Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
Mención en Ingeniería del Software

Análisis comparativo de motores de juego para juegos en navegador: rendimiento y aspectos técnicos

---

Alumno: Alberto Aguado del Caño

Tutor: Diego Rafael Llanos Ferraris
Tutor: Jaime Finat (Rapture-Games)

University of Valladolid

School of Computer Engineering **Bachelor's Degree in Computer Engineering**

Specialisation in Software Engineering

Comparison of Game Engines for Browser-Based Games: Performance and Technical Features

---

Student: Alberto Aguado del Caño

Supervisor: Diego Rafael Llanos Ferraris
Supervisor: Jaime Finat (Rapture-Games)

Resumen

En la industria actual del videojuego, donde la accesibilidad y la eficiencia energética son factores clave, optimizar el rendimiento de los juegos en plataformas limitadas se ha vuelto una necesidad creciente. Este trabajo se enmarca en el contexto de la exportación de videojuegos ligeros para navegadores web, con especial atención a su ejecución en sistemas operativos de bajo consumo como Raspberry Pi OS. Se ha desarrollado un prototipo de minijuego con funcionalidades básicas, implementado en tres motores de videojuegos ampliamente utilizados en la industria: Unity, Unreal Engine y Godot. El prototipo incluye dos modos de funcionamiento: un modo manual, donde el usuario controla directamente el personaje, y un modo automático, en el que el sistema recorre una serie de puntos predefinidos y dispara a objetivos de forma autónoma.

El objetivo principal de este trabajo es evaluar el rendimiento comparativo de cada motor al exportar el juego a formato web y ejecutarlo en diferentes entornos, incluyendo un ordenador de especificaciones modestas y una Raspberry Pi. Para ello, se definieron una serie de parámetros clave a medir —como tasa de fotogramas por segundo (FPS), uso de CPU y memoria, entre otros— y se utilizaron herramientas de análisis de rendimiento (profilers) disponibles en navegadores web. Cada motor fue sometido a un mínimo de cinco ejecuciones por entorno, asegurando así la fiabilidad estadística de los resultados obtenidos.

Finalmente, se realiza un análisis comparativo de los resultados para determinar qué motor ofrece un mejor equilibrio entre rendimiento, calidad gráfica y facilidad de exportación a navegador. Las conclusiones de este trabajo permiten ofrecer recomendaciones fundamentadas para desarrolladores que buscan crear experiencias web optimizadas, especialmente en aplicaciones de realidad virtual o en dispositivos con recursos limitados.

Abstract

In today's video game industry, where accessibility and energy efficiency are increasingly important, optimising performance for games on limited hardware platforms is a growing concern. This thesis focuses on the development and web-based deployment of lightweight games intended to run on low-resource systems such as Raspberry Pi OS, through web browsers without the need for local installation. A prototype mini-game was developed using three major game engines — Unity, Unreal Engine, and Godot — each supporting the implementation of basic gameplay mechanics.

The mini-game features two operational modes: a manual mode, where the user directly controls the character, and an automatic mode, in which the system navigates through a set of predefined waypoints whilst autonomously shooting targets. The core objective of this work is to assess and compare the performance of each engine when exporting the game to WebGL or equivalent formats, and executing it in constrained environments — specifically, a low-end personal computer and a Raspberry Pi.

To conduct a fair comparison, performance metrics such as frame rate (FPS), CPU usage, and memory consumption were monitored using browser profiling tools. Each engine underwent a minimum of five test runs per environment, ensuring reliable average performance data. The gathered results were then analysed to determine which engine provides the best trade-off between performance efficiency, visual quality, and ease of deployment to the web.

This study offers valuable insights and practical recommendations for developers seeking to optimise game performance on low-power or embedded devices, with applications in web-based gaming, educational tools, or even virtual reality environments where resource constraints are critical.

Contents

Chapter 1

Introduction

This chapter presents the general context that underpins this Final Year Project, which focuses on the comparison of game engines designed for browser-based deployment. Traditionally, video games have been developed and deployed primarily on desktop platforms and gaming consoles, where local hardware performance played a critical role in delivering a satisfactory user experience. However, recent advancements in web technologies — such as WebGL and WebAssembly — have enabled the execution of increasingly complex games directly within web browsers, eliminating the need for installation while offering competitive performance.

In addition to the technological background, this chapter outlines the main objectives of the project, which involve evaluating and comparing different game engines in terms of performance, technical features, and their suitability for developing browser-based games.

Finally, a review of the state of the art is provided, examining previous comparative studies on game engines. This analysis aims to identify existing research, methodologies employed, and current gaps in the literature, thereby supporting the relevance and originality of the present work.

1.1 Context

The gaming industry is rapidly evolving with advancements in AI, Cloud gaming, and immersive experiences such as VR and AR. Developers must innovate to stay competitive. They must improve user experiences, optimise operational efficiency, and use advanced graphics to create more realistic, engaging games. This transformation requires adapting to new technologies and market demands to maintain relevance and success in an increasingly digital and immersive environment.

Among others, some of the main detected causes for this rapidly evolving enviroment would be:

- **New forms of human–computer interaction**: Traditional methods of playing video games, such as controllers or keyboards and mice, are becoming outdated. Some of these methods have been on the market for years but were not fully developed. However, in recent years, significant research and investment have made them more appealing[1]. Some of these are:
  - **Voice recognition**: various solutions have been proposed in the field, such as keyword recognition, emotion recognition, interactive games, and command recognition. From the early models of voice interaction—such as the japanese version of *The Legend of Zelda* for the NES console (known in Japan as the Famicom), where certain enemies could be defeated by shouting into the microphone of the controller[2], to more recent developments like the use of Convolutional Neural Network (CNN) to control the classic Snake in Python through voice commands such as “up”, “down”, “left”, and “right”, the field has undergone significant evolution[3].
  - **Facial recognition**: it has been demonstrated that the use of this technology in video games enhances player experience, making it more engaging. Furthermore, if a difficulty system

were to be implemented based on facial expressions, it could suggest lowering the difficulty level when the player shows signs of frustration, or increasing it when signs of satisfaction are detected. This would allow for a clearer representation of the player's learning curve[4].
It is true that, among all the advances in human-computer interaction, this is one of the least emphasized. In the field of virtual reality, where the subject of study is based, there are challenges in capturing facial expressions or all facial features, as the face is often obscured. Nevertheless, thanks to the use of Machine/Deep learning, significant results can be achieved. One such example is the use of CNN, which can accurately categorise emotions using only facial features extracted from areas of the face other than the eyes and eyebrows, even when facial information in those regions is absent. This suggests that the model can effectively detect emotions across different gaming scenarios[5].

– **Gesture control**: it is the area that has received the most emphasis in recent years, from the early tracking models, such as Kinect. Which is an input device developed by Microsoft, originally released for the Xbox 360 gaming console and later made compatible with Xbox One and PC, this device uses a series of sensors, such as an RGB camera, a depth sensor (which uses infrared technology), and a microphone, to capture the user's movement and actions, the used algorithms are k-nearest neighbors (KNN) or Support Vector Machine (SVM)[6].
Currently, VR is used by implementing CNN as an algorithm[7].

- **Diversification of platforms and business models**: in recent years, the number of platforms where any video game can be developed and deployed has increased, the rapid growth of this industry has created a need for business models that align with this expansion:

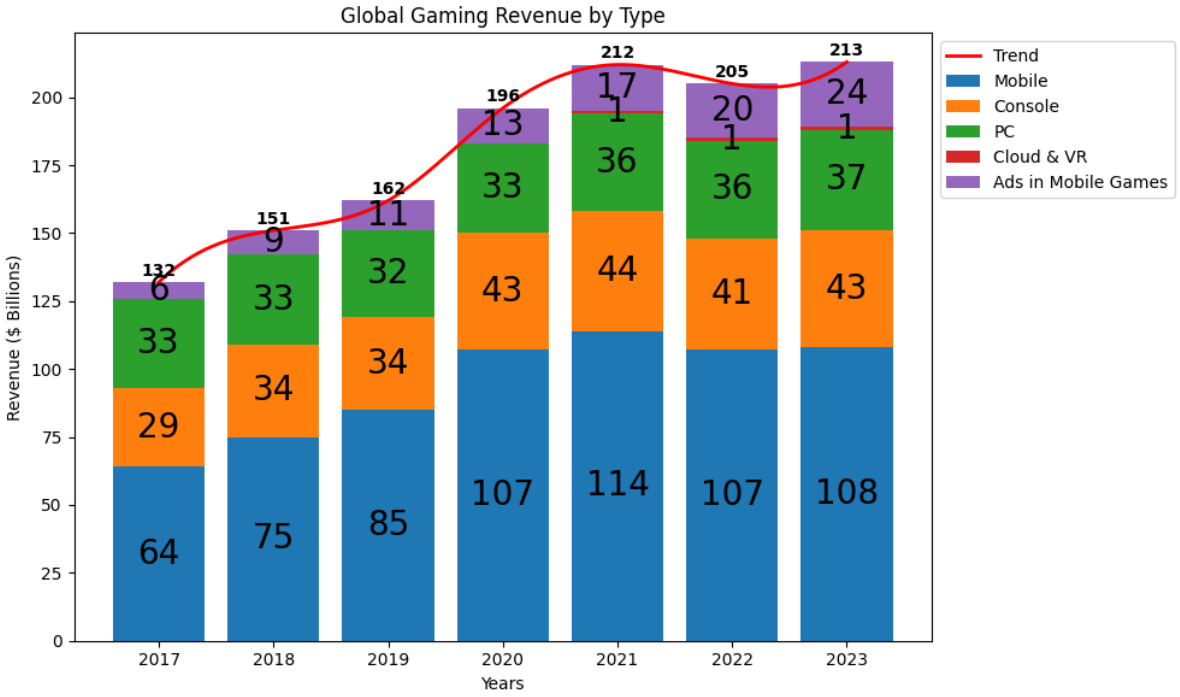

Figure 1.1: The Boston Consulting Group examines how the gaming industry's revenue surged from $131 billion to $211 billion between 2017 and 2021, reflecting a compound annual growth rate of 13%, influenced by diversified platforms and business models.[8]

  Therefore, it has been reflected in new market strategies, such as: developing a subscription strategy, embracing AI and automation for efficiency, or focusing on untapped demographics among others.[9]
- **Increased social and cultural acceptance**: video game music is gaining cultural recognition, as seen in events like the London Soundtrack Festival, which celebrates video game scores alongside traditional film and TV music[10].
- **Impact of the COVID-19 pandemic**; the pandemic led to a surge in gaming, with millions of new players purchasing games and consoles during lockdowns, significantly boosting the industry's growth.[11]
- **Rise of streaming and content creators**: the rise of streaming platforms and the influence of content creators have had a significant impact on the video game industry, transforming both the way games are consumed and developed. Streamers have played a crucial role in spreading and popularising various titles. For instance, games like *Among Us* and *Fornite* saw a notable increase in their player base thanks to live streams conducted by influencers.[12]

As can be seen, the industry is consolidated, so we must place great importance, as computer engineering professionals, on following engineering processes to deliver the highest quality product possible. This will allow us to achieve better revenue and ensure backward compatibility when creating better products than previous ones.

1.2 Objectives

Main objective

To analyse and compare different game engines focused on web browser game development, primarily assessing their performance.

Specific objectives:

- **Investigate main game engines**: a memory of each engine will be prepared highlighting the pros and cons, detected weaknesses and best case scenarios.
- **Define metrics**: at least the following metrics will be measured for each engine: FPS, RAM, size of the exported game, among others that might arise during the development of the project.
- **Create test prototypes**: configurable automatic execution for every engine/browser, automatically outputting the aforementioned metrics.
- **Analyse and comparation**: look the results and determine the strengths and weaknesses of each engine in different development scenarios.
- **Draft conclusions and recommendations**: identifying the most suitable engine based on different use cases, such as 2D games, 3D games, multiplayer experiences, or lightweight interactive applications.

1.3 State of the art

The focus of this thesis will be to determine which game engine performs the best in web browsers, specifically on low-resource computers or embedded systems, such as virtual reality glasses.

First of all, let's conduct a state of the art related to this subject. Previous studies on performance metrics in libraries that run natively in the browser using WebGL, such as ThreeJS and BabylonJS (later, the nature of these libraries will be explained.), have been considered, In the following tables:

Library	**Chrome**		**Firefox**	
	CPU (%)	GPU (%)	CPU (%)	GPU (%)
Three.js	6–8	31–41	6–8	38–41
Babylon.js	5–7	31–41	5–8	36–38

Table 1.1: CPU/GPU usage on high-end systems in 3D graphics in browsers[13]

Library	**Chrome**		**Firefox**	
	CPU (%)	GPU (%)	CPU (%)	GPU (%)
Three.js	53–67	40–42	43–49	36–38
Babylon.js	46–48	44–45	32–40	36–40

Table 1.2: CPU/GPU usage on low-end systems in 3D graphics in browsers[13]

the usage of CPU and GPU can be observed in high-end and low-end computers respectively, for the rendering of 3D graphics, it can be seen that GPU usage does not vary significantly between both systems, whereas CPU usage increases considerably. Therefore, it can be concluded that programs must be optimised in order to avoid bottlenecks that could render the video game unplayable.

This becomes even more evident in the following tables:

Library	Chrome		Firefox	
	@30FPS	@60FPS	@30FPS	@60FPS
Three.js	29998	15157	15844	8977
Babylon.js	6360	3490	4820	2610

Table 1.3: Simultaneous sprites reached to achieve 30 and 60 FPS on high-end systems in 3Dgraphics in browsers[13]

Library	Chrome		Firefox	
	@30FPS	@60FPS	@30FPS	@60FPS
Three.js	13030	5449	5670	2340
Babylon.js	2530	1080	1940	710

Table 1.4: Simultaneous sprites reached to achieve 30 and 60 FPS on low-End systems in 3Dgraphics in browsers[13]

which show how many sprites the engine can render without the performance dropping below a given target framerate (in these cases, 30 and 60 FPS respectively). It can be seen that at 30 FPS, both high-end and low-end systems are able to render nearly twice as many sprites as at 60 FPS. This is because fewer sprites must be displayed at 60 FPS in order to maintain smoother performance, avoiding excessive processing that could impact the frame rate. Nevertheless, the performance gap between high-end and low-end systems remains evident, reinforcing the conclusion mentioned earlier. These 3D graphics libraries will be explored in more detail in a later section.

Following the research, it has been found that comparative studies between engines have been conducted, but always from a superficial perspective and without examples based on simple video games, instead relying on specific hardware features[14].

In conclusion, it can be seen that there are no studies that have made a real comparison with prototypes or that also specify programming-related aspects.

The key terms used in this state of the art study have been: *Comparison of Game Engines for Browser-Based Games*, *Browser-Based Games performance*, *game engines performance comparison*

1.3.1 Why use web browsers?

We will focus on a part of video games that may generate the least revenue as shown in the image:

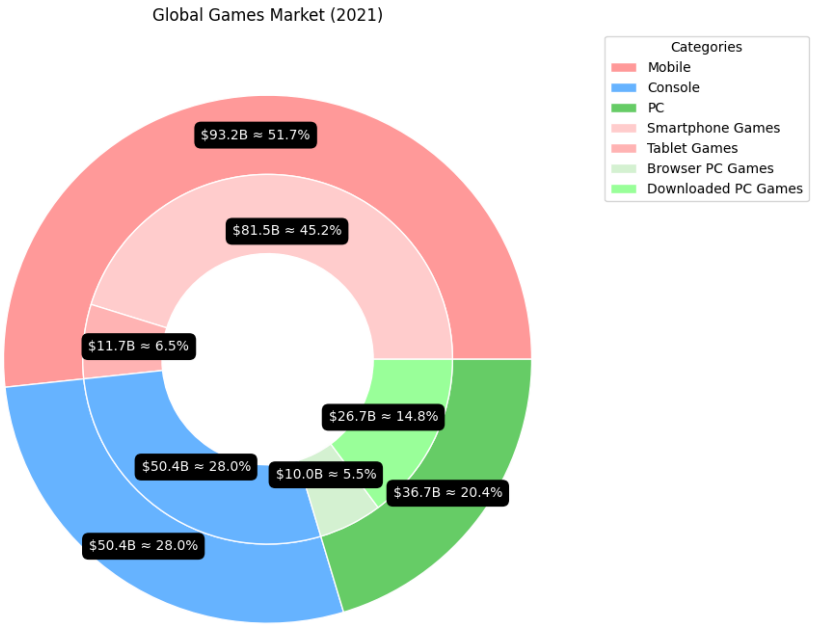

Figure 1.2: The graph below shows the market share percentage, clearly segmenting gaming revenues based on platforms and gaming categories for 2021.[1]

but is still important, which is **browser-based games**. Browser-based games have indeed lost some of their appeal due to their basic graphics, almost non-existent artistic design, and the fact that many of them are now programmed using AIs like ChatGPT or third-party AIs that create the game without understanding how it works internally, resulting in high-performance costs. However, they remain relevant and important for several factors, which are:

- **Easy accessibility**: one major reason that browser-based games remain popular is their unbeatable ease of access. In contrast to classic video game methods that normally would require specific hardware or installing specific software, browser games are instantly playable on any device with an Internet connection. This is very convenient, and users can enjoy gaming experiences during short breaks at work while using public transportation, or from the comfort of their home.
- **Cost-effectiveness**: design and graphics play a crucial role in the overall development cost of applications. Games that incorporate advanced graphic assets, such as 3D graphics, offer an impressive visual experience and an attractive user interface. However, these resources require intensive use of time and resources, significantly increasing development costs. Furthermore, it is noted that 3D graphics are more expensive than 2D graphics, what can be seen in this comparison:

Scale and Complexity	– Basic 2D graphics with simple assets and animation may cost around $10,000 to $10,000 – Complex 2D graphics with high-resolution assets and advanced animation could cost around $10,000 to $50,000
3D Graphics	– Basic 3D graphics with simple animations could cost around $5,000 to $20,000 – Advanced or complex 3D graphics with high resolution, animations, and special effects could cost $20,000 or more.

Table 1.5: A comparison of 2D vs 3D graphics costs in mobile games, also applicable to browser-based games.[15]

- **Educational potential**: browser-based games can also be used in an educational setting. The ease of access and low barriers to entry make them perfect tools for interactive learning experiences. Games such as Duolingo have completely changed the way people look at learning languages by providing playful, browser-based exercises, and making education a game. In addition to language learning, browser-based games can help kids (and adults) with math, biology, geography, and history.

A good example of these factors is **netQuake** (`https://www.netquake.io/`). This browser edition of the globally famous first-person shooter game proves that graphically games can now be enjoyed directly from a web browser too.

1.3.2 And why in low-end systems?

In a high-end computer, even if performance optimisation is carried out, the difference will not be noticeable, as the CPU, GPU, and RAM more than compensate for the shortcomings of poor optimisation. This is especially true considering recent technologies such as NVIDIA's Deep Learning Super Sampling (DLSS) and AMD's FidelityFX™ Super Resolution (FSR). However, in low-performance systems, optimising our programs is critical to prevent a poor user experience.

For the tests and evaluations, trials will be conducted not only on a desktop computer, but also on a **Raspberry Pi 4 Model B**, which features the following specifications[16]:

- **CPU**: Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
- **RAM**: 8GB
- **Graphic interface**: OpenGL ES 3.1, Vulkan 1.0
- **Temperature**: 0–50 degrees C ambient

As can be noted, there is no external GPU present, as it is integrated within the CPU itself. In this case, the integrated GPU is the **VideoCore VI**, which delivers a performance of approximately 0.3 TFLOPS[17]. This highlights just how far its performance falls behind even that of entry-level desktop PCs or gaming consoles, for example, the Playstation 5 have 10.28 TFLOPS[18].

In addition to this, it must be noted that the Raspberry Pi OS is a GNU/Linux distribution, which means that the majority of games cannot be run, as most of them are developed for Windows platforms, this is due to that the majority of games on the market are compatible with DirectX, that is a set of APIs that serve as a bridge between games and a PC's hardware, allowing game developers to create games without requiring knowledge of the specific CPU, GPU, RAM, motherboard, or other components of a given system. is a middleman between the hardware drivers and the game.[19]

This makes libraries such as OpenGL or Vulkan less attractive for programming, as they require explicit management of hardware resources. Additionally, most games are developed with compatibility for DirectX.

However, if we run the games in a browser, Chromium and Firefox browsers can enable hardware acceleration. This allows the GPU to dedicate all its resources to rendering graphics, making the CPU less burdened and thereby improving performance.

1.3.3 Game engines and libraries

The game engines to be studied are:

- **Unity**: widely used game engine that allows developers to create games for PC, mobile platforms, consoles, and the web. It is known for its ability to create highly detailed graphics and user experiences, which can be challenging for devices with limited resources.[20]
- **Unreal Engine**: is another well-known game engine used in the development of AAA games and other interactive applications. Unreal Engine is famous for its impressive graphics capabilities and powerful physics engine, which may demand more resources and pose challenges when running on low-performance systems.[21]
- **Godot**: an open-source game engine that has gained popularity for its flexibility, ease of use, and cross-platform support. Godot tends to be lighter in terms of resource usage, making it an attractive option for games on systems with limited hardware. However, it has a smaller community compared to the two engines mentioned, and its visual appearance is not as polished as theirs.[22]

Having established the motivation, objectives, and background of the project, the next step is to define a formal analysis model that supports the development and evaluation process. The following chapter introduces the scope of the system, user requirements, and use cases, which together provide the technical framework necessary to design and implement the comparative prototype for browser-based game engines.

Chapter 2

Analysis Model

This chapter outlines the analysis phase of the project, which establishes the technical and functional foundations for the design and implementation stages. The objective of this phase is to define the scope of the system, identify user requirements, and model the core functionality that the prototype must support in order to serve as a valid platform for comparing game engines in a browser-based environment.

The chapter begins by describing the scope and planning of the project, followed by the methodology adopted and an assessment of potential risks. It then presents a detailed specification of functional, non-functional, and information requirements. Based on these requirements, a set of use cases is defined to illustrate user interactions and system responses, including gameplay actions and test automation. Finally, a domain model is introduced to represent the key entities and relationships within the game system.

2.1 Model and project planning

2.1.1 Scope of the project

The project scope will be defined, focusing on three fundamental points, which are:

- **FPS development**: a basic FPS game prototype will be developed in Unity, Unreal, and Godot.
- **Performance testing in web browsers**: compare the performance of these engines under the same conditions (*) using browser profilers.
- **Performance testing on different hardware platforms**: the tests will be executed on two types of hardware platforms: PC and Raspberry Pi.

(*) Equivalent conditions will be ensured, within reasonable expectations, by implementing in-game logic that automatically completes the game with the same behaviour in each engine.

2.1.2 Methodology project

An agile methodology will be adopted for the development of this project, following a hybrid approach that combines elements of both Kanban and Scrum frameworks, to adapt to the different stages of work.

In the initial phase, which will involve research and the early development of the prototype (starting with the Unity engine), a Kanban approach will be used. This model will allow tasks to be managed continuously. It is true that Kanban also manages time, but it is more flexible than Scrum, which makes it suitable for the first phase, as research is being conducted on how to use the aforementioned game engines, as well as the use of profilers in web browsers.

Once the exploratory phase is completed and a stable foundation of the project is in place, the Scrum framework will be adopted, using two-week sprints. In each sprint, specific tasks will be planned, progress will be monitored regularly, and reviews will be conducted to ensure objectives are met. A retrospective will follow each sprint to evaluate what was achieved and improve future iterations.

This hybrid model will make it possible to start with flexibility and then introduce greater control and structure as the project progresses, adapting both to technical requirements and to the time constraints of the final year project.

Given that the work team consists of two people, the agile methodology has been adapted (combining Scrum and Kanban principles) to suit the project's needs. Two main roles have been defined:

- **Developer**: responsible for the technical implementation of the project, including code development, functional testing, and technical documentation. The role is also responsible to organise and prioritise the work using the Kanban board in Jira.
- **Reviewer and Validator**: responsible for overseeing the work done, reviewing code quality, and validating deliverables against the established objectives. The roles also ensure that the requirements are met and suggest improvements once the development process has been marked as completed.

2.1.3 Project planning

The following estimated schedule is based on a standard 40-hour workweek, with the project divided into phases using Kanban and Scrum methodologies as appropriate:

Phase	Weeks	Methodology	Hours/week	Estimated total hours
Training in profiling and game engines	1–4	Kanban	40	160
Sprints 1 to 4: prototype creation and automatic mode	5–12	Scrum	40	320
Sprints 5 to 7: testing, conclusions, and adjustments	13–18	Scrum	40	240
Sprint 8: documentation and closure	19–20	Scrum	40	80
			Total project hours	**800**

Table 2.1: Estimated project planning using Kanban and Scrum

The project begins with one month under the Kanban methodology for training in profiling and game engines, allowing flexibility and focus on initial learning, with an approximate dedication of 160 hours. Subsequently, Scrum is implemented during the development, testing, and closing phases, applying structured sprints totaling 640 hours to ensure incremental deliveries and adaptability.

2.1.4 Risk management plan

The risks that have been identified are:

1. **Graphics driver incompatibility with engines**: some engines such as Unity, Unreal, or Godot may require specific versions of drivers that may not be available or may not function correctly on certain platforms, such as Raspberry Pi OS (Raspbian) or other Linux-based distributions.
2. **Raspberry Pi performance limitations**: the limited processing power and memory of the Raspberry Pi could hinder performance results or even cause failure to execute tests.
3. **Issues capturing profiling data**: problems may arise when collecting, saving, or interpreting the profiling metrics during performance tests.
4. **Automation script failures**: errors or unexpected behaviour in movement or shooting automation may disrupt the test flow.
5. **Cross-browser performance variation**: differences in system configuration can skew results, making them hard to compare.
6. **Delays in documentation delivery**: final report writing may be delayed due to technical workload or poor time management.
7. **Developer illness**: a potential sick leave or health issue affecting the only developer may halt progress and delay project deadlines.
8. **Communication breakdown between developer and reviewer**: poor communication between the developer and the reviewer could result in misunderstandings regarding the requirements, development progress, or technical implementation details, potentially leading to undetected errors.

The risk management plan is presented in this table:

ID	Name	Probability	Impact	Mitigation Strategy
R01	Graphics driver incompatibility with engines	Medium	High	Check engine compatibility with drivers from early stages, use LTS or stable versions, and maintain a documented execution environment.
R02	Raspberry Pi performance limitations	Medium	High	Lower graphical quality and adjust settings to fit the hardware limitations.
R03	Issues capturing profiling data	Medium	Medium	Use tested profiling tools and validate data capture from the start.
R04	Automation script failures	Medium	Medium	Modularise scripts, implement detailed logging, and perform unit testing.
R05	Cross-browsers performance variation	High	Medium	Standardize the testing environment and ensure consistent testing conditions across browsers.
R06	Delays in documentation delivery	Low	Medium	Set internal deadlines and allocate time specifically for documentation creation and review.
R07	Developer illness	Low	High	Plan partial deliverables, maintain up-to-date technical documentation and foresee a reasonable extension of the schedule if needed.
R08	Communication breakdown between developer and reviewer	Medium	Medium	Regular meetings to review progress, ensure clear documentation, and use project management tools for task tracking.

Table 2.3: Risk management plan table

Now proceed to make an estimation of time in days and cost. The expected risk is calculated using the following formula:

$$\text{Risk} = \text{Probability} \times \text{Impact (days)} \tag{2.1}$$

Risk	Probability (D)	Impact (days)	Risk (days)
R01	0.5 (Medium)	7 (High)	3.5
R02	0.5 (Medium)	7 (High)	3.5
R03	0.5 (Medium)	5 (Medium)	2.5
R04	0.5 (Medium)	5 (Medium)	2.5
R05	0.7 (High)	5 (Medium)	3.5
R06	0.3 (Low)	5 (Medium)	1.5
R07	0.3 (Low)	7 (High)	2.1
R08	0.5 (Medium)	5 (Medium)	2.5
Total expected cost if all risk occur			**21.6**

Table 2.4: Expected risk value in days

The conclusions drawn from this indicate that, as priorities, risks R01, R02, and R05 should be mitigated as soon as possible.

2.2 Cost estimation

The following assumptions are considered for the cost estimation:

- **Annual cost of a junior engineer**: €40,000.
- **Working hours**: 52 weeks per year, 40 hours per week.
- **Hardware cost** (computer and monitor): €1,000.
- **Hardware amortisation period**: 3 years (36 months).
- **Estimated project duration**: 20 weeks (approximately 6 months).

Cost calculation

- **Engineer hourly rate**:
$$\frac{40,000\,€}{52 \times 40} \approx 19.23\,€/\text{hour}$$
- **Total labour cost excluding risks**:
$$800\,\text{hours} \times 19.23\,€/\text{hour} = 15,384\,€$$
- **Proportional hardware cost (assuming 36-month amortization)**:
$$\frac{1,000\,€}{36\,\text{months}} \times 6\,\text{months} = 166.68\,€$$
- **Additional cost for risk contingency**: Estimated 21.6 extra days × 8 hours/day = 172.8 hours.
$$172.8\,\text{hours} \times 19.23\,€/\text{hour} = 3,323\,€$$

Estimated total project cost

$$15,384\,€ + 166.68\,€ + 3,323\,€ = \mathbf{18,873.68}\,€$$

This estimate includes a contingency margin to cover possible unforeseen events arising from the identified risks, thus improving the reliability of the budget.

2.3 User requirements

The requirements to be provided are:

2.3.1 Functional requirements

1. **FR01**: the system must allow the player to move freely in a 3D first-person environment.
2. **FR02**: the system must allow the player to shoot with a weapon via an input command.
3. **FR03**: the system must allow the weapon to be reloaded either through a player input command or automatically when the ammunition is depleted.
4. **FR04**: the system must allow the inclusion of shooting targets acting as enemies within the environment.
5. **FR05**: the system must handle the disappearance of a shooting target when hit by the player's shot and update the count of remaining targets in the scene.
6. **FR06**: the system must provide the player with the option to restart the game or exit the game either after a session ends or from the victory menu.
7. **FR07**: the system must activate the profiler at the start of the automated run and stop it upon completion, collecting all performance data.
8. **FR08**: the system must automatically execute a predefined path in the scenario, where the player moves without user input when execute the performance testing.
9. **FR09**: the system must automatically aim and shoot at each target panel in sequence until all enemies (targets) have been eliminated, completing the performance test.

2.3.2 No-functional requirements

1. **NFR01**: the system must be compatible with three game development engines: Unity, Unreal Engine, and Godot.
2. **NFR02**: the system must be compatible with most web browsers, primarily those based on Chromium and Firefox.
3. **NFR03**: the system must be compatible with WebGL to render 3D graphics in web browsers.
4. **NFR04**: the system must start as fast as possible to reduce the initial waiting time. For web browser video games, load times are expected to range between 3 and 6 seconds under standard conditions for a smooth user experience.
5. **NFR05**: the system must be executable on a Raspberry Pi with a compatible browser, in order to evaluate performance on low-cost, energy-efficient hardware.

2.3.3 Information requirements

1. **IR01**: the system shall display a HUD to the player showing the crosshair, ammunition, the current number of targets, and the elapsed time.
2. **IR02**: the system must display a victory message and the elapsed time when the player eliminates all targets in the scene.
3. **IR03**: the system must store performance profiling results in JSON files, enabling further analysis and visualisation.

2.4 User Cases

The resulting use case diagram is as follows:

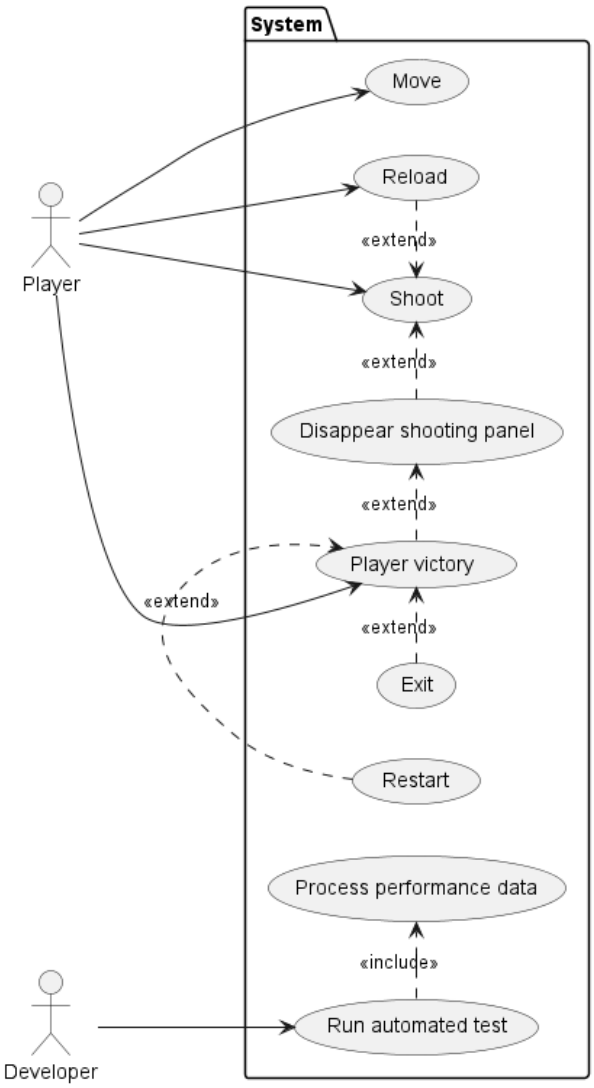

Figure 2.1: Use cases diagram

2.4.1 UC01: Move

ID	**UC01: Move player**
Related requirement(s)	FR01
Actor(s)	Player
Description	This use case describes how the player can move freely within the 3D first-person environment using input controls.
Preconditions	• The system is initialized. • The game scene is loaded. • The player is in control (not in a cutscene or paused).
Main flow	1. The player presses directional keys (W, A, S, D or arrow keys). 2. The system interprets the input and computes the new position. 3. The system updates the player's position in real time in the 3D environment.
Postconditions	• The player's position has changed according to input. • The 3D environment reflects the new position.
Alternative flows	• **A1: Movement blocked by obstacle** – At step 2, if there is a wall or object in the direction of movement, the system prevents the player from passing through it. – The player's position remains unchanged.

Table 2.5: Use Case: Move

2.4.2 UC02: Shoot

ID	UC02: Shoot
Related requirement(s)	FR02
Actor(s)	Player
Description	This use case describes how the player can shoot with the weapon using an input command, such as pressing a button or key on the controller or keyboard.
Preconditions	• The player has a weapon equipped. • The player is in the game environment (not in a menu or paused screen).
Main flow	1. The player presses the command to shoot (e.g., "left mouse click" or "spacebar"). 2. The system interprets the shoot command and checks if the weapon is ready to shoot. 3. The system executes the shot, performing the impact calculation. 4. The system plays the shooting animation and the corresponding visual and audio effects (shooting sound). 5. The system updates the HUD, reflecting the remaining ammunition. 6. The system calculates whether the shot has hit a target (panel shooting, objects, or nothing).
Postconditions	• The player's ammunition has been reduced accordingly. • The game state has been updated, reflecting the impact of the shot (if applicable).
Alternative flows	• **A1: Ammunition exhausted** – In step 2, if the player's ammunition has been exhausted, the system displays indicator in the HUD (e.g., "0/30") and does not allow shooting. – The system triggers the use case **UC03: Reload the weapon**. • **A2: Weapon not ready to shoot** – In step 2, if the weapon is not ready to shoot (e.g., if it is in use case **UC03: Reload the weapon**), the system prevents the shot. • **A3: Target hit** – In step 6, if the shot hits a target and the target is a "shooting panel", the system triggers the use case **UC04: Make panel disappear**. – otherwise, no further action is taken.

Table 2.6: Use Case: Shoot

2.4.3 UC03: Reload

ID	**UC03: Reload**
Related requirement(s)	FR03
Actor(s)	Player
Description	This use case describes how the player can reload the weapon manually via input, or automatically when the ammunition is depleted.
Preconditions	• The system is initialised. • The weapon is equipped. • Current ammunition is below its maximum capacity.
Main flow	1. The player presses the reload key. 2. The system refills the weapon to its maximum allowed capacity. 3. The system updates the HUD to reflect the new ammunition amount.
Postconditions	• The weapon is reloaded. • The HUD correctly displays the updated ammunition.
Alternative flows	• **A1: Automatic reload** – At step 1, if the player attempts to shoot with no ammunition, the system automatically triggers the reload process. – Continue from step 2 of the main flow.

Table 2.7: Use Case: Reload

2.4.4 UC04: Disappear shooting panel

ID	**UC04: Disappear shooting panel**
Related requirement(s)	FR05
Actor(s)	
Description	This use case describes how the system removes a shooting panel (target) when it is hit by a shot.
Preconditions	• The system is initialised. • The player has fired and the shot has hit a target. • The hit target is a shooting panel.
Main flow	1. The system removes the panel. 2. The system checks if there are any remaining targets in the scene. 3. If there are more targets, no further action is taken. 4. The system updates the number of remaining targets on the HUD.
Postconditions	• The shooting panel has disappeared from the scene. • The number of remaining targets is correctly displayed on the HUD.
Alternative flows	• **A1: The Panels Shooting Dissapeared** – At step 2, if there are no remaining targets, the system triggers the use case **UC05: Player victory**.

Table 2.8: Use Case: Disappear shooting panel

2.4.5 UC05: Player victory

ID	**UC05: Player victory**
Related requirement(s)	FR06
Actor(s)	Player
Description	This use case describes how the system declares player victory once all targets have been eliminated, and presents options to either restart the game or exit.
Preconditions	• The system is initialised. • All targets have been removed from the scene.

Main flow	1. The system displays a victory message to the player.
	2. The system halts active gameplay logic.
	3. The system disables the player control.
	4. The system presents two buttons: “Restart” and “Exit”.
	5. The player selects one of the two options.
	6. The system triggers the corresponding use case:
	  • If “Restart” is selected, **UC06: Restart** is triggered.
	  • If “Exit” is selected, **UC07: Exit** is triggered.

Postconditions	• The player has been informed of their victory.

Table 2.9: Use Case: Player Victory

2.4.6 UC06: Restart

ID	UC06: Restart
Related requirement(s)	FR06
Actor(s)	Player
Description	This use case describes how the player can restart the scenario after achieving victory.
Preconditions	• The player has been informed of their victory. • The player has selected the “Restart” option.
Main flow	1. The system resets the game variables (targets, ammo, HUD, etc.). 2. The system reloads the initial scene from the beginning. 3. The system enables the player to regain control.
Postconditions	• The game is reset and ready to play. • The player can move and act normally.

Table 2.10: Use Case: Restart

2.4.7 UC07: Exit

ID	**UC07: Exit**
Related requirement(s)	FR06
Actor(s)	Player
Description	This use case describes how the player can close the application after achieving victory.
Preconditions	• The player has been informed of their victory. • The player has selected the “Exit” option.
Main flow	1. The system gracefully shuts down the application.
Postconditions	• The application has been closed safely.

Table 2.11: Use Case: Exit

2.4.8 UC08: Run automated test

ID	UC08: Run automated test
Related requirement(s)	FR7, FR8, FR9
Actor(s)	Developer
Description	This use case describes how the developer initiates an automated test that runs a predefined path and shoots targets in the scenario without user input. The system handles the movement and actions.
Preconditions	• The profiler is initialised and capturing performance data. • The system is initialised. • The scene is loaded and contains the targets.
Main flow	1. The developer starts the automated test. 2. The system moves the player through the scenario without user input following a predetermined path. 3. The system automatically aims and shoots at each target in sequence until all targets are eliminated. 4. The system stops the automated test. 5. The system triggers the use case **UC09: Process performance data**.
Postconditions	• The automated test has been completed.
Alternative flows	• **A1: Error in automated test** – If the system detects an error during the test (e.g., a failure in the path or shooting), the test is interrupted, and error data is displayed.

Table 2.12: Use Case: Run automated test

2.4.9 UC09: Process performance data

ID	UC09: Process performance data
Related requirement(s)	IR3
Actor(s)	
Description	This use case describes how the system saves the performance data generated during the automated tests for later analysis.
Preconditions	• The automated test has finished. • The performance data has been stored by the profiler.
Main flow	1. The system opens the execution profile generated during the automated test. 2. The system analises and stores the filtered data in a file or database. 3. The system displays a message confirming that the data has been saved successfully. 4. The system checks if the number of iterations for the test has been reached (5 by design). 5. The system generates the average values per execution and stores the result in a separate file.
Postconditions	• The performance data has been successfully saved. • The data is available for later analysis.
Alternative flows	• **A1: Error saving/reading data** – If an error occurs while opening files with the data (e.g., lack of space or database failure, reading permissions, etc.), the system displays an error message and asks the user to fix it and try again. • **A2: Number of repetitions for the test not reached** – In step 4, if the maximum number of iterations is not reached the system triggers the use case **UC08: Run automated test**.

Table 2.13: Use Case: Process performance data

2.5 Domain Model

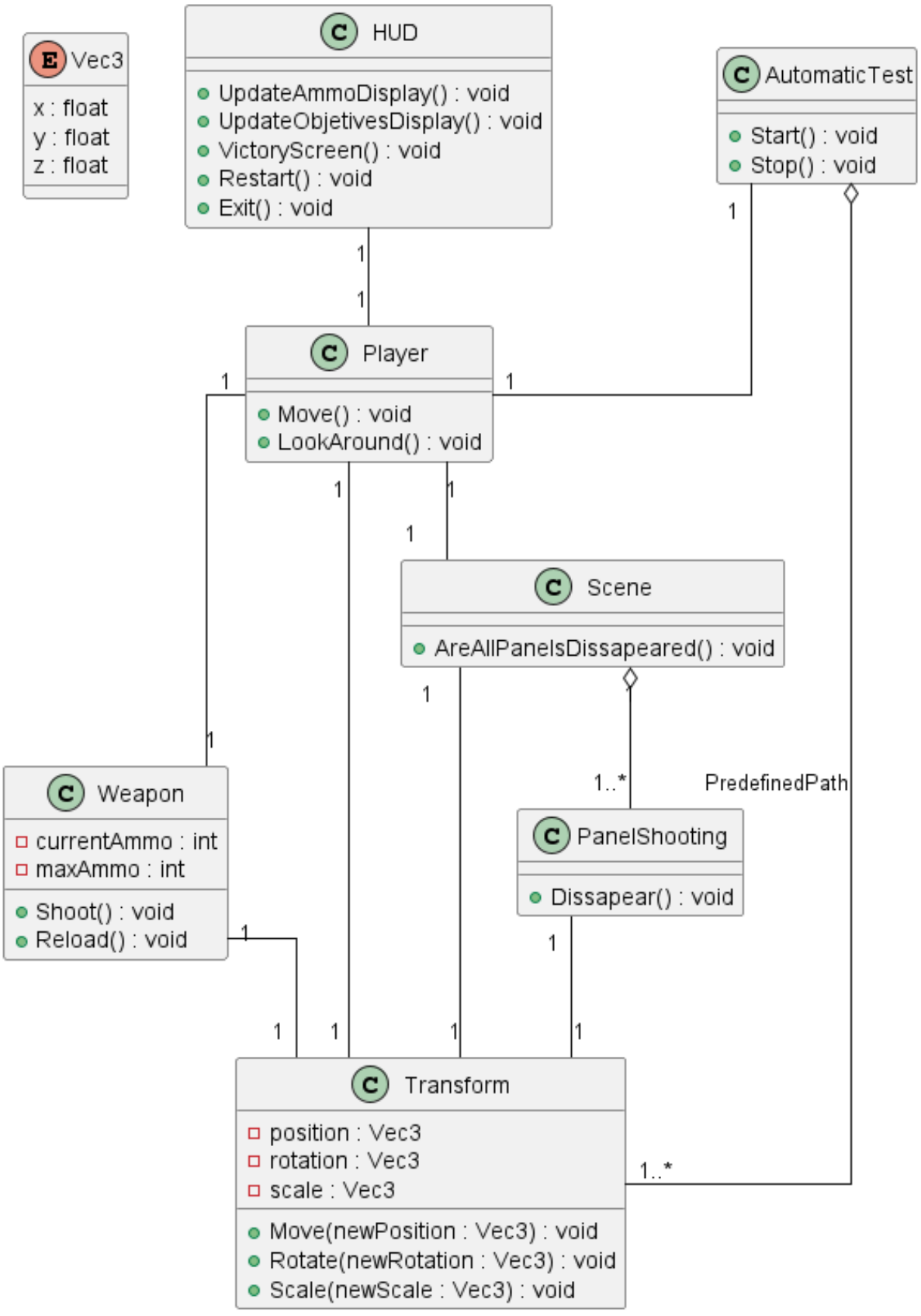

Figure 2.2: Domain Model in analysis

This domain model illustrates the key entities within the system: the player (`Player`) with their weapon (`Weapon`) and position (`Transform`); the interface components (`HUD` and `PanelShooting`); the environment (`Scene`); and the automated testing system (`AutomaticTest`).

The relationships capture how these objects interact and are positioned within the 3D space using vectors (`Vec3`) and transformations.

In summary, this chapter has established the analytical framework of the project by defining the scope, functional and non-functional requirements, use cases, and an initial domain model. This foundation provides a clear understanding of the expected functionalities and system behaviours. The following chapter will focus on detailed design, covering sequence diagrams, a refined domain model, state machine diagrams, and the selection of technologies to be used in the development process.

Chapter 3

Design

This chapter addresses the design phase of the project, where the structure and behaviour of the system are explored in depth based on the previously defined analysis model. A redefinition of the domain model will be carried out, detailing the core classes, their responsibilities, and relationships. Additionally, sequence diagrams illustrating the flow of the fundamental use cases will be presented, providing a clear view of the interactions between system components.

Furthermore, the state machines governing weapon animation and the overall game state will be described, which are key elements for the proper functioning of the internal logic. Finally, the technologies selected for implementation will be outlined, with their selection justified according to the project's requirements and objectives.

3.1 Redefinition of the domain model

The domain model proposed will be refined based on the operation of most video game engines, in order to subsequently draw the sequence diagrams for the various use cases that have been identified. The refined domain model is as follows:

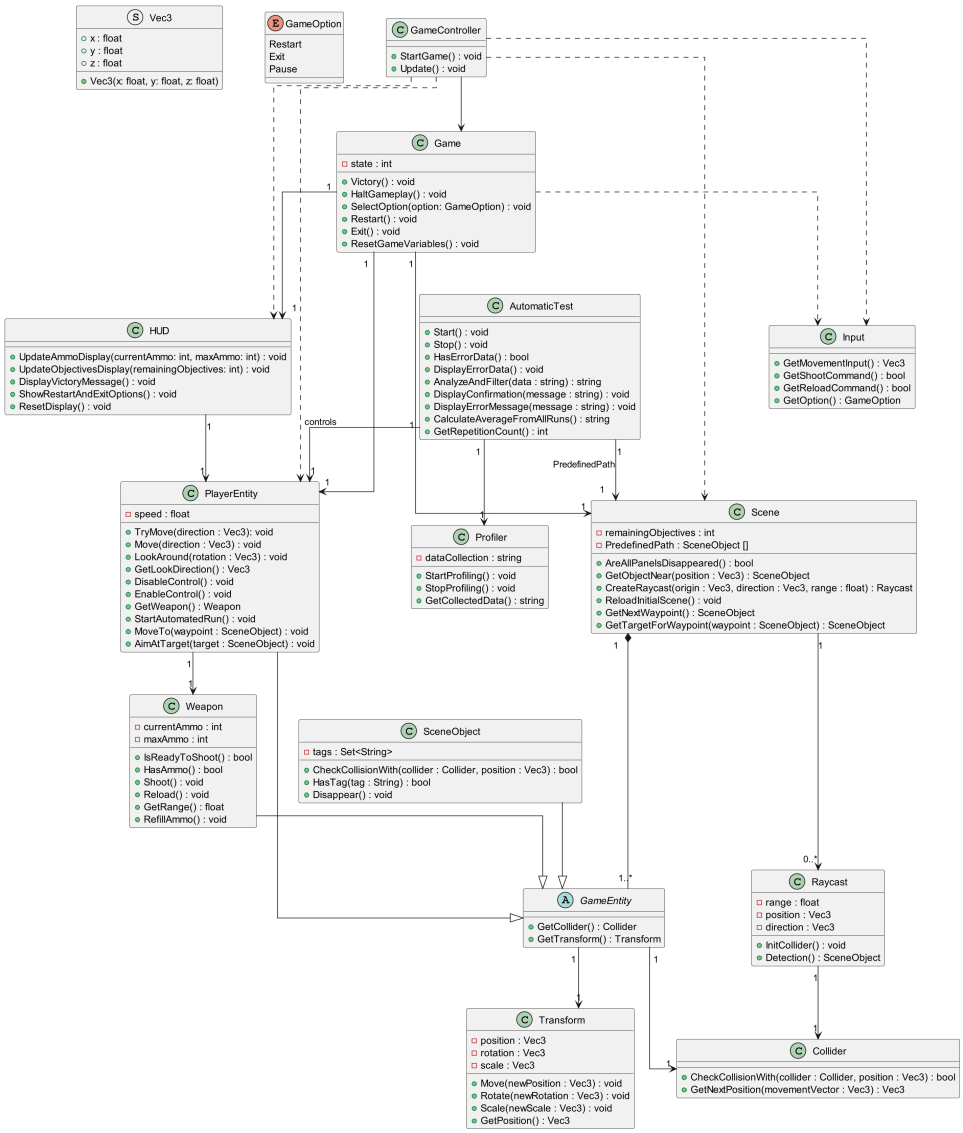

Figure 3.1: Domain model in design

Below is a description of the main classes and their relationships:

3.1.1 Core Classes and Responsibilities

- **Game**: manages the overall game state and logic, such as handling victory conditions, halting gameplay, restarting, and exiting the game. It holds references to key components like the `HUD`, player entity, and scene.
- **GameController**: orchestrates the game lifecycle by starting the game and updating the game state every frame. It interacts with the `Game`, `Input`, `PlayerEntity`, `Scene`, and `HUD` components.
- **HUD**: responsible for displaying the user interface elements, such as ammo count, objectives, victory messages, and menu options for restart or exit. It is closely linked to the player entity to reflect the current status.

- **PlayerEntity**: represents the player character with capabilities to move, look around, aim, shoot, and control the equipped weapon. It can also be controlled automatically during tests and can have control enabled or disabled by the game logic.
- **Weapon**: represents a weapon with ammunition management, shooting, reloading, and ammo refill functionalities. It is owned by the player entity.
- **Scene**: contains the game environment, including targets and predefined paths for automated tests. It provides methods to reload the scene, retrieve waypoints, check if all targets have disappeared, and perform raycasting.

3.1.2 Supporting Classes

- **Transform**: Defines the spatial properties of game entities such as position, rotation, and scale. It provides methods to move, rotate, and scale objects in 3D space. Every `GameEntity` owns a `Transform`.
- **Collider**: Handles collision detection and spatial queries. It can check for collisions with other colliders and compute potential positions based on movement vectors. Every `GameEntity` also owns a `Collider`.
- **SceneObject**: Represents interactive objects in the scene, such as targets or obstacles. They have tags to identify their type and can check for collisions. They can disappear when hit.
- **Raycast**: Facilitates raycasting operations for line-of-sight or shooting mechanics by detecting objects along a ray within a specified range. It uses a collider and spatial data for detection.

3.1.3 Additional Components

- **AutomaticTest**: Supports automated gameplay testing by controlling the player entity through predefined paths and shooting targets without user input.
- **Profiler**: Collects performance metrics during gameplay or automated tests. It allows starting and stopping profiling and provides the collected data for analysis.
- **Input**: Manages player input such as movement commands, shooting, reloading, and menu selections.

3.1.4 Class Relationships

- `GameEntity` is an abstract class that represents entities in the game world. It owns exactly one `Transform` and one `Collider`.
- `PlayerEntity`, `Weapon`, and `SceneObject` inherit from `GameEntity`, gaining spatial and collision properties.
- The `PlayerEntity` has a composition relationship with a `Weapon` — the player owns exactly one weapon.
- The `HUD` is associated with the `PlayerEntity` to reflect player status information such as ammo.
- `Scene` aggregates multiple `GameEntity` objects that represent scene elements including targets and waypoints.
- The `Raycast` class depends on `Collider` to perform hit detection. Furthermore, a `Scene` will be able to create the raycasts it deems appropriate.
- The `Game` class holds references to `HUD`, `PlayerEntity`, and `Scene`, orchestrating game logic across these components.
- The `GameController` acts as the main coordinator, interacting with the `Game` and other components like `Input`, `PlayerEntity`, `Scene`, and `HUD`.
- `AutomaticTest` controls the `PlayerEntity` for automated testing and communicates with the `Profiler` and `Scene` to execute tests and gather performance data.

3.2 Sequences diagram for use cases

3.2.1 UC01: Move

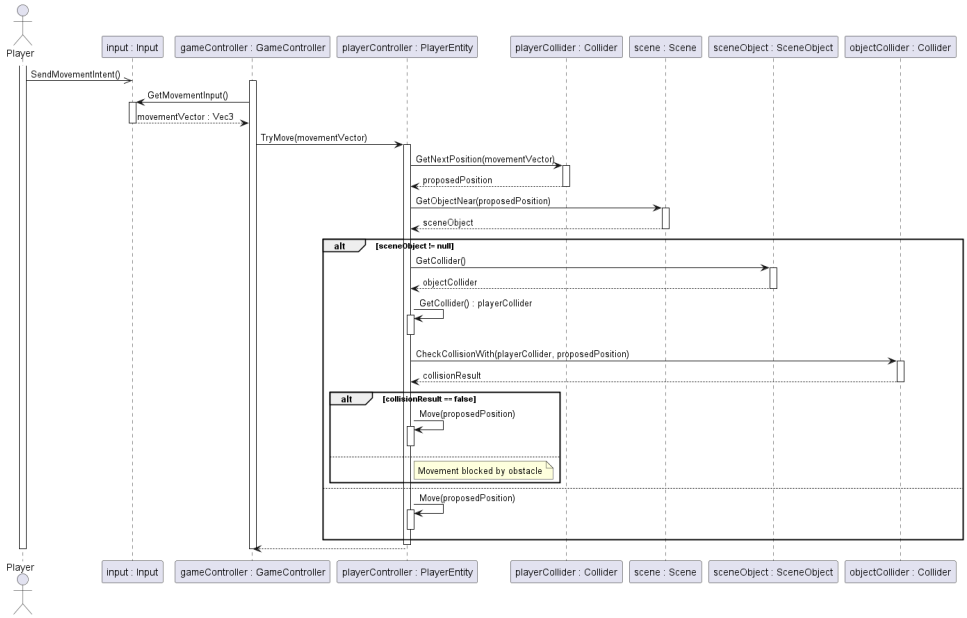

Figure 3.2: sequence diagram use case 01

1. The *Player* actor executes the movement command. The input is captured asynchronously by the `Input` component within the game cycle.
2. The `GameController` requests the movement input vector from the `Input` system by invoking `GetMovementInput()`.
3. The `Input` system returns the movement vector of type `Vec3` to the `GameController`.
4. The `GameController` instructs the `PlayerEntity` to attempt movement using `TryMove(movementVector)`.
5. The `PlayerEntity` queries its `Collider` to compute the next proposed position by calling `GetNextPosition (movementVector)`.
6. The `Collider` returns the proposed position to the `PlayerEntity`.
7. The `PlayerEntity` queries the `Scene` for any nearby objects at the proposed position using `GetObjectNear(proposedPosition)`.
8. The `Scene` responds with a nearby `SceneObject`, if any exists, or `null` otherwise.
9. If a `SceneObject` is detected:
   (a) The `PlayerEntity` obtains the `Collider` of the nearby object by calling `GetCollider()`.
   (b) The `PlayerEntity` also retrieves its own `Collider` for collision checking.
   (c) The `PlayerEntity` requests collision detection by invoking `CheckCollisionWith(playerCollider, proposedPosition)` on the object's `Collider`.
   (d) If no collision is detected (`false`), the `PlayerEntity` moves to the proposed position by calling `Move(proposedPosition)`.
   (e) If a collision is detected (`true`), the movement is blocked, and the `PlayerEntity` maintains its current position.
10. If no nearby object exists, the `PlayerEntity` moves directly to the proposed position.
11. The control flow returns from the `PlayerEntity` to the `GameController`, completing the movement action.

3.2.2 UC02: Shoot

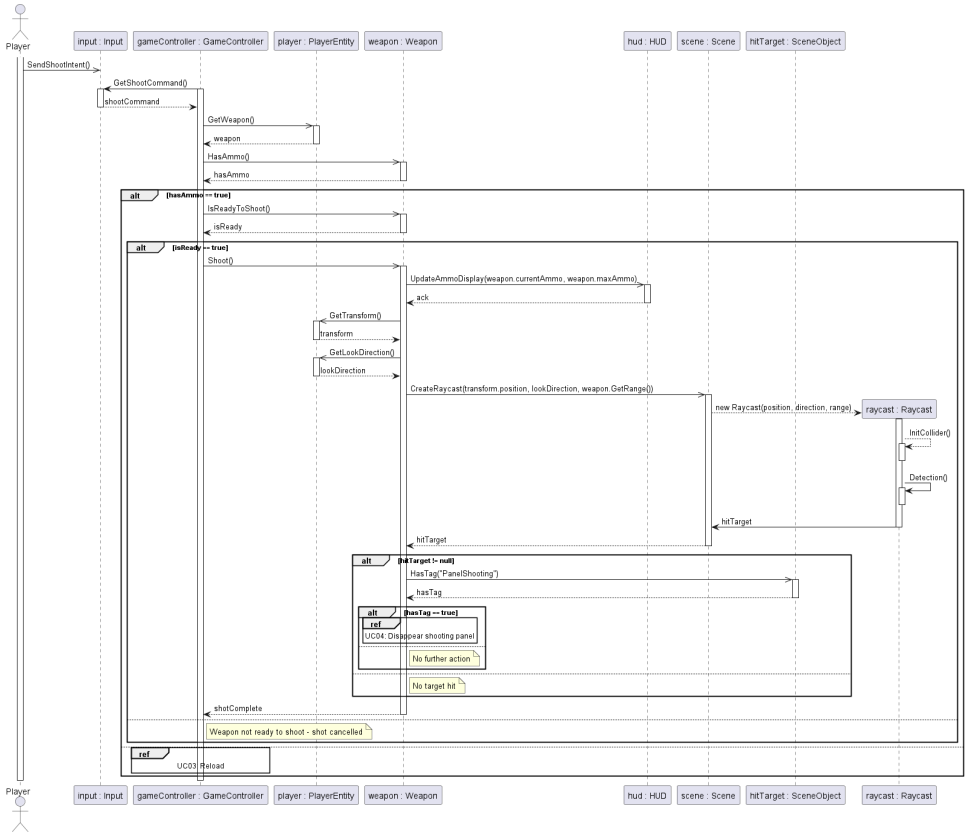

Figure 3.3: sequence diagram use case 02

1. The actor *Player* executes the shoot command. The input is captured asynchronously by the `Input` component.
2. The `GameController` requests the shoot command status from `Input`.
3. If a shoot command is detected, the `GameController` queries the `player` for the currently equipped `weapon`.
4. The `GameController` asks the `weapon` whether it has available ammunition.
5. If ammunition is available, the `GameController` verifies whether the `weapon` is ready to shoot (`IsReadyToShoot()`).
6. If the `weapon` is ready, the shooting action is executed:
   - The `HUD` is updated to reflect the current ammunition count.
   - The `weapon` requests the `player`'s transform (position and rotation) and look direction.
   - Using this information, the `weapon` requests the `scene` to create a `raycast` with the appropriate origin, direction, and range.
   - The `raycast` performs collision detection in the scene and returns the object hit, if any.
7. If the impacted object has the tag `PanelShooting`, the use case **UC04: Disappear shooting panel** is triggered.
8. If the `weapon` is not ready to shoot or ammunition is exhausted, the shooting action is not executed.
9. In the event of ammunition exhaustion, the `HUD` is updated to display zero ammunition and the use case **UC03: Reload** is triggered.

3.2.3 UC03: Reload

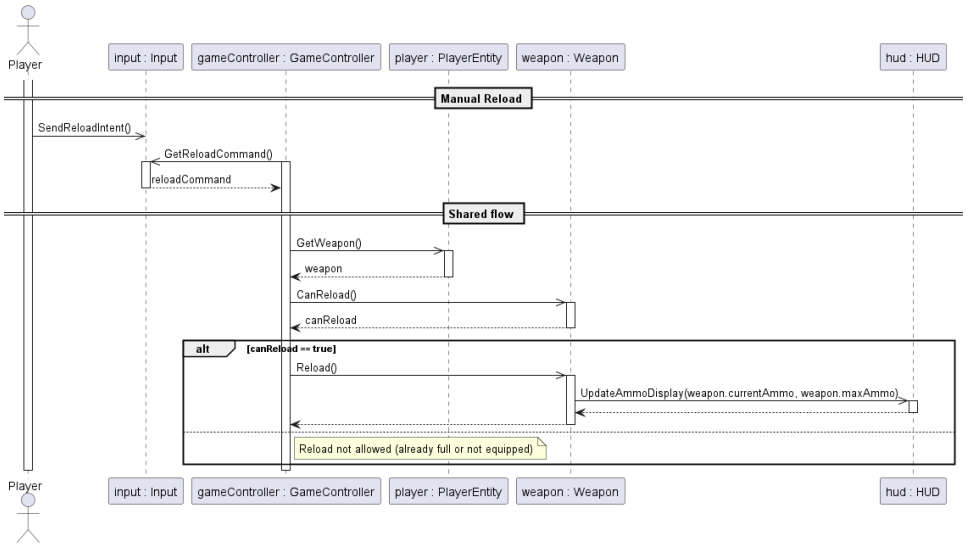

Figure 3.4: sequence diagram use case 03

- **Manual reload:**
  1. The actor *Player* executes the reload command. The input is captured and stored by the `Input` component.
  2. The `GameController` queries the `Input` for the reload command.
  3. Upon receiving the command, the `GameController` retrieves the currently equipped `Weapon` from the `PlayerEntity`.
  4. The `GameController` checks whether the weapon can be reloaded (`CanReload()`).
  5. If allowed, the `GameController` instructs the `Weapon` to reload.
  6. After reloading, the `Weapon` sends an update to the `HUD` to refresh the ammunition display.
  7. The `HUD` reflects the updated ammunition values in the graphical interface.
- **Automatic reload (triggered from UC02 - when attempting to shoot without ammo):**
  1. During UC02, if the player tries to shoot with no remaining ammo, the `GameController` triggers UC03 internally.
  2. The `GameController` retrieves the equipped `Weapon` and verifies reload availability.
  3. If allowed, it sends the `Reload()` command to the `Weapon`.
  4. The `Weapon` refills its ammunition.
  5. The `HUD` is updated to reflect the new ammunition values.

3.2.4 UC04: Disappear shooting panel

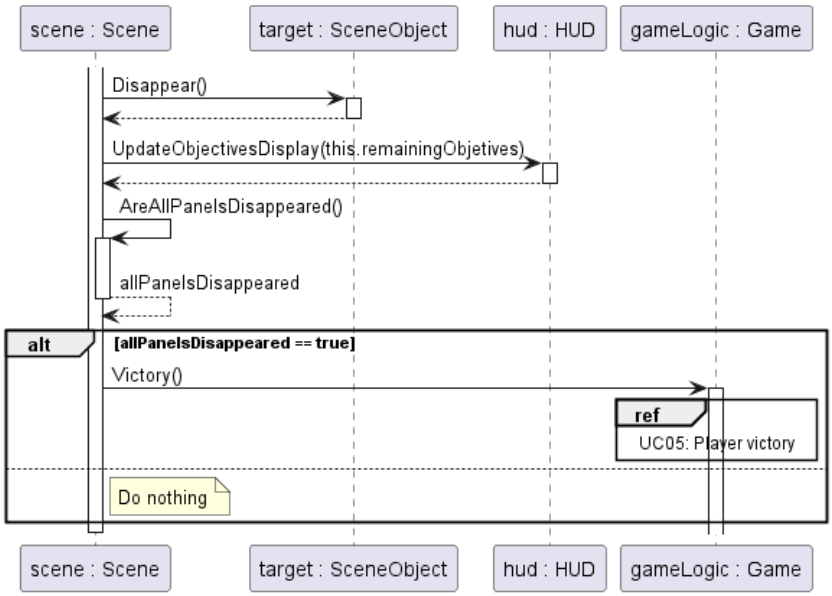

Figure 3.5: sequence diagram use case 04

1. The `scene` calls the method `Disappear()` on the `target` retrieved in UC02, hiding the panel from the game.
2. Then, the `scene` updates the `HUD` by invoking `UpdateObjectivesDisplay(this.remainingObjectives)` to reflect the new number of remaining objectives.
3. The `scene` checks if all panels have disappeared by calling `AreAllPanelsDisappeared()`.
4. If all panels have disappeared, the `scene` sends the message `Victory()` to `gameLogic` and the use case **UC05: Player victory** is triggered.
5. Otherwise, no further action is taken.

3.2.5 UC05: Player victory and UC07: Exit

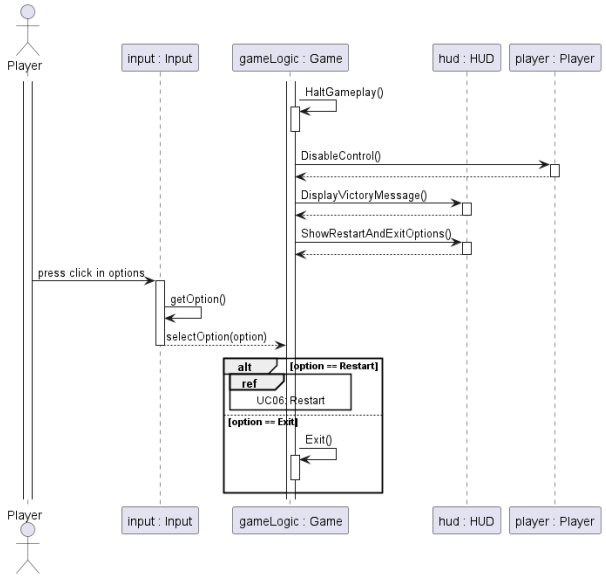

Figure 3.6: sequence diagram use case 05 and 07

1. `gameLogic` performs a self-call with the message `HaltGameplay()`, indicating that it internally halts gameplay.
2. `gameLogic` sends a message to `playerEntity` with `DisableControl()`, disabling player control during the victory sequence.
3. `gameLogic` requests the `HUD` to display the victory message via the message `DisplayVictoryMessage()`.
4. `gameLogic` requests the `HUD` to show the restart or exit options via the message `ShowRestartAndExitOptions()`.
5. The player selects one of the options displayed on the interface (“Restart” or “Exit”).
6. This interaction is captured by the `input` component, which executes `getOption()` to identify the selected option.
7. The selected option is evaluated:
   - If the option is **Restart**, the referenced use case `UC06: Restart` is triggered.
   - If the option is **Exit**, `gameLogic` sends the `Exit()` message to gracefully shut down the application.

3.2.6 UC06: Restart

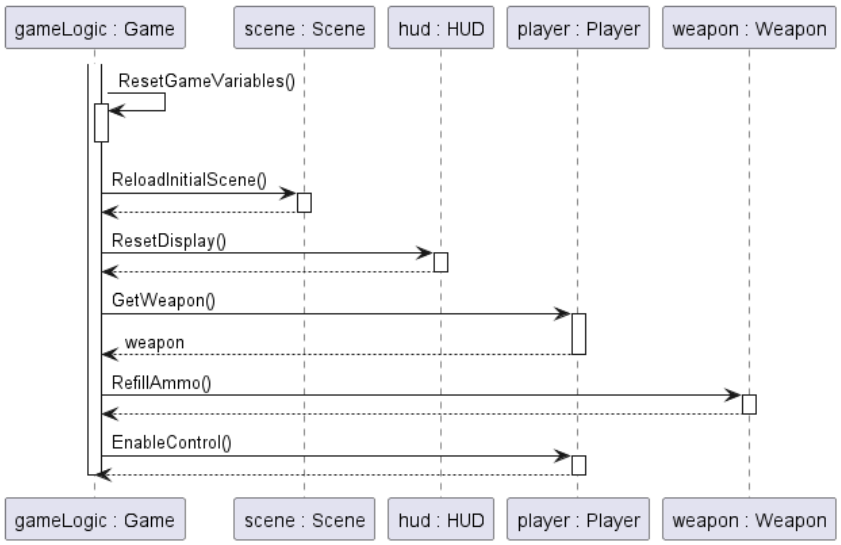

Figure 3.7: sequence diagram use case 06

1. `gameLogic` invokes `ResetGameVariables()` to reset all necessary game variables such as objectives, ammunition, etc.
2. `gameLogic` requests `scene` to reload the initial scene by sending the message `ReloadInitialScene()`.
3. `gameLogic` requests `HUD` to reset the visual interface via `ResetDisplay()`.
4. `gameLogic` invokes `RefillAmmo()` on the `weapon` object to restore ammunition to its maximum value.
5. `gameLogic` re-enables player control by calling `EnableControl()` on the `player` object.

3.2.7 UC08: Run automated test

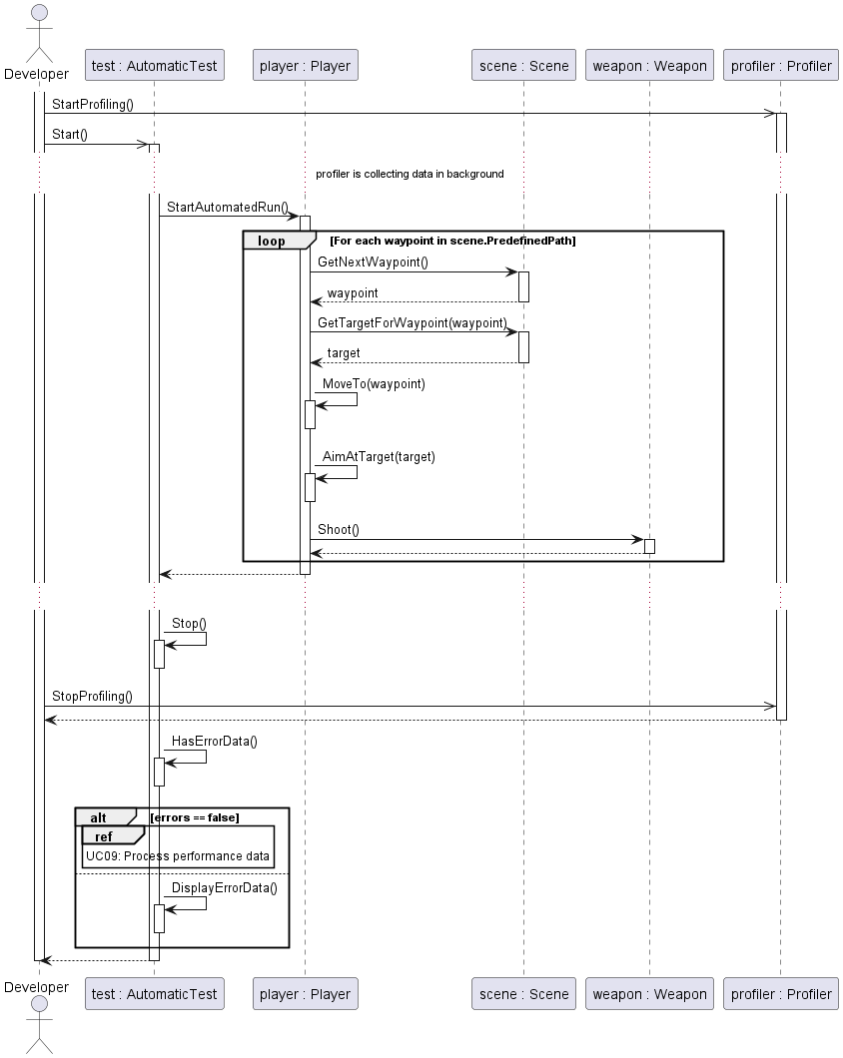

Figure 3.8: sequence diagram use case 08

1. The `Developer` starts the data collection by sending the message `StartProfiling()` to the `profiler` object of type `profiler`. From this moment, the `profiler` remains active collecting data in the background while the test runs.
2. The `Developer` starts the automated test by sending the message `Start()` to the `test` object of type `AutomaticTest`.
3. Next, `test` instructs the `player` to begin the automated run using `StartAutomatedRun()`, enabling player control in test mode.
4. For each `waypoint` in the `scene.PredefinedPath` list, the following sequence is executed:
   - `player` requests the next `waypoint` from `scene` via `GetNextWaypoint()`, and the scene responds by sending a `SceneObject`.
   - `player` requests from `scene` the `target` associated with that `waypoint` via `GetTargetForWaypoint(waypoint)`, and the scene replies with another `SceneObject`.
   - `player` moves to the `waypoint` by executing `MoveTo(waypoint)`.
   - Then, `player` aims at the `target` using `AimAtTarget(target)`.
   - Finally, `player` shoots the target by sending `Shoot()` to the `weapon` object.
5. Once all `waypoints` have been completed, `player` notifies `test` that the route is finished.
6. `test` ends the test by executing `Stop()`.
7. Afterwards, `Developer` requests the `profiler` to stop data collection.
8. `test` evaluates whether any errors were detected using `HasErrorData()`:
   - If no errors are detected, use case **UC09: Process performance data** is triggered.

- Otherwise, `test` executes `DisplayErrorData()` to show the detected errors.

3.2.8 UC09: Process performance data

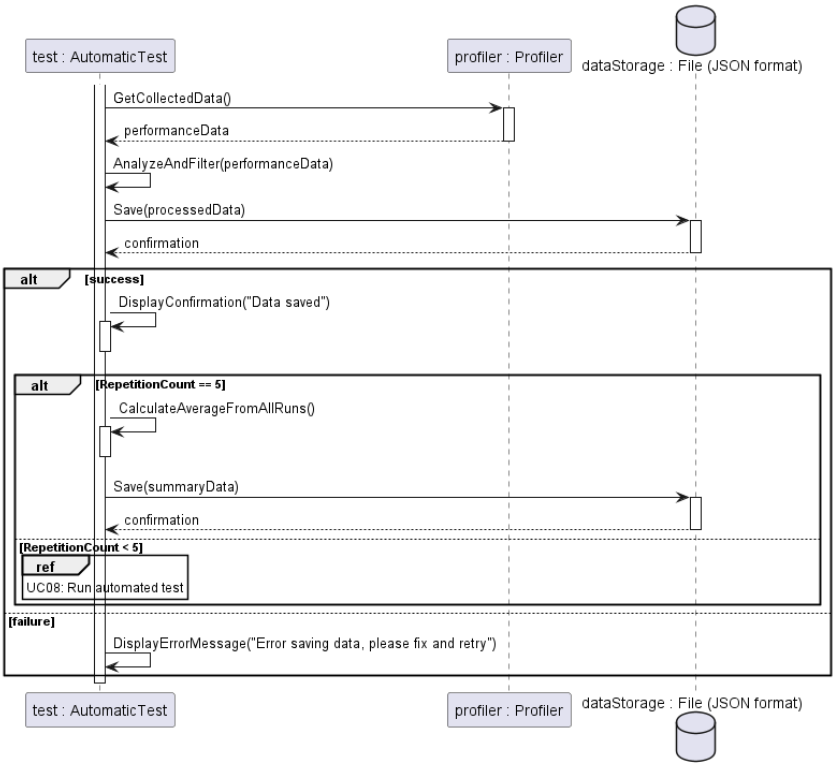

Figure 3.9: sequence diagram use case 09

1. The `AutomaticTest` component initiates the processing of performance data once the automated test has completed.
2. The test requests the collected data from the `Profiler` by sending the message `GetCollectedData()`. The profiler responds with the recorded performance data.
3. The test analyses and filters the received data to prepare it for storage.
4. The test attempts to save the processed data into persistent storage, represented as a JSON file, by sending `Save(processedData)`. The storage confirms successful saving.
5. Upon successful saving:
   - The test displays a confirmation message indicating that the data has been saved successfully.
   - The test checks whether the maximum number of test iterations (Five iterations have been deemed sufficient to obtain a stable output; however, if any of these five iterations present outlier values, additional repetitions may be performed to better filter out anomalous data caused by background processes executed by the operating system.) has been reached.
   - If the maximum iterations have been reached:
     - The test calculates average values from all executed runs.
     - It saves this summary data into persistent storage.
   - If the maximum iterations have not been reached:
     - The test triggers the use case **UC08: Run automated test** to perform another iteration.
6. If an error occurs while saving or reading data:
   - The test displays an error message prompting the user to resolve the issue and retry.
7. The process concludes and the `AutomaticTest` component deactivates.

3.3 Weapon animation machine state

Weapon animations will be implemented to enhance the sense of movement and observe how this impacts performance. To achieve this, a state machine diagram will be created, which is shown as follows:

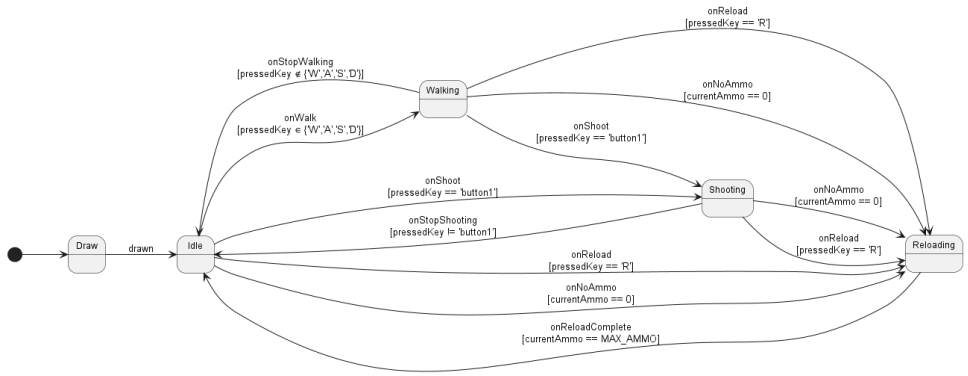

Figure 3.10: Weapon animation state machine diagram for Player

The state machine consists of the following states:

- **Draw**: the weapon is being drawn.
- **Idle**: the weapon is stationary and ready.
- **Walking**: the player is moving while holding the weapon.
- **Shooting**: the weapon is firing.
- **Reloading**: the weapon is being reloaded.

The transitions between states are defined as follows:

- **Draw** → **Idle**: occurs automatically once the weapon is fully drawn.
- **Idle** ↔ **Walking**: triggered by pressing or releasing any movement key (W, A, S, or D).
- **Idle** / **Walking** → **Shooting**: occurs when the left mouse button is pressed.
- **Shooting** → **Idle**: occurs when the left mouse button is released.
- **Idle**, **Walking**, or **Shooting** → **Reloading**: occurs when the `R` key is pressed or automatically when `currentAmmo == 0`.
- **Reloading** → **Idle**: occurs once `currentAmmo == MAX_AMMO`.

The state machine will also be implemented in the NPC for the automated performance test, albeit with a slight modification. The diagram in question is as follows:

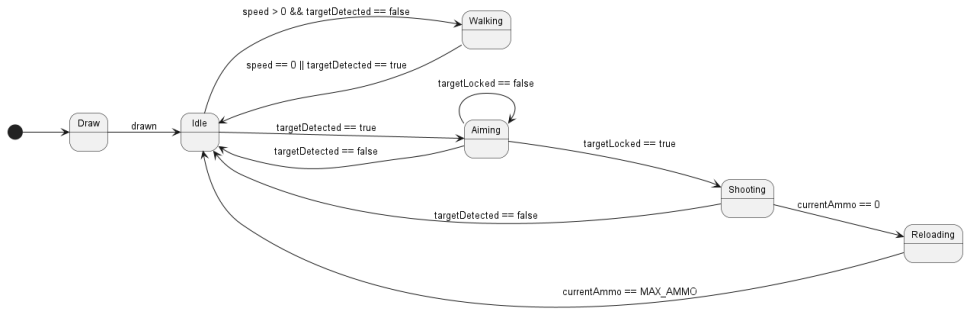

Figure 3.11: Weapon animation state machine diagram for NPC

The NPC weapon state machine consists of the following states:

- **Draw**: the weapon is being drawn.
- **Idle**: the NPC is stationary, deciding whether to move or engage a target.
- **Walking**: the NPC is moving towards the next point, with no target detected.
- **Aiming**: the NPC has detected a target and is aiming at it.
- **Shooting**: the NPC is firing at the target.
- **Reloading**: the weapon is being reloaded.

The transitions between states are defined as follows:

- **Draw** → **Idle**: occurs automatically once the weapon is fully drawn.
- **Idle** → **Walking**: occurs when the NPC's speed is greater than zero and no target is detected.
- **Walking** → **Idle**: occurs when the NPC's speed is zero or a target is detected.
- **Idle** → **Aiming**: occurs when a target is detected.
- **Aiming** → **Shooting**: occurs when the NPC locks on the target.
- **Aiming** → **Aiming**: remains aiming while the NPC has not locked the target yet.
- **Aiming** → **Idle**: occurs if the target is lost.
- **Shooting** → **Idle**: occurs when the NPC has finished with the target (e.g., eliminated).
- **Shooting** → **Reloading**: occurs when the NPC runs out of ammo.
- **Reloading** → **Idle**: occurs once ammo is fully replenished.

3.4 Game state machine

Lastly, but importantly, the game state transitions are represented using a classic state diagram. This diagram illustrates when victory, restart, and exit conditions are triggered. In this case, it should be clarified that there are no plans to implement a pause menu with resume, restart, or exit options, as the prototype focuses solely on the victory condition. The diagram is as follows:

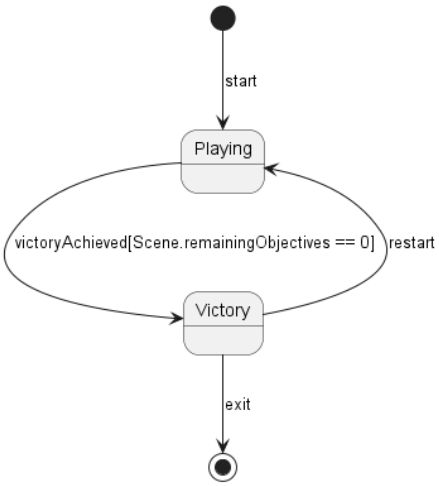

Figure 3.12: Game state machine diagram

The game state machine consists of the following states:

- **Playing**: the game is actively running.
- **Victory**: the player has achieved victory; end-of-game screen.

The transitions between states are defined as follows:

- **Initial** → **Playing**: occurs when `start` is triggered.
- **Playing** → **Victory**: occurs when all objectives have been completed, specifically when `Scene.PanelShootings[] == 0`.
- **Victory** → **Playing**: occurs when `restart` is triggered.
- **Victory** → **Final**: occurs when `exit` is triggered.

3.5 Technologies used

The technologies chosen for the development of the project are as follows:

- **Unity 6**: release 6000.0.40f1
- **Unreal engine 4**: release 4.27.2
- **Godot**: release 4.4.1
- **Visual Studio Community**: An IDE used with Unity that allows code to be written and compiled instantly for direct use within Unity, without the need for additional preliminary steps.
- **Visual Studio Code**: is a lightweight, open-source code editor that is highly customisable through extensions.
- **Plastic SCM**: is a cross-platform, commercially distributed version control tool, available for Microsoft Windows, Linux, and other operating systems. The tool includes a command-line interface, native graphical user interfaces, a diff and merge tool, and integration with various IDEs. It provides a complete version control solution that is not based on Git, although it can interact with it.[23]

The use of the engines in their respective versions is due to the development environment in which the

project is being carried out. For example, Unreal Engine 5 was found to have compatibility issues with integrated GPUs, such as those included in the CPU.

3.6 Research about profilers

3.6.1 Objective

The objective of this research is a methodological verification to ensure that the conclusions drawn from the conducted tests are reliable.

Important aspects must be taken into account, among them are:

- **Metrics**: what are the most interesting metrics? Should all of them be used?
- **Accessibility**: which profiler is the most suitable? depending on the metrics, which one is the most interesting to us?
- **Cross-browsing**: is the profiler valid for all browsers?

3.6.2 Methodology

The methodology to follow will be to determine which metrics are fundamental, and from there, decide which profiler is necessary, in our case, when it comes to executing mini-games made with Unity, Unreal or Godot, The methodology will determine which metrics are:

- **FPS (frames per second)**: measures the frequency at which the game loop is executed; the higher the value, the better the performance.
- **RAM**: the memory occupied by the mini-game in execution, calculating how much the scene, elements, textures, lighting, sound, etc. occupy.
- **Initial Load Time**: the time it takes for the game to load from the moment the user accesses it until they can start playing.
- **Render Time**:the time required to render a 3D scene, especially when loading new scenes or animations. While the FPS is measuring a whole cycle for the game loop, the Render Time only considers the time required for visual calculations
- **GPU**: there are different metrics that encompass the measurement of GPU power, duration, frequency, etc.
- **Input lag**: the time it takes for the game to respond to the player's input (for example, pressing a key or moving the mouse).

3.6.3 Profilers most used

Next, let's look at the most commonly used profilers and examine their advantages and disadvantages. These are:

- **Chrome DevTools**: a tool developed by Google, integrated within the browser, and it only works in the Google Chrome browser (Chromium)
- **FireFox DevTools**: a tool for the Firefox browser, it is integrated and only works in that browser.
- **LightHouse**: open-source tool created by Google for web browsers based on Chromium.
- **playwright**: a test automation tool for web applications that works with most web browsers.

3.6.4 Comparative table

For the investigation, a comparison will be made to determine which profiler is of most interest, depending on the metrics and whether it is compatible with most browsers. This is shown in the following table:

Profiler	**FPS**	**RAM**	**Initial Load Time**	**Render Time**	**GPU Usage**	**Input Lag**	**Cross-Browser**
Chrome DevTools	yes	yes	yes	yes	yes	yes	Only Google Chrome or extensions
Firefox DevTools	yes	yes	yes	yes	yes	yes	Only Firefox or extensions
Lighthouse	no	no	indirectly (FCP, LCP)	indirectly (FCP, LCP)	no	yes	Chromium browsers, integrated in DevTools
Playwright	no	no	indirectly (FCP, LCP)	indirectly (FCP, LCP)	no	yes	Yes, but not focused on profiling WebGL

FCP: First Contentful Paint; LCP: Largest Contentful Paint.

Table 3.1: Comparison of profiling tools and available metrics.

3.6.5 Conclusion

After reviewing the comparison table, it is concluded that the most suitable profilers are those from Google and Firefox, as all the metrics needed are provided without requiring additional calculations, and they are the most representative browsers. However, if a specific browser not based on Chromium were needed, the Playwright API would be used.

3.7 Importing GLTF files with Sketchfab

To save time on the work, a pre-modeled 3D environment [1] and weapon [2] with various details will be added to the FPS prototype to assess their potential impact on performance. The models will be sourced from Sketchfab.

Sketchfab (`https://sketchfab.com/feed`) is an online platform designed for uploading, sharing, discovering, and trading 3D models, VR, and AR content. It offers a powerful WebGL viewer, enabling users to interact with 3D assets directly within web browsers, whether on mobile, desktop, or virtual reality devices.

For the import, the decision has been made to choose GLTF, which is a standard file format for three-dimensional scenes and models. A GLTF file uses one of two possible file extensions: `.gltf` (JSON/ASCII) or `.glb` (binary). This format has been chosen as it is compatible with the majority of the engines and libraries selected. Nevertheless, some adjustments will be necessary in certain engines, such as Unreal Engine, since in version 4.27 this technology was still in beta, in this case, the FBX format will be used

To address this issue, since the environment does not support FBX import, Blender –an open-source 3D creation suite– will be used to convert the GLTF format to FBX.

The steps to perform the conversion are as follows:

1. Open Blender.
2. Go to: `File > Import > glTF 2.0 (.glb/.gltf)`.
3. Select your file and click **Import glTF 2.0**.
4. Verify that meshes, materials, textures, and animations (if any) are correctly imported.
5. Go to: `File > Export > FBX (.fbx)`.
6. Set the following important export settings:
   - **Path Mode**: set to “Copy”, and click the icon next to it to embed textures.
   - **Apply Transform**: enable this to bake transformations.
   - **Forward**: -Z Forward.
   - **Up**: Y Up (matches Unreal Engine’s coordinate system).
   - It should look something like this:

[1] `https://sketchfab.com/3d-models/lowpoly-fps-tdm-game-map-d41a19f699ea421a9aa32b407cb7537b`
[2] `https://sketchfab.com/3d-models/ak47-831519a097d84e079fd8bc4b15e5b57d`

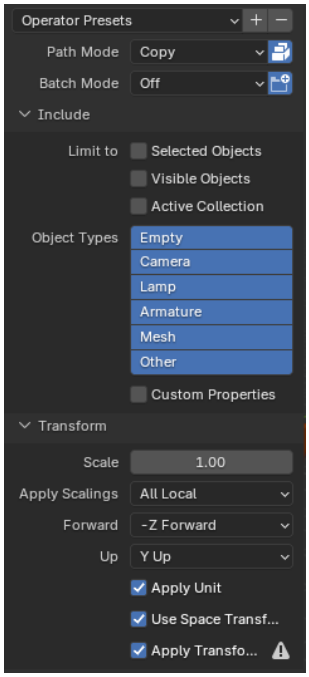

Figure 3.13: settings export to FBX in Blender

7. Rename the file and click **Export FBX**.

After this, to import the environment into Unreal:

1. Go to: `File > import into level`
2. Choose location for importing the scene content (e.g /content directory)
3. In the Static Meshes section, under the normal import method, select **Import Normals and Tangents**.
4. It should look something like this:

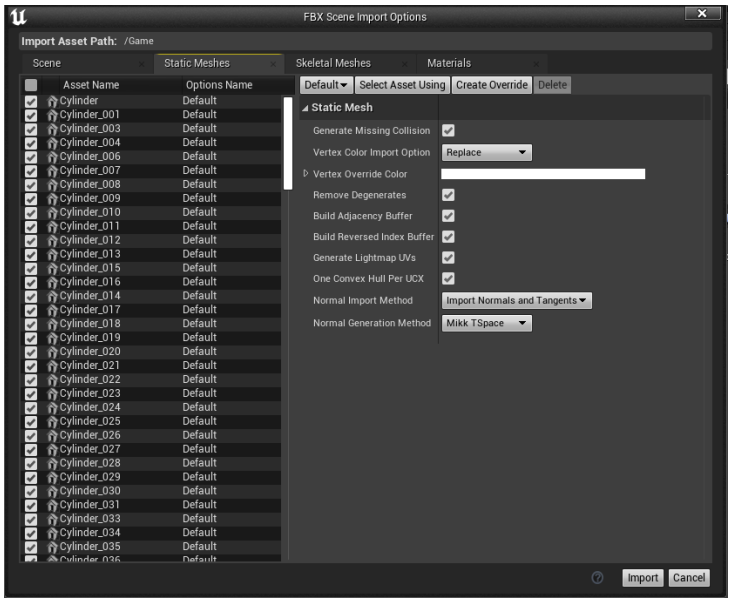

Figure 3.14: settings import from FBX in unreal in the Static Meshes section

5. If the 3D model has a skeleton or animations, make sure that **import Animations** is selected under the Skeletal Meshes section.
6. It should look something like this:

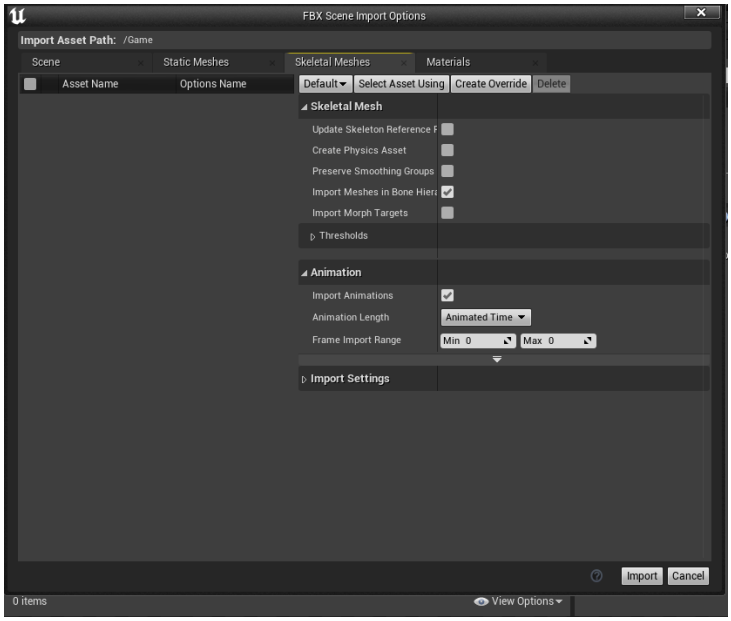

Figure 3.15: settings import from FBX in unreal in the Skeletal Meshes section

This chapter has developed a detailed system design, redefining the domain model and specifying sequence diagrams for the main use cases. Additionally, state machines controlling weapon animation and the overall game state have been defined, and the most suitable technologies for development have been selected.

Building on this solid design foundation, the next chapter will proceed to the implementation phase, starting with the development of the player-controlled manual mode prototype.

Chapter 4

Implementation (manual mode)

This chapter presents the implementation of the FPS game prototype in manual mode, where the player directly controls the character. The development process across three different engines — Unity, Unreal Engine, and Godot — is detailed. For each engine, the environment setup, collision system implementation, and relevant code analysis are provided. This comparison aims to highlight the particularities and differences between each platform.

4.1 Prototipe development

The first step is the minimum viable prototype defined for conducting performance tests. It has already been mentioned that this is a simple FPS. The technical decisions made were:

1. The player will move using the W, A, S, and D keys, with W moving forward, A moving backward, S moving right, and D moving left.
2. The player fires by pressing or holding down the left mouse button.
3. The player reloads the weapon using the R key.
4. The weapon will have animations to indicate to the player the action being performed at any given moment, as well as to assess whether it impacts performance.
5. To determine whether a shot has been fired, a gunshot sound file will be played upon execution of the action.
6. The player's HUD will consist of the following elements:
   - **Crosshair**: used for aiming at targets. By default, the crosshair is white, but when aiming at an enemy, it will change to red.
   - **AmmoDisplay**: displays the remaining ammunition in the weapon alongside the maximum capacity.
   - **ObjetiveDisplay**: displays the number of remaining targets.
   - **GameTimer**: displays the elapsed game time.
7. When all targets are defeated, the HUD will update to display a victory message, the time elapsed until victory, and two menu options: *Restart* and *exit*.

4.2 General Structure in Game Engine

This section provides an overview of the structure and interaction between the technologies used to implement the FPS game prototype in manual mode. The development is carried out across three main game engines: Unity, Unreal Engine, and Godot. Each engine serves as a foundational platform supporting game logic, user input management, collision systems, and the visual interface.

Although each engine possesses its own features and particularities, they all share a common structure

that facilitates comparison and evaluation of the prototype across different environments. This structure comprises the following fundamental components:

- **Player control:** User input is primarily managed through keyboard and mouse, allowing movement of the character and execution of actions such as shooting or reloading. The W, A, S, and D keys control movement, while the mouse is used for aiming and firing.
- **Collision and physics system:** Each engine implements mechanisms to detect interactions between the player, the environment, and targets via colliders. This system is crucial for determining when a shot hits an enemy and for managing the appropriate response within the game.
- **User interface (HUD):** A set of visual elements provides real-time information to the player, including the crosshair, remaining ammunition, number of targets left, and elapsed time. This interface is consistent across all engines, ensuring a comparable user experience.
- **Animations and sounds:** To enhance immersion and provide visual and auditory feedback, animations indicating weapon actions (shooting, reloading) and accompanying sounds are integrated, also contributing to the assessment of performance impact.

This modular and common structure ensures the prototype maintains consistent functionality across different platforms, enabling evaluation and comparison of both technical capabilities and implementation efficiency within each game engine.

The following sections will analyse in detail the environment setup, collision system implementation, and code review for each engine, building upon this shared general structure.

4.3 Unity

4.3.1 Introduction

Unity is the first game engine used to implement the FPS prototype in manual mode. Its role in this project is to serve as a reference base for developing and testing the core game functionalities: player control, collision system, shooting and reloading logic, the HUD interface, and the audiovisual elements required for a functional gameplay experience.

Unity was chosen due to its extensive documentation, ease of rapid prototyping, and flexibility in scripting with C#. Moreover, as the first engine employed, additional time was allocated to explore its potential and establish a solid foundation. This foundation could then be replicated across the other engines to ensure fair and consistent comparisons.

Development was carried out using Unity version **6000.0.40f1**. The following sections will detail the environment configuration, collision system implementation, and source code analysis, while maintaining the same functional prototype structure to ensure coherence across engines.

4.3.2 Environment

The environment is divided into clearly defined sections, which are as follows:

- **Project window**: this is the window where the project can be viewed and edited. It can be organised according to user preference, but by default, a Unity project is structured according to the types of assets it contains or the packages it uses.
  An asset is a representation of any item that can be used in a game or project. An asset may originate from a file created outside Unity, such as a 3D model, an audio file, an image, etc. There are also other types of assets that can be created within Unity itself, such as an Animator Controller or materials.
  A package can be considered analogous to a library in programming; it provides the remaining tools necessary for the engine to function, such as the graphics pipeline or the physics API. Additionally, it is possible to add external packages to the project.

- **Hierarchy window**:this is the window where all objects placed in the scene are displayed. These can be organised according to user preference. There are two types of objects:
  - **Scene**:these represent the game levels. This is where game objects for the respective level are placed. They can later be saved and added to the project structure in order to be loaded for editing or to transition to other scenes via code. They are represented by the white Unity logo.
  - **Game Object**: these are the game elements, including shapes, cameras, lights, and the HUD. However, a GameObject cannot perform any action by itself; it requires properties to be assigned before it can become a character, environment, or special effect. To provide a GameObject with the necessary properties, components must be added to it.
    A Component in Unity is a functional unit or module attached to a GameObject that imparts specific properties and behaviours. Examples of components include the Transform, which indicates the position, rotation, and scale of the GameObject within the scene; the Mesh Renderer, which defines the GameObject's shape; and the Rigidbody, which enables the GameObject to interact with Unity's physics system.
- **Scene view**: this is where direct editing on the scene takes place. Objects can be moved, rotated, and scaled without the need to access their Transform component directly. Objects can be dragged into the scene from the hierarchy window.
- **Game view**: this is where the game can be tested. It can be paused to allow for review in the Console window in case of errors or during debugging.
- **Inspector Window**: this is where the components of GameObjects can be viewed and interacted with directly, without the need for coding.

It should be noted that this is the default version in which the environment is presented, although it is possible to choose which tabs to display, as well as their arrangement, through the *Layout* menu located at the top right of the interface.

An overview of a Unity project is depicted in this image:

Figure 4.1: Unity environment

4.3.3 Collider implementation

In Unity, assigning *colliders* to objects can be done through various types, ranging from simple shapes like boxes (*Box Collider*) and spheres (*Sphere Collider*) to specialized components such as the *Character Controller*, which combines a capsule with spheres at its ends to facilitate character movement and collisions.

For objects with more complex geometries, the *Mesh Collider* is used, which approximates collision based on the object's actual mesh. However, this type of *collider* is significantly more performance-intensive, especially when characters frequently interact with these objects. For this reason, it is recommended to limit its use and prefer simple *colliders* whenever possible to optimise game smoothness and avoid FPS drops.

Correct selection and configuration of *colliders* is crucial to balance collision accuracy and performance—an essential aspect in developing an FPS prototype where responsiveness and stability take priority.

4.3.4 Code explanation

In Unity, programming is primarily conducted through C# scripts that inherit from the MonoBehaviour class. MonoBehaviour acts as the base class from which user-defined scripts derive to interact with the Unity game engine's lifecycle.

Within this class, methods can be defined to interact with the GameObjects in the scene and access their components. In this way, scripts can be created that reflect behaviours within the game without explicitly specifying what those objects must do. Therefore, each of the scripts will inherit from this class.

A view of the proposed class diagram would be as follows:

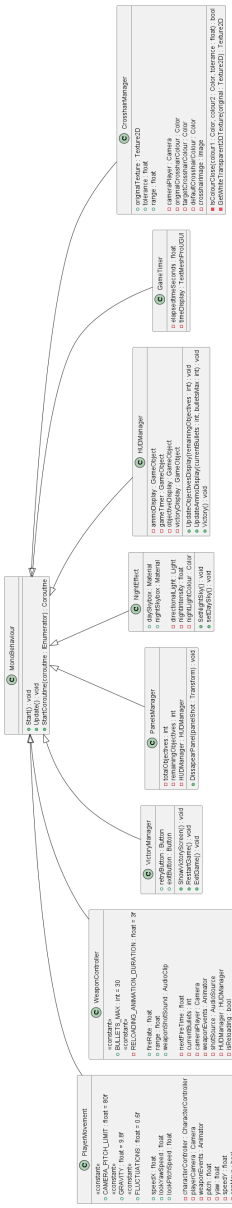

Figure 4.2: Scripts Unity class diagram

The methods of interest for the MonoBehaviour class have been the following:

- `Start()`: is executed once the script is enabled, just before the first `Update()`. This allows dependencies between objects to be assigned after they have been instantiated in the scene.[24]
- `Update()`: is executed every frame of the game after `Start()`.[25]
- `StartCoroutine(coroutine)`: Coroutines can be paused at any point using the `yield` statement, which causes execution to resume in a subsequent frame. They are ideal for implementing behaviours that span multiple frames. When `StartCoroutine(coroutine)` is called, control returns after the first `yield`.[26]

These functions will be overridden as needed during the construction of our scripts,

Another important aspect in these scripts is the behaviour of the `public` and `private` access modifiers. In addition to their standard role in object-oriented programming, declaring a variable as `public` allows it to be exposed in the Unity Inspector of the object where the script is attached. This makes it possible to test and adjust parameters during runtime without the need to constantly recompile scripts, facilitating rapid iteration and debugging.

the following behaviours will be defined:

PlayerMovement.cs

1. In `Start()`, the dependencies are obtained: the `CharacterController`, which represents the player in the FPS using its built-in physics system; the `playerCamera`, which provides the camera inside the player; and the `Animator`, which controls the weapon animation state machine in Unity. The mouse cursor is hidden and locked to the centre of the screen. Additionally, initiating movement of the player and the camera is allowed. The sequence diagram representation of this is as follows:

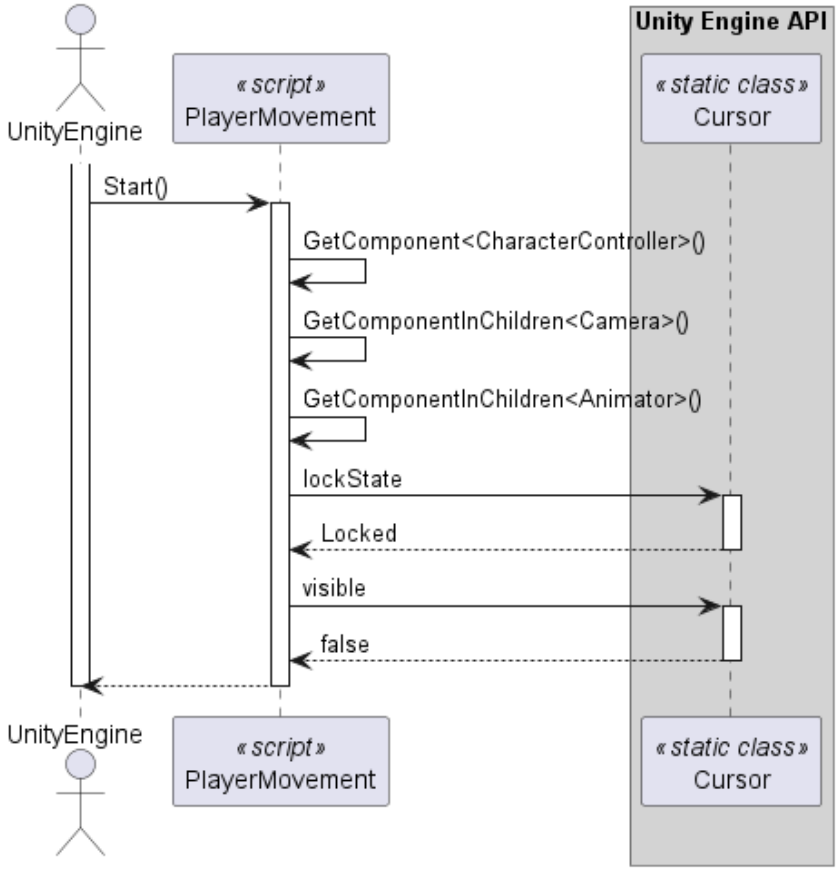

Figure 4.3: Sequence diagram of Start function of PlayerMovement script in Unity

2. In `Update()`, it is first checked whether movement is allowed, in case the player is in a victory state. If true, the following methods are then called
   - `MovePlayer()`: the first step is to retrieve the player's movement inputs. This is done by calling the static methods `GetAxis(string axisName)` from the Input class. By default in Unity, "Horizontal" corresponds to left/right movement (A/D or Left/Right arrow keys), and "Vertical" corresponds to forward/backward movement (W/S or Up/Down arrow keys).[27] After this, the velocity on the X and Z axes is calculated. Since the `GetAxis` function returns a value between -1 and 1, it is multiplied by the corresponding local axes of the character, represented by `transform.right` (X axis) and `transform.forward` (Z axis). These vectors are summed and then multiplied by `speedX`, which represents the movement intensity.
     For the Y component of the velocity, which represents gravity, it is checked whether the character is grounded using `characterController.isGrounded()`. If not, the movement is calculated as free fall, which is mathematically represented as follows:

$$v_{t+\Delta t} = v_t - g \cdot \Delta t$$

     where:
     - v_t: vertical velocity at the current time step.
     - $v_{t+\Delta t}$: vertical velocity after a time step Δt.
     - g: acceleration due to gravity (e.g. $9.81\,\mathrm{m/s^2}$).
     - Δt: time interval between frames (in Unity, `Time.deltaTime`).

Otherwise, the `speedY` will be zero.
After this, the character can be moved using `characterController.Move(Vector3 velocity)`. However, it should be noted that the fall represents an acceleration, so it must be multiplied again by `Time.deltaTime` to ensure realism.
Finally, the weapon animation must be performed while walking. The handling of animations in Unity will be explained later. For this case, the magnitude of the velocity is taken. In Unity, due to ground collisions and the setup, it was observed that the velocity never reached zero but fluctuated with an error of approximately 0.3–0.5. To address this, the constant `FLUCTUATIONS` is used to ignore this error. If the velocity is less than this fluctuation, it is considered zero. Afterwards, the animation state machine method for the weapon is called to check whether it is possible to transition from the "idle" state to the "walking" state based on the velocity magnitude(`weaponEvents.SetFloat(''movement'', magnitude)`).

The sequence diagram for `MovePlayer()` in PlayerMovement is as follows:

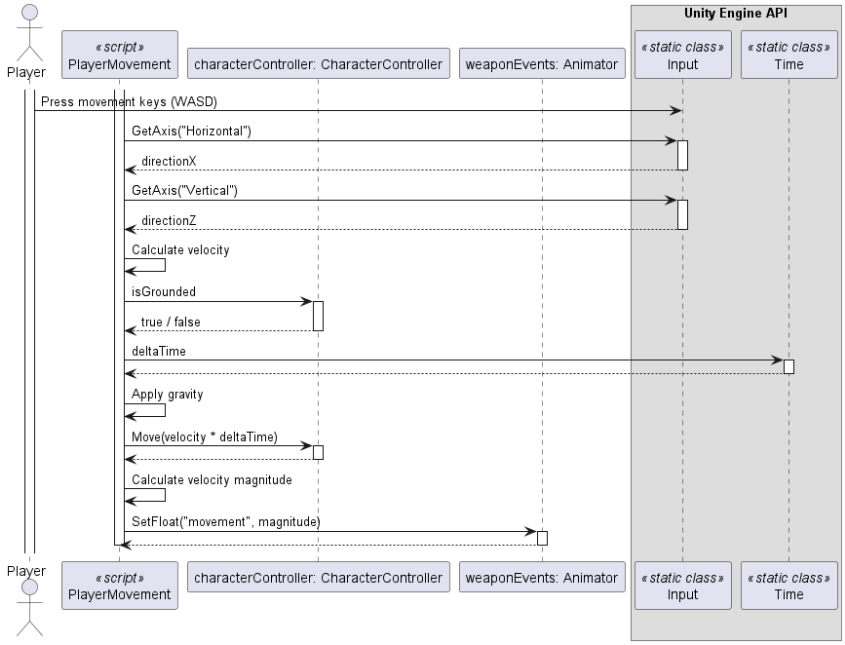

Figure 4.4: Sequence diagram of MovePlayer function of PlayerMovement script in Unity

- `LookAround()`: similarly to `MovePlayer()`, the input axes are obtained using `GetAxis(string axisName)`, in this case the X axis ("Mouse X") and Y axis ("Mouse Y") of the mouse.
  For the X axis, the value is simply multiplied by lookYawSpeed, which reflects the mouse sensitivity, and then subtracted, indicating that moving the mouse to the right causes the player to rotate to the right.
  For the Y axis, the same procedure is followed, except that to prevent the player from looking completely backwards or the camera from rotating excessively, `Mathf.Clamp(float value, float min, float max)` is used to limit the pitch value between -/+CAMERA_PITCH_LIMIT, which represents an angle of 160 degrees.
  To manage the character's orientation and rotations, Euler angles and quaternions are used. The full theory on Euler angles, the gimbal lock problem, and how quaternions solve this issue is developed in detail in Appendix A. Here, we focus on the practical application in the different engines and the code analysis.
  Horizontal rotation is applied to the player's body. The function `Quaternion.Euler(float x, float y, float z)` is used to convert an Euler angle into a quaternion, both for the rotation of the body and the rotation of the camera.
  In the case of the body rotation, it is applied in the negative direction because, when mapping the mouse movement to Unity's axis system, the rotation direction needs to be inverted to match the expected behavior. Specifically, moving the mouse to the right produces a positive input value, but a positive rotation around the Y-axis in Unity corresponds to turning left (counterclockwise). Therefore, to make the character rotate to the right as the mouse moves right, the rotation is applied in the negative direction (clockwise), and vice versa. Finaly, using `localRotation` on the camera is important so that it only tilts vertically without affecting the player's global rotation.

The sequence diagram for `LookAround()` in PlayerMovement is as follows:

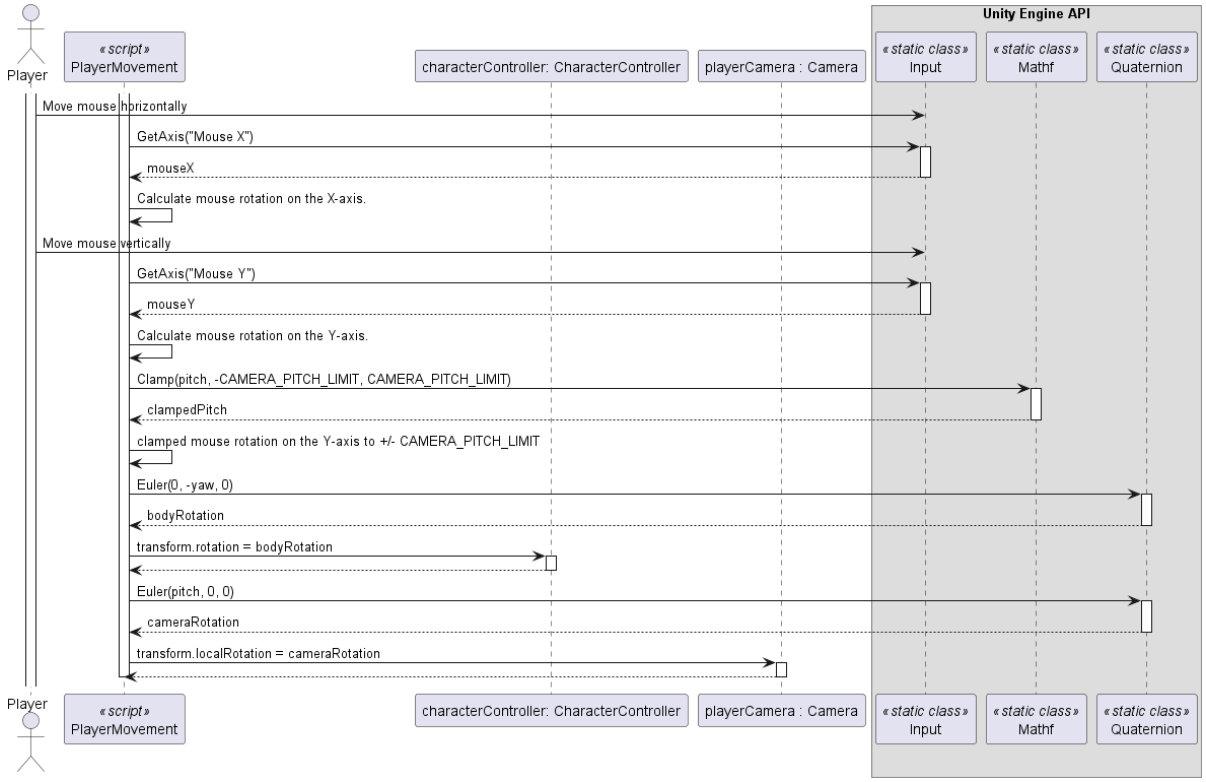

Figure 4.5: Sequence diagram of LookAround function of PlayerMovement script in Unity

The sequence diagram for `Update()` in PlayerMovement is as follows:

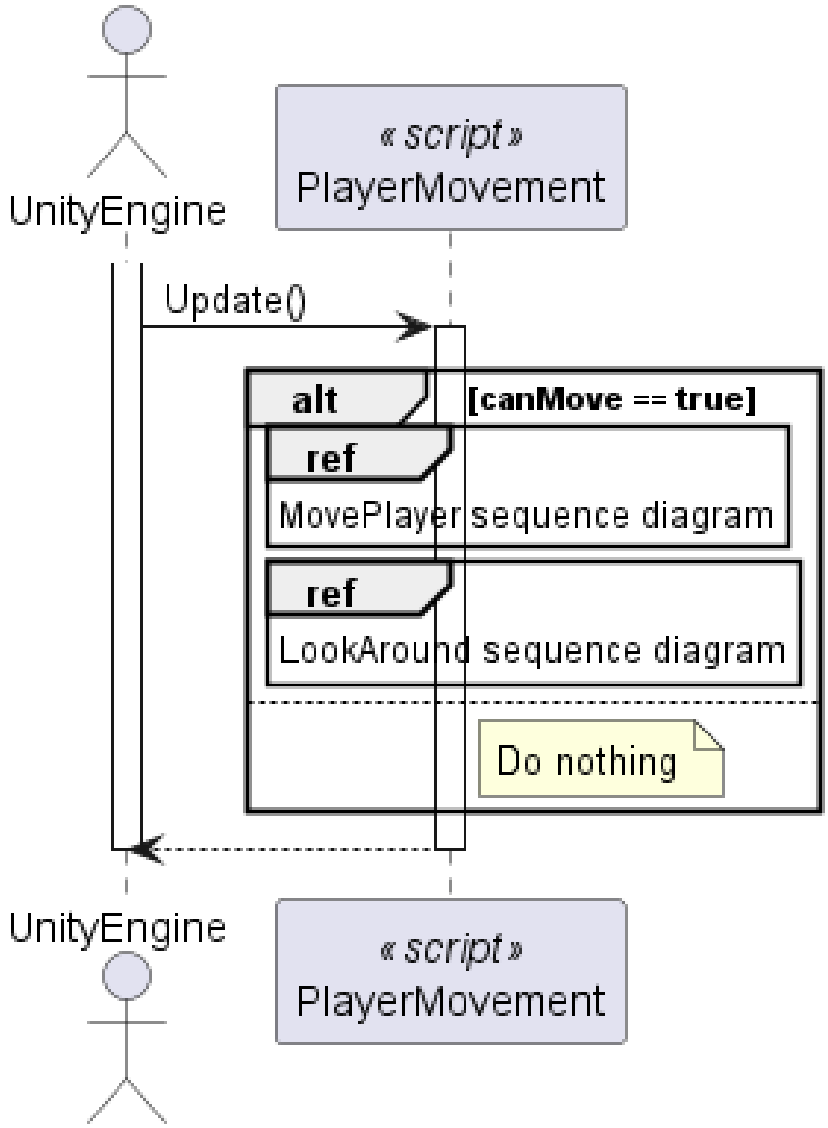

Figure 4.6: Sequence diagram of Update function of PlayerMovement script in Unity

3. Lastly, there is the `StopPlayerMovement()` function, which is used to stop both the character and camera movement. The reason and context for calling this function will be explained later.

The implemented code is as follows:

```
using System.Reflection;
using UnityEngine;
using UnityEngine.InputSystem.XR;
using UnityEngine.SceneManagement;

public class PlayerMovement : MonoBehaviour
{
    private const float CAMERA_PITCH_LIMIT = 80f;
    private const float GRAVITY = 9.8f;
    private const float FLUCTUACTIONS = 0.6f;   // Due to physics, idle time in Unity is not strictly 0

    [Header("Movement Settings")]
    public float speedXZ = 2f;

    [Header("Mouse Look Settings")]
    public float lookYawSpeed = 1.5f;     // Mouse rotation speed on the X-axis
    public float lookPitchSpeed = 1.5f;   // Mouse rotation speed on the Y-axis

    private CharacterController characterController;
    private Camera playerCamera;
    private Animator weaponEvents;         // Weapon animation event handler.

    private GameController gameController; // Check if the system is running in automatic test mode
    private float pitch = 0f;              // Mouse movement on the Y-axis
    private float yaw = 0f;                // Mouse movement on the X-axis
    private float speedY = 0f;             // Gravity
    private bool canMove;                  // Used When the player win, allows/blocks the player AND camera movements

    void Start()
    {
        gameController = GameObject.Find("Game Controller").GetComponent<GameController>();
        characterController = GetComponent<CharacterController>();
        ValidationUtils.CheckNotNull(gameController, "gameController script is missing.", this);
        ValidationUtils.CheckNotNull(characterController, "CharacterController component is missing.", this);
        // Skip this update cycle
        if (gameController.automaticMode)
        {
            // The CharacterController must be disabled during automatic test mode because having both a Collider
            // (through CharacterController) and a NavMeshAgent enabled at the same time is not supported during runtime
            characterController.enabled = false;
            this.enabled = false;
        }


        playerCamera = GetComponentInChildren<Camera>();
        weaponEvents = GetComponentInChildren<Animator>();

        ValidationUtils.CheckNotNull(playerCamera, "Camera component is missing.", this);
        ValidationUtils.CheckNotNull(weaponEvents, "Animator component for weapon events is missing.", this);

        Cursor.lockState = CursorLockMode.Locked; // Locks the cursor to the centre
        Cursor.visible = false;
        canMove = true;
    }

    void Update()
    {
        if (canMove)
        {
            MovePlayer();
            LookAround();
        }

    }

    private void MovePlayer()
    {
        // Unity's default input axes:
        // Input.GetAxis("Horizontal") corresponds to left/right movement (A/D or Left/Right arrow keys).
        // Input.GetAxis("Vertical") corresponds to forward/backward movement (W/S or Up/Down arrow keys).
        float directionX = Input.GetAxis("Horizontal");
        float directionZ = Input.GetAxis("Vertical");

        Vector3 velocity = (transform.right * directionX + transform.forward * directionZ) * speedXZ;

        // Apply gravity to the vertical speed
        if (!characterController.isGrounded)
```

```
78          {
79              speedY -= GRAVITY * Time.deltaTime;
80          }
81          else
82              speedY = 0f;
83
84          velocity.y = speedY;
85          // Mind the Y component will be multiplied by the deltaTime twice because the t^2
86          characterController.Move(velocity * Time.deltaTime);
87
88          // It is checked that the player moves:
89          // If the velocity's magnitude is greater than 0, the animation changes from idle to walking.
90          // else, the animation changes from walking to idle
91          float magnitude = velocity.magnitude;
92          if (magnitude < FLUCTUACTIONS) magnitude = 0; // To avoid issues with the weapon animations,
                 it is set to 0.
93          weaponEvents.SetFloat("movement", magnitude);
94      }
95
96      private void LookAround()
97      {
98          // Rotation on the Y-axis (left/right)
99          yaw -= Input.GetAxis("Mouse X") * lookYawSpeed;
100         // Rotation on the X-axis (up/down) clamped to the +/- CAMERA_PITCH_LIMIT
101         pitch -= Input.GetAxis("Mouse Y") * lookPitchSpeed;
102         pitch = Mathf.Clamp(pitch, -CAMERA_PITCH_LIMIT, CAMERA_PITCH_LIMIT);
103
104         // Rotate only the body on the Y-axis (yaw)
105         characterController.transform.rotation = Quaternion.Euler(0, -yaw, 0);
106
107         // Rotate only the camera on the X-axis (pitch)
108         playerCamera.transform.localRotation = Quaternion.Euler(pitch, 0, 0);
109     }
110
111     public void StopPlayerMovement()
112     {
113         canMove = false;
114     }
115 }
```

WeaponController.cs

1. In `Start()`, the dependencies are obtained: in addition to `playerCamera` and `weaponEvents`, obtain the `HUDManager`, which will be the script responsible for managing all HUD elements, and the `ShotSource`, which will handle the shooting sounds in 3D environments. Start with all bullets (30 by default) and perform a null check on all variables. If that is the case, force the game to exit through another script called `VictoryManager`, which will be described later.

The sequence diagram for `Start()` in WeaponController is as follows:

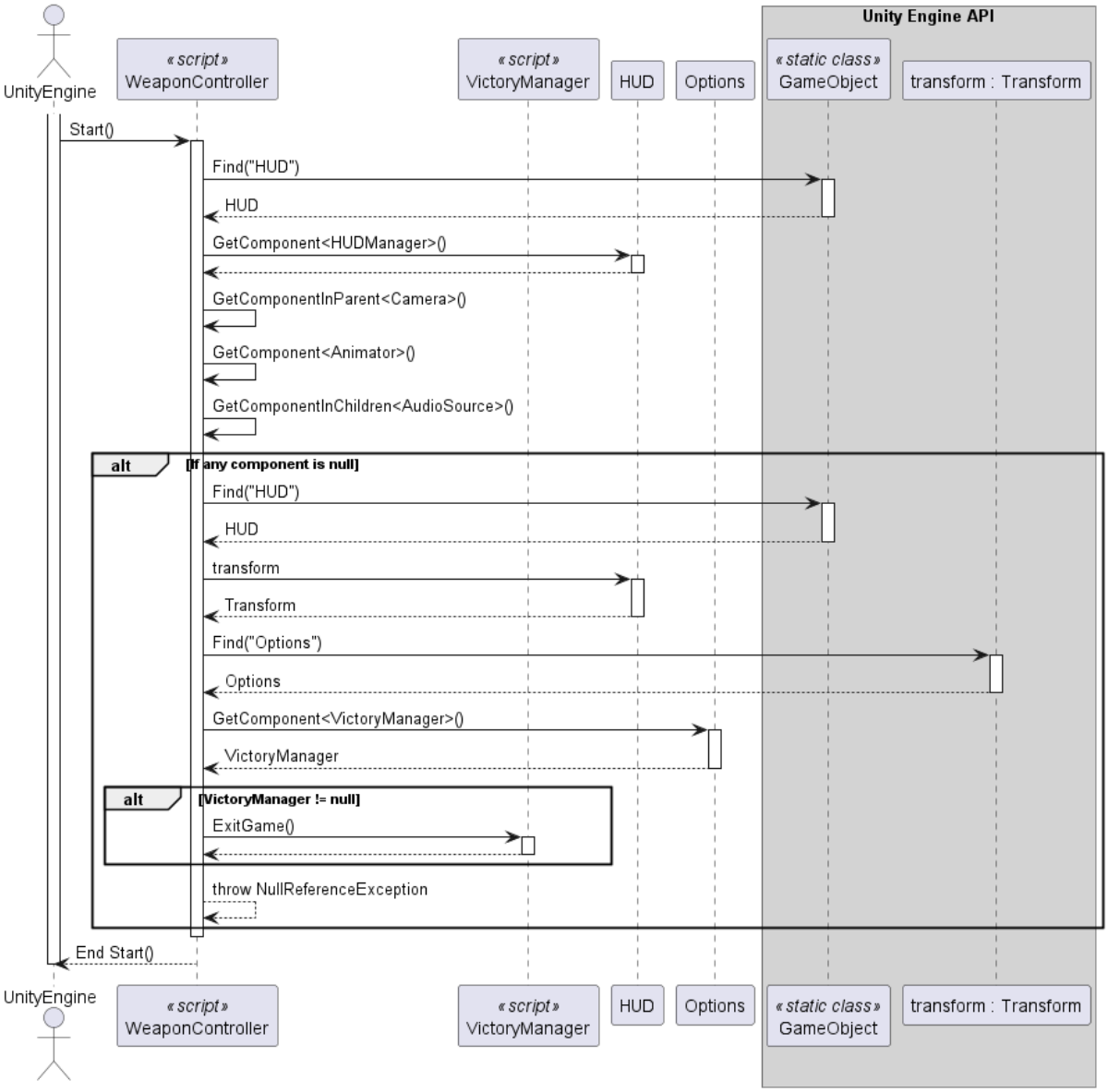

Figure 4.7: Sequence diagram of Start function of WeaponController script in Unity

2. In `Update()`, first, define how a state machine is used to control animations, as this will clarify the implementation of the Update method.
   In Unity, there is the `Animator` GameObject, which controls both 2D and 3D animations. Within this interface, it is possible to define exactly what a state machine is and assign an animation to each state.
   The view appears as follows:

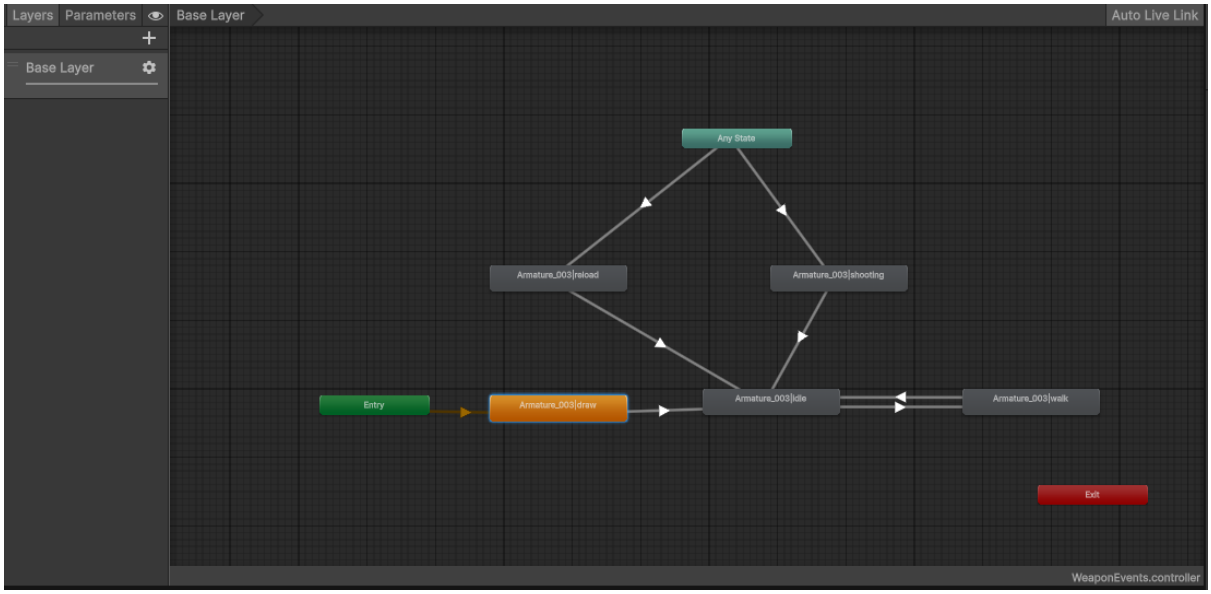

Figure 4.8: Unity environment animator Menu

As can be seen, in Unity, to keep the diagram cleaner, the *Any State* function can be used when it is desired to transition from any defined state to the desired states. Then, each state is assigned the animation that should be executed.

The area of interest is the transitions interface, where conditions for transitioning between states can be defined. Additionally, it is possible to require that the current animation must finish before the transition takes place, using the *Has Exit Time* parameter:

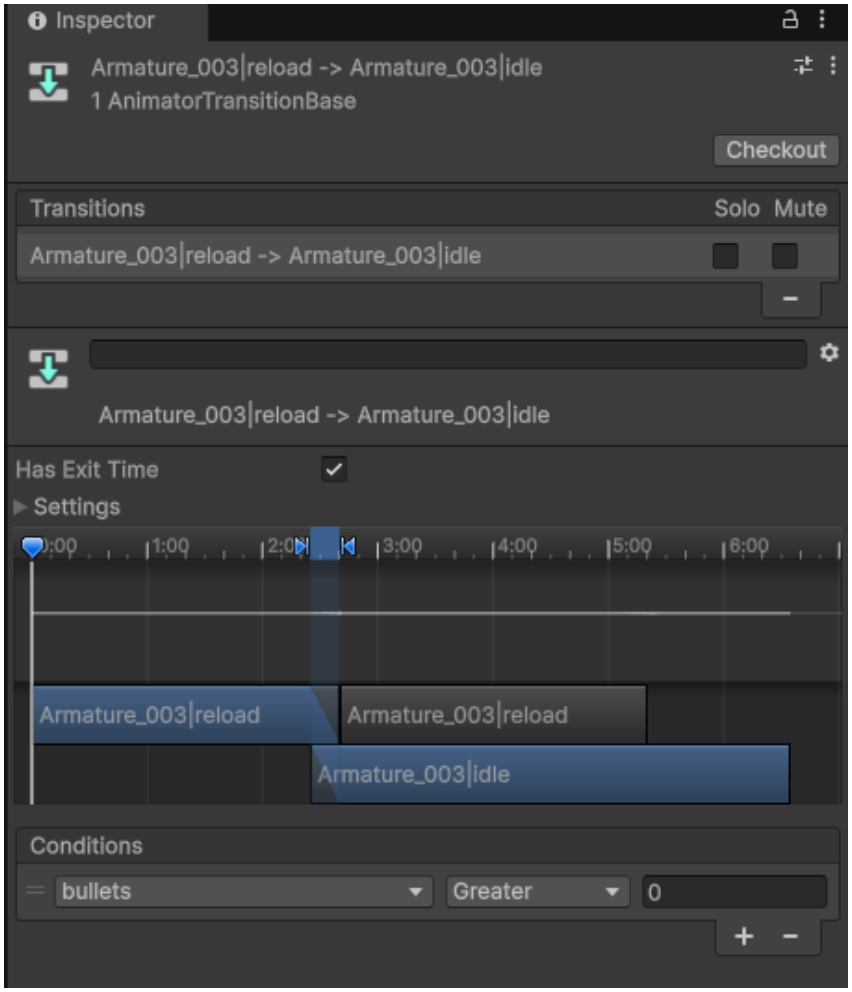

Figure 4.9: Unity environment animator Transition Menu

With this clarified, return to the code. The first step is to check whether the R key is pressed using (`Input.GetKeyDown(KeyCode.R)`) or if there are no bullets remaining, which triggers automatic reloading. Reloading can be performed only if a reload is not already in progress and the magazine is not full. In such a case, a coroutine will be called to handle the reloading process.
Now move on to shooting. Shooting is triggered by holding down the left mouse button. However, shooting cannot occur while the reload function is active. Additionally, a shooting cooldown has been implemented to simulate the fire rate of a real weapon, preventing the ammunition from being depleted too quickly. If the conditions are met, the shot is executed; otherwise, the shooting animation is set to false to allow returning to “idle” or “walking” states.
After this, simply update the HUD with the remaining bullets.

The sequence diagram for `Update()` in WeaponController is as follows:

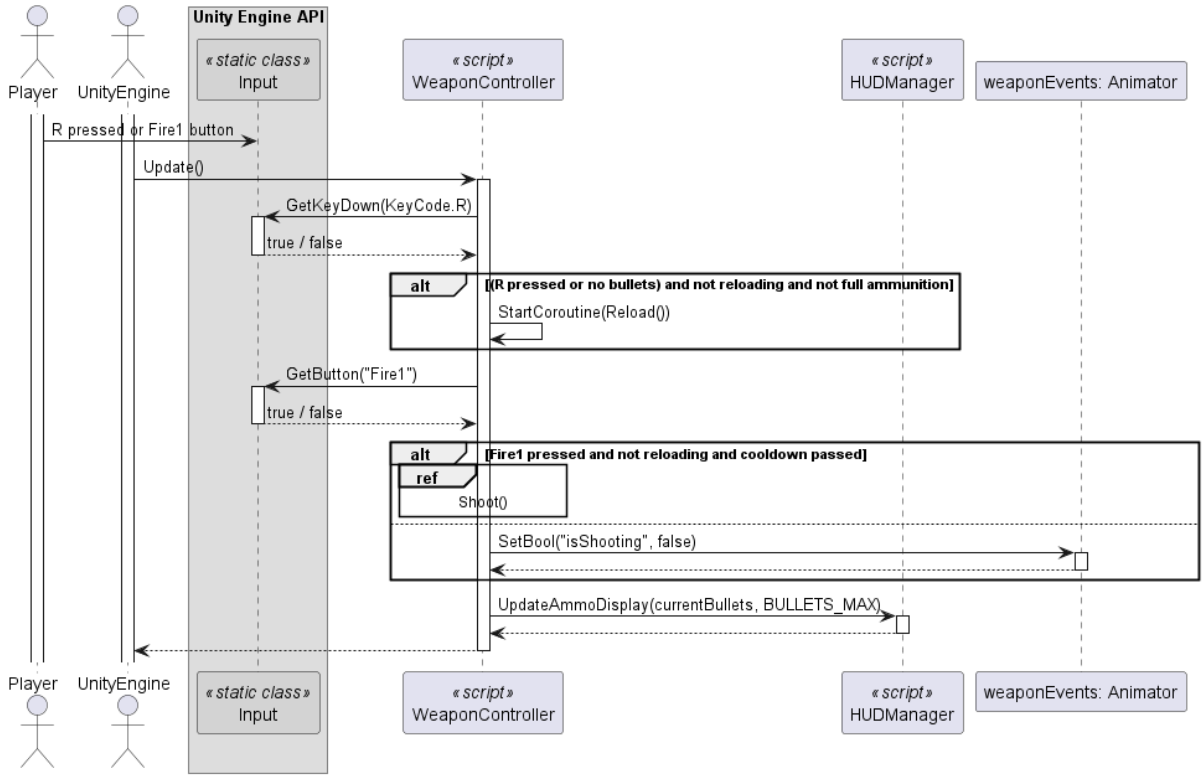

Figure 4.10: Sequence diagram of update function of WeaponController script in Unity

3. Now proceed to explain both functions: reloading and shooting:
   - **Reload**: defines a coroutine, allowing you to pause execution and resume after a delay, at this point, ensure that no other animations occur, as the reload animation takes priority. Triggers the reloading animation and waits for the duration of the reload animation before proceeding. This simulates the time it takes to reload. When the reload animation finishes, refill the weapon with the maximum ammunition and trigger a transition to the "idle" state. After this, allow the checks to continue in the following frames.
   - **Shoot**: activate the shooting animation and play the shooting audio file(Defined externally in the Inspector). After this, create the raycast, starting from the player's camera position, and its direction is that of the camera. The raycast detects if any object with a collider is in its path within the `range` distance. If the raycast collides with an object, that information is stored in hit. If, in addition, the hit object is a shooting panel (this is determined by the `GameObject` having a tag named "panel_shooting"), then The `PanelsManager` script, which manages the shooting panels, is called and the panel is made to disappear. The details of this function will be explained later. In any other case, no action is taken. Regardless of what happens, one bullet is subtracted from the weapon.

The sequence diagram for `Shoot()` in WeaponController is as follows:

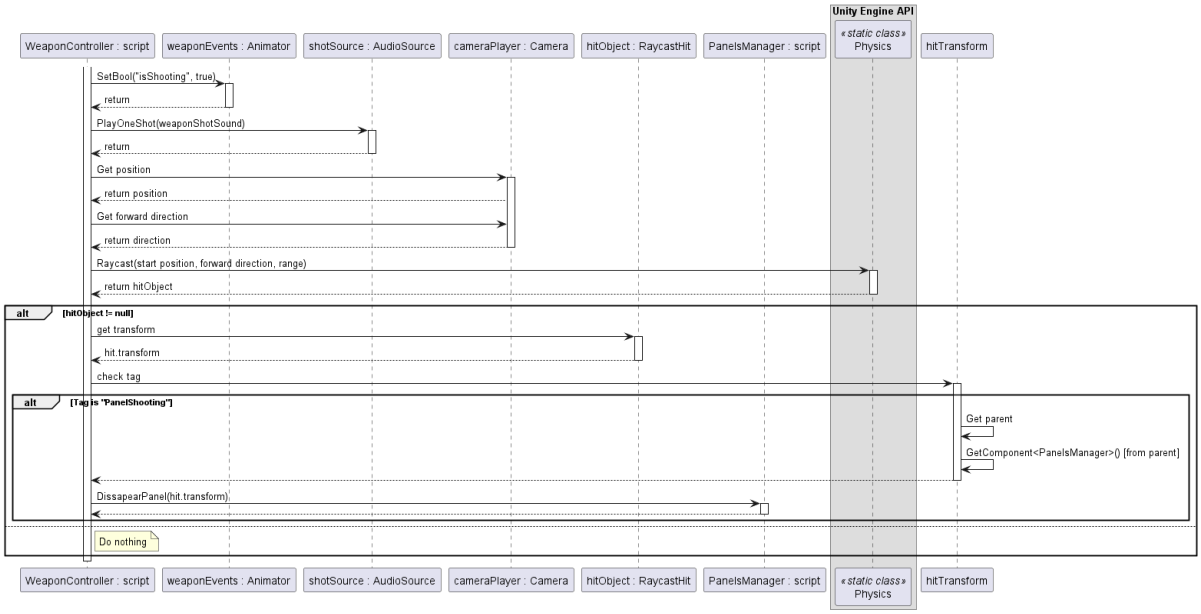

Figure 4.11: Sequence diagram of Shoot function of WeaponController script in Unity

The implemented code is as follows:

```
using System;
using System.Collections;
using System.Reflection;
using TMPro;
using Unity.Burst.Intrinsics;
using Unity.VisualScripting;
using UnityEditor;
using UnityEngine;
using UnityEngine.InputSystem;
using UnityEngine.UIElements;
using static UnityEngine.GraphicsBuffer;

public class WeaponController : MonoBehaviour
{
    public const int BULLETS_MAX = 30;

    private const float RELOADING_ANIMATION_DURATION = 3f;

    [Header("Weapon Settings")]
    public float fireRate = 0.3f;    //Time between shots, It is dependent on the shooting animation.
    public float range = 100f;
    public AudioClip weaponShotSound;

    private float nextFireTime = 0f; //Time in which shooting will be allowed again.
    private int currentBullets;
    private Camera playerCamera;
    private Animator weaponEvents;   // Weapon animation event handler.
    private AudioSource shotSource;
    private GameController gameController;
    private HUDManager HUDManager;

    // Prevent firing while reloading or to prevent reloading from repeating.
    private bool isReloading = false;


    void Start()
    {
        HUDManager = GameObject.Find("HUD").GetComponent<HUDManager>();
        playerCamera = GetComponentInParent<Camera>();
        weaponEvents = GetComponent<Animator>();
        shotSource = GetComponentInChildren<AudioSource>();
        gameController = GameObject.Find("Game Controller").GetComponent<GameController>();

        // If any element is missing, the game is aborted.
        ValidationUtils.CheckNotNull(HUDManager, "HUDManager script is missing.", this);
        ValidationUtils.CheckNotNull(playerCamera, "Camera component is missing.", this);
        ValidationUtils.CheckNotNull(weaponEvents, "Animator component for weapon events is
            missing.", this);
        ValidationUtils.CheckNotNull(shotSource, "AudioSource component for shotSource is missing.",
            this);
        ValidationUtils.CheckNotNull(weaponShotSound, "AudioClip component for weaponShotSound is
            missing.", this);
        ValidationUtils.CheckNotNull(gameController, "gameController script is missing.", this);

        currentBullets = BULLETS_MAX;
    }
    void Update()
    {
        HUDManager.UpdateAmmoDisplay(currentBullets, BULLETS_MAX);
        // Skip this update cycle
        if (gameController.automaticMode)
            return;

        // Reloading when desired with R button or when the bullets run out.
        // Reloading is avoided multiple times at the same time.
        if ((Input.GetKeyDown(KeyCode.R) || currentBullets == 0) && !isReloading && currentBullets
            != BULLETS_MAX)
        {
            StartCoroutine(Reload());
        }

        // The shot is fired by holding down the left mouse button and
        // it is ensured that it does not reload at the moment
        // in addition, Shooting is only allowed if the waiting time has passed.
        if (Input.GetButton("Fire1") && !isReloading && Time.time >= nextFireTime)
        {
            nextFireTime = Time.time + fireRate;
            Shoot();
        }
    }

    public void Shoot()
    {
        weaponEvents.SetTrigger("isShooting");
        shotSource.PlayOneShot(weaponShotSound);
```

```
82
83          // A raycast hit is created, starting from the player's camera position, and its direction
                is that of the camera.
84          RaycastHit hit;
85          Vector3 start = playerCamera.transform.position;
86
87          //if the raycast hits an object that has a collision.
88          if (Physics.Raycast(start, playerCamera.transform.forward, out hit, range))
89          {
90              // If it is a shooting panel, it is hidden.
91              if (hit.transform.CompareTag("PanelShooting"))
92              {
93                  PanelsManager panelsManager = hit.transform.parent.GetComponent<PanelsManager>();
94                  ValidationUtils.CheckNotNull(panelsManager, "PanelsManager script is missing.",
                        this);
95                  panelsManager.DissapearPanel(hit.transform);
96              }
97          }
98
99          currentBullets--;
100     }
101     private IEnumerator Reload()
102     {
103         isReloading = true;
104         // Activate the reloading animation
105         weaponEvents.SetTrigger("reloading");
106         // It is waited for during the time the reload animation lasts.
107         yield return new WaitForSeconds(RELOADING_ANIMATION_DURATION);
108         currentBullets = BULLETS_MAX;
109         // Reloading is finished.
110         weaponEvents.SetInteger("bullets", BULLETS_MAX);
111         isReloading = false;
112     }
113 }
```

HUDManager.cs

1. In `Start()`, all visual elements of the HUD have been referenced in the code for management during gameplay. Specifically, the following objects have been defined:
   - **ammoDisplay**: represents the amount of ammunition displayed to the player.
   - **gameTimer**: displays the elapsed game time.
   - **objectiveDisplay**: indicates the number of remaining active targets.
   - **victoryDisplay**: corresponds to a menu that is automatically activated once all targets have been eliminated.

   A null check was subsequently performed, followed by hiding the victoryDisplay element from the HUD view.

The sequence diagram for `Start()` in HUDManager is as follows:

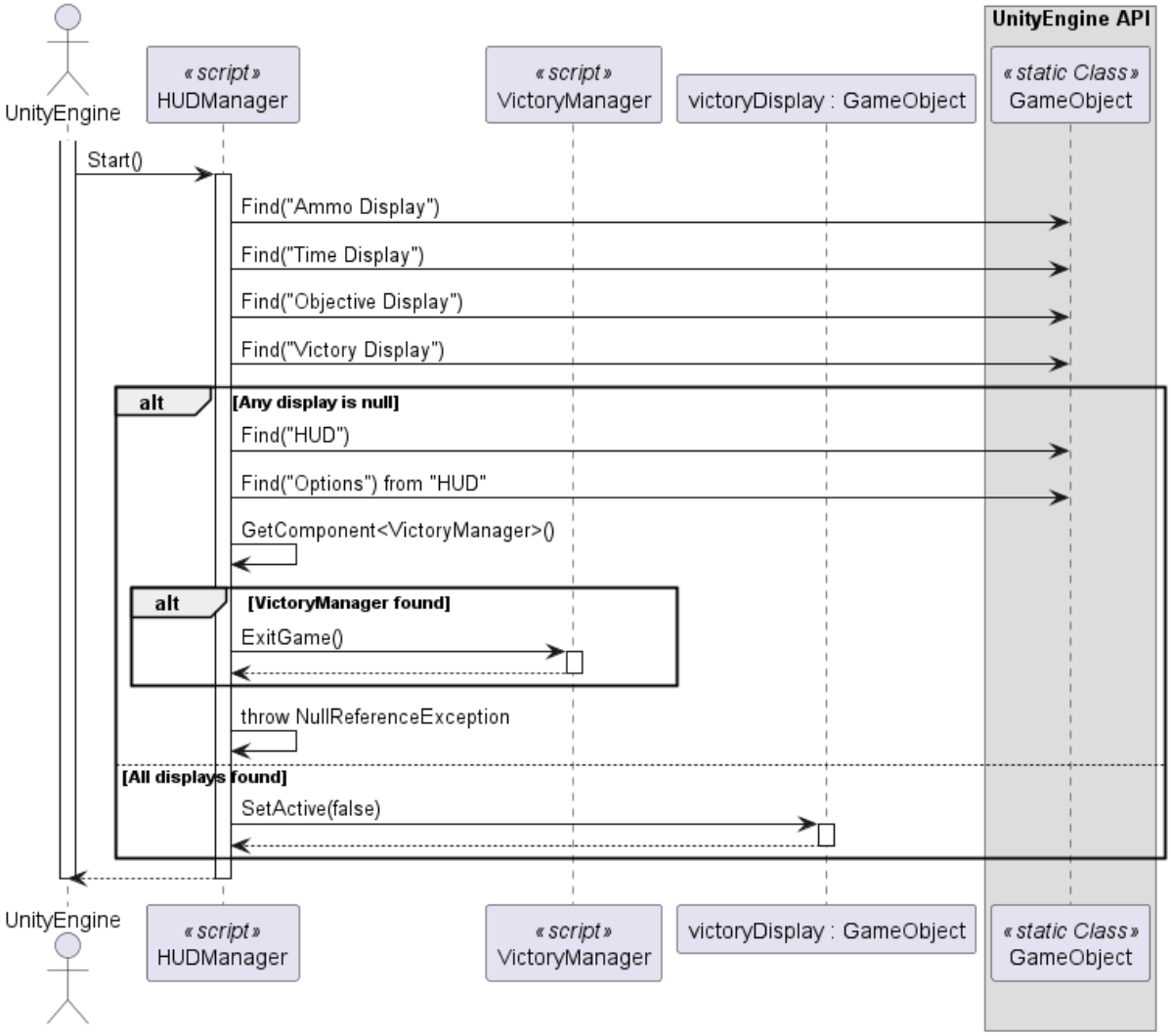

Figure 4.12: Sequence diagram of Start function of HUDManager script in Unity

2. Subsequently, a set of functions is defined solely for updating the content of the display, which are invoked from other scripts.(`UpdateObjectivesDisplay(int remainingObjectives)` and `UpdateAmmoDisplay(int currentBullets, int bulletsMax)`)
3. Finally, the `Victory()` function is executed when all shooting targets have been defeated. Within this function, all HUD elements are hidden, the `NightEffect` script is called to change the sky and lighting, and the `GameTimer` script is invoked to retrieve the total elapsed time and display it in the `victoryDisplay` along with a congratulatory message.
   It should be noted that in Unity, the `Find(string nameObject)` function only works with active objects. Therefore, the system is instructed to search for inactive objects as well, delegating responsibility to the `VictoryManager` to display the victory options.

The sequence diagram for `Victory()` in HUDManager is as follows:

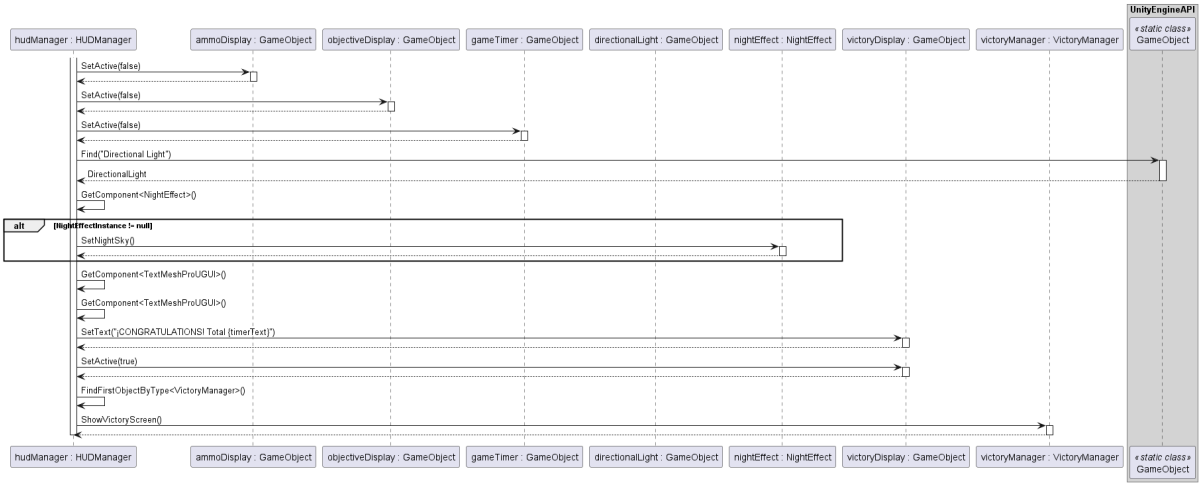

Figure 4.13: Sequence diagram of Victory function of HUDManager script in Unity

The implemented code is as follows:

```
using System;
using System.Reflection;
using TMPro;
using UnityEditor;
using UnityEngine;
using UnityEngine.SceneManagement;

/* A script that manages all the player's HUD components. */
public class HUDManager : MonoBehaviour
{
    private GameObject ammoDisplay;         // The ammunition indicated in the player's HUD.
    private GameObject gameTimer;           // the game Time indicated in the player's HUD.
    private GameObject objectiveDisplay;    // The number of targets that remain alive.
    private GameObject victoryDisplay;      // A menu that is enabled when all the targets are
        eliminated.
    private GameController gameController;

    void Start()
    {
        gameController = GameObject.Find("Game Controller").GetComponent<GameController>();
        ammoDisplay = GameObject.Find("Ammo Display");
        gameTimer = GameObject.Find("Time Display");
        objectiveDisplay = GameObject.Find("Objective Display");
        victoryDisplay = GameObject.Find("Victory Display");

        ValidationUtils.CheckNotNull(ammoDisplay, "ammoDisplay is missing.", this);
        ValidationUtils.CheckNotNull(gameTimer, "gameTimer is missing.", this);
        ValidationUtils.CheckNotNull(objectiveDisplay, "objectiveDisplay is missing.", this);
        ValidationUtils.CheckNotNull(victoryDisplay, "victoryDisplay is missing.", this);
        ValidationUtils.CheckNotNull(gameController, "gameController script is missing.", this);

        victoryDisplay.SetActive(false);
    }

    public void UpdateObjectivesDisplay(int remainingObjectives)
    {
        objectiveDisplay.GetComponent<TextMeshProUGUI>().text = $"Remaining Objectives:
            {remainingObjectives}";
    }

    public void UpdateAmmoDisplay(int currentBullets, int bulletsMax)
    {
        ammoDisplay.GetComponent<TextMeshProUGUI>().text = currentBullets + "/" + bulletsMax;
    }

    /* A function that manages the player's victory when all the targets have been eliminated. */
    public void Victory()
    {
        // The game elements are hidden.
        ammoDisplay.SetActive(false);
        objectiveDisplay.SetActive(false);
        gameTimer.SetActive(false);

        // Applies the night effect to the scene.
        NightEffect nightEffect = GameObject.Find("Directional Light").GetComponent<NightEffect>();
        ValidationUtils.CheckNotNull(nightEffect, "NightEffect script is missing.", this);
        nightEffect.SetNightSky();

        // The game time is taken and displayed with a victory message.
        string timerText = gameTimer.GetComponent<TextMeshProUGUI>().text;
        victoryDisplay.GetComponent<TextMeshProUGUI>().text = $"CONGRATULATIONS! Total {timerText}";
        victoryDisplay.SetActive(true);

        //he victory menu is opened with the available options.
        VictoryManager victoryManager =
            FindFirstObjectByType<VictoryManager>(FindObjectsInactive.Include);
        ValidationUtils.CheckNotNull(victoryManager, "VictoryManager script is missing.", this);
        if (!gameController.automaticMode)
            victoryManager.ShowVictoryScreen();
        else
            victoryManager.ExitGame();
    }

}
```

VictoryManager.cs

1. In `Start()`, the victory panel is hidden at the start (`gameObject.SetActive(false)`), as it should not be displayed while the game is in progress. Subsequently, functions are assigned to the `retryButton` and `exitButton` to ensure they respond to user clicks.
2. In `ShowVictoryScreen()`, The method is invoked when the player wins the game. First, Pause the game's time to stop all in-game activity, activate the victory panel to make it visible. Subsequently, unlock and display the cursor so the player can interact with the HUD. Finally, disable player movement by calling the `StopPlayerMovement()`.
3. In `Restart()`, the game is resumed and the current level is reloaded.
4. In `Exit()`, the game is closed; if running within the Unity editor, the mode simply returns to edit mode.

The implemented code is as follows:

```
using System.Collections;
using UnityEngine;
using UnityEngine.SceneManagement;
using UnityEngine.UI;


public class VictoryManager : MonoBehaviour
{

    public Button retryButton;
    public Button exitButton;

    void Start()
    {
        //The panel is hidden at the start.
        gameObject.SetActive(false);

        ValidationUtils.CheckNotNull(retryButton, "retryButton component is missing.", this);
        ValidationUtils.CheckNotNull(exitButton, "exitButton component is missing.", this);

        // Functions are assigned to the buttons when using the mouse.
        retryButton.onClick.AddListener(RestartGame);
        exitButton.onClick.AddListener(ExitGame);


    }

    public void ShowVictoryScreen()
    {
        Time.timeScale = 0f;        // Paused game
        gameObject.SetActive(true); // The panel is shown.
        Cursor.lockState = CursorLockMode.Confined; // The mouse is within the game screen.
        Cursor.visible = true;

        PlayerMovement playerMovement = GameObject.Find("Player").GetComponent<PlayerMovement>();
        ValidationUtils.CheckNotNull(playerMovement, "PlayerMovement script is missing.", this);
        playerMovement.StopPlayerMovement();
    }
    public void RestartGame()
    {
        Time.timeScale = 1f;                                        // Resume game
        SceneManager.LoadScene(SceneManager.GetActiveScene().name); //Reload the scene
    }
    public void ExitGame()
    {
        Application.Quit();                                 // Close Game
        #if UNITY_EDITOR
            UnityEditor.EditorApplication.isPlaying = false; // Stop in the editor
        #endif
    }
}
```

PanelsManager.cs

1. In `Start()`, locate the `HUDManager`, count all shooting panels, and assign the total to the remaining targets counter.
2. Use the `DissapearPanel(Transform panelShot)` function to receive the panel that has been hit and simply hide it from the scene. Then, update the remaining targets and check whether all panels have been removed.
3. In `AreAllPanelsDissapeared()`, If all targets have been eliminated, call the `HUDManager` to handle the victory sequence.

The implemented code is as follows:

```
using TMPro;
using Unity.VisualScripting;
using UnityEngine;

public class PanelsManager : MonoBehaviour
{
    private int totalObjectives;
    private int remainingObjectives;
    private HUDManager HUDManager;

    void Start()
    {
        HUDManager = GameObject.Find("HUD").GetComponent<HUDManager>();
        ValidationUtils.CheckNotNull(HUDManager, "HUDManager script is missing.", this);
        totalObjectives = transform.childCount; // Gets the number of active panels in the scene
        remainingObjectives = totalObjectives;
    }
    /**
     * A function that makes the panel disappear from the scene, considering it as one less target.
     *
     * @param panelshot, The panel to disappear.
     */
    public void DissapearPanel(Transform panelShot)
    {
        panelShot.gameObject.SetActive(false);
        remainingObjectives--;
        HUDManager.UpdateObjectivesDisplay(remainingObjectives);
        AreAllPanelsDissapeared();
    }

    private void AreAllPanelsDissapeared()
    {
        if (remainingObjectives == 0 && HUDManager != null)
            HUDManager.Victory();
    }
}
```

NightEffect.cs

This script simply adds a night-time effect when the game ends. Initially, `setDaySky()` is used to apply a sunny daytime environment at the start of the game. Once the game ends, replace the sky with a night setting and adjust the directional light to create a night-time atmosphere. The implemented code is as follows:

```
using UnityEngine;


/* A script that implements a night sky effect when the game is won. */
public class NightEffect : MonoBehaviour
{
    public Material daySkybox; // Material for the sky during the day
    public Material nightSkybox; // Material for the starry sky during the night

    private Light directionalLight; // Directional light (sun)
    private float nightIntensity = 0.2f; // Intensity of the night light
    private Color nightLightColour = Color.blue; // Colour of the night light

    void Start()
    {
        directionalLight = GetComponent<Light>();
        ValidationUtils.CheckNotNull(directionalLight, "Light is missing.", this);

        ValidationUtils.CheckNotNull(daySkybox, "DaySkybox is missing.", this);
        ValidationUtils.CheckNotNull(nightSkybox, "NightSkybox is missing.", this);

        setDaySky();
    }

    public void SetNightSky()
    {
        RenderSettings.skybox = nightSkybox;            // Change the Skybox to a starry one
        directionalLight.intensity = nightIntensity;    // Reduce the intensity of the sun's light
        directionalLight.color = nightLightColour;      // Change the light's colour
        RenderSettings.ambientLight = nightLightColour; // Adjust the ambient light
    }

    public void setDaySky()
    {
        RenderSettings.skybox = daySkybox;
    }
}
```

GameTimer.cs

This script simply displays the elapsed time on screen after the game starts, formatting it into minutes and seconds using two digits, similar to a digital clock.

The floor function rounds down to the smallest integer that is not greater than the number. Floor is used because the time between one minute and the next has not yet fully passed. For example, `Floor(59.9)` returns 59 seconds, which is correct as the full minute has not elapsed. If `Round(59.9)` were used instead, it would return 60 seconds, potentially causing the timer to display 1 minute and 0 seconds prematurely, since it rounds to the nearest integer.

The implemented code is as follows:

```
using TMPro;
using UnityEngine;

public class GameTimer : MonoBehaviour
{
    private float elapsedtimeSeconds = 0f;
    private TextMeshProUGUI timeDisplay; // UI where the time is displayed.

    void Start()
    {
        timeDisplay = GetComponent<TextMeshProUGUI>();
        ValidationUtils.CheckNotNull(timeDisplay, "TextMeshProUGUI component in timeDisplay is
            missing.", this);
    }
    void Update()
    {
        elapsedtimeSeconds += Time.deltaTime;
        // We convert the total time into a digital clock format.
        int minutes = Mathf.FloorToInt(elapsedtimeSeconds / 60);
        int seconds = Mathf.FloorToInt(elapsedtimeSeconds % 60);
        timeDisplay.text = $"Time: {minutes:D2}:{seconds:D2}"; // MM:SS Format
    }

}
```

CrosshairManager.cs

1. In `Start()`, initialise the crosshair texture and apply a base version using the `GetWhiteTransparent2DTexture()` function. Then, rescale it and position it at the centre of the screen. inally, retrieve the reference to the player's camera.
2. In `Update()`, the same principle used for shooting is applied: create a raycast that detects whether it hits an object. If the raycast hits a shooting panel, change the crosshair colour to red to highlight the target; otherwise, keep it white by default.
3. In `GetWhiteTransparent2DTexture(Texture2D original)`, generate a new texture based on the original by applying two transformations:
   - All white pixels (RGB > 0.9) are made fully transparent (alpha $= 0$).
   - All pixels with a colour similar to `originalCrosshairColour`(Colour similar to red), (using `IsColourClose`) are replaced with white.

   To understand the reason behind this, simply consider how the original crosshair looked:

Figure 4.14: Original Crosshair used in Unity

4. In `IsColourClose(Color colour1, Color colour2, float tolerance)`, compare two colours and determine whether their RGB components are similar within a given tolerance. This is mainly used to identify colours in the texture that are similar to the original crosshair colour and need to be replaced with white.

The implemented code is as follows:

```
using GLTFast;
using NUnit.Framework;
using System;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.UI;
using static UnityEngine.GraphicsBuffer;

/* Script that implements a crosshair configuration, resetting it to white.*/
public class CrosshairManager : MonoBehaviour
{
    public Texture2D originalTexture;
    public float tolerance = 0.1f;
    public float range = 100f;

    private Camera cameraPlayer;
    private Color originalCrosshairColour = new Color(190f/255f, 57f/255f, 31/255f);
    private Color targetCrosshairColour = Color.red;
    private Color defaultCrosshairColour = Color.white;
    private Image crosshairImage;

    void Start()
    {
        ValidationUtils.CheckNotNull(originalTexture, "Texture2D component is missing.", this);
        Texture2D changedTexture = GetWhiteTransparent2DTexture(originalTexture);

        crosshairImage = GetComponent<Image>();
        ValidationUtils.CheckNotNull(crosshairImage, "Image component is missing.", this);
        crosshairImage.sprite = Sprite.Create(changedTexture, new Rect(0, 0, changedTexture.width,
            changedTexture.height), new Vector2(0.5f, 0.5f));

        cameraPlayer = GameObject.Find("Player").GetComponentInChildren<Camera>();
        ValidationUtils.CheckNotNull(cameraPlayer, "Camera component is missing.", this);
    }

    void Update()
    {
        // A raycast hit is created, starting from the player's camera position, and its direction
            is that of the camera.
        RaycastHit hit;
        Vector3 start = cameraPlayer.transform.position;
        Vector3 direction = cameraPlayer.transform.forward;
            //if the raycast hits an object that has a collision.
        if (Physics.Raycast(start, direction, out hit, range))
        {
            if (hit.transform.CompareTag("PanelShooting"))  // If what is hit is a panel shooting
            {
                crosshairImage.color = targetCrosshairColour; // change colour to target colour
                return;                                       // Prevents it from turning default
                    colour again further down.
            }
        }
        // Otherwise, the colour is default
        crosshairImage.color = defaultCrosshairColour;
    }

    /**
     * Checks if two colours are practically the same within a given tolerance.
     *
     * @param colour1 The first colour to compare.
     * @param colour2 The second colour to compare.
     * @param tolerance The allowed difference between the RGB components of the colours.
     * @return true if the colours are similar within the tolerance, false otherwise.
     */
    private bool IsColourClose(Color colour1, Color colour2, float tolerance)
    {
        float rDiff = Mathf.Abs(colour1.r - colour2.r);
        float gDiff = Mathf.Abs(colour1.g - colour2.g);
        float bDiff = Mathf.Abs(colour1.b - colour2.b);


        return rDiff < tolerance && gDiff < tolerance && bDiff < tolerance;
    }

    private Texture2D GetWhiteTransparent2DTexture(Texture2D original)
    {
        Texture2D editableTexture = new Texture2D(original.width, original.height,
            TextureFormat.RGBA32, false);
        editableTexture.SetPixels(originalTexture.GetPixels());
        editableTexture.Apply();

        // Modify white pixels to be transparent
        for (int y = 0; y < editableTexture.height; y++)
        {
            for (int x = 0; x < editableTexture.width; x++)
```

```
82              {
83                  Color pixel = editableTexture.GetPixel(x, y);
84                  if (pixel.r > 0.9f && pixel.g > 0.9f && pixel.b > 0.9f)
85                  {
86                      pixel.a = 0;
87                      editableTexture.SetPixel(x, y, pixel);
88                      continue;
89                  }
90                  if (IsColourClose(pixel, originalCrosshairColour, tolerance))
91                  {
92                      pixel = Color.white;
93                      editableTexture.SetPixel(x, y, pixel);
94                  }
95
96              }
97          }
98          editableTexture.Apply();
99          return editableTexture;
100     }
101 }
```

4.4 Unreal Engine

4.4.1 Introduction

Unreal Engine is the second engine used to implement the FPS prototype in manual mode. Its role in this project is to provide a technically advanced alternative to Unity, enabling a comparison of the implementation of the same set of functionalities in a different environment, with a more powerful graphics system and a distinct architecture.

The same functional prototype has been implemented: player control, collision system, shooting and reloading logic, HUD interface, and audiovisual elements. This ensures that the comparison between engines is based on identical technical and design conditions.

Unreal Engine version **4.27.2** was used due to graphical incompatibilities with newer versions on the development system. Nevertheless, this version retains all the necessary capabilities to develop and evaluate the prototype without functional limitations.

The following sections describe the environment configuration in Unreal, followed by the implementation of the collision system and game logic, replicating the structure previously established in Unity.

4.4.2 Environment

The environment is divided into clearly defined sections, which are as follows:

- **Content browser**:it works in the same way as Unity's Window project, with the difference that the packages are C++ preprocessor directives for engine resources.
- **World outliner**:it works in the same way as Unity's Hierarchy window. Scene objects are referred to as actors. These can range from simple forms—such as an empty character or a stage light, which include the minimal components of a game object (transform, collision, etc.)—to complex forms, typically composed of groups of actors that share similar behaviour
- **Place actors**:this panel allows searching for and placing predefined actors in the scene. Actors can be added to the world easily by simply dragging them from a list of common elements into the viewport.
- **Viewport**: same function as Unity's Scene view.
- **Game bar**: this is where the game can be tested. Unreal Engine allows viewing the in-game logs for debugging purposes, displaying them by default at the top right of the screen of the game.
- **Details**: this is where the components of actors can be viewed and interacted with directly, without the need for coding.

It should be noted that this is the default version in which the environment is presented, although it is possible to choose which tabs to display, as well as their arrangement, through the *window* menu located at the top left of the interface.

An overview of a Unreal Engine project is depicted in this image:

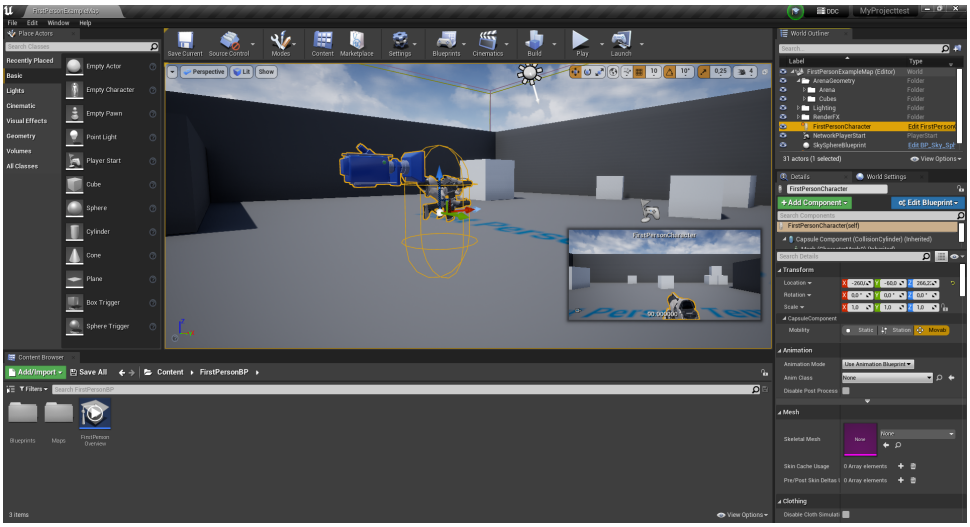

Figure 4.15: Unreal environment

4.4.3 Collider implementation

In Unreal Engine, collision management is quite automatic and efficient. By default, the engine generates collisions from the actor's mesh using a simplified approximation known as a *collision mesh* or *collision hull*. This approximation reduces the complexity of the original model to optimise collision calculations without sacrificing accuracy in most cases.

For static or less complex objects, Unreal automatically creates *simple collision shapes* (such as boxes, capsules, or spheres) that encompass the original geometry. For more complex models, *complex collision* can be used, which detects collisions based on the actual mesh, albeit at a higher performance cost.

In the development of the prototype, the automatically generated collisions were sufficient, providing a good balance between precision and efficiency. However, in more advanced projects, it may be necessary to create custom colliders or adjust collision parameters to optimise performance and gameplay experience.

4.4.4 Code explanation

In Unreal, programming is primarily conducted through **Blueprint Visual Scripting**, that provides a comprehensive method to create game logic through a node-based visual interface within the Unreal editor. Instead of writing traditional code, users connect nodes representing functions, events, and variables to define behaviours and interactions[28].

From an object-oriented programming perspective, a `Blueprint Class` is a visual and editable class that extends a base engine class (such as `Actor`, `Pawn`, or `Character`) and contains both data and behavioural logic.

To access them, it is necessary to use the actor's Details panel and select `Blueprint/Add Script` or `Edit Blueprint` if one already exists. This action associates a `Blueprint Class` with the actor and opens a window for editing the Blueprint. The layout of that window appears as follows:

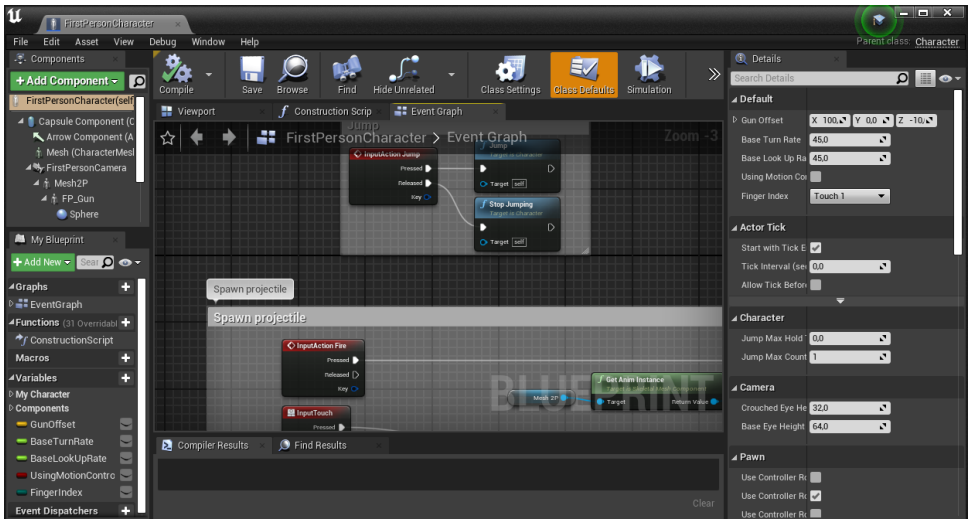

Figure 4.16: Unreal Blueprint editor environment

In the Blueprint editor, the Details and World Outliner windows (in this case named *Components*) are available. Additionally, there is other important windows such as *My Blueprint* that displays the elements that belong to the given class. These elements include:

- **Variables**: act as the attributes of a class, defining their type and class scope.
- **Functions**: Just like traditional programming functions.
- **Graphs**: the Blueprint's primary execution flow occurs within the **Event Graph**, which processes both system-level events—such as those related to the game's lifecycle—and input events from devices like the keyboard or mouse. Additionally, the **Anim Graph** is used to simulate a state machine, primarily for controlling animation behaviour. the various types of graphs available, these two are the most relevant for the current context.

The explanations of the various Blueprints will describe the functions of the nodes and their connections. For the prototype design, four Blueprints and one Animation Blueprint—a specialised type of Blueprint responsible for handling animations—have been chosen. The weapon's animation state machine will be implemented within the Animation Blueprint.

NOTE: to visualise the entire execution flow of the various functions and procedures within the blueprint, the **Sequence** node has been used, which divides the execution into separate blocks. To make an analogy, it is similar to instructing a program to execute code blocks by pointing to their memory addresses where each block begins.

All these Blueprints will be detailed below:

BP_Player

The Player Blueprint is responsible for managing both the player's behaviour and that of the weapon they possess. In this case, since only a single weapon is used, all related logic has been included within a single Blueprint. If multiple weapons were to be implemented, a separate Blueprint would be required for each one.

The behaviour has been managed through several groups of nodes, each responsible for specific functionality. These are:

- **VariableSettings**: this section initialises variables by assigning references to other Blueprints, such as `BP_Scene` and `WeaponAnimationController`. It also sets the initial display of the weapon's bullet count. This is displayed as follows:

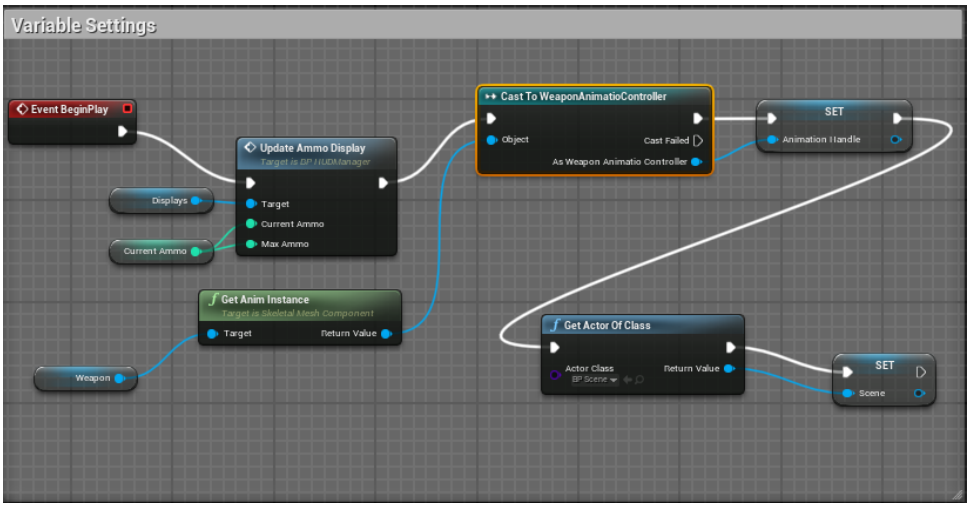

Figure 4.17: Variable settings in BP_player in unreal

- **Movement**: The first thing to consider is that the input system works differently from Unity. To assign keys, gamepad buttons, or mouse inputs, the following steps must be taken:
  1. Open the project in Unreal Engine.
  2. Navigate to `Edit > Project Settings`.
  3. In the left-hand panel, select **Input** under the **Engine** category.
  4. Under the **Bindings** section, two primary types of mappings are available:
     - **Action Mappings**: used for discrete inputs, such as key presses or mouse clicks.
     - **Axis Mappings**: used for continuous inputs, such as directional movement.
  5. Click the + button next to either *Action Mappings* or *Axis Mappings*.
  6. Name the mapping, e.g., `MoveForward`.
  7. Assign one or more keys:
     - For example, assign `W` with a scale of +1.
     - Assign `S` with a scale of -1.
  8. For Action Mappings, assigning the corresponding key is sufficient, as it behaves like a boolean value.

It must have this appearance:

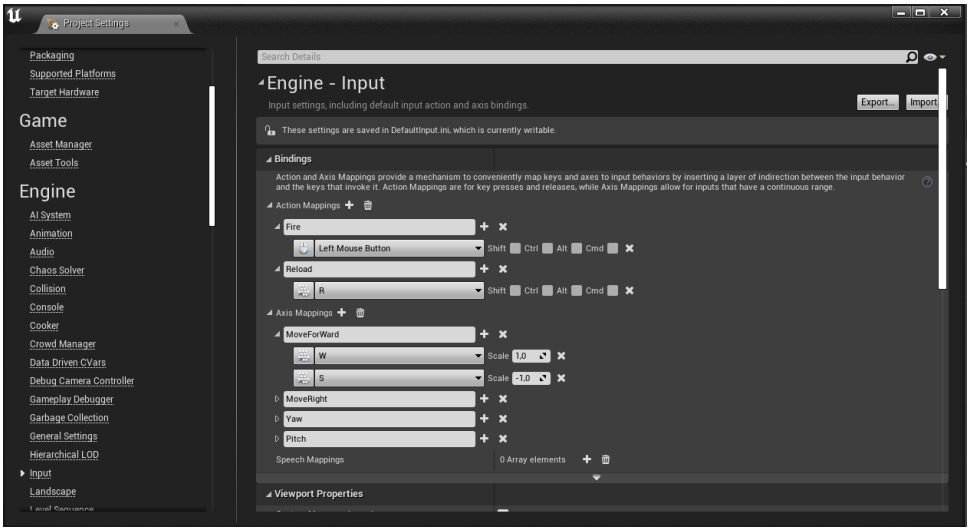

Figure 4.18: Input in project settings in unreal

After explaining this, the focus returns to the behaviour. The direction axis is multiplied by the player's forward vector according to the key pressed, and this resulting movement is applied to the player.
The image showing this is as follows:

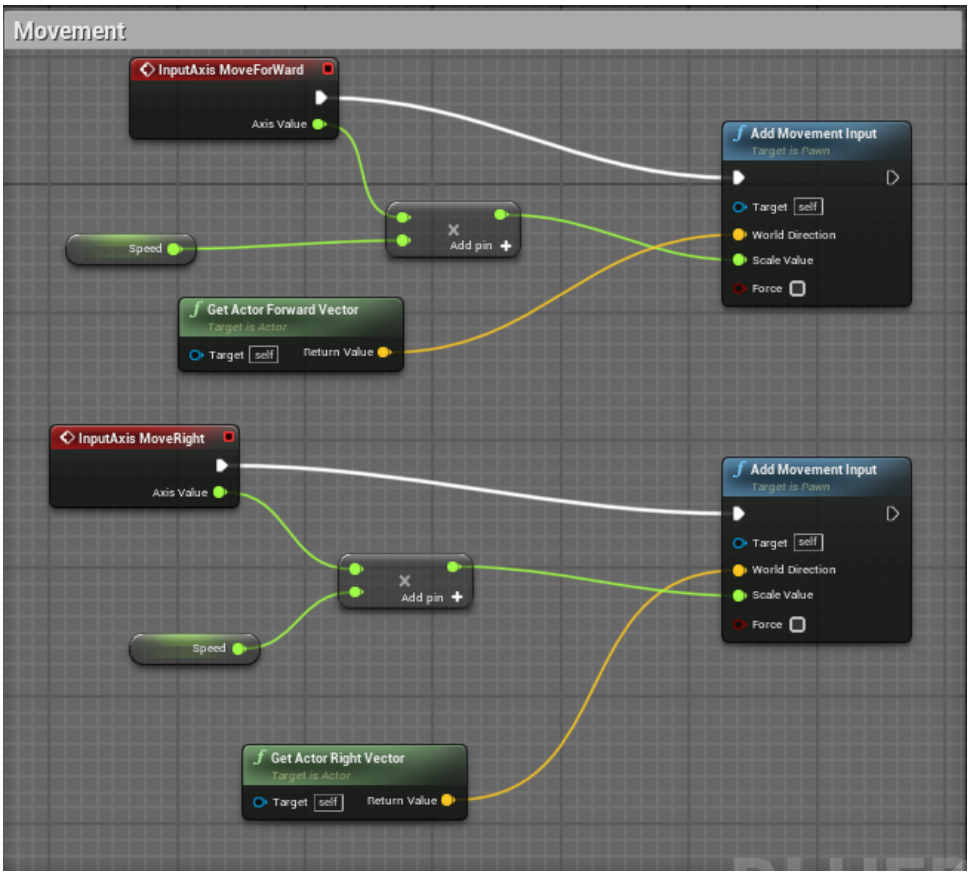

Figure 4.19: Movement in BP_player in unreal

One point to highlight is that Unreal automatically handles gravity, so it is not reflected in the Blueprint behaviour and does not need to be calculated.

- **LookAround**: the implementation is similar to that of Unity, with the difference that gimbal lock does not occur directly because Unreal handles this issue automatically.
  The image showing this is as follows:

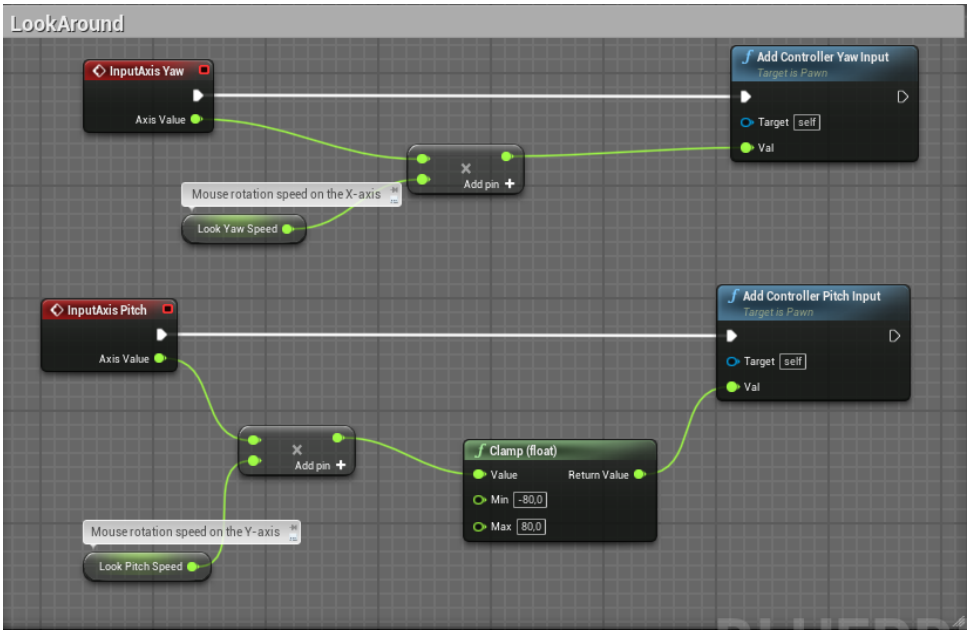

Figure 4.20: LookAround in BP_player in unreal

- **Walking animation**: the walking animation, which in this case affects only the weapon, is controlled as follows. The first element to consider is the `Event Tick` node, which executes the event every frame. This is similar to the `Update()` function in Unity. This mechanism will be used in other behaviours later; however, in this instance, each frame verifies whether the player's speed is greater than zero, and the resulting logical value is assigned to the transition variable isWalking in the WeaponAnimationController.
  The image showing this is as follows:

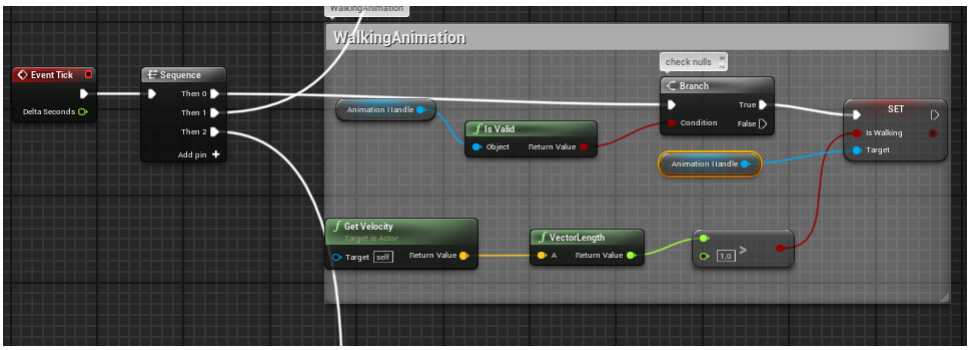

Figure 4.21: Walking animation in BP_player in unreal

An explanation will be provided regarding the state machine for weapon animations in Unreal. As in Unity, a robust system is available to represent state machines for animations through the *Animation Blueprint*.

The appearance of the state machine in Unreal is as follows:

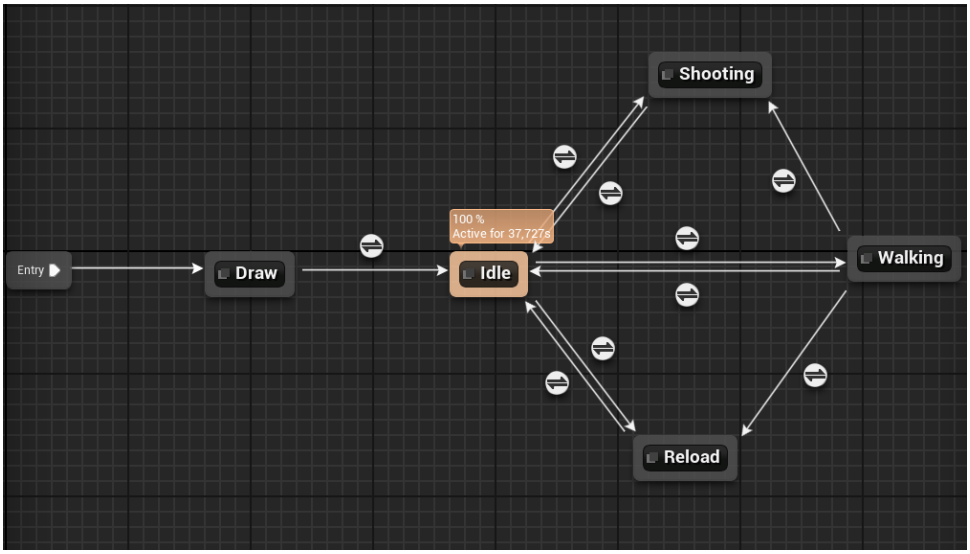

Figure 4.22: Animation Blueprint state machine in Unreal

Transitions are defined as boolean nodes that verify whether a transition can occur, while states correspond to the execution of weapon animations, similar to Unity.
Finally, some animations need to be played entirely before proceeding, such as draw, reload, and shooting. For these, *notifies* are defined, which notify when an animation has reached a specific frame. When this occurs, a state change can be triggered.
To add notifies, the following steps must be taken:

1. In the project, open the animation in which the notify is to be added. This may be an `Animation Sequence`.
2. In the animation window, observe the timeline curve where frames can be played and viewed.
3. Right-click on the notifies bar (below the timeline).
4. Select `Add Notify` → `New Notify`.
5. Assign a name to the notify, for example: "`EndShooting`".
6. Drag or position the notify at the exact frame where the event should be triggered, approximately near the end in this case.
7. Open the `AnimInstance` that manages the animation (`Animation Blueprint`).
8. Locate the node `Anim Notify` or a similar event (it may be called `anim NotifyEndShooting`).
9. Within the event, use the notify's name to distinguish which notify is executing and trigger the desired logic (for example, playing a sound).

It should look something like this:

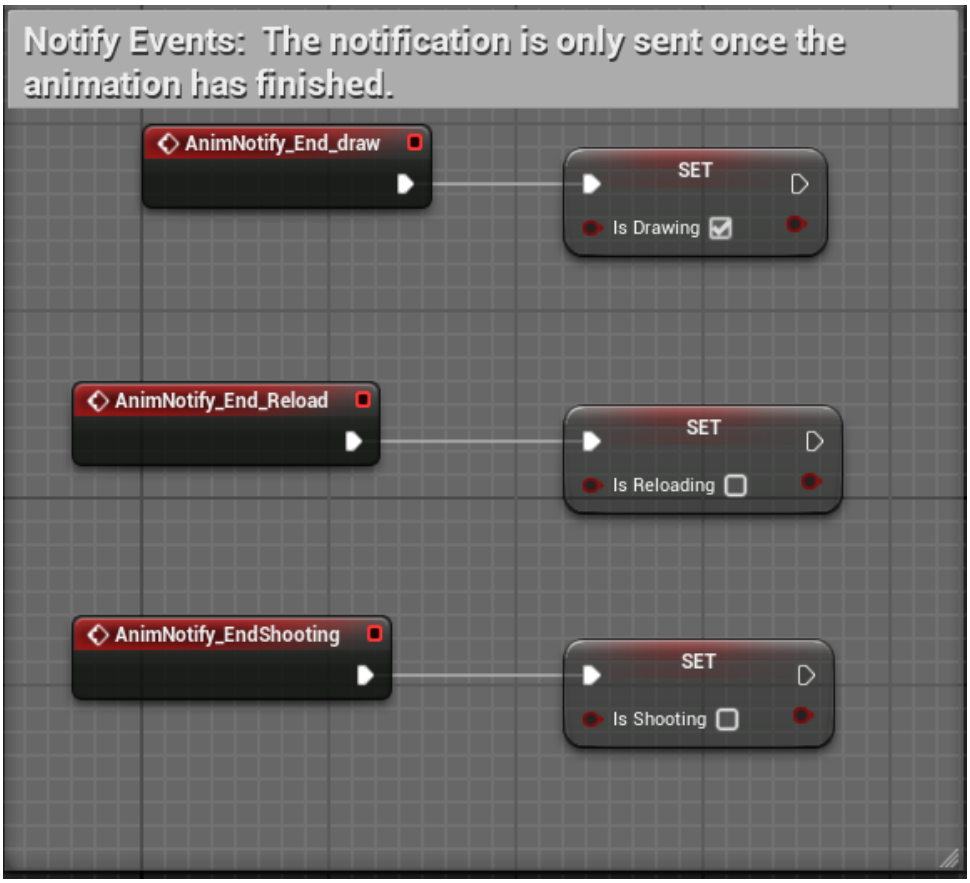

Figure 4.23: Notify events in Animation Blueprint in Unreal

- **Reload state**: the approach is similar to Unity. It is assumed that the `R` key has been previously assigned for reloading. If the weapon is not currently reloading and the ammunition is not at maximum capacity, the reload is triggered by calling the `Reload` function.
  The image showing this is as follows:

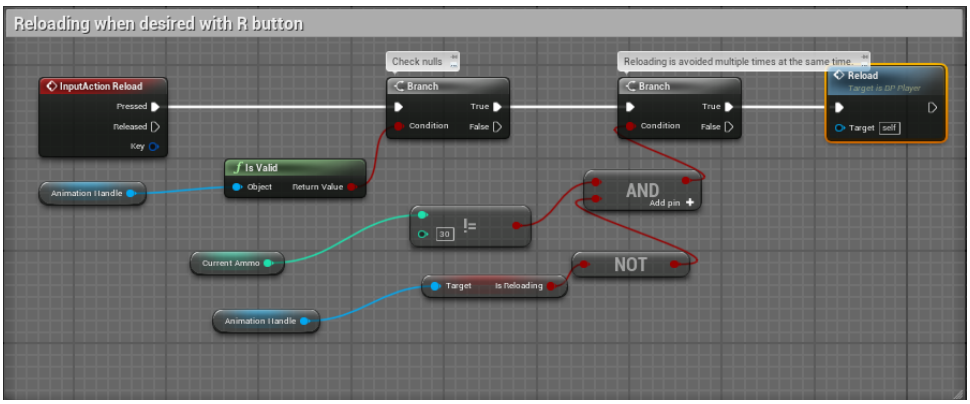

Figure 4.24: Reload state in BP_player in Unreal

In the Reload function, the reload animation is triggered first, followed by a delay of 2.7 seconds to allow the animation to complete before continuing the execution flow. Once the delay has finished, the HUD is updated to reflect the maximum number of bullets.

The image showing this is as follows:

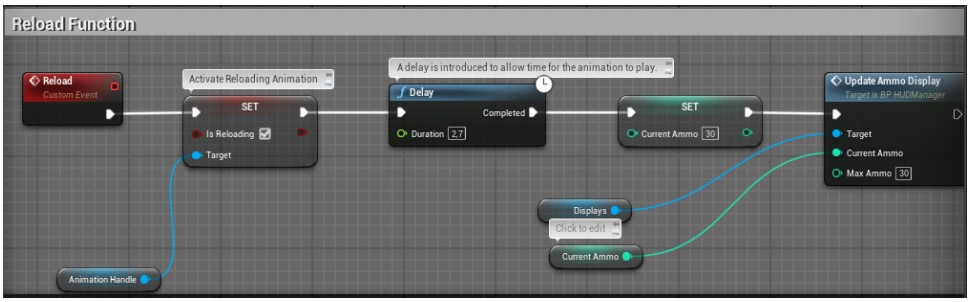

Figure 4.25: Reload Function in BP_player in Unreal

- **Shooting state**: to simulate automatic weapon fire, the `Gate` node is used. While the left mouse button (previously bound) is held down, the execution flow enters the firing behaviour. Once the button is released, the flow is interrupted, as this logic is also driven by the Event Tick. If the weapon is not currently reloading, the `Get Time Seconds in World` node (which represents the total time elapsed since the level was loaded) must be greater than Next Time Fire, and there must be available ammunition. If these conditions are met, the weapon can fire (as defined in the corresponding `Shoot` function). After firing, if no ammunition remains, the automatic reload process described previously is triggered.
  The image showing this is as follows:

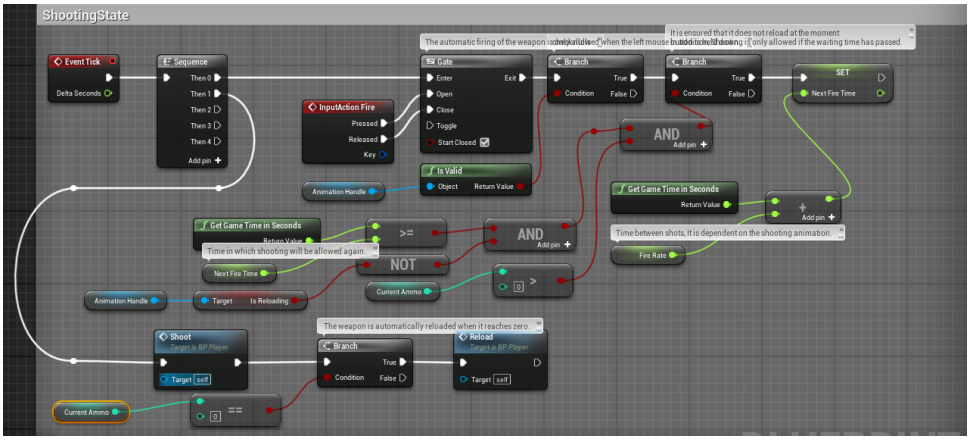

Figure 4.26: Shooting state in BP_player in Unreal

In the `Shoot` function, the weapon's animation is changed to the shooting animation, and a sound simulating the gunshot is played. Subsequently, the `CalculateRayCast` function is invoked, which will be defined later. This function returns a boolean, `isHit`, indicating whether an object in the scene has been hit, and `HitComponent`, which refers to the object hit or null if nothing was struck. If a hit is detected and the object is a shooting panel, the `Disappear` function from `BP_Scene` is called. In any case, the ammunition display is updated to reflect the decreased amount.

The image showing this is as follows:

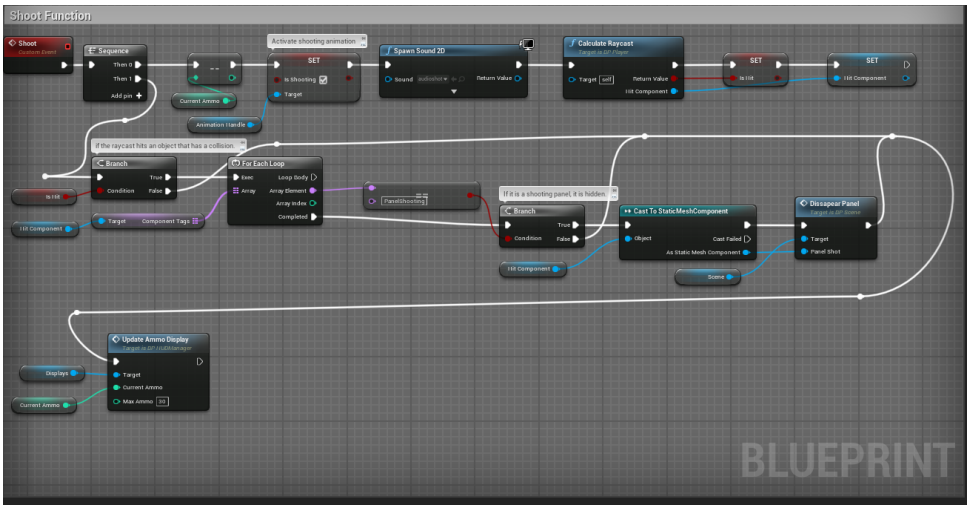

Figure 4.27: Shooting function in BP_player in Unreal

In the `CalculateRayCast` function, the camera position is obtained using `Get World Location` to determine the start point of the ray. For the end point, the unit direction vector of the camera is taken, multiplied by the weapon's range, and added to the initial camera position. Then, the `Line Trace By Channel` node is called, which returns a boolean indicating whether something has been hit, and an object called `hit`. Using the `break` node, the hit object is decomposed into its various attributes, and its `static mesh` component is obtained, as this will be required for later removal. The image showing this is as follows:

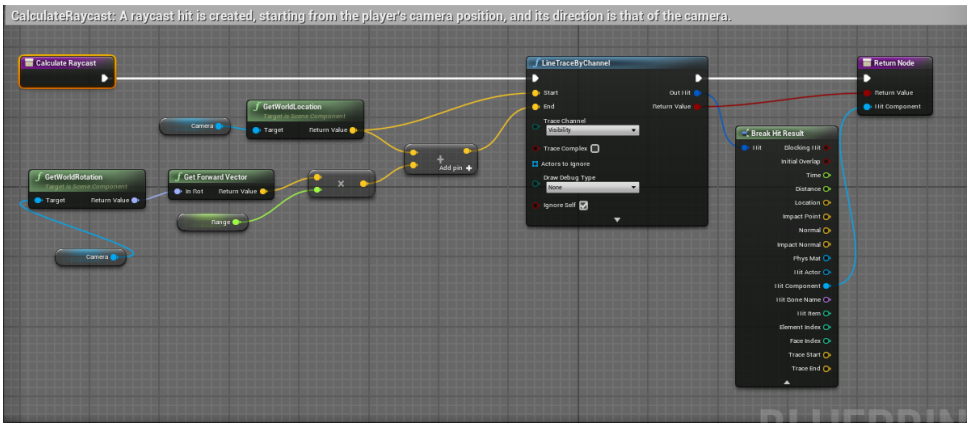

Figure 4.28: Calculate ray cast in BP_player in Unreal

- **Update Crosshair**: the same principle as with shooting is applied. In this case, if something is hit and the object is the shooting panel, the crosshair is updated to red (using a crosshair texture in PNG format). Otherwise, the crosshair remains white by default.
  The image showing this is as follows:

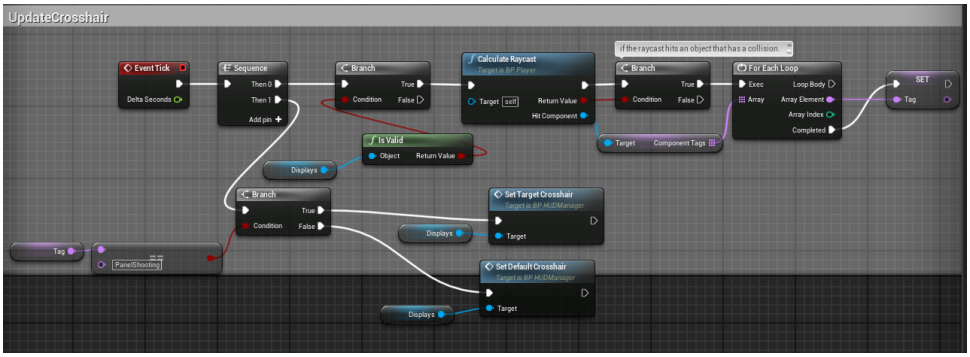

Figure 4.29: Update crosshair in BP_player in Unreal

BP_Scene

The blueprint for the scenario is responsible for managing the creation and removal of the shooting panels.

The behaviours and functions it contains are as follows:

- **Count Panels**: this function is called when the scenario is initially loaded (as seen in BP_GameModeFPS). Essentially, all the panels under the parent actor `Panels Shooting` are retrieved, counted using `remainingObjectives`, and assigned the tag "Panel_Shooting" for identification purposes. Afterwards, the HUD is updated.
  The image showing this is as follows:

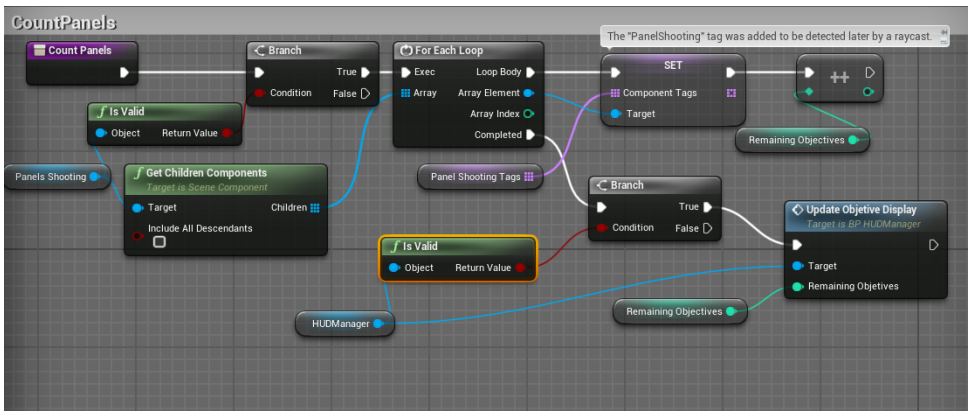

Figure 4.30: CountPanels in BP_Scene in Unreal

- **DissapearPanel**: unlike in Unity, when objects are hidden in Unreal, they continue to interact with the engine's physics system. As a result, the raycast still detects their colliders. Therefore, after hiding the object, its collider is disabled. Following this, the behaviour remains the same.
  The image showing this is as follows:

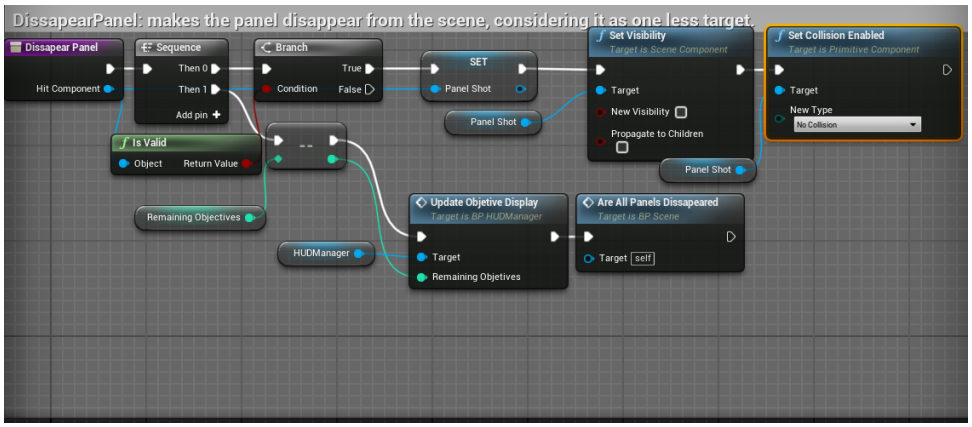

Figure 4.31: DissapearPanel in BP_Scene in Unreal

- **AreAllPanelsDissapeared**: as in Unity, the difference lies in which system is responsible for determining the player's victory — in this case, it is handled by BP_GameModeFPS

The image showing this is as follows:

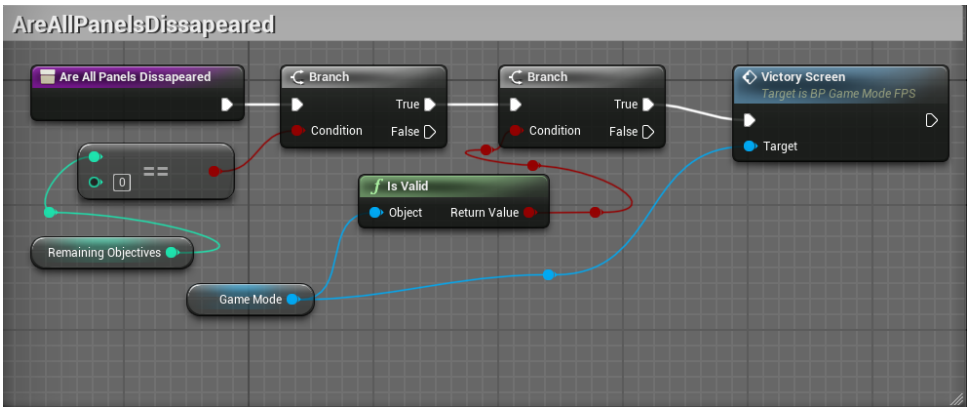

Figure 4.32: AllArePanelsDissapeared in BP_Scene in Unreal

BP_HUDManager

All HUD elements are managed here. Additionally, each of these elements is hidden individually as needed.

The behaviours and functions it contains are as follows:

- **Event Graph**: the click events for the restart and exit buttons are assigned here. The image showing this is as follows:

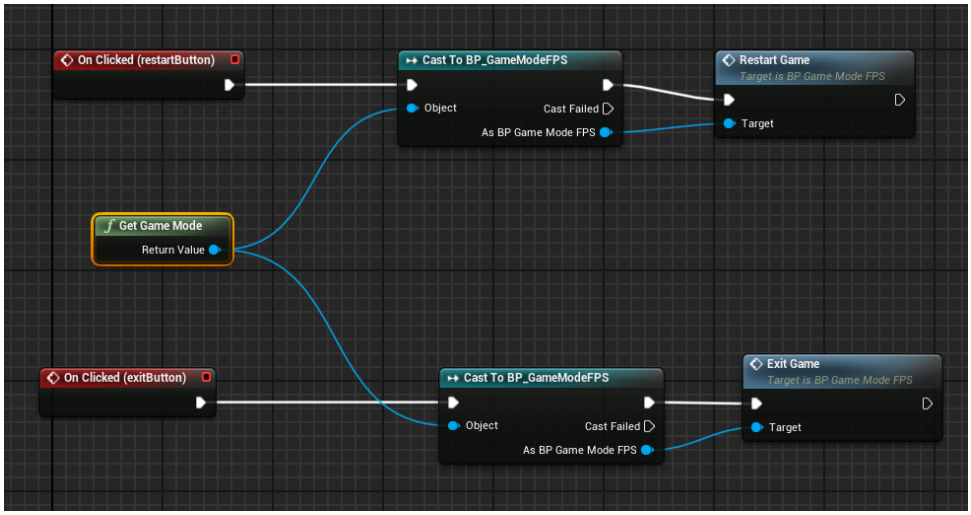

Figure 4.33: Event graph in BP_HUDManager in Unreal

- **UpdateAmmoDisplay and UpdateObjectivesDisplay**: notably, the `Format` node is used here to set the desired values within a text.
  The image showing this is as follows:

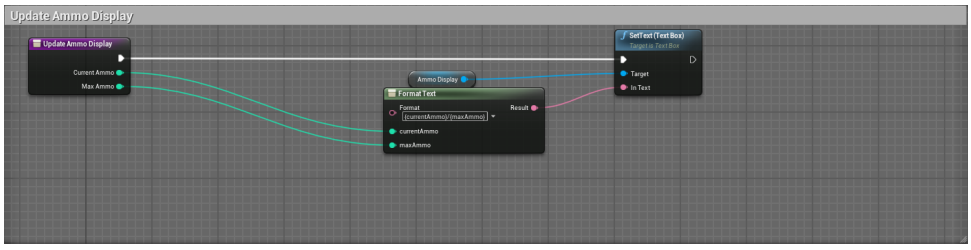

Figure 4.34: Update HUD example in BP_HUDManager in Unreal

- **ShowFinalMessage**: the victory message is displayed along with its associated elements. This function is called from `BP_GameModeFPS`.
  The image showing this is as follows:

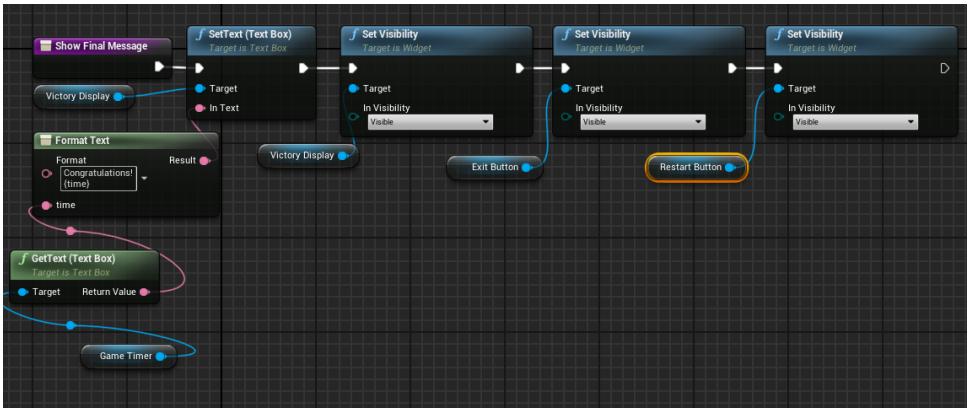

Figure 4.35: ShowFinalMessage in BP_HUDManager in Unreal

BP_GameModeFPS

The game restart and exit are controlled here, in addition to establishing the game victory.

The behaviours and functions it contains are as follows:

- **VariableSettings**: here, the HUD is created, and all the shooting panels are retrieved. Additionally, all references to the various blueprints are set.
  The image showing this is as follows:

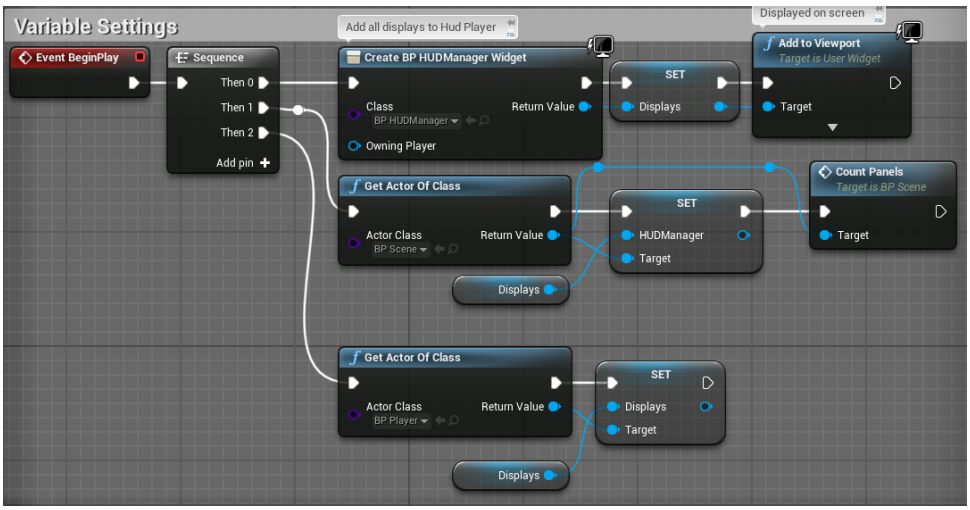

Figure 4.36: Variable settings in BP_GameModeFPS in Unreal

- **VictoryScreen**: the game is paused, the cursor is made visible, and the game elements are hidden. Then, the `ShowFinalMessage` function from `BP_HUDManager` is called.
  The image showing this is as follows:

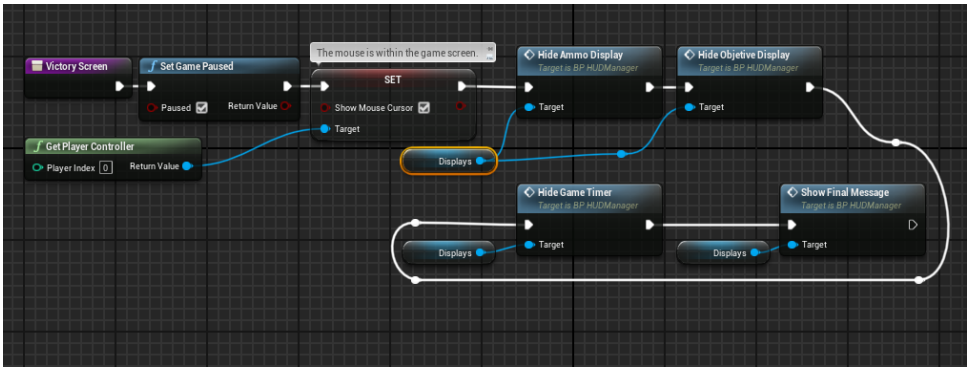

Figure 4.37: VictoryScreen in BP_GameModeFPS in Unreal

- **RestartGame**: the cursor is simply made visible again, the game is unpaused, and the level is reloaded.
  The image showing this is as follows:

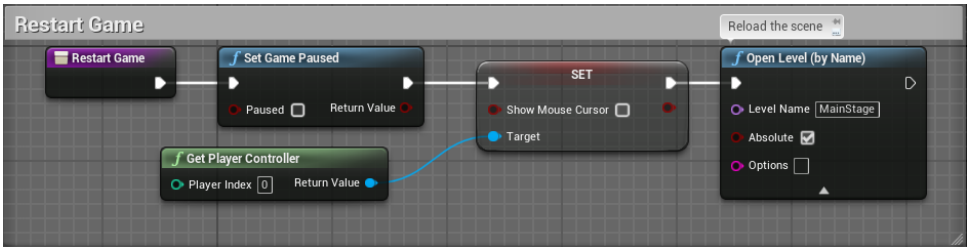

Figure 4.38: RestartGame in BP_GameModeFPS in Unreal

- **ExitGame**: it is simply a node that closes the application.
  The image showing this is as follows:

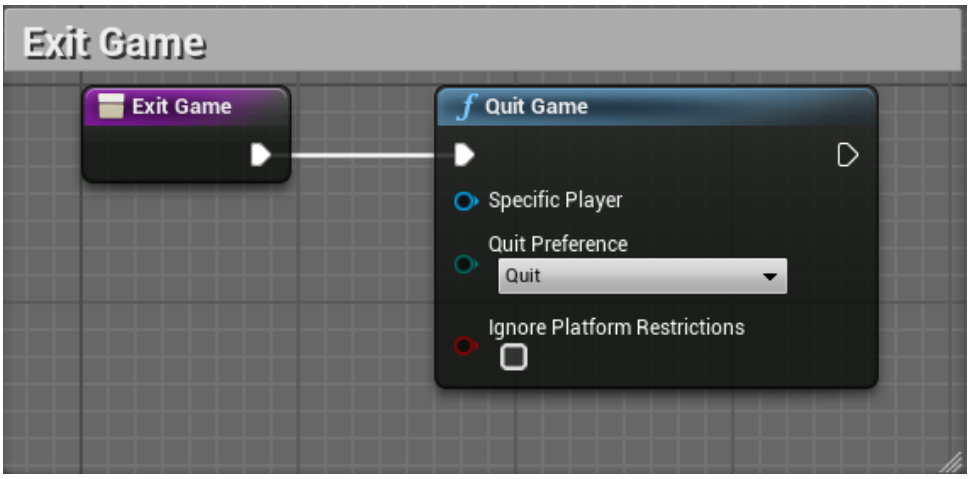

Figure 4.39: ExitGame in BP_GameModeFPS in Unreal

4.5 Godot

4.5.1 Introduction

Godot is the third game engine used to implement the FPS prototype in manual mode. Its role in this project is to provide a different perspective compared to more established engines such as Unity or Unreal, particularly in terms of lightweight design, open architecture, and ease of adaptation to projects requiring low resource consumption.

The implementation in Godot replicates the same prototype functionalities: player control via keyboard and mouse, collision system, HUD, animations, and shooting and reloading mechanics. This enables a comparison of how each engine addresses the same technical challenges and what advantages or limitations each one presents, especially in the context of low-performance systems or web-based deployment.

Version 4.4.1 of Godot was used, one of the most recent and stable releases at the time of development. This version introduces key improvements to the physics system, animations, and performance, which are particularly relevant when assessing the engine's viability for the development of a basic FPS like the one proposed in this study.

The following sections detail the environment configuration, collision system implementation, and the logic of the prototype developed in Godot, following the same structure used with the other engines.

4.5.2 Environment

The environment is divided into clearly defined sections, which are as follows:

- **Scene**: the place where different types of scene objects or different scenes can be selected. Unlike other engines, a scene in Godot does not have to represent a complete level; it can instead represent a single game element or a grouping of such elements, which can later be added to other scenes. In this case, the Scene 3D type has been used, which typically contains elements called `Node3D`, similar to Unity's GameObjects with affine 3D transformation. However, for the prototype, Canvas nodes — which are used for the HUD — have also been combined.
- **FileSystem**: similar to Unity's Project window.
- **Inspector and Nodes**: The properties of scene objects can be accessed and modified directly from the editor interface or through code. The Nodes panel allows the creation of functions that respond to signals and events, without requiring manual connections in code.
- **Viewport**:Scene elements can be interacted with from this area. The top toolbar allows switching between different editing modes:
  - 2D, for working with Canvas or `Node2D` elements
  - 3D, for editing `Node3D` elements
  - Script, to view and edit the code of various scripts
  - Game, to preview the project in execution
  - Asset Library, to import required assets or libraries

An overview of a Unity project is depicted in this image:

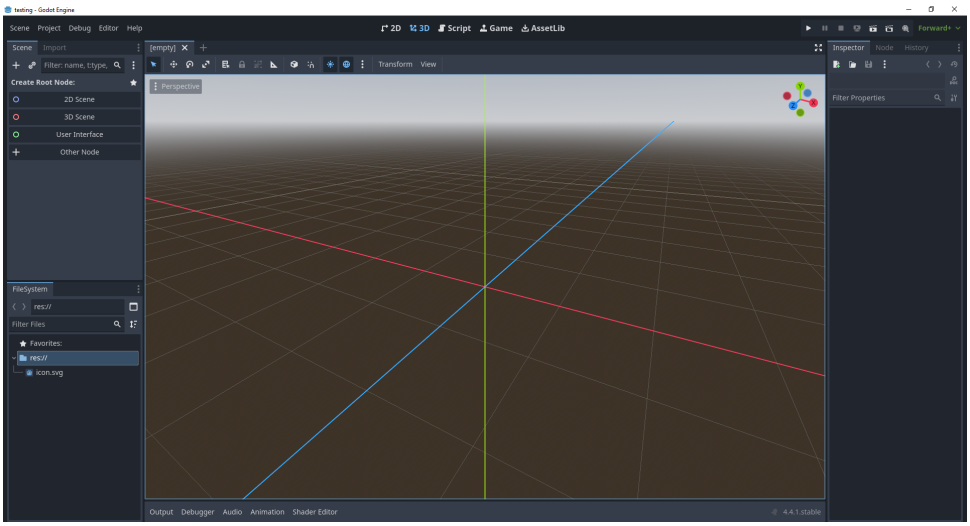

Figure 4.40: Godot environment

4.5.3 Collider implementation

In Godot, a hierarchical structure must be created for each scene object assigned a collider. For this purpose, the following three elements are defined:

- `MeshInstance3D`: node that represents a visual 3D object using a mesh. It is used to display 3D models, either imported (e.g., ġlb files) or primitives such as boxes, spheres, and cylinders. This node does not possess collision or physical behaviour on its own and typically acts as a parent or sibling of a physics node that contains the collision logic.
- `StaticBody3D`: node that represents a static physical body. It does not move or react to physical forces or collisions and serves as a fixed object in the scene, such as the ground or walls.
- `CollisionShape3D`: node that defines the geometric shape for collision detection. It does not possess physical behaviour on its own and must be associated as a child of a physical body

With this structure, it is possible to create the required collider. Colliders can be created manually for simple shapes or automatically for complex shapes. In this case, the latter is necessary for ramps, barrels, and similar objects. The following steps should be followed:

1. Select the `MeshInstance3D`.
2. In the Inspector, locate the Mesh property.
3. Ensure that a mesh is assigned (e.g., ġlb, or a `PrimitiveMesh`).
4. In the 3D editing view, use the top toolbar to select the Mesh option.
5. Window will appear with two options to choose from. The following selections should be made:
   - **Collision Shape placement**: choose “Static Body Child”
   - **Collision Shape Type**: choose “Trimesh”

With this, the collider is now assigned based on the object’s mesh and ready to interact with the player.

4.5.4 Code explanation

Godot features its own programming language, known as **GDScript**. It is a High-level, object-oriented and scripting language, with a syntax similar to that of `Python`[29].

As in Unity, scripts are typically attached to scene objects in order to access their references without the need to search through all nodes, unless strictly necessary.

he implemented functionality is as follows:

PlayerMovement.gd

Responsible for the character's movement, specifically its `CharacterBody3D`. In Godot, all 3D objects inherit from a single base class called `Node3D`. It has the ability to possess a transform, as well as to hide and show itself along with all its children.

Variable initialization is similar to that in Python; it can be implicit or explicit when type enforcement is required. Similar to JavaScript, `const` represents the program's constants, while `var` denotes variables that will change throughout the execution.

It must be taken into account that the game objects' lifecycle is important. Since the order in which objects are created in the scene is unknown, and as these objects may reference others—resulting in the possibility that a reference may be null when accessed—the `@onready` tag is used. This tag ensures that the object is not referenced until the `_ready()` function is called, which will be explained later.

There are two ways to reference an object in Godot. One is by using the object's absolute path within the FileSystem through the `get_node(object_localization: String)` function. The other is by using the $ symbol, which allows the use of a relative path that accesses the child nodes of the current object directly.

The `_ready()` function corresponds to Unity's `Start()` function; it initialises the object associated with the script. In the case of this object, the only operation performed is to verify that the referenced objects are not null. To this end, a script named `Utils.gd` has been created, which is responsible for performing this verification. The method by which this is done will be explained later.

The `physics_process()` function, inherited from `Node`, is executed during each lifecycle iteration of the object. It does not behave like Unity's `Update()` (that role is fulfilled by `_process()` in Godot, which will be discussed later)as it runs at a constant rate independent of the framerate. This makes it particularly suitable for simulating physics, collisions, or realistic movement, as is required in this case.

The character is simply moved using the same underlying logic as in Unity: input is received, the movement direction on the horizontal plane is calculated and multiplied by the speed. A similar approach is applied for gravity. After this, the weapon's animation states (walking or idle) are updated accordingly.

Subsequently, the function `is_panel_shooting()` is called to verify whether the aim is directed at a shooting panel. If confirmed, the crosshair changes to red by invoking the hudmanager.gd script, which will be detailed later.

The next function, `_unhandled_input(event)`, detects events of any kind. In this case, if the event corresponds to mouse movement, the function `look_around()` is invoked, following the same principle as that implemented in Unity. It should be clarified that the use of quaternions is not necessary in Godot, as this engine does not suffer from the gimbal lock issue. Rotation using degrees is sufficient.

The implemented code is as follows:

```
extends CharacterBody3D

var utils = load("res://assets/scripts/Utils.gd")

const GRAVITY : float = 9.8
const LOOK_SENSITIVITY : float = 0.5    # Mouse rotation speed
const CAMERA_PITCH_LIMIT : float = 80.0

var speed_xz : float = 2.0
var speed_y : float = 0.0   # Gravity

var yaw : float = 0.0       # Mouse movement speed on the X-axis
var pitch : float = 0.0     # Mouse rotation speed on the Y-axis

@onready var camera_player = $CameraPlayer
# A raycast hit is created, starting from the player's camera position, and its direction is that of the camera.
@onready var raycast_player = $CameraPlayer/RayCastPlayer
@onready var hud_manager = get_node("/root/MainStage/HUD")
@onready var weapon = $CameraPlayer/Weapon

func _ready():
    # Null checker
    utils.check_node_validity(camera_player, "cameraPlayer")
    utils.check_node_validity(raycast_player, "raycastPlayer")
    utils.check_node_validity(hud_manager, "HUD")
    utils.check_node_validity(weapon, "Weapon")

func _physics_process(delta):
    move_player(delta)
    is_panel_shooting()

func _unhandled_input(event):
    if event is InputEventMouseMotion:
        look_around(event)

func move_player(delta_time : float):

    var input_x = Input.get_action_strength("move_left") - Input.get_action_strength("move_right")
    var input_z = Input.get_action_strength("move_forward") - Input.get_action_strength("move_back")

    var direction = (transform.basis.x * input_x + transform.basis.z * input_z).normalized()

    velocity.x = direction.x * speed_xz
    velocity.z = direction.z * speed_xz

    # Apply gravity to the vertical speed
    if not is_on_floor():
        speed_y -= GRAVITY * delta_time
    else:
        speed_y = 0.0
    velocity.y = speed_y

    move_and_slide()
    var magnitude = velocity.length()
    if magnitude > 0:
        weapon.set_walking_animation(true)
    else:
        weapon.set_walking_animation(false)



func is_panel_shooting():
    var is_target_panel_shooting : bool = false
    if raycast_player.is_colliding():
        var target = raycast_player.get_collider()
        is_target_panel_shooting = target.is_in_group("panel_shooting")
    hud_manager.update_crosshair(is_target_panel_shooting)

```

```
69 func look_around(event : InputEventMouseMotion):
70     # Rotation on the X-axis (left/right)
71     yaw -= event.relative.x * LOOK_SENSITIVITY
72     # Rotation on the Y-axis (up/down) clamped to the +/- CAMERA_PITCH_LIMIT
73     pitch -= event.relative.y * LOOK_SENSITIVITY
74     pitch = clamp(pitch, -CAMERA_PITCH_LIMIT, CAMERA_PITCH_LIMIT)
75 
76     # Rotate only the body on the Y-axis (yaw)
77     rotation_degrees.y = yaw
78 
79     # Rotate only the camera on the X-axis (pitch)
80     camera_player.rotation_degrees.x = pitch
```

WeaponController.gd

Responsible for the weapon's functions and its animations

The weapon's import from Sketchfab results in unusual animation names. To minimise alterations to the import, a dictionary has been created where the key corresponds to the weapon's state and the value is the name of the respective animation in the import.

Within the `_ready()` function, null values are checked and the weapon's initial animation, draw, is triggered. Upon completion of this animation (handled via a signal that subsequently invokes the `animation_weapon_finished()` function) the HUD is called to update the ammunition display. However, it was identified during development that the HUD had not yet been initialised or constructed, resulting in a null value. To resolve this issue, the `call_deferred()` function is used. This function defers execution until all scene objects have been fully instantiated. Once this condition is met, the target function can be called by passing its name along with the required parameters.

Unlike `physics_process()`, the `_process()` function is framerate-dependent, as realistic behaviour is not required for this particular functionality. Its operation remains consistent with the original design. However, in this case, the weapon's animation state machine cannot be implemented robustly using the engine's built-in tools. As a result, it has been manually and minimally modelled.

The first condition to check is whether the draw animation has completed, as no further actions can be performed while it is in progress. This is represented by the `is_drawing` variable. Subsequently, it is verified whether the weapon is in a reloading or firing state. If neither condition is met, the remaining possibility is that the weapon is either idle or in motion. This ensures that animations are neither interrupted nor overlapped.

Reloading operates in the same manner; the only difference is that, upon completion of the animation, a signal is emitted to invoke the `animation_weapon_finished()` function, which handles the remaining functionality. As for firing, the behaviour is standard: a Raycast is created, and if it strikes a panel, the panel is removed.

The implemented code is as follows:

```
extends Node3D

var utils = load("res://assets/scripts/Utils.gd")

const BULLETS_MAX: int = 30
const MS_TO_SECONDS: float = 1000.0

var fire_rate: float = 0.35             # Time between shots, It is dependent on the shooting animation.
var current_ammo: int = BULLETS_MAX
var is_reloading: bool = false      # Prevent firing while reloading or to prevent reloading from repeating.
var is_drawing: bool = true
var is_walking: bool = false
var next_fire_rate: float = 0.0     # Time in which shooting will be allowed again.
var elapsed_time : float = 0.0

@onready var hud_manager = get_node("/root/MainStage/HUD")
@onready var weapon_events = $AnimationPlayer       # Weapon animation event handler.
# A raycast hit is created, starting from the player's camera position, and its direction is that of the camera.
@onready var raycast_player = get_parent().get_node("RayCastPlayer")
@onready var panels_manager = get_node("/root/MainStage/Scene/scene/Panels Shooting")
@onready var shoot_sound = get_node("ShootSound")

var animation_names = {
    "draw": "Armature_003|draw",
    "idle": "Armature_003|idle",
    "walking": "Armature_003|walk",
    "shooting": "Armature_003|shooting",
    "reload": "Armature_003|reload"
}

func _ready():
    # Null Checker
    utils.check_node_validity(hud_manager, "HUD")
    utils.check_node_validity(weapon_events, "AnimationPlayer")
    utils.check_node_validity(raycast_player, "raycastPlayer")
    utils.check_node_validity(panels_manager, "Panels Shooting")
    utils.check_node_validity(shoot_sound, "ShootSound")

    weapon_events.play(animation_names.draw)
    # The HUD is waited for to be loaded in the scene to avoid a NullReferenceException.
    hud_manager.call_deferred("update_ammo_display", current_ammo, BULLETS_MAX)

func _process(_delta):
    # Prevent any animation from playing during the weapon deployment.
    if is_drawing:
        return
    # Reloading when desired with R button or when the bullets run out.
    # Reloading is avoided multiple times at the same time.
    if (Input.is_action_just_pressed("reload") or current_ammo == 0) and not is_reloading and current_ammo != BULLETS_MAX:
        reload()

    # The shot is fired by holding down the left mouse button and
    # it is ensured that it does not reload at the moment
    # in addition, Shooting is only allowed if the waiting time has passed.
    if Input.is_action_pressed("shoot") and not is_reloading:
        elapsed_time = Time.get_ticks_msec() / MS_TO_SECONDS
        if elapsed_time >= next_fire_rate:
            next_fire_rate = elapsed_time + fire_rate
            shoot()

    # If it's neither firing nor reloading, then it can only be either idle or walking.
    update_movement_animation()


func shoot():
    weapon_events.play(animation_names.shooting)
    # A raycast hit is created, starting from the player's camera position, and its direction is that of the camera.
```

```
68      if raycast_player.is_colliding():
69          var target = raycast_player.get_collider()
70          if target.is_in_group("panel_shooting"):
71              panels_manager.disappear_panel(target)
72
73      current_ammo -= 1
74      hud_manager.update_ammo_display(current_ammo, BULLETS_MAX)
75
76      shoot_sound.play()
77
78  func reload():
79      is_reloading = true
80      weapon_events.play(animation_names.reload)
81
82  func set_walking_animation(active: bool) -> void:
83      is_walking = active
84
85  func update_movement_animation():
86      # Only proceed if the current animation is not shooting or reloading,
87      # to avoid interrupting those critical animations and
88      # to ensure there's no overlapping
89      if weapon_events.current_animation not in [animation_names.shooting, animation_names
            .reload]:
90          if is_walking:
91              weapon_events.play(animation_names.walking)
92          else:
93              weapon_events.play(animation_names.idle)
94
95  # This function is automatically called when an animation finishes playing
96  func animation_weapon_finished(anim_name: StringName) -> void:
97      match anim_name:
98          animation_names.draw:
99              is_drawing = false
100         animation_names.reload:
101             is_reloading = false
102             current_ammo = BULLETS_MAX
103             hud_manager.update_ammo_display(current_ammo, BULLETS_MAX)
```

PanelsManager.gd

There are no differences from the previously established approach. Null values are still checked, and `call_deferred()` is used to delay the function call. As in Unreal, visually removing objects does not automatically remove their colliders, so they are explicitly disabled.

The implemented code is as follows:

Responsible for removing the panels and checking whether any targets remain.

```
extends Node3D

var utils = load("res://assets/scripts/Utils.gd")

var total_objectives: int = 0
var remaining_objectives: int = 0
@onready var hud_manager = get_node("/root/MainStage/HUD")

func _ready():
    # Null checker
    utils.check_node_validity(hud_manager, "HUD")
    # Gets the number of active panels in the scene
    total_objectives = get_child_count()
    remaining_objectives = total_objectives
    hud_manager.call_deferred("update_objective_display", remaining_objectives)

# A function that makes the panel disappear from the scene, considering it as one less target.
# @param panel_shot, The panel to disappear.
func disappear_panel(panel_shot: StaticBody3D):
    # Null checker
    utils.check_node_validity(panel_shot, "Panel Shooting")
    panel_shot.get_parent().hide()
    panel_shot.get_node("CollisionShape3D").disabled = true
    remaining_objectives -= 1
    hud_manager.update_objective_display(remaining_objectives)
    _check_if_all_panels_disappeared()

func _check_if_all_panels_disappeared():
    if remaining_objectives == 0:
        hud_manager.victory()
```

HUDManager.gd

Management includes both the HUD elements and the handling of player victory, game reset, and exit procedures.

The HUD functionality remains consistent with previous implementations: updating ammunition, displaying the number of targets, and showing elapsed time.

In the event of a game victory, `get_tree()` is called to retrieve the current scene, allowing it to be paused and enabling decisions regarding restart or exit. The operation of the exit and restart calls is managed via events sent from mouse input to the HUD buttons.

The implemented code is as follows:

```
extends CanvasLayer

var utils = load("res://assets/scripts/Utils.gd")

var elapsed_time : float = 0.0

@onready var crosshair = $Crosshair
@onready var default_crosshair_texture = preload("res://assets/textures/defaultCrosshair.png")
@onready var target_crosshair_texture = preload("res://assets/textures/targetCrosshair.png")
@onready var ammo_display = $AmmoDisplay            # The ammunition indicated in the player's HUD.
@onready var objective_display = $ObjetiveDisplay   # The number of targets that remain alive.
@onready var game_timer = $GameTimer                # the game Time indicated in the player's HUD.
@onready var victory_panel = $VictoryPanel      # A menu that is enabled when all the targets are eliminated.

func _ready():
    # Null checker
    utils.check_node_validity(crosshair, "crosshair")
    utils.check_node_validity(ammo_display, "ammoDisplay")
    utils.check_node_validity(objective_display, "objetiveDisplay")
    utils.check_node_validity(game_timer, "gameTimer")
    utils.check_node_validity(victory_panel, "victoryPanel")

    utils.check_child_exists(victory_panel, "VBoxContainer/VictoryText")
    utils.check_child_exists(victory_panel, "VBoxContainer/GameTimerVictory")
    utils.check_child_exists(victory_panel, "VBoxContainer/RestartButton")
    utils.check_child_exists(victory_panel, "VBoxContainer/ExitButton")

    # Hide the cursor and lock it to the centre of the screen.
    Input.set_mouse_mode(Input.MOUSE_MODE_CAPTURED)
    victory_panel.hide()
    ammo_display.show()
    objective_display.show()
    game_timer.show()

func _process(delta):
    elapsed_time += delta
    update_game_timer()

func update_crosshair(is_panel_shooting: bool):
    if is_panel_shooting:
        crosshair.texture = target_crosshair_texture
    else:
        crosshair.texture = default_crosshair_texture

func update_ammo_display(current_ammo: int, max_ammo: int):
    ammo_display.text = "%d / %d" % [current_ammo, max_ammo]


func update_objective_display(remaining_objectives: int):
    objective_display.text = "Remaining Objetives: %d" % [remaining_objectives]

```

```
52
53 func update_game_timer():
54     # Convert the total time into a digital clock format.
55     var minutes : int = int(elapsed_time / 60)
56     var seconds : int = int(elapsed_time) % 60
57     game_timer.text = "Time: %02d:%02d" % [minutes, seconds]
58
59 # A function that manages the player's victory when all the targets have been eliminated
       .
60 func victory():
61     # Paused game
62     get_tree().paused = true
63
64     # The game elements are hidden.
65     crosshair.visible = false
66     ammo_display.visible = false
67     objective_display.visible = false
68     game_timer.visible = false
69
70     # The game time is taken and displayed with a victory message.
71     victory_panel.get_node("VBoxContainer/VictoryText").text = "Congratulations!"
72     victory_panel.get_node("VBoxContainer/GameTimerVictory").text = game_timer.text
73     victory_panel.show()
74
75     # The mouse is within the game screen
76     Input.set_mouse_mode(Input.MOUSE_MODE_VISIBLE)
77
78 func exit() -> void:
79     get_tree().quit()   # Close Game
80
81 func restart() -> void:
82     get_tree().paused = false # Resume game
83     get_tree().reload_current_scene() # Reload Scene
```

Utils.gd

This script is responsible for performing basic checks to ensure that nodes and their children exist and are not null at runtime.

- **check_node_validity**: verifies that a node is not null. If it is, an error is thrown and execution is halted in debug mode. This helps detect broken or improperly initialized references.
- **check_child_exists**: checks that a specific child node exists within a given parent node. If it does not, an error is also thrown and execution is paused in debug mode. This ensures the scene structure is as expected.

The implemented code is as follows:

```
extends Node

# Validates that a node is not null; if it is, shows an error and pauses execution in debug mode.
static func check_node_validity(node: Object, nodeName: String) -> void:
    if node == null:
        push_error("%s is null!" % nodeName)
        assert(false)

# Validates that a child node exists within a parent node and pauses execution in debug mode if not found.
static func check_child_exists(parent: Node, child_path: String) -> void:
    if not parent.has_node(child_path):
        push_error("Child node '%s' not found in '%s'" % [child_path, parent.name])
        assert(false)
```

This chapter has detailed the manual implementation of the FPS prototype in Unity, Unreal Engine, and Godot, analysing the environment setup, collision system, and associated code in each engine. Using the manual prototype as a foundation, the following chapter will present the implementation in automatic mode, where the player does not directly control the character but instead navigation and NPC behaviour systems are employed to simulate gameplay and conduct performance testing.

Chapter 5

Implementation (automatic mode)

This chapter describes the implementation of the automatic control system, in which the player is replaced by an NPC that navigates the environment autonomously, without user input. This functionality is essential for simulating realistic behaviours in interactive environments, such as patrolling, obstacle avoidance, or free movement across a closed map.

The starting point was the manual prototype described in the previous chapter, which enabled direct control of the character using keyboard and mouse. However, to conduct tests in closed environments in a more controlled, reproducible, and scalable manner, it became necessary to develop an automated system. The manual prototype is retained as a reference and as a useful tool for future performance tests or validation scenarios that may require direct user interaction.

Based on this foundation, three equivalent implementations were developed using Unity, Unreal Engine, and Godot. In each engine, two core components were addressed: the generation and use of a navigation mesh (NavMesh), and the logic that enables the NPC to move automatically by following the NavMesh as a guide.

5.1 Prototipe development

After establishing the prototype using the manual model, the next step is to develop the automated version for execution during performance testing. To achieve this, a non-player character NPC in first person will be created, with its movement controlled by AI (further details on the implementation will be provided later). The proposed requirements are as follows:

1. The scenario will include a set of points referred to as waypoints. These waypoints will be positioned near the shooting panels and will serve as the targets towards which the NPC must move.
2. Upon reaching the target, regardless of its current facing direction, the NPC will gradually adjust its aim towards the objective, simulating human-like behaviour
3. Once the NPC is aiming in the direction of the shooting panel, the weapon will behave identically to the manual mode. After the target has been eliminated, the NPC must move to the next waypoint and repeat the cycle.
4. The HUD elements are the same as in manual mode.
5. The victory conditions are the same as those in manual mode.

5.2 General Structure in Game Engine

Before diving into the specific implementation details of each engine, it is important to provide an overview of how the involved technologies interact and the role each plays within the system.

The project centres on simulating autonomous movement of character NPC within a virtual environment. To this end, three widely used game engines have been selected: Unity, Unreal Engine, and Godot. Each of these engines offers its own set of tools, libraries, and APIs to manage navigation and character control.

The general structure of the system in each engine follows a similar pattern, divided into two main components:

- **Navigation Mesh (NavMesh):**
  This represents the navigable space within the map. The NavMesh is either generated automatically or manually configured depending on the engine, and it defines the areas where NPCs can move. This mesh is fundamental for pathfinding and avoiding static obstacles (elements that do not move within the environment). Further details on this concept will be covered in the append B.
- **NPC Control and Logic:**
  Building on the foundation provided by the NavMesh, the logic responsible for autonomous NPC control is developed. This includes route planning to enable characters to move towards specific points in the scene, as well as managing behaviours such as aiming, shooting at targets, and reacting to obstacles.

Thus, although each engine employs its own tools and methods, the core concept is shared: first define where the NPC can move (NavMesh), and then how it behaves and moves within that space (control and logic). This global perspective will facilitate later comparison and evaluation of each engine in terms of ease of use, performance, and flexibility.

5.3 Unity

The implementation in Unity utilises its native Navigation Mesh system to define the navigable space, facilitating pathfinding and obstacle avoidance. NPC logic is developed in C#, enabling seamless integration with the engine's tools to control movement and autonomous behaviours. Version **6000.0.40f1** is employed to ensure a stable and reproducible environment for development and testing.

5.3.1 Navigation mesh implement

In Unity, the generation of the navigation mesh is carried out using a component called `NavMesh Surface`. This component is responsible for constructing the geometry of all elements tagged as "walkable" by default. Unity calculates the geometry in two ways: either using the meshes or using the colliders of the game objects assigned to the walkable layer. For this prototype, it has been decided to use the meshes due to the presence of complex shapes.

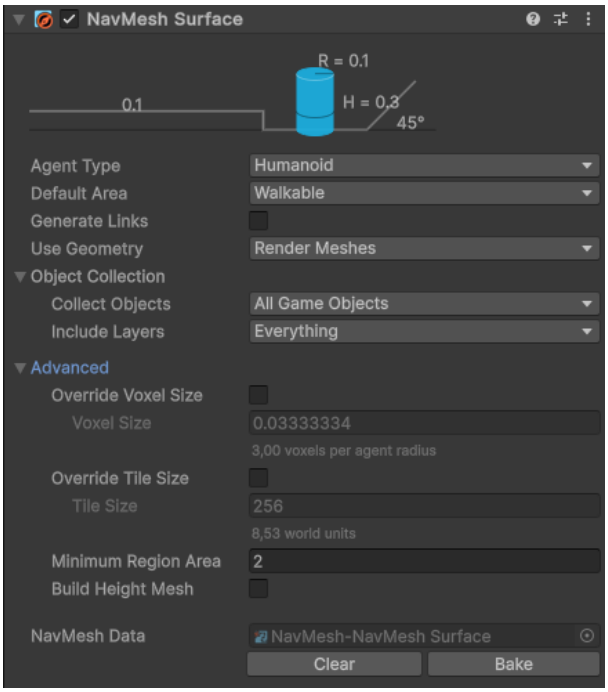

Figure 5.1: Navigation Mesh Properties in Unity

Other parameters to consider include the shape of the agent, which is a CharacterController with additional properties, among which are:

1. **Radius and Height**: relating to the shape of the agent's CharacterController
2. **Step Height**: it is the maximum height that an agent can *step up* onto, similar to a stair or curb.
3. **Max Slope**: it is the maximum slope angle of the terrain that the agent can walk on.

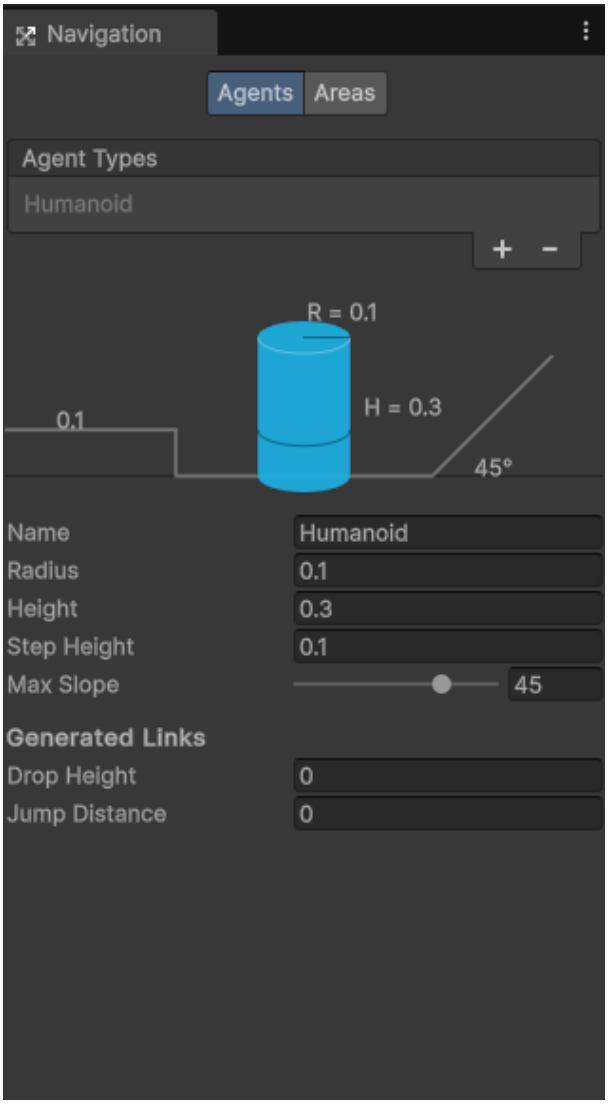

Figure 5.2: Navigation Agent Properties in Unity

After setting the parameters, Unity calculates it and displays the shape of the navigation mesh in the viewport using the colour blue:

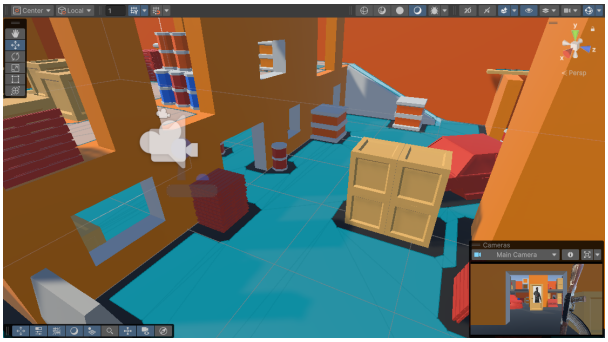

Figure 5.3: Navigation mesh result in Unity

5.3.2 NPC implement

The implementation in Unity, as with the manual mode, will use a script inheriting from `MonoBehaviour` that performs this behaviour, which will be called Patrol.cs

1. In `Start()`, the dependencies are obtained: The `GameController` is a class that has a public attribute called `automaticMode`, which can be enabled or disabled in Unity's inspector. This allows switching between game modes. If the game is not in automatic mode, both the agent and the script itself are disabled. This is because the agent and the player share the same `GameObject`, which has a `CharacterController` component to represent the collider. The `GameObject` also includes a `NavMeshAgent` component, which behaves similarly to a `CharacterController` but is constrained to the navigation mesh. Since both components act as colliders, they can overlap, potentially preventing any movement or causing erratic behaviour.
   Therefore, the agent is disabled. Additionally, by disabling the script, Unity halts its execution cycle—specifically, `Update()` will no longer be called.
   Once the objects have been initialised, if the game is in automatic mode, the agent is instructed to move to the position of the assigned waypoint.
2. In `Update()`, for each frame, if the NPC is indeed moving, a desired rotation is calculated using `Quaternion.LookRotation(Vector3 forward)` to face the direction of movement. The player's camera `playerCamera` is then smoothly rotated towards this direction using `Quaternion.Slerp (Quaternion a, Quaternion b, float t)`, which performs spherical interpolation for a smooth and natural result. The factor `Time.deltaTime * cameraRotateSpeed` controls the speed of the rotation, ensuring it remains frame-rate independent. This simulates human behaviour, as if a mouse were being used to look around. The weapon's animation is then updated while in the walking state.
   Finally, if the agent is not currently aiming, the NavMeshAgent has finished calculating the path (i.e., it is not pending), and is already very close to the current destination (waypoint), the coroutine `AimShotAndMove()` is invoked. This accounts for cases where the agent might never reach the exact target position due to the precision limitations of the `NavMeshSurface`.
3. The function `AimShotAndMove()` manages the entire NPC behaviour cycle upon reaching a waypoint. It begins by setting the `isAiming` flag to true, preventing the aiming routine from being triggered multiple times concurrently. Then, the `Aim()` coroutine is called and awaited, ensuring progression halts until the camera is properly aligned with the target. Once aiming is complete, the weapon's `Shoot()` method is invoked. Subsequently, the NPC moves to the next waypoint via `MoveToNextWaypoint()`. Finally, the `isAiming` flag is reset to false, allowing the process to repeat as necessary during subsequent patrol cycles.
4. The function `Aim()`, This routine is responsible for smoothly aligning the NPC's camera with the current target. It works by calculating the direction vector from the camera to the target and determining the required rotation using `Quaternion.LookRotation(Vector3 forward)`. The angular difference (angle) between the current rotation and the target rotation is then measured using `Quaternion.Angle(Quaternion a, Quaternion b)`. Within a do-while loop, the camera's rotation is interpolated each frame using `Quaternion.Slerp(Quaternion a, Quaternion b, float t)`, allowing a smooth and natural rotation over time. After each interpolation step, the routine yields for one frame to allow the update. The process continues until the angle between the current and target rotation falls below a predefined threshold (`ERROR_MARGIN_ANGLE`), at which point the camera is considered sufficiently aligned. This approach ensures visually pleasing, -rate-independent movement that simulates human-like behaviour when turning to face a target.
5. Finally, the function `MoveToNextWaypoint()` assigns the next waypoint to the agent, ensuring continuous movement along the path. If the agent reaches the final waypoint in the list, the index is reset to the beginning, effectively creating a circular patrol route.

The implemented code is as follows:

```
using System;
using System.Collections;
using System.ComponentModel;
using UnityEngine;
using UnityEngine.AI;

public class Patrol : MonoBehaviour
{
    private const float ERROR_MARGIN = 0.1f;        // Due to physics, idle time in Unity is not strictly 0
    private const float MIN_DISTANCE = 0.1f;        // The minimum distance for the waypoint to be considered reached.
    private const float ERROR_MARGIN_ANGLE = 1.0f;  // The maximum allowed angular deviation (in degrees) when comparing orientations.

    public NavMeshAgent agent;      // The NPC's navigation agent used for pathfinding and movement.
    public Transform[] waypoints;   // The points that define the predefined path the agent follows.
    public Transform[] targets;

    public float cameraRotateSpeed = 3f;
    public float range = 100f;

    private int destinationPoint = 0;
    private bool isAiming = false;
    private WeaponController weapon;
    private Animator weaponEvents;
    private Camera playerCamera;
    private GameController gameController;

    void Start()
    {
        gameController = GameObject.Find("Game Controller").GetComponent<GameController>();
        ValidationUtils.CheckNotNull(gameController, "gameController script is missing.", this);
        // Skip this update cycle
        if (!gameController.automaticMode)
        {
            // The NavMeshAgent must be disabled during manual mode because having both a Collider
            // (through CharacterController) and a NavMeshAgent enabled at the same time is not supported during runtime
            agent.enabled = false;
            this.enabled = false;
        }

        weapon = GetComponentInChildren<WeaponController>();
        weaponEvents = GetComponentInChildren<Animator>();
        playerCamera = GetComponentInChildren<Camera>();


        ValidationUtils.CheckArrayLengths(waypoints, targets, "Waypoints and targets validation failed.", this);

        for (int i = 0; i < waypoints.Length; i++)
        {
            ValidationUtils.CheckNotNull(waypoints[i], $"Waypoint at index {i} is not assigned.", this);
            ValidationUtils.CheckNotNull(targets[i], $"Target at index {i} is not assigned.", this);
        }

        ValidationUtils.CheckNotNull(weapon, "WeaponController component is missing.", this);
        ValidationUtils.CheckNotNull(weaponEvents, "Animator component for weapon events is missing.", this);
        ValidationUtils.CheckNotNull(agent, "NavMeshAgent component for agent is missing", this);
        ValidationUtils.CheckNotNull(playerCamera, "Camera component is missing.", this);

        if (gameController.automaticMode)
            agent.destination = waypoints[destinationPoint].position;
    }

    void Update()
    {
        Vector3 Agentdirection = agent.velocity;
        Agentdirection.y = 0;

        if (Agentdirection.magnitude > ERROR_MARGIN)
        {
            Quaternion agentRotation = Quaternion.LookRotation(Agentdirection);
            // The camera is smoothly rotated towards the agent's direction while moving.
            playerCamera.transform.rotation = Quaternion.Slerp(playerCamera.transform.rotation, agentRotation, Time.deltaTime * cameraRotateSpeed);
        }

        weaponEvents.SetFloat("movement", Agentdirection.magnitude);

        // Only proceed if the NavMeshAgent has finished calculating the path
        // and the agent is very close to its current destination
```

```
78          if (!isAiming && !agent.pathPending && agent.remainingDistance < MIN_DISTANCE)
79              StartCoroutine(AimShotAndMove());
80      }
81
82      private IEnumerator AimShotAndMove()
83      {
84          isAiming = true;
85
86          // Wait until aiming is complete
87          yield return StartCoroutine(Aim());
88          weapon.Shoot();
89
90          MoveToNextWaypoint();
91
92          isAiