Pathfinder, the player must roll a d20, which is a 20-sided die, and use the number obtained in the die to determine if the attack was successful. This is confirmed by adding the attack bonus, which is specific to every character, to the roll of the die, and then comparing it to the target's AC. If the Armor Class of the target is greater than the number rolled plus its bonus, the attack misses. If the number rolled is equal or superior to the target's Armor Class, the attack is successful. There is an extra layer of depth to this system though, since any attack that surpasses the AC value by 10 or more points is considered a "critical hit" to the target, dealing double the damage it normally would in a non-critical hit. The following stage after hitting an attack is to calculate how much damage is done, which also required the use of a die. Normally in Pathfinder the player has different weapons that each deal different amounts of damage numbers but, for the sake of balancing and reducing complexity, the damage die in-game is set at a d10 (10-sided die) for the melee range damage and a d6, (6-sided die) for any ranged attacks. After rolling for the damage numbers, the modifier for damage is added to the roll, each character having its own modifier, and doubling it if the attack was a "critical hit" (rolled more than 10 above the AC of the target). With the damage calculated, the only thing left is to deduce the number of points of damage to the targets HP. If this value ever reaches 0 or below, the target is considered dead. The Pathfinder logic presented was accurately adapted to the game. The attacking process implemented uses all the mentioned modifiers and die rolls to determine their success and damage. Since the Pathfinder also has a grid-based combat system and has similar abilities, this logic transitions seamlessly to the video game in development.

This consolidation of the attack logic is visually represented in Figure 37.

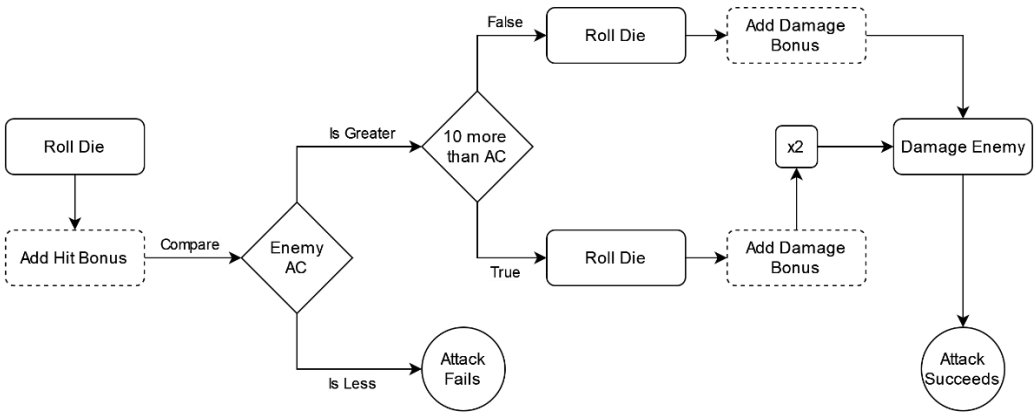


Figure 37 – Visually representation of the game Attack Logic, inspired by Pathfinder 2e.

5.3.3 Combat Mechanics

With the combat statistics now defined, it is time to move on to establishing the combat mechanics, or how the game will work on each turn. The combat mechanics for the game needed to be simple enough due to the low complexity of the game but at the same time allow for enough variety to happened between different turns. Here are the defined actions that can be performed by both the player and enemy characters:

- Attack – costs 1 action and requires the target to be in adjacency, meaning in the 8 squares that surround his current position.
- Ranged Attack – costs 1 action and can be performed at a range of 2 up to 6 squares of distance from the wanted target.
- Defend – costs 1 action and has a target of self, meaning that it can be performed anywhere, raising the AC by 2.
- Heal – costs 1 action and has a target of self, healing its HP a pre-determined amount.
- Improve – costs 2 actions and has a range of self, and allows the character to level up, increasing their AC and attack and damage bonuses. This action is exclusive to the AI since the player shouldn't improve outside the levelling system.
- Move – costs 1 action and has a range determined by the combat encounter file where all entities stats are specified.

The action picked are the ones that were deemed crucial to have an interesting variation of actions and something that the RL model could then work with without increasing its complexity unnecessarily.

5.3.4 Game States

With the mechanics and other concepts now explained, there was the need to enhance the definition of the game state. Describing the state based on the distance to the target alone was no longer possible due to the increased complexity granted by the newly added game mechanics.

First, with the goal for any of the entities to eliminate their identified target, the game needed to have a win state that in the simulation and during RL model training could represent a win scenario. There must also be a loss state to represent the death of the entity. Although an entity cannot put itself in the loss state, meaning that by picking his own actions, it can never directly lose the combat, outside entities can. The only entities that can do this are the player character in the case of an adversary agent, and the adversary agent in the case of the player character. Both enter the state when reduced to 0 HP due to being attacked.

Taking into consideration the new stat-based combat system and the mechanics that were previously explained, it is also important to consider the entity's stats when defining states. Actions like Heal and Improve (exclusive to AI agent) are important and can grant advantages to the entity, but in a scenario where they can attack, they will probably choose to do so. This led to the creation of new states for the state machine. These new states need to consider the most important part of the entity's stats which is its HP. Once this stat goes to 0, no more actions can be taken, so there was the need to create states where the entity is in critical health. The only thing left to consider is how many new states must be created to accommodate this new idea. Since there are 3 distance related states (range increments), it is only logical that there would also be 3 more distance related states with a critical health condition added to them. This means that the final number of entity states is 8.

Win State:

1. Win state – has reduced its adversary to 0 HP.

Melee Reach States (Within reach):

2. In melee attack reach– the melee ability can reach the target.
3. In melee attack reach but health is critical – the melee ability can reach the target, but the entity has less than 30% health.

Ranged Reach States (Within reach):

4. In ranged attack reach – can reach the target using the ranged attack ability if it has one.
5. In ranged attack reach but health is critical – can reach the target using the ranged attack ability, if it has one, but the entity has less than 30% health.

Out of Reach States:

6. Out of reach – cannot reach the target with an attack.
7. Out of reach and health is critical – cannot reach the target with an attack and entity has less than 30% health.

Loss State:

8. Dead State – was reduced to 0 HP.

Like previously mentioned, there are some states that the agent cannot purposefully enter. These are states that would not in any circumstance be beneficial to it. An example of this would be to purposefully enter a critical health state or the dead state. This means that there is no state transition between these states and the other "normal" states.

With the actions and states now defined, the video game state machine can be created. For this the previously defined actions and range increments need to be considered. The differentiation of the range increments will be crucial in the creation of the agent's logic, but it isn't for the state machine. While the agent needs to know

if it is in melee or ranged reach, for the state machine it is only important if the agent can or not attack. For this reason, and only in the state machine, the agent's range increments can be grouped into Within Reach states and Out of Reach states.

In this state machine it is important to represent the point of view of one agent while acknowledging the intervention of the adversary. Since the adversary turn is completely out of control of the current agent, the health state does not matter, being excluded from the state machine. To distinguish the adversary states from the current agent states, they are marked with a dotted line.

This results in a state machine that has 10 states. This state machine can be viewed in Figure 38. Once again, it only represents the point of view of one entity. This entity can be either an AI agent or the player character. Lastly, the state machine is presented in an easy way for the user to understand, not considering design standards.

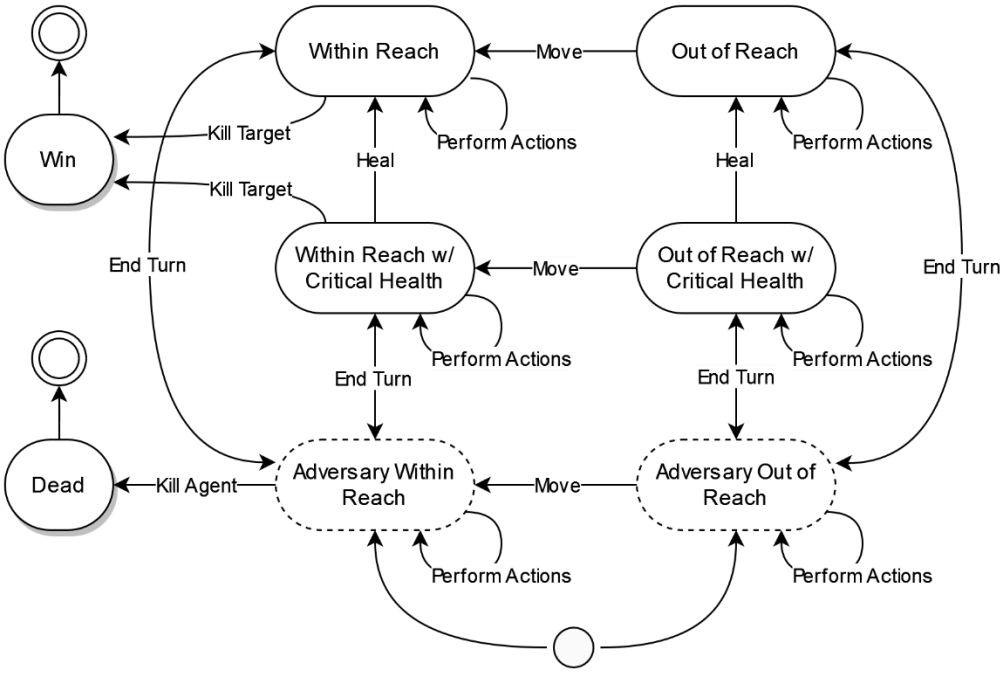


Figure 38 – State Machine of the video game in development. Note: Perform Action includes all the actions possible including attacking and moving. Some actions are isolated when they generate state transitions.

5.3.5 Combat Encounter Storing and Processing

Similar to other previously explained systems, the combat system needs to have pre-established combat encounters. These would be created in advance by the developer with all the entities to spawn once the combat started and their respective stats. Once again, a CSV file was created to store this information, which can be seen in Figure 39.

	A	B	C	D	E	F	G	H	J	K	L	M
1		CombatID	EnemyID	EnemyName	HP	AC	HitBonus	DamageBonus	XP	Abilities	Movement	Mode
2	1	1	1	Enemy 1	25	15	7	7	30	(1,4)	50	1
3	2	1	2	Enemy 2	30	17	5	5	10	(1,2,4)	40	2
4	3	1	3	Enemy 3	20	14	5	5	10	(1,2,4)	40	1

Figure 39 - CSV file structure for combat encounter related data storing.

Once again, following the Unreal Engine importing requirements, the first column of the file has no header, representing the row name for the Data Structure created in engine.

Following the same thought process as all other information stored in CSV files, each line matches an entity that would be spawned in the combat map. Each enemy has a unique ID that identifies it in-game and a name that is presented on top of them during combat as a way for the players to identify them easily. This is done with the help of a widget that will be explained in detail later.

Like the storing method used in other systems, although each line represents an entity to be spawned in combat, there is a grouping column that specifies which combat encounter each entity belongs to. This is the Combat ID variable, a unique ID that represents each combat encounter and that is used to tell the Combat Trigger what level to load when it is triggered.

The other columns in the file represent the statistics of the entity, like HP, AC, and their bonuses to hit and damage. The abilities column specifies what abilities each entity has, allowing for a creation of different types of entities that can have more or less of the abilities previously enumerated. Movement represents the movement in feet, that is then translated into squares. This comes from the original inspiration for the grid-based movement: "Dungeons and Dragons", which uses the imperial system to measure distance travelled by a character in a grid, with each square of the grid representing 5 feet. This means that a character with 30 feet of movement can move 6 squares on the grid.

Lastly, the mode column represents what mode the entity is operating on. This was added later in the development of the combat system, creating an alternative way for the entities to behave, with mode 1 being powered by the RL model, and mode 2 having randomized actions. A toggle like this was required for testing of the results the model, comparing them to a random choice algorithm.

When the Combat Trigger is activated, a new level is loaded. This was a decision made to ensure that all the new systems that were in play in the combat mode of the game didn't come into contact or conflict with the ones from the exploration mode. When the combat ends, the previous level is loaded back up.

When loading the combat level, the Combat ID in the trigger is read and if it represents a known combat encounter, it loads the correct level for that encounter. Initially the objective was to spawn entities according to the ones present in the encounter, but there were difficulties spawning entities in random spots on the map

and without the knowledge of how to create spawn areas for entities, the focus shifted to populating the already spawned entities with their stats. This was achieved by loading them at the same time the level was being created, in its starting script, and then all the placed entities would correctly be populated one by one, with the developer needing to ensure that all spawned entities had a stat that could be attributed to them.

To create the pawns in the combat arena, a new class was created called, "Combat Interaction". Like a normal NPC Interaction Trigger, this class has a 3D mesh of human attached to it to represent the adversary and it also has a few more features. Firstly, it incapsulates another class called "Spawned Entity". This is a generic class, used in the turn management system, that stores all the information of the adversary like HP, AC, movement, as well as its abilities. Since this is a generic class that isn't tied to either the player character or the adversaries, it is used to represent both in the turn management system that will be explained later. The combat interaction also has a special widget, as seen in Figure 40, that is tied to the 3D mesh representing the enemy's health and name. This widget is always set facing towards the camera and it has a listener directly connected to its spawned entity health variable. As soon as the damage is updated in the spawned entity class, the widget automatically updates itself showing the health, converted into percentage, of the adversary. There is also a camera that is attached to the entity, at a much higher altitude, that will be used to represent its point of view when its turn to act starts.

The main character is the one that is truly spawned in the level. When it loads, the character has a specific place on the map, representative of where the combat encounter started in the previous exploration level. Its stats are more complex than the adversaries, but not all are required for the combat encounter, so the stats of the PC are copied over from the Player Controller class that holds them and another spawned entity object is created. This stores all the same stats as the adversaries except for the "Is Ally" Boolean flag, that allows the game to differentiate the two types of entities from one another.

Figure 40 – Example of combat interaction entity with name and health widget floating on top of its 3D mesh.

5.3.6 Turn Management System

As established earlier, the combat system is meant to run as a turn-based system, where several entities cycle turns in combat, each making their own actions before finishing their turn. For this to be possible, a turn management system had to be created since Unreal Engine 5 did not possess a feature like this by default.

The first step was to create a way to represent every entity in the turn order, making it then easier to cycle between them. As explained earlier, with the creation of the "Spawned Entity" class, this problem was already solved. This class could represent both player characters and adversaries in the turn.

When the combat level was loaded for the first time, and after the already spawned hollow entities were populated, the level would also add them to the Game Mode class that adds them to an array. After all of them are added, the array is then shuffled, with the first element of that array always being the player character.

When a character's turn is over, either by pressing "End Turn" in the case of the player character or by running out of actions to perform, the Game Mode then calls the function "Advance Turn" as seen in Figure 41. This is one of the most complex functions in the game, verifying numerous conditions and managing the AI algorithms at the same time.

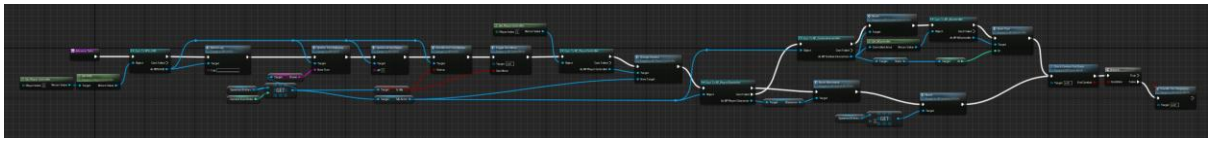


Figure 41 – Function "Advance Turn" programmed with Blueprints, exemplifying the complexity of some of the functions created for the game.

This function has the task of updating all the elements required to continue the combat on the next turn, like updating the log to represent a new turn as started, or resetting the action of the character that finished his turn to be ready for when their turn begins again. This function also controls the camera that each agent has, switching between them as the turn moves along. When a new turn begins, the AI Behaviour trees need to be signalled that they can start running again, allowing for the entity to perform their actions powered by the RL model. Lastly, the UI is updated to show on the top centre of the screen which entity's turn it is.

5.3.7 Combat Grid

Once it was decided that the combat would work on top of a grid system, a new challenge arose. Although it is easier to create and manage game stats with a grid system, the new challenge was how to create a system that generates the grid where the spawned entities will walk on. This required research to be done, since no real solution is immediately clear.

The solution that came up was a creation of a grid using the default cube mesh Unreal Engine provides. In this approach a complex mathematical equation had to be programmed to allow for the creation of a flexible size grid. This system was also able to create an instanced or not instanced grid, meaning that the grid could be processed as several cubes, or as just one object. The parameters were also important to establish. Not only was the grid flexible in size, with the developer being able to insert the width and height of the entire grid in cubes, the scale of each cube in relation to the default cube asset was also customizable. This meant that cubes could have the size required and that the developer can create a grid the size they desire with no compromise. The combat grid can be viewed in Figure 42.

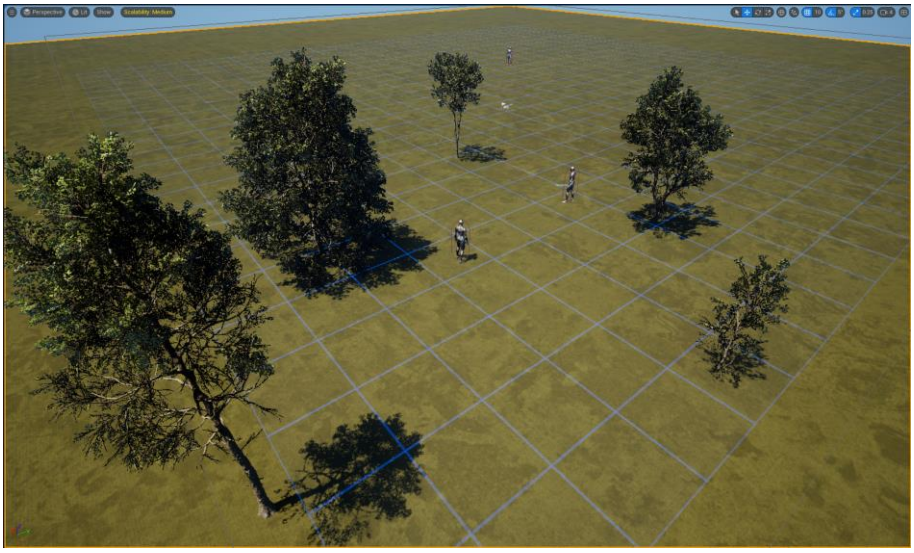


Figure 42 – Example of combat level with a Combat Grid of 25x25 squares and Combat Interaction class entities placed in the environment.

There were problems that needed to be solved with the creation of the grid. The first one was how the entities would understand their position relative to the combat grid in geometric coordinates. The second problem was how to represent the combat grid in a way that the player would be able to see the edges of each grid cube but also the environment level below.

After some time understanding how each instance of the grid functioned, it was clear that it wasn't possible for each cube instance to know its own coordinates the way it was created. There was the option to go back and redo the grid creation system but with its complexity and no clear better options, another alternative had to be conceived. By placing the cubes at the absolute world coordinates of (0,0) the placement of the centre of each cube could be calculated using the cubes dimensions and the actor's location. Using the same method used for the Triggers, once a character was over a cube, the event "On Component Begin Overlap" would start a calculation of its position and transfer that information to respective spawned entity. This calculation must consider the size of the cube inserted in the combat grid parameters. Since there is no easy way to allow for the cube to know its own dimensions, the size of it became hardcoded at 2.5x the normal cube scale. With this

scale set, it is easy to calculate the coordinates of the spawned entity, by dividing the cube actor location by the cubes dimensions, resulting in x and y coordinates.

Initially, the combat grid was either totally visible or totally hidden in-game like one of the triggers developed. This was not the ideal scenario since although the game knew where the player and other entities were, there was no way for the player to know the relative location to their targets. This problem had to be resolved in a creative way, and since there was no way to keep the grid below the environment without affecting the position detection of the entities, the material used for the cube had to change. After some research and testing, a new material was created that had a transparency factor to it. This allowed for a clear layer to be present on top of the map, all that was required now was to find a way for the edges to have a different look, that would allow them to be visible. This material is presented in Figure 43 applied to its cube mesh.

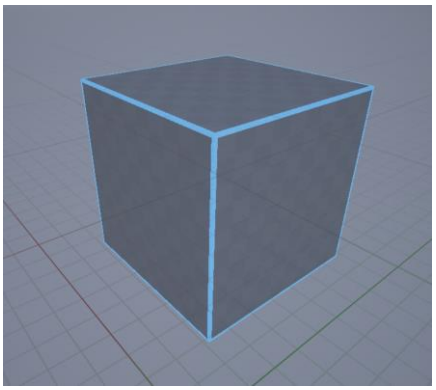


Figure 43 – Default UE5 cube mesh with altered material to allow for transparency and edge glow. Note: the transparency in the cube was reduced for the purpose of demonstration.

With all these techniques and rules implemented, the combat system was ready to be played. There was only the necessity to test the end and win scenarios of the combat. The win scenario of the player character was when the adversary entities were eliminated (reduced to 0 HP). At this point the game would load back the exploration level and the game would continue with a completed flag connected to the combat encounter, so the Game Mode class knew that the combat was already completed successfully. When an adversary entity is reduced to 0 HP, their body physics are turned off and it enters a "ragdoll" mode slumping to the ground. When the enemy entities win, the combat enters a game over scenario and starts again.

Like previously explained, to play the game, the player was required to first choose the action they were taking and then choosing its target. This was achieved by listening to the "On Clicked Mesh" event in each Combat Interaction object, which would inform the Game Mode class that this was the intended attack target. The class would them process all the necessary stat die rolls and determine the outcome based on them. Each roll that was performed was sent to the UI to be printed in the stat box so that the player could understand what was happening. The Game Mode class is also responsible for managing the other types of actions the

player can take like healing, defending, and moving, allowing for movement based on x and y coordinates of the combat grid.

5.4 Cut or Truncated Systems

During the description of the game in chapter 4 several systems were mentioned and explained that had to be cut or truncated during development. These decisions were all in order to focus the development more on the parts that really mattered to reach the final goals, ensuring that the RL model was fleshed out and functioning within the combat system.

There were a few minor systems that got cut or truncated during the development of the game. The first one was the class system that got completely cut. This was an auxiliary system which objective was to allow the player to play as a few different classes, granting greater variety and even some "replayability" to the game. This means that when starting the game, the player has no option to create a character and choose their class. Since the menu system was one of the last ones to be created, it makes sense that this feature was also left for later in development, ending up being cut due to the lack of time and the fact that it didn't add much value to the game since it wasn't going to be complete. Although truncated, the main character class and information still has the statistics variables and the class options for them to possibly be added in a future work scenario.

Another system that was truncated was the levelling system. This one has a bigger impact because it was directly connected to the player character's progression and would allow for new abilities to be unlocked and eventually used in combat. This would have added complexity to the combat system and to the RL model, which was already complex and difficult to implement correctly. The levelling system itself did end up being created, with the character levelling up as they move through the missions and defeat adversaries, but the only significant change is in the HP the character has. This means that the character has more HP for combat scenarios which is always helpful. The progress bar in the UI and the level number are updated. Similarly, when the character gains enough XP to level up, the game presents a pop-up saying that the character levelled up.

Some other smaller systems that were abandoned during development were related to combat and those were the armour, resistances, and weapons. Resistances was something that a character could gain over time, mostly on the side of the adversaries, with the player needing the help of armour to gain them, not only increasing its AC but also reducing some of the damage received. The final version of the game has fixed damage dies that are rolled for each type of attack, but at one point, the game was meant to have different types of weapons that would grant different modifiers and attack and damage bonuses to their wielder. This would have made the combat system more robust and would have been a good reward system, allowing players could find armour and weapons on the world and equip them. This

idea fell through once the inventory system was also removed from a possibility even though it is a staple of all RPGs. Without an inventory to manage the weapons and armour, there was no need to create a system that allowed them to function.

At one point it was also considered the introduction of player companions. These would be controlled by the player in combat and be important characters in the story, that would join the main PC on their journey. Remnants of this can be seen in the NPC Interaction Trigger where it still has a flag that marks if the NPC is a player companion. Ultimately, the addition of companions was something that would have made the game more interesting and would allow for the player to get to know the NPCs better. The problem was that since the development didn't reach that far into the levels, where the main character would meet the companions, the system was deemed unnecessary for the current state of the game. This feature was taken into consideration when creating most of the systems, like the NPC Interaction Trigger and combat and could easily, with time, be completed and introduced, increasing the game's value and complexity.

These were some of the systems that had to be cut or truncated for the game to reach its final objective in the time established. Although some were interesting and would have added value to the game, they were not crucial to reach the goal of the project and thus were removed or changed.

5.5 Breakdown of Level Creation

Creating a level is not an easy task to do, since there are a lot of elements and moving parts that need to be considered and constructed. To allow for a better understanding of how the levels are created, this subchapter will focus precisely on that, a brief overview of how a developer would create a level in this game with the tools that have been developed to do so. The example provided will be of the first level of the game.

First the terrain needs to be morphed. Since this is a small village, the terrain is mostly flat so there's only a bit of deformation needed just to add some texture to the ground. Then a ground material instance is chosen. There is the option to create smart instances that automatically change depending on terrain elevation and this was what was used. There are random patches of grass, some darker and others lighter to avoid the sense of a flat and repeated texture filling the ground.

After the terrain, the village layout needs to be determined. Since this is meant to be a small settlement there is usually one or two main roads where everything revolves around. With the street defined, the main pathways are marked down with the help of a terrain brush, marking the spots of most traffic that lead to the grass being removed. To populate the village with houses, there are a few websites online that offer free 3D assets at a relatively good quality, as seen in Figure 44. Quixel has a lot of assets but never full houses or complex objects like that. With some 3D

houses picked they are placed in the terrain, rotating some and slightly changing their colours to give the impression of a greater variation in design.

Figure 44 – Example of a wooden house asset found online for free.

Next, with the city layout and houses in place, the other assets are added to the scene. The most common assets are trees and vegetation, which can affect the performance of a level, so they must be placed carefully using the placement tools, all around the houses while avoiding the main street. Some wells and horse racks are placed on the main street, and it is complete. Although it might seem simple, all the steps take time to perform and research with the level creation taking several hours or days depending on the size and complexity required for the level.

Now the story content needs to be placed in the level. For this first level, since it is just an introduction the main character only needs to run down the street where people are gathering. The navigation mesh and bounds are placed to keep the player from moving outside the village. Now the Stage Triggers and NPC Interaction Triggers need to be placed. These must be thought out alongside the story so in this case tree Stage Triggers will be required to display tree objectives and then load the next level at the end. Two NPC Interaction Triggers will also be required so that the player can interact with the village inhabitants and get a sense for what is happening at that moment. For some context, the player character just got woken up by people shouting in the morning and came out to investigate. The triggers are placed in the path of the player towards the final objective with the NPC ones being optional interactions and the level is complete. Some tweaks can be made to the lighting or time of day, and other assets can be placed in the scene to make it richer, but the most important aspects are done. An example of the levels that were built are in Figure 45 and Figure 46.

In total, 4 levels were created for the game, with the first two being shown in Figure 45 and Figure 46, those being by far the most complex ones, with the other two being combat levels that would be loaded once combat started in the first and second level.

Figure 45 – Example of the types of levels that were built for the game. This level is called "Village of Wornborough".

Figure 46 – Example of the types of levels that were built for the game. This level is called "First Journey Alone" and presents a greater number of 3D assets compared to Figure 45 making it heavier on the GPU.

6 REINFORCEMENT LEARNING APPLIED TO THE VIDEO GAME

This chapter is complementary to chapter 5, continuing the explanation of how the video game was developed, focusing especially on the RL model that was implemented in the combat system.

Once again, this part of the project required a lot of research, given the plethora of options when implementing a RL model. Narrowing down these options required testing as well, since in a real-time environment there's always the concern of the resources being used. This meant that the model being implemented needed to run in real-time and be completely transparent to the player, in the sense that they should not notice the model running, just like in a conventional AI algorithm. In article [50] the author explores the use of RL models to train a variety of strategic behaviours. Among those was the training of NPC attacking and defending. This became the starting point for development.

In the beginning of the project, the idea was to minimize the number of inputs and states that the game could have. This became especially important when the combat system changed to grid-based movement, where not all the positions on the map could realistically be considered. For a 20x20 grid there would be 400 inputs alone to determine the location of the agent on the map, and as previously explained, it wasn't necessary. After this realization, the focus turned to how the other stats of each character could be represented and if they should. In the end, and as previously explained, only the health statistic really mattered to the behaviour of the agent.

The development process consisted of creating a state machine to understand the different ways the agent could transition between states and after that, choosing a model and implement it, adapting any of the previously defined concept to suit it.

In this chapter, the development process will be explained, starting with explaining the model implementation and ending by describing some of the challenges and difficulties of implementing the model, both in UE5 and within a game.

6.1 Agent AI and Q- Learning

After much testing and deliberation, with the statistics and agents at play, there was an RL algorithm that became more and more adequate to use, with its possible implementation being clearer than some other algorithms. As the subchapter already displays, this RL algorithm is Q-Learning.

6.1.1 AI Behaviour Tree Implementation

Before the RL model can be explained, it is necessary to explain how the general AI Behaviour Tree for the agent was created. This was a necessary step

between the agent and the model itself since even though the major decisions are determined by the model, the agent is still present in the video game environment, thus needing a control class in which to map its behaviours.

Using the sequence type of tree node, the game's Behaviour Tree was created. It starts running when the agent's turn begins, called for by the "Advance Turn" function previously explained. It is connected to a blackboard that houses all its control variables that are used to make the tree function. The variables are:

- Is My Turn – Boolean.
- Action Picked – Boolean.
- Is Moving – Boolean.
- Needs to Move – Boolean.
- Actions – Integer.
- Ability – Class "Ability Type".
- Q-Table Initialized – Boolean.
- Q-Table – Class "Q-Table".
- Target – Class "Targets".
- Move Coordinates – Integer x2.

The final iteration of the AI Behaviour Tree created can be seen in Figure 47. During the rest of the chapter, small sections of the tree will be explained in detail, each with a figure to help visualize their behaviour.

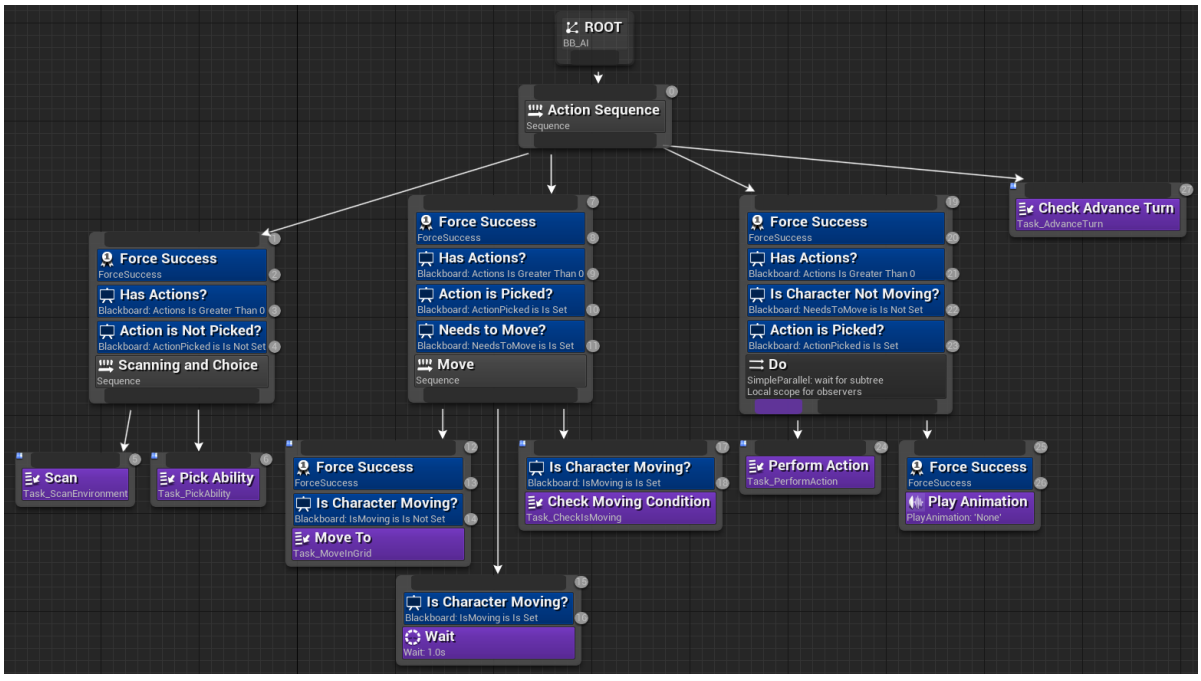


Figure 47 – AI Behaviour Tree created for the game.

The Boolean flags are the ones that make the sequences function properly. They are used in the conditional nodes (blue nodes) so that branches that are being executed can be controlled. Since a sequence only succeeds when all children succeed, this means that if one of the conditional nodes fails, the tree fails and stops running. For example, looking at the first branch (furthest left in the tree) seen in Figure 48, there is a condition to test if there are enough actions left to be performed (actions left are greater than 0, since the least expensive action is 1). If this condition fails, this branch fails. With the help of the "Force Success" node, the branch failed, but the sequence thinks it succeeded, moving on to the next branch. If this condition is true though, the branch is executed, and the condition "Action Picked" is tested. This checks if the action is picked already or not. In a normal circumstance this would not be required, but it keeps the tree from entering this branch if the target and action have already been picked but the action not yet performed. If they already were performed and the condition is false (Action Picked "is not set" means it must be false to progress), in normal circumstances, the sequence would fail because the branch fails. Once again this is nullified due to the Force Success node previously mentioned. The Force Success node, as already explained, makes the branch or node succeed even if its conditions failed.

If all the conditions in the left branch are passed, the tasks in purple are executed, once again from left to right. First the agent scans the environment for its target. This occurs inside the "Scan" task. At this point in development the agent does not rely on "line of sight" to scan the environment, having full omni awareness of its surroundings, meaning that it knows where all agents are. This is not ideal if the development were to continue, and it would be interesting to create a system of "line of sight". With its target selected, the blackboard variables are updated, and the class Target is created, saving within it the most important information about the target. The distance to it is calculated and saved alongside the spawned entity to further reference in the hit and damage calculations. From now on, this entity is the attack target for the controlled entity until an action is performed. The next task is called "Pick Ability", and this is where the RL model comes into play with the Behaviour Tree. The code and calculation part of the equation will be explained in the next subchapter but for now, the important thing to understand is that the Q-Learning model file is executed and then the code waits for a file to be created, signalling that the training process has ended. This would be a no-go for most AI systems, but since the process is quick due to its low complexity, it is feasible. The main problem that could arise with this implementation is the fact that it is vulnerable to lower-end computer specs. A slower CPU could make the files creation and IO accessing visible to the user (the game would freeze). This is the biggest problem with the current implementation. When the flag file is created, the Blueprint reads the Q-Table created by the model, creates a class in-game to save it, and checks the highest reward action. If it is possible to perform this action with the current number of actions, the action is picked, if not, the next action down is picked, until

an action that is doable with the current number of actions is found (in case of a tie, a random one is picked). This action is then saved in the Ability blackboard variable, within its custom Ability Type class, and the flag file is deleted for the next action to come. The Action Picked flag is also set to true allowing the tree to manage the branch access, now allowing access to the other branches but not this one.

With the target and ability set, the Pick Ability task, has one more thing to compute. If required by the ability, the agent might need to move to reach a certain distance from the target. This is calculated using the imperial distance of each square (5 feet per square), making the calculations more accurate than using just the number of squares to the target. If the agent needs to move, the "Needs to Move" flag is set to true and then, a possible position is calculated for the agent setting the "Move Coordinates" related to the grid.

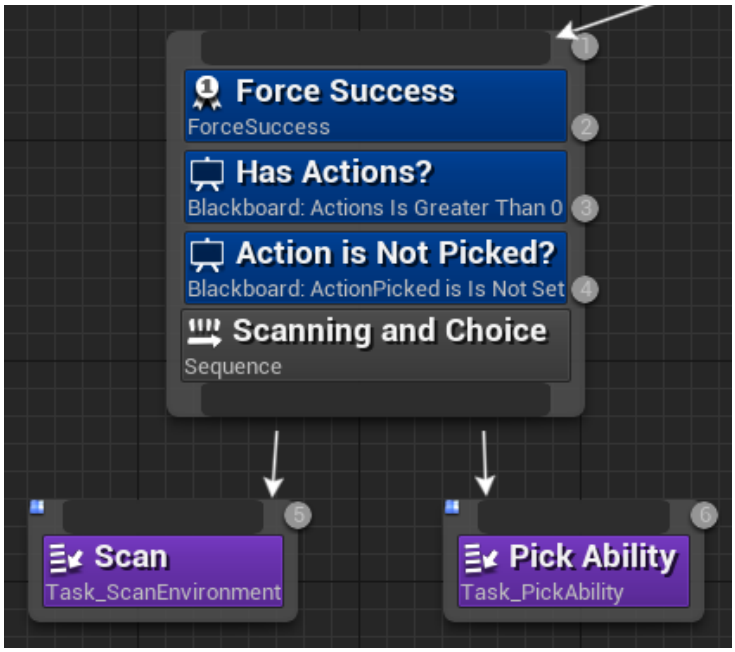


Figure 48 – First branch of the AI Behaviour Tree.

Moving on to the second branch seen in Figure 49, once again, the number of actions is checked to confirm if its greater than 0 allowing for at least one action to be picked. After this condition, there are few more conditions to be checked. The first one is the previously set Action Picked being true, letting the tree know if the condition was picked and not allowing the tree to progress if for some reason the condition was not picked, and the tree still moved forward. The second one is the Needs to Move being true, to see if this branch is required to even be executed. If the character is already on the right spot, there is no need for the agent to move to perform the chosen action. If both are set to true, the sequence will then begin. If not, then the branch has a Force Success node that keeps the tree going. The following sequence is composed of 3 tasks that loop. The first task is called "Move To" and is meant to make the agent move to the determined Move Coordinates set prior. This is a process that is not immediate, with the Combat Interaction that represent the agent on the map needing to physically move, respecting the

movement rules and speed. To allow for the next branch only to be executed after the movement is complete, the "Wait" task makes the sequence wait for 1 second, and then the next task called "Check Movement Condition" checks if the agent has reached the desired location. This process is then repeated and since the sequence is aborted when one of the children fails, the first task must have a Force Success node, allowing the sequence to run without it, only checking if the agent reached its location. The other two don't need this node because the Wait task always succeeds and the Check Movement Condition is responsible for stopping the sequence when the movement is finished, so it cannot have a Force Success node. To only execute the necessary tasks when they're needed, each task has a condition based around the "Is Moving" flag. This flag is set to true when the first time the task Move To is executed. From then on, while looping in the sequence, the first task Move To is never executed again (not stopping the sequence due to the Force Success node), while the other two, Wait and Check Movement Condition are. When the agent has reached its destination, the Check Movement Condition task fails, stopping the sequence which also fails. Since the sequence has a Force Success node, this is used to signal that it is time to move to the next branch. The task also changes the Is Moving and Needs to Move flags to false.

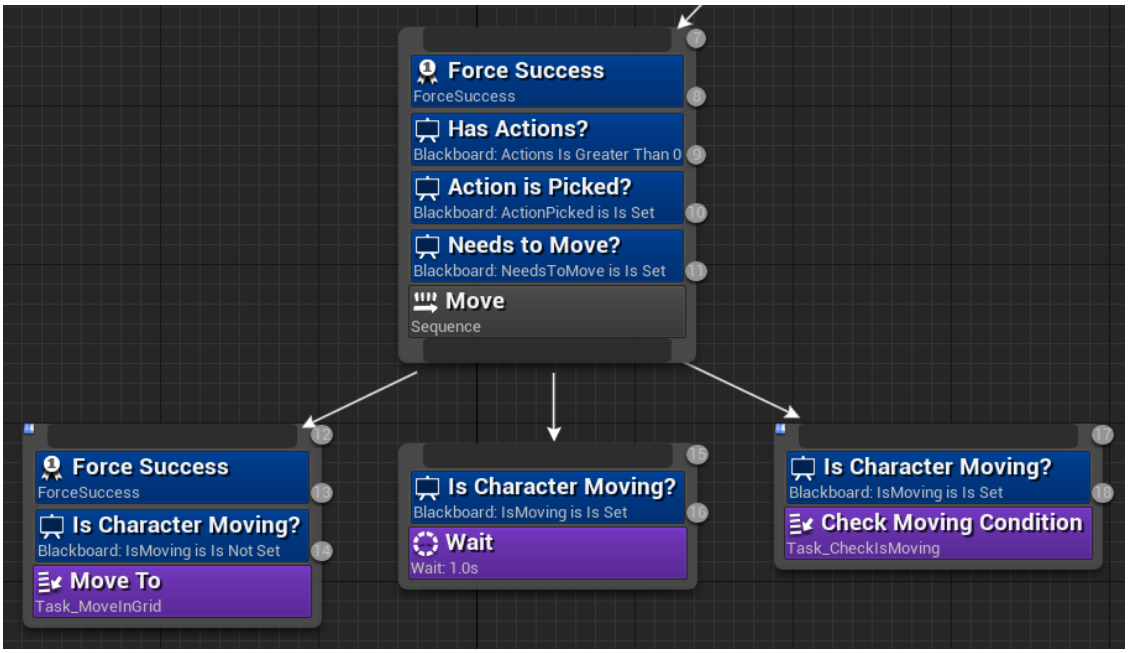


Figure 49 – Second branch of the AI Behaviour Tree.

In the third branch seen in Figure 50, once again, the number of actions is checked. If it succeeds, 2 more conditions need to be checked preventing wrong access to the branch in case something goes wrong with the other branches. The first one is the Needs to Move flag being false, preventing a moving character from performing actions. This flag, by default, is false and was set to false again after the end of the movement in the previous branch so this condition, under normal circumstances, is met. The next condition is Action Picked being true. Once again, if the logic all executed correctly, this flag is set to true, and the branch can begin. In

this branch, the simple parallel node was used. It was ideal for this situation in which the action had to be performed in terms of logic, but it also needed to be performed in terms of animation. With the use of the parallel node, both actions would occur at the same time, with the main one being the Perform Action task, and the secondary the Play Animation. The secondary task is set to "delayed", meaning that when the main task ends, the main node waits for the animation to finish playing. In this case, there is no animation to play when the action is performed, but it is already set to receive possible animations in future work. Within the logic of the Perform Action task, the ability and target are fetched from the blackboard and the action is performed. After they are completed, the action cost is discounted from the total number of actions per turn and saved in the blackboard for the tree to check in the main sequence.

Figure 50 - Third branch of the AI Behaviour Tree.

When the 3 main branches finish their execution, the main sequence has only last branch, composed of a single task with no conditions. The task name is Check Advance Turn and what is does is it checks for two conditions: Is My Turn true; and Actions is equal to 0. If both are true, then the task fails, and since there is no Force Success node, the sequence also fails, stopping the execution of the tree. Eventually, with actions performed discounting the total number of actions available, the first 3 branches of the sequence will fail (with their Force Success nodes), leaving this task to manage the execution. The tree will then be restarted by the Advance Turn function in the Game Mode class, resetting the number of actions in the blackboard and starting the tree once again, making the agent act.

6.1.2 Q-Learning and Markov Decision Process

As already previously explained, Q-Learning is a Reinforcement Learning, off-policy algorithm (policy used to explore can differ from policy learned) that allows the agent to discover the next best action, from its current state. It does this without knowing the environment, choosing during training a random action available in the current state (random exploration) with the aim of maximizing the reward. After many iterations of random exploration, the model learns through trial and error what is the best action to maximize its reward. This was the perfect model to implement in the game given its low computational requirements and easy input management. It was also ideal since the agent did not know the probabilities of each state transition or the rewards associated with those transitions. In turn, the model would learn them, allowing for an optimal policy to be created for the agent.

Initially planned to be implemented in C++, the code itself was created in Python since the UE5 Blueprints required a lot more coding to achieve the same level of robustness compared to Python.

With the algorithm already chosen, the important part was to decide how it would run and train in a real time scenario while the game was playing. Through Blueprints, command line calls could be performed, allowing for a Python file to be easily launched. The problem with this approach is that, while running an external file with code, there is no way of transferring that information back to the Blueprint. The solution found was to create a file logic that would be used to transfer information between the Python file that ran the model, a CSV file that would store the information, which in this case would be the Q-Table, and the UE5 Blueprints that were running within the AI Behaviour Tree.

After determining the logic, a couple of situations arose that needed to be figured out before implementing the model in Python and committing to this approach. First, there was no way to synchronize the Q-Table being updated and the Behaviour Tree reading the CSV file. This required a flag to represent the updated state of the Q-Table. The solution was to create a text file called "flag file" that would be created once the Q-Table had been updated. This would then allow for the Blueprint to check if the file existed and if it didn't, tried again until the file was created by the model's code, signalling that the table was prepared to be read. The second problem came out of the first, once the Q-Table was updated there was no way to read the file in real time. Although, UE5 allows for importing of CSV files when not running the game, like done with the game information files that stored story, dialogue, and combat information, there was no way to perform this real time reading in Blueprints. This is a limitation of Blueprints that stems from it being a high-level programming language. Another alternative was necessary, and after much research and even considering creating a connection with a C++ file that failed, a module for Blueprints was purchased that allowed for files to be read in real time. This module, was written in C++ and performed the task required without any problem, being instrumental to solving the problem at hand. The connection

purpose was all tested and verified before starting development on the model. With these two problems solved, the model implementation could commence.

Compared to the state machine, the MDP would require a different structuring process. In it, all the range increment states are once again relevant, since now the focus is on the agent's actions independent of outside factors. The first change was to restore the 3 levels of range increments. The second change was to remove the Dead state. This was done because there is no action the agent can perform that puts it in the Dead state. Therefore, there is no need to consider the state in the MDP since its focus is on actions that can be performed by the agent. Once again, as a reminder, the Dead state can only be reached by having the player character reduce the agent to 0 HP.

To systematize the states being used on the MDP, they are:

1. Win state – has reduced its adversary to 0 HP.
2. In melee attack reach – the melee ability can reach the target.
3. In melee attack reach but health is critical – the melee ability can reach the target, but the entity has less than 30% health.
4. In ranged attack reach – can reach the target using the ranged attack ability if it has one.
5. In ranged attack reach but health is critical – can reach the target using the ranged attack ability, if it has one, but the entity has less than 30% health.
6. Out of reach – cannot reach the target with an attack.
7. Out of reach and health is critical – cannot reach the target with an attack and entity has less than 30% health.

The number given to each state is important, and in future reference, the states will be referred by their number following the prefix "S". Example, S1 means "state 1", which represents the win state.

Before even writing a line of code on the model, it was important to establish some fundamental data. Starting with the State Machine previously shown it was used as reference to create the game's Markov Decision Process. This was required because it was important to establish not only how the state transitions could be performed by picking a certain action, but also the rewards that would be given to the agent once he picked that action. For this it is important to remember the actions the agent can perform:

0. Move.
1. Melee Attack.
2. Ranged Attack.
3. Defend.
4. Heal.

5. Improve.

Each action's number will be used in future reference as an abbreviation. Example, A0 means "action 0" which is move the agent.

The first thing to establish was a hierarchy of importance for the actions in each state that the agent could perform. This hierarchy would serve as a point of reference for the creation of the rewards that would be given to the agent in training. The most important one had to be the ones that could lead to the win state (S1), meaning that the damage dealing actions had greater rewards compared to most other actions in non-critical health scenarios. When in a critical health scenario, the situation was different. To encourage the agent to perform healing instead of attacking, the reward for healing needed to be greater than the one for attacking. This is all under the knowledge that the Q-Learning algorithm will find the policy that has greater rewards. Through the reward establishment process, an ideal path can be created, allowing for easy visual debugging in the initial stages of development. The benefit of a reward system is that it can be easily tweaked to allow for the outcome of the training to be more of what is intended and expected.

During the creation of the MDP, the concept of Actions Space Shaping explored in the Related Work chapter [24] was used to shape the actions an agent had in each state. This means that the actions that are optional or unnecessary can be removed to reduce the amount of training the agent has to do to reach an optimal policy. For instance, when in a non-critical health state, meaning either states S2, S4 or S6, the agent cannot perform the heal action since its use would be objectively worse than performing attacks, if in S2 or S4, or move to the target if in S6. In those states, considering the agent's health condition, the optimal action will always be to attack since that's how it can reach the goal, state S1. Another use of this strategy was with the attack actions in S2 and S4. When in S2 the A2 action cannot be used. There is a restriction on the use of ranged actions that required the target to be at further than 2 squares away from target (10 feet). In S4 though, since the agent is not in melee range, there's no need to test A1 in that state, reducing training time once again. Similarly, when in S6, no attack can be performed. These are all examples of Action Space Shaping reducing the complexity of the model, and time needed to train. Once again, it is important to reiterate that the use of strategies like this requires careful consideration on the side of the developer. Inappropriate use can create a scenario that limits the agent's ability to learn. In this case, Action Space Shaping was only implemented in scenarios that the developer was sure wouldn't affect the agent in any way.

The final rewards for each action in each state can be seen in Table 1. The positive number indicate positive rewards, the negative number indicate negative rewards and the zeros indicate that no reward is attributed to that action in that state. If an action is not possible to perform in a specific state, a dash will take its spot. Since there are no action that can be performed in S1, the state is omitted from the table. There are several actions that have their reward determined by outside factors,

meaning that different rewards can be attributed to the same action in the same state. These will be demonstrated with a slash separating the two rewards.

Table 1 – Rewards given to agents for actions in each state.

	A0	**A1**	**A2**	**A3**	**A4**	**A5**
S2	+10/-5/0	+20/+10	-	+8	-	+2
S3	+10/0	+12/+4	-	+2	+20	-2
S4	-5/+10	-	+15/+8	+6	-	+1
S5	-10	-	+10/+4	0	+12	0
S6	+20	-	-	0	-	+10
S7	+10/+5	-	-	-12	+15	+6

With the rewards established, it was time to solve the issue related to the several rewards for the same action in a state. While a hidden probability could be used to solve the issue, there was a cleverer way of doing it. For this specific model, the rewards that required them were A1 in S2 and S3, A2 in S4 and S5, and A0 in S2, S3, S4 and S7. For the first two, this means that the outcome and therefore the reward was dependent on successfully achieving S1 (win state) which meant reducing their target to 0 HP. If the HP of the target and the minimum guaranteed damage dealt by the entity were to be give as parameters in the training of the model, it was possible to calculate if the damage could reduce the target's HP below 0. For A0 the condition is different. In each respective state it might be beneficial to move or not somewhere depending on the actions the agent has available. For example, in S4, its rewards mean that if A1 is not available, there is no benefit in moving towards the target, therefore there is a negative reward (-5) for doing so. If A1 is available though, the reward is positive (+10), encouraging the agent to move closer to its target. Similarly, in S2, there are 3 rewards, because the agent is tested to see if it is missing A1 (+10 reward) or A2 (-5 reward). If both conditions fail, there is no reward. This logic in turn, allows the model to award the highest or lowest rewards for an action depending on the conditions. This method proved to be effective, completely solving the problem.

With the rewards now established, the Markov Decision Process can be finally shown. Using the same nomenclature previously established and the rewards, the MDP was constructed.

As seen in Figure 51, the layout of the MDP was divided with a dotted line to aid with visual understanding of the design. The columns establish the agent's current health condition and if it has won the combat. From left to right they represent the win state, the non-critical health states, and then the critical health states. The rows divide the different states between their respective range increments. From top to bottom they represent the Melee Reach, the Ranged Reach, and the Out-of-Reach states.

Once again, it is important to reiterate why there's no Dead state. It would be a good reward mechanism for the training process, allowing for a very large negative reward. It might even incentivize the agent to pick the heal ability more when

entering the critical health states, but since the agent cannot, willingly and by itself enter the Dead state (no action can lead the agent into the Dead state), it was removed from the MDP while remaining in the State Machine.

This MDP and its rewards were used to structure the environment used to train the RL model. With its random exploration policy, the agent will only know what the consequences for each choice are after receiving the respective rewards. For the same reasons, there is no need to add probabilities to each action in the MDP since transitions are dependent on other factors (agent statistics) besides probabilities. Even then, the agent has no knowledge of what factors are associated with what state transitions. This is important to reiterate since there would be no need for Q-Learning if the agent already knew the environment.

The MDP seen in Figure 51 shows the states, actions, and rewards for each transition, which are colour coded. Green arrows represent positive rewards given to the agent, red arrows represent negative rewards, blue arrows represent neutral rewards, black arrows represent decisions taken, and lastly purple arrows represent transitions with multiple rewards as seen in Table 1. Each reward arrow has the corresponding number(s) attached.

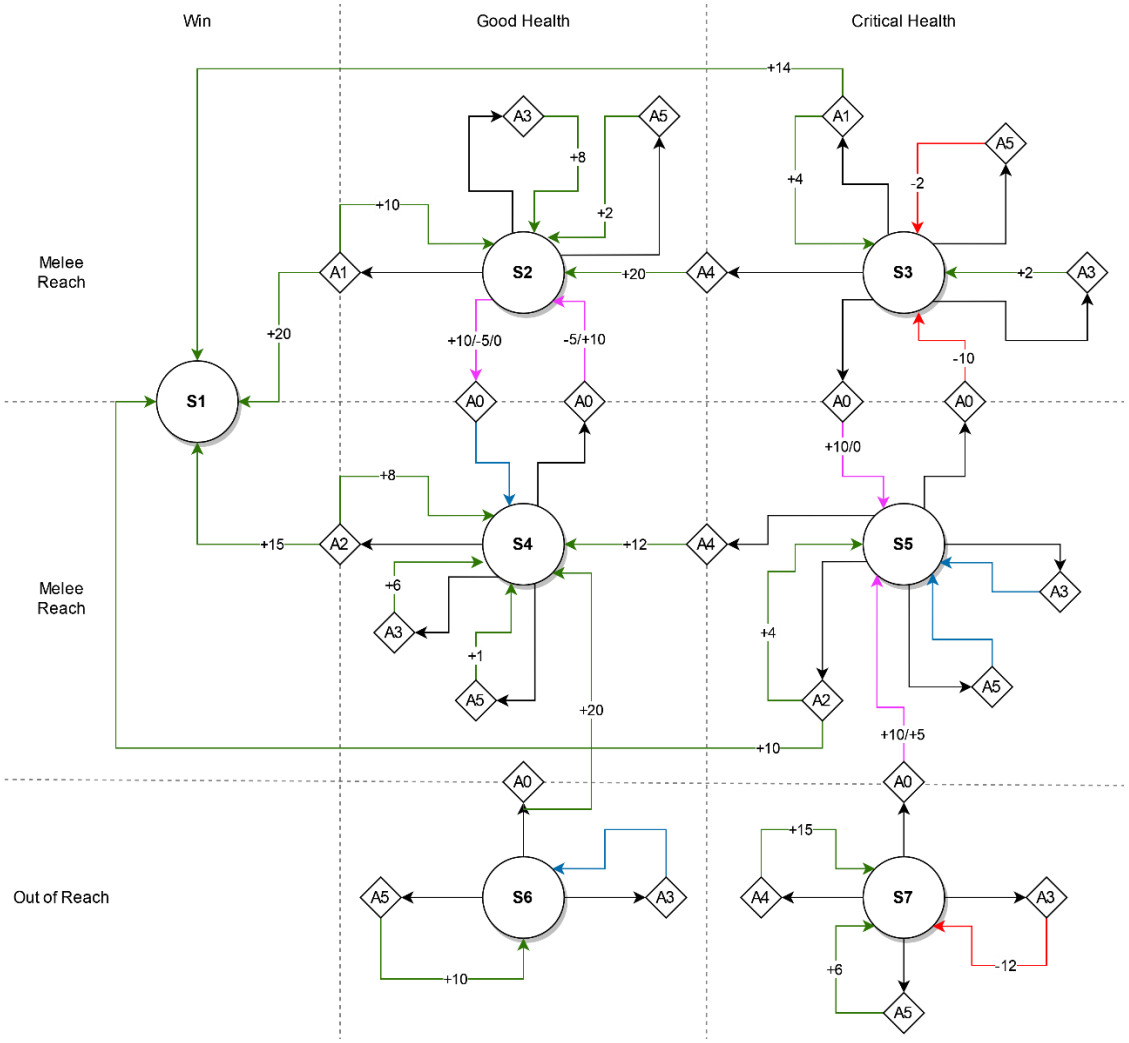


Figure 51 – Markov Decision Process established for the game's decision-making.

Now with the MDP complete, visually representing for the reader and developer the combat environment from the perspective of the AI agent, it is time to begin construction of the Q-Learning model in Python.

The first thing that needed to be established, knowing how the script would be called using a command line call, was what parameters were needed for the training process. After deliberation and taking into consideration everything established previously, these were the parameters passed to the Python script:

- Abilities – Integer Array.
- Current State – Integer.
- Certain Damage – Integer.
- Target's Health – Integer.

The use of the Abilities in the model was required since, as seen in the combat chapter explanation, not all combat entities have the same number of abilities, with some for example only being able to attack at melee reach and others only at ranged reach. This is important to know because it will reduce considerably the number of trials the model has to perform to learn the optimal policy. Since each Spawned Entity within the game's logic already has their Ability Type class arrays created, it is easy to send each ability ID as a part of the array to the Python script.

The current state is crucial to pass as a parameter given that it is the main determining factor on what actions are available and being tested. This state is first calculated within the UE5 Blueprints. It considers all the possible states that are not the win state, calculating distance from the target and checking to see where the entity is within the reach scale (out of reach, ranged reach, or melee reach), even checking if the entity has the ability to perform ranged attacks. After the range situation is determined, it checks if the character is in a critical health condition. This required an establishment of what a critical health condition was. After some deliberation it was determined that and entity below 25% of their maximum HP, would be in a critical health state. This was then updated later to be 30%, allowing for more variation in the types of actions chosen. From these calculations, the state number was determined and sent as a parameter.

The final two parameters were necessary to determine if there was a possibility to reach the win state. Like previously explained, there are actions that can grant two types of rewards to the agent, depending on the outcome. These are the attack action A1 and A2 in any of the states that allow for attack actions, S2, S3, S4 and S5. With certainty of how much damage an attack could deal, coupled with the current HP of the target, the model will know when the attack will trigger a win state, thus awarding the highest reward for that action. The certain damage dealt by the agent is calculated using the damage bonus it has, which is always fixed, and adding 1 since that's the lowest number that the die can roll to calculate damage. For instance, in a melee attack where the damage would be 1d10 (10-sided die, goes from 1 to 10) plus a

damage bonus of 5, the certain damage would be 1 (from the die) plus 5 (from the bonus) equalling 6.

With the parameters explained, the Python script was the next thing to be developed. The main focus of the development was the implementation of the Q-Learning update rule, previously explained in the Background chapter. For reference the equation will be shown again later in this subchapter alongside the code for the model.

The model is called in the Blueprint, with the command created and composed of the file call plus all parameters in the correct order. The script then begins to compute the parameters, setting the initial variables of the model. The initial variables are as follows:

- Alpha – Learning rate – Represents the size of steps taken towards the solution.
- Decay – Learning rate decay – With this the learning rate is slowly reduced as the number of iterations increases, allowing for finer detail learning.
- Discount – Discount factor – Represents the discount rate and regulates what rewards the agent takes more into consideration. Short term rewards (closer to 0), or long term rewards (closer to 1).

To train the model, the number of iterations is pre-defined by the developer and the code seen in Figure 52 is executed. For the model trained, the number of iterations in the loop were 10000.

Within the loop logic there are two major functions that need to be explained. These are the ones on line 154 and 155 of Figure 52. The first one is the "exploration policy" function. The job of this function is to determine what action the agent will take next. This is achieved using random exploration. Basically, each time the agent is in a state, it has an equal probability of choosing any of the actions possible for that state. While picking these actions, the agent has no idea of what rewards are associated with them, or what state they'll lead to. It is also important to consider the actions the agent can perform, since not all agents are created equal, with some having less actions that others. This is why the abilities variable is a parameter of the function, alongside the state. With the next action chosen, it is time to determine the outcome and the corresponding reward. For this, the second function called "step" is used. The step function works by taking the state, action, target health, certain damage and abilities as parameters, to determine the outcome and reward. Each action has specific outcomes and one reward when in a specific state. As previously explained, in sistuations where there are several possible outcomes and rewards, those being S2, S3, S4 and S5, the other 3 parameters are used to calculate the outcome. If the certain damage the agent will do is greater or equal to the adversary's HP, the reward is maximized and if not, the lowest value is used instead. When the certain damage is not enough to reduce the target's HP below 0, that certain damage is deducted from the target's HP to allow for a chain of several attacks to take into

account all previously performed attacks. With this, the agent can perform 2 attacks that reduce the target's HP below the threshold that would guarantee a kill with another hit, and know the next hit would result in S1, granting the greater reward for the third attack.

To estimate the optimal future reward, the model fetches the next value, which is the one in the Q-Table with greater reward as seen in line 156 before beginning the calculations for the Q-Learning Update Rule.

During the Update Rule calculations, the alpha is decayed considering the number of iterations already performed, multiplying them by the decay, and using them to divide alpha zero which represents the initial learning rate. With this, the current learning rate is obtained, and the calculations can continue. After, the old value is saved in a temporary variable (this process is not necessary and was only done to aid the code's ease of understanding) and then the Update Rule itself is calculated. As seen in Figure 52, the line of code 162 matches perfectly with the previously shown below.

$$Q^{new}(s_t, a_t) = \underbrace{Q(s_t, a_t)}_{old\ value} + \underbrace{\alpha}_{learning\ rate} * \overbrace{\left(\underbrace{r_t}_{reward} + \underbrace{\gamma}_{discount} * \underbrace{max\ Q(s_{t+1}, a)}_{best\ future\ value\ estimate} - \underbrace{Q(s_t, a_t)}_{old\ value} \right)}^{TD\ error}$$

```
152 for iteration in range(num_iterations):
153     # Discover a Reward for this state
154     action = exploration_policy(state, abilities)
155     hp, reward = step(state, action, target_hp, certain_dmg, abilities)
156     next_value = np.max(q_values[action])
157     target_hp -= hp
158 
159     alpha = alpha_zero / (1 + iteration * decay)
160     old_value = q_values[action]
161     # Q-Learning Update Rule
162     q_values[action] = old_value + alpha * (reward + discount * next_value - old_value)
```

Figure 52 – Python code for the Q-Learning model implemented compared to the Q-Learning Update Rule.

To communicate with UE5 again, the Q-Table was written into a CSV file, overriding the information already contained within. This would be a bad practice if the decision-making process was extremely complex, preventing the agent from learning throughout several iterations. With the limited number of actions and with the result of training always making clear which action the agent should choose, the current implementation had no effect on the learning ability of the agent. In future iterations, when the combat becomes more complex and the number of states increases, it would be ideal to switch to the aforementioned approach. Now that the file is written, a flag file is created to signal the UE5 Blueprint, currently waiting to read the updated Q-Table, to read it and proceed with the Behaviour Tree's logic. Each time the CSV file is read, the Blueprint deletes the flag file to allow a new one to be created the next time the model runs.

During development, considerations were made to check if there was the need to dive into a Deep Q-Learning implementation. This would be different from the normal Q-Learning approach and instead of using a loop and a table to map each state-action pair to its correspondent Q-Value, it uses a Neural Network to map the input states to action-value pairs. This would be beneficial in situations where there were a lot of states to consider. In this case, there is no need to create a NN, since the number of states is small, remaining unaffected by the scalability limitations present in Q-Learning.

It was also considered the use of a different exploration policy for the model. The original exploration policy relies on random exploration, where the agent randomly picks one of the possible actions in a state and accesses its reward. There were alternatives considered like "Epsilon-Greedy Exploration". In this strategy the agent is not just exploring the environment but exploiting its knowledge. This is called exploration-exploitation trade-off, and it means that instead of randomly exploring, the agent rolls a percentage and if that percentage is higher that the pre-determined epsilon parameter, the agent will instead take the best-known action for its current state. This approach ended up not being used. After some deliberation and testing it was determined that it wouldn't bring performance updates to the model since it didn't converge to an optimal policy sooner. This would have allowed the developer to reduce the number of iterations, which would also reduce any needed waiting in the Blueprint that could stutter or even freeze the game [51], but it ultimately didn't.

6.2 Development Issues

During the implementation of the model in the game, several issues and difficulties slowed down the development process. In this subchapter, the major issues will be explained, granting some context for decisions that were made during development.

As already explained, UE5 can use either Blueprints or C++ as their main programming languages. Initially Blueprints were chosen, allowing for easier implementation of logic without the worries of the C++ programming language. There was always the option, to switch back to C++ from Blueprints or even create the game in a hybrid fashion using both methods. The project was initially created for version 5.0 of the Unreal Engine 5. After a year of development several updates came along with the latest version now being 5.2. When starting development on the model, the original idea was to create everything on a C++ file, allowing for seamless integration with the UE5 Behaviour Tree, without the need to use CSV files like the current implementation requires. This wasn't possible since when any C++ source file was compiled, several errors arose and though some were simple to resolve, others weren't. After much testing and debugging to uncover where the errors were originating, it was decided that the model implementation needed to

migrate elsewhere to allow for the project to resume development. These errors were likely caused by the starting project being Blueprints, mixed with several updates that caused problems in dependencies and linked libraries.

Although carefully examined, it was hard to begin the development of the model itself. There are some many possible implementations that understanding what's the best on for the current project was not easy. Ultimately, the simplest answer was the one that ended up being used, allowing for a clear layout of a plan and development structure to be created.

The creation of the game's state machine and MDP also took a considerable amount of time suffering several iterations before arriving at the final ones. This was due to the combat system being in active development at the same time the model was being conceptualized. Since the combat system suffered several iterations of its own, so did the possible implementations of the model and the states definitions.

Lastly, closer to the end of development, with the model already built and working, there was the realization of the fact that there was no way to in real time, read a CSV file. This created a big problem, since redoing things was not an option and with the current implementation showing promising results, not even being considered. Creating code in C++ to read a file was a possibility, but with the already mentioned integration problems with C++, it was also out of the question. The solution ended up being to purchase a plugin for UE5 Blueprints that allowed the use of specially created nodes, that when a valid file path was given, it allowed a file to be read in lines. This was not ideal, since a small monetary investment was made, but ended up being the only feasible one.

7 TESTS AND VALIDATION

To understand the level of performance achieved with the implementation of the model in the game, it is required to perform some testing and validation of the results obtained.

Since it is a live environment, the testing is hard to perform, with agents having to move in the environment to reach their desired targets. For this reason, and without changing the nature of the game, most of the tests are performed on the decision-making model, and less on the combat system itself.

During this testing and validation process, there was also the occasional need to tweak and balance some of the model's finer details like the decay or even the action rewards.

7.1 Establishing Metrics

Anytime there is testing and validation of results, there must also be metrics established to use as reference points. In some cases, numbers are predominant and can be used as metrics themselves to then compare to other results, but it is not the case here, with a more complex scoring system needed.

For testing and comparison, metrics need to be established that could cover both in-game testing, as well as model testing using its outputted Q-Tables for each state. These metrics need to be worth testing, with their values meaning something to the developer to allow for critical assessment of the model's behaviour.

The metrics made for analysing the model, are less number based and require the direct analysis of the Q-Tables. For this the following metrics were created, each with a reference nomenclature:

- M1 – Percentage of optimal position corrections – a position is corrected when the agent moves when its beneficial to move. Example: S4 to S2 when it has A1, or from S2 to S4 when does not have A1.
- M2 – Percentage of optimal heals – Number of times the agent healed when it was in critical health.
- M3 – How often was defending viable in attack ranges – How many times is defending in the top 2 options for the agent to choose within the attack ranges (S2, S3, S4, S5).
- M4 – Percentage of optimal choices – optimal choices are the best for the agent to take in each state, setting up the next possible action.

These were the metrics that were established for the game, continuing with the reference nomenclature from the model's metrics:

- M5 – Average number of rounds alive.
- M6 – Average amount of damage dealt in a round.
- M7 – Average amount of damage received in a round.
- M8 – Average health restored by healing in a round.
- M9 – Number of combat encounters won.

It is important to note that all game metrics are made for one-on-one encounters with the player in combat.

7.2 Testing

Due to the statistics-based nature of this combat system, it is also important to compare two entities with the same statistics, to guarantee impartial testing conditions.

To effectively test something there must also be a baseline to compare results too. This wasn't originally intended but the solution was to create two new agent types. The first type is an "Attacking Agent". This agent is a barebones implementation of a conventional AI, that's just focussed on attacking the target without carrying for its own health. This could cause more damage to be dealt to the target, but there are other metrics to ensure that this is not the only thing being analysed. The second type is a "Random Agent". As the name implies, this agent's purpose is to provide a baseline for how a random decision-making agent would behave.

Something that cannot be worked around is the percentage-based nature of the game's attack and damage rolls. This makes it so that the numbers might be skewed to one side based purely on coincidence created by the random number generator. The only way the testing process can reach fairness is with many tests. Although the testing process takes a long time and is performed entirely by hand, a compromise needs to be reached to ensure proper testing. That way, a fair average of all metrics can be obtained, granting impartiality to the otherwise impartial random nature of the game.

First the model will be tested to assure that the results are good and can be used to control the video game agents. After that, the in-game tests will be performed, knowing that the model works properly. Any lingering issues will be solved to avoid disruption of the testing process. If any changes are performed during testing that could affect the previously performed test results, all tests must be redone.

7.2.1 Model Testing and Analysis

To test the model's reliability, several scenarios were created by structuring Python commands, that the UE5 Blueprint could have created in a normal play condition and analysing the game state. These were separated between states and several conditions were tested. They varied from the opportunity to reduce the adversary below 0 HP to several different combinations of abilities and states to see how agents reacted. In total 28 model tests were performed, divided by states. This means that 4 tests were performed for each state, with an additional 4 for specific combat instances. The metric scored results can be seen in Table 2.

Serving as a reference point to the model performance the Random Agent will be used. Since the agent picks all actions at random, the baseline for each metric is approximately 17% effectiveness (100% divided by 6 actions). In theory, this type of agent represents the behaviour of the model before being trained, having a random behaviour.

To better understand the table, its columns need to be explained. In Table 2, each column represents:

- Total Instances – total amount of instances that can be counted for the metric. For example, in M1 (Percentage of optimal position corrections), only the test instances where the move action would be optimal are considered.
- Right Model Choices – the among of instances, out of the total that were correctly performed by the model.
- Percentage – result for the metric calculated by dividing the correct model choices by the total instances.

Table 2 – Q-Learning Agent metric scoring for the model testing performed.

Metrics	Total Instances (units)	Right Model Choices (units)	Percentage (approx.)
M1	14	12	85.7%
M2	7	7	100%
M3	21	5	23.8%
M4	28	26	92.8%

The metrics overall showed a decent percentage of results, but there were some instances where the model struggled.

Starting with the best results, the model made very few choices that made little logical sense, and they all belonged to S7, which proved to be the model's "Achilles Heel". Still, this is better than randomly selecting actions like the Random Agent does. The model was also efficient in terms of healing, with 100% effectiveness. This is a significant improvement when compared to the Random Agent that only had 17%. This grants the agent a longer life expectancy in-game, with it prioritizing the heal action above all others when available. This behaviour is also to be expected with the healing rewards being greater than most other rewards

in the game on purpose. Lastly, the model also has a great ability to correctly correct the agent's position, requesting move actions when it was ideal to do so with 85.7% effectiveness. This means that the agent, more often than not, is in the correct position to perform the best action next, something that the Random Agent just cannot provide given its nature. Once again, there is a situation on S7 that caused the model to fail in its decision-making process.

When the model is in S7, which is the out of range in critical condition state, it tends to heal itself as it should, before doing anything else. This process is correctly performed and free of problems. The problem arises when the agent is conditioned in terms of abilities available (cannot perform all actions). The movement in S7 to other states is only directly connected to states S4 and S5. This is rewarded if the agent has the action A2 to use, but when A1 is not available the agent chooses not to move. Looking into the step function to understand the problem, it became clear why. Although the movement of A2 is available is rewarded, the movement for A1 remains with no reward. This means that when there is only A1 to perform as an attack action, the model will never choose to move, thus being stuck for at least one turn. To fix this a reward was added to the S7 to encourage movement in situations like the one described. There was an immediate change in the result of the model, and although the agent still prefers to heal (as it should) it now also recognized the benefit in moving towards the target.

The model currently is working just as intended by the developer. It takes the rewards correctly into consideration and almost always picks the optimal one to perform in each state. This will translate to the in-game testing where the Combat Interaction will benefit from the refined agent training.

7.2.2 In-Game Testing and Analysis

When it comes to in-game testing, it proved to be the most difficult part. After testing the model's decision-making and effectiveness, it is time to apply these into the game via the already established connection between the UE5 Blueprints and the Python model.

This testing will have two baselines to test against. The first baseline is called an Attacking Agent with its behaviour being solely focused on attacking, and the second is called a Random Agent with its behaviour being random. The Attacking Agent should be better at dealing damage to the target but will have less endurance, since it has no self-preservation in mind. The Random Agent should create interesting scenarios where a greater variety of actions are picked (besides the optimal or attack actions).

In total, 27 tests were performed, 9 for each type of agent (Q-Learning Agents, Random Agents, and Attacking Agents). These were then split into 3 categories of 3 tests each. These were slight variations in agent statistics and characteristics like AC, HP, and available actions. All these tests were performed by hand in the combat encounter mode of the game, in one-on-one combat scenarios.

These 3 variations were created to try to understand the scalability of the model, and how effective it is at controlling better agents. They were also fighting the player character, being controlled by the player, but obeying the same rules of combat for actions and their costs.

The agent templates created can be seen in Table 3 along with their stats.

Table 3 – Statistics of different agent templates created for in-game testing of the combat system.

Name	Abilities	Starting State	HP	AC	Hit & Damage Bonus	Move
Agent 1	A0, A1, A3, A4	S2	25	16	5	40 feet
Agent 2	A0, A2, A3, A4, A5	S4	35	17	6	40 feet
Agent 3	A0, A1, A2, A3, A4, A5	S6	50	15	8	50 feet

To compare all agents amongst each other fairly, they share the same statistics provided by the template agents.

For reference it is important to establish the statistics of the PC, and these are present in Table 4. The same character was used for all the combat tests.

Table 4 – Statistics of Player Character used for in-game combat testing of the RL model.

Name	Abilities	HP	AC	Hit & Damage Bonus	Move
Player	A0, A1, A2, A3, A4	50	15	5	40 feet

These are the results of the tests performed to the RL model in-game using the combat system. Table 5 shows the average results of the tests performed to the Q-Learning Agents, Table 6 shows the average result of the tests performed to the Attacking Agents, and Table 7 shows the average result of the tests performed to the Random Agent. To present the averages of the numbers obtained, they were all rounded down.

Table 5 – Metric Scoring obtained during testing of the in-game model implementation for Q-Learning Agents.

Metrics	Q. Agent 1 (average)	Q. Agent 2 (average)	Q. Agent 3 (average)
M5 (rounds)	3	4	10
M6 (damage)	27	33	229
M7 (received)	37	51	89
M8 (healing)	7	13	50
M9 (wins)	0	0	2

The Q-Learning Agents have a clear progress with the increase of the agent's actions and HP. As expected, the more HP the agent has, the longer they can last in a one-on-one fight with their target. There is a clear progression in the number of rounds the agent can last (M5) and it is even possible for Q. Agent 3 to win a couple of fights (M9). These wins stem from a mixture of the right dice rolls and how proficient the agent is at healing itself when needed. A human player when they see

their character in a low health scenario, tends to spend almost all actions healing the character, ending up not attacking due to the limits imposed by the 3-action economy of the combat system. On the other hand, the agent can balance that risk reward of its HP by always keeping itself above 30% health and using the rest of their actions to attack the target. This can also be seen in the amount of damage received the agent has (M7), due to it being able to heal when it is optimal, while still dealing huge amounts of damage (M6). It is also important to acknowledge the giant damage increase the Q. Agent 3 has over the Q. Agent 2. By massively increasing the number of rounds alive, the agent can deal much more damage to the player's character. Once again this leads to the player having to heal itself instead of attacking, granting the upper hand to the Q. Agent 3. Lastly, with a similar number of rounds alive, both Q. Agent 1 and 2 have their metrics very close to each other, with better numbers on Q. Agent 2 as expected with the increased stats.

Table 6 – Metric Scoring obtained during testing of the in-game model implementation for Attacking Agents.

Metrics	**A. Agent 1 (average)**	**A. Agent 2 (average)**	**A. Agent 3 (average)**
M5 (rounds)	2	3	4
M6 (damage)	25	41	54
M7 (received)	28	33	51
M8 (healing)	0	0	0
M9 (wins)	0	1	1

When it comes to the Attacking Agents, they also show a clear progression with the agent's stat increase. This can be seen in the rounds survived (M5), as well as in the damage dealt and received (M6 and M7 respectively). They all share an improvement when the agent gets stronger and has more health points. Another thing that's important to highlight is that fact that both A. Agent 2 and 3 won a fight. This happened due to a lack of care for self-preservation, focusing just on getting into position to attack and doing it as many times as possible. The player will then heal, attempt to run away, or risk a single attack, those being its best options. This leads to an influx of damage from the agent that cannot be retaliated by the player unless it risks dying sooner from not healing. If the agent can deal enough damage, it can wither the player character slowly so that, even with healing, it cannot keep up and loses the fight.

Table 7 – Metric Scoring obtained during testing of the in-game model implementation for Random Agents.

Metrics	**R. Agent 1 (average)**	**R. Agent 2 (average)**	**R. Agent 3 (average)**
M5 (rounds)	2	6	8
M6 (damage)	7	28	94
M7 (received)	31	61	77
M8 (healing)	0	2	1
M9 (wins)	0	0	0

Finally, when it comes to the Random Agents, once again, they show a clear progression with the agent's stat increase. The number of rounds survived increase (M5), as well as the damage dealt and received (M6 and M7 respectively). The same cannot be said for the amount of healing performed with it being inconsistent during the testing performed. This can likely be attributed to the specific conditions where the healing action can exclusively be performed. No Random Agent was able to beat the player in combat. This is to be expected since the agent does not care for the health of the player's character or its own so all it does is randomly pick actions, granting variability to its turn, but removing any sense of combat logic.

Comparing the first two tables (Q. Agents versus A. Agents) grants a clear vision of how the Q. Agent is much better at staying in the fight. Compared to the A. Agent, they can remain in the fight for much longer, with Q. Agent 3 having encounters with 10 rounds on average and winning some of them. This is possible due to the ability the agent has of knowing when it is optimal to heal themselves while still attacking in the same turn. The benefit of lasting more rounds can also be seen in the damage dealt and received. The average damage dealt by the Q. Agent 3 (average of 229) completely surpasses the damage dealt by A. Agent 3 (average of 54). The intelligent agent is also healing several times, with an average of 3.33 heals per fight, around 50 HP which is the exact amount they start with. This proves that the agent is much more efficient and optimal when picking actions, if these are either moving (A0), dealing damage (A1 and A2), or healing themselves (A4). The other actions are not picked by either of the agents.

The main problem with the intelligent agents (Q. Agents) is that they are not likely to pick any of the other actions even if they are good options, as long as they're not optimal. When an algorithm is created to be optimal, there will always be lack of variety in the actions determined by it. The testing performed with the R. Agents proves that variety can be a good thing in combat. These agents can outlast the A. Agents by varying their actions so that A3 (defend) and A5 (improve) are more likely to be picked, making it harder to be hit by the player's attacks while at the same time making it easier to perform their own. Even with their choices being random, when comparing A. Agent 3 with R. Agent 3, the latter outperforms the former in all metrics except wins. It deals more damage (54 for the A. Agent versus 94 for the R. Agent) and at the same time can withstand more (51 for the A. Agent versus 77 for the R. Agent). Variety in actions can be beneficial to the agent, with the ideal behaviour being a mix between the Q. Agent's optimal choices and the R. Agent's variation in actions.

One possible way to solve the lack of variety could be to add a mechanism to either the model or the in-game reading of the Q-Table that takes into account previously taken actions, and tries to perform other, non-optimal actions that are still beneficial to the agent. For instance, the use of A3 in the Q. Agent 2 would make their AC 19, meaning that it would be increasingly difficult for the player to hit it with an attack. Similarly, the use of A5, an action that would grant the benefits seen in the tables of statistic increases, is never used and could be very beneficial to

the agent. This can also be a balancing act since, as it stands, the model is too powerful to fight the player in group battles, especially Q. Agent 3. In a normal combat encounter, the player is faced with around 3 to 5 enemies and as it currently stands, the strongest agents will not provide a challenging yet balanced combat experience. The goal of creating a fair challenge for the player was achieved in one-on-one combat, with the other forms of combat requiring careful planning of the agent's stats before creating the combat encounter. Another way of balancing the decision-making would be to tweak the rewards given to the model during training to try and vary the actions picked. This is complicated and hard to balance, so it would require a lot of trial and error to figure out the best approach.

8 CONCLUSION

In this final chapter, the conclusion of the project will be structured, presenting the closing thought about the project and the process to reach it. The objectives will be analysed, and their completion checked. Ideas for future work will be outlined in their section and finally, at the end, some closing remarks on the project in its final form.

8.1 Objectives Completed

At the end of a project, it is usual to perform an evaluation of what was created, comparing it to what was originally intended. As mentioned in the beginning of the report, these were the objectives laid out for the author:

1. Create a short video game.
2. Create a combat system where a RL model can be implemented and tested.
3. Create the RL model and incorporate it in the combat system.

The first objective was the most successful in terms of achievements. The final product is far from finished, but most of the systems required to create a video game that could be published are there. From the triggers to the story presentation and dialogue systems, the project has a good foundation for future development. Some systems had to be reduced or cut from the final version of the delivered game, but they were still considered during development and can easily be integrated as part of a development continuation effort.

When it comes to the creation of the combat system, the result was once again good. Although much simpler than originally envisioned, the combat system is functional and in a stable state. It was affected by the lack of new abilities that would have come from the class and levelling system that were not finished for the main game, but the base skills still allow for an interesting combat experience, especially when fighting the agents created.

Finally, analysing the success of the model might be the hardest part. Looking at it from a performance point of view, the goal was completed, with a model that runs real-time in the game, and is the sole decision-maker for the agent's actions. When looking at the model as a system that could replace conventional ones, that's not as easy to affirm. It is too focused on the optimal strategies, and although that created interesting scenarios where the agent might beat a human player, it also can become repetitive soon, with the same actions always picked. Overall, the system is considered a moderate success, with space for improvements.

8.2 Future Work

This subchapter is meant to describe the possible iterations that can be performed not only on the RL model but also on the game.

The systems built are suitable for the author to continue development on the game, with the prospects of one day even publishing it if the final product is of high enough quality. For this to be achieved, other systems will have to be constructed as well, like the levelling and class systems that were initially mentioned but ended up being truncated, and the inventory systems with the accompanying armours and weapons. Other could be eventually added, but these are the main ones that were initially envisioned for the game, and that could make its existence worth publishing.

The model itself is a place that has a lot of possible improvements that can be performed. With the eventual increase in the number of abilities, the model would also need to change to fit these. It would also be an ideal opportunity to create a system that would vary the actions used from time to time, considering other options besides the optimal ones. A possible way of implementing this would be to create negative feedback outside the model, that would reduce the value of an action that was already used multiple times, making the system pick the next best action instead. This wouldn't be optimal but could be an interesting addition. Alternatively, there is the possibility of further refining the rewards given to the model. These can always be further tweaked to offer a better, more balanced training to the model.

The last idea for future work would be the addition of animations and more bespoke 3D models to the game. This would require a great amount of commitment and a new skillset the author currently does not possess, but it would add a lot to the game in the visual department, which is where it might be lacking the most.

8.3 Closing Remarks

In this final subchapter, the author's final thoughts on the whole development process will be expressed.

It would be incorrect to say that the project met all the expectations initially set by the author. With a tight development schedule and no experience, it was certain to be a demanding and daunting task, and so it was. The number of new systems and tools that had to be learned was very high, all of them of high complexity levels. Although research can be performed to try and mitigate the lack of knowledge and experience, the only way to really consolidate any sort of knowledge is through trial and error, by trying to do things and failing.

The experience of creating a video game is completely different from what was originally envisioned. As in any creative endeavour, there is a level of uncertainty linked to it. When deciding on what's best to do in each situation there are usually no wrong answers and only different options on how to do things. Doing a project

like these solo demands a great amount of discipline to keep all the pieces working together and synergizing, which wasn't always possible. There isn't a team to bounce ideas with, there is only the developer trying things and seeing how they work. This is then further complicated by not being able to do the things envisioned due to the lack of experience. Overall, the challenge level of developing the game was increased not only by the scope of the goal set, but also by the level of quality the developer wanted to imbue into it.

The product, although competent in trying to prove that it is possible to implement a model in real-time inside a video game, can't quite prove the point that this approach is better than conventional ones. Instead, it shows that it is an alternative that, even though not suited for every game, could be broadly implemented.

Overall, the experience of development was rewarding and enriching. These types of creations are something that can take years to make and yet, even after a year of work, it remained engaging and exciting throughout. The development will surely continue, even if it is with some conceptual or mechanical changes, in hopes of creating something great, worth publishing one day.

REFERENCES

[1] IBM, “What is Machine Learning? | IBM,” Jul. 15, 2020. https://www.ibm.com/cloud/learn/machine-learning (accessed Nov. 30, 2022).

[2] IBM, “What is Supervised Learning? | IBM,” Aug. 19, 2020. https://www.ibm.com/cloud/learn/supervised-learning (accessed Nov. 30, 2022).

[3] IBM, “What is Unsupervised Learning? | IBM.” https://www.ibm.com/topics/unsupervised-learning (accessed Jul. 10, 2023).

[4] B. Osiński and K. Budek, “What is reinforcement learning? The complete guide - deepsense.ai,” Jul. 05, 2018. https://deepsense.ai/what-is-reinforcement-learning-the-complete-guide/ (accessed Nov. 30, 2022).

[5] OpenAI, “GPT-4,” 2023. https://openai.com/gpt-4 (accessed Jul. 03, 2023).

[6] D. Silver *et al.*, “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play,” *Science (1979)*, vol. 362, no. 6419, pp. 1140–1144, Dec. 2018, doi: 10.1126/SCIENCE.AAR6404.

[7] R. Jagtap, “Markov Decision Process Explained | Built In,” Nov. 21, 2022. https://builtin.com/machine-learning/markov-decision-process (accessed Jul. 04, 2023).

[8] A. Choudhary, “Deep Q-Learning | An Introduction To Deep Reinforcement Learning,” Apr. 18, 2019. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/ (accessed Jun. 27, 2023).

[9] Y. Septiana, “Implementing State Machine in Android App | by Yolanda Septiana | Medium,” Jun. 03, 2020. https://medium.com/@yolapop/implementing-state-machine-on-android-app-634b2f75b08e (accessed Jun. 27, 2023).

[10] “Common game development terms and definitions | Game design vocabulary | Unity.” https://unity.com/how-to/beginner/game-development-terms (accessed Jul. 13, 2023).

[11] Mozilla Hubs, “What is a Nav Mesh?,” Feb. 25, 2022. https://hubs.mozilla.com/labs/what-is-a-nav-mesh/ (accessed Jul. 04, 2023).

[12] Night Quest Games, “Quick Introduction To Skeletal Animation for Video Game Development,” *https://www.nightquestgames.com/quick-introduction-to-skeletal-animation-for-video-game-development/*, Feb. 18, 2023.

[13] Potter 1992, “Level of Detail | potter1992,” Sep. 26, 2013. https://potter1992.wordpress.com/2013/09/26/level-of-detail/ (accessed Jun. 26, 2023).

[14] OpenAI, “DALL·E 2,” 2022. https://openai.com/dall-e-2 (accessed Jul. 03, 2023).

[15] D. Silver *et al.*, “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play,” *Science (1979)*, vol. 362, no. 6419, pp. 1140–1144, Dec. 2018, doi: 10.1126/SCIENCE.AAR6404.

[16] S. Madhavan and M. Sturdevant, “Machine learning and gaming - IBM Developer,” May 20, 2021. https://developer.ibm.com/articles/machine-learning-and-gaming/ (accessed Oct. 31, 2022).

[17] S. Nam and K. Ikeda, “Generation of diverse stages in turn-based role-playing game using reinforcement learning,” *IEEE Conference on Computatonal Intelligence and Games, CIG*, vol. 2019-August, Aug. 2019, doi: 10.1109/CIG.2019.8848090.

[18] B. Geisler, “Integrated Machine Learning For Behavior Modeling in Video Games”.

[19] N. Justesen, P. Bontrager, J. Togelius, and S. Risi, “Deep learning for video game playing,” *IEEE Trans Games*, vol. 12, no. 1, pp. 1–20, Mar. 2020, doi: 10.1109/TG.2019.2896986.

[20] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, "A Survey of Deep Reinforcement Learning in Video Games," Dec. 2019, doi: 10.48550/arxiv.1912.10944.
[21] "StarCraft: Remastered." https://starcraft.blizzard.com/en-us/ (accessed Jul. 30, 2023).
[22] E. Pagalyte, M. Mancini, and L. Climent, "Go with the Flow: Reinforcement Learning in Turn-based Battle Video Games," *Proceedings of the 20th ACM International Conference on Intelligent Virtual Agents*, 2020, doi: 10.1145/3383652.
[23] L. Han *et al.*, "Grid-Wise Control for Multi-Agent Reinforcement Learning in Video Game AI." PMLR, pp. 2576–2585, May 24, 2019. Accessed: Oct. 26, 2022. [Online]. Available: https://proceedings.mlr.press/v97/han19a.html
[24] A. Kanervisto, C. Scheller, and V. Hautamaki, "Action Space Shaping in Deep Reinforcement Learning," *IEEE Conference on Computatonal Intelligence and Games, CIG*, vol. 2020-August, pp. 479–486, Aug. 2020, doi: 10.1109/COG47356.2020.9231687.
[25] T. Miller, "Reward shaping — Introduction to Reinforcement Learning." https://gibberblot.github.io/rl-notes/single-agent/reward-shaping.html (accessed Dec. 06, 2022).
[26] R. R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Perez-Liebana, "Deep Reinforcement Learning for General Video Game AI," *IEEE Conference on Computatonal Intelligence and Games, CIG*, vol. 2018-August, Oct. 2018, doi: 10.1109/CIG.2018.8490422.
[27] T. Karras, T. Aila, S. Laine, A. Herva, and J. Lehtinen, "Audio-driven facial animation by joint end-to-end learning of pose and emotion," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, Jul. 2017, doi: 10.1145/3072959.3073658.
[28] S. Nivas, "The promise of voice AI in game development | VentureBeat," Sep. 14, 2020. https://venturebeat.com/games/the-promise-of-voice-ai-in-game-development/ (accessed Dec. 07, 2022).
[29] S. S. Esfahlani, H. Shirvani, J. Butt, I. Mirzaee, and K. S. Esfahlani, "Machine Learning role in clinical decision-making: Neuro-rehabilitation video game," *Expert Syst Appl*, vol. 201, p. 117165, Sep. 2022, doi: 10.1016/J.ESWA.2022.117165.
[30] J. Jonsdottir, R. Bertoni, M. Lawo, A. Montesano, T. Bowman, and S. Gabrielli, "Serious games for arm rehabilitation of persons with multiple sclerosis. A randomized controlled pilot study," *Mult Scler Relat Disord*, vol. 19, pp. 25–29, Jan. 2018, doi: 10.1016/J.MSARD.2017.10.010.
[31] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-Learning Algorithms: A Comprehensive Classification and Applications," *IEEE Access*, vol. 7, pp. 133653–133667, 2019, doi: 10.1109/ACCESS.2019.2941229.
[32] P. Patel, N. Carver, and S. Rahimi, "Tuning computer gaming agents using Q-learning | IEEE Conference Publication | IEEE Xplore." https://ieeexplore.ieee.org/abstract/document/6078182 (accessed Jul. 06, 2023).
[33] C. J. Lin, J. Y. Jhang, C. L. Lee, H. Y. Lin, and K. Y. Young, "Using a Reinforcement Q-Learning-Based Deep Neural Network for Playing Video Games," *Electronics 2019, Vol. 8, Page 1128*, vol. 8, no. 10, p. 1128, Oct. 2019, doi: 10.3390/ELECTRONICS8101128.
[34] Paradox Interactive, "Pillars of Eternity - Paradox Interactive," 2015. https://www.paradoxinteractive.com/games/pillars-of-eternity/about (accessed Jul. 03, 2023).
[35] Larian Studios, "Divinity: Original Sin 2," 2017. https://divinity.game/ (accessed Jul. 03, 2023).
[36] "The Witcher 3: Wild Hunt." https://www.thewitcher.com/us/en/witcher3#home (accessed Jul. 15, 2023).

[37] "Dragon Age™: Origins." https://www.ea.com/games/dragon-age/dragon-age-origins (accessed Jul. 15, 2023).
[38] S. Nunneley-Jackson, "Starfield features over 250,000 lines of dialogue, which is more than Skyrim and Fallout 4 combined | VG247," Oct. 13, 2022. https://www.vg247.com/starfield-250000-lines-of-dialogue-more-than-skyrim-and-fallout-4 (accessed Jun. 22, 2023).
[39] "The Witcher 3: Wild Hunt | Interface In Game | Video game UI." https://interfaceingame.com/games/the-witcher-3-wild-hunt/ (accessed Jul. 10, 2023).
[40] Oxford Dictionary, "replayability noun - Definition, pictures, pronunciation and usage notes | Oxford Advanced Learner's Dictionary at OxfordLearnersDictionaries.com." https://www.oxfordlearnersdictionaries.com/definition/english/replayability (accessed Jun. 25, 2023).
[41] C. Hall, "Pillars of Eternity 2 feels great as a turn-based game, update arrives this week - Polygon," Jan. 23, 2019. https://www.polygon.com/2019/1/23/18194374/pillars-of-eternity-2-deadfire-preview-impressions (accessed Jul. 03, 2023).
[42] M. Wutz, "'Into the Breach' is monsters, mechs and a reset for strategy games | Engadget," 2023. https://www.engadget.com/2018-02-27-into-the-breach-subset-games-ftl-sequel-gameplay-video.html?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAABflR5mBQUUlRMUxUr0b_iO3v5lzi4uIgvFiBLki1yYzKRFCerzWILhYxRtioEeaHl_DXFr7QaSb2byCrI9DhsfzWoQTh5zaoDFXmoRqmP-Vjmpq-f9lKL2zdD7f7_GQfleOWO72oxIepgncgepXqJHz7sB6QLUtKSWk5x5fmtNS (accessed Jun. 14, 2023).
[43] M. Thomsen, "History of the Unreal Engine - IGN," Jun. 14, 2012. https://www.ign.com/articles/2010/02/23/history-of-the-unreal-engine (accessed Jul. 03, 2023).
[44] C. Plante, "Better with age: A history of Epic Games," *Polygon*, Oct. 2012, Accessed: Jul. 03, 2023. [Online]. Available: https://www.polygon.com/2012/10/1/3438196/better-with-age-a-history-of-epic-games
[45] Unreal Engine, "Frequently Asked Questions - Unreal Engine." https://www.unrealengine.com/en-US/faq (accessed Jul. 03, 2023).
[46] D. Buckley, "Unreal Engine Blueprints Tutorials - Complete Guide - GameDev Academy," Feb. 09, 2023. https://gamedevacademy.org/unreal-blueprints-tutorial/ (accessed Jul. 10, 2023).
[47] Unreal Engine, "Behavior Trees in Unreal Engine | Unreal Engine 5.0 Documentation," 2022. https://docs.unrealengine.com/5.0/en-US/behavior-trees-in-unreal-engine/ (accessed Jun. 26, 2023).
[48] B. Conway, "3D Scanning and Photogrammetry Explained | VNTANA." https://www.vntana.com/blog/3d-scanning-and-photogrammetry-explained/ (accessed Jun. 26, 2023).
[49] Paizo, "Pathfinder Roleplaying Game: Unleash Your Hero! | Paizo," 2019. https://paizo.com/pathfinder (accessed Jun. 22, 2023).
[50] A. Krumins, "Creating Next-Gen Video Game AI With Reinforcement Learning | by Aaron Krumins | Towards Data Science," Oct. 19, 2020. https://towardsdatascience.com/creating-next-gen-video-game-ai-with-reinforcement-learning-3a3ab5595d01 (accessed Jul. 04, 2023).
[51] M. Wang, "Deep Q-Learning Tutorial: minDQN. A Practical Guide to Deep Q-Networks | by Mike Wang | Towards Data Science," Nov. 18, 2020. https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc (accessed Jul. 10, 2023).

Instituto Superior de Engenharia
Politécnico de Coimbra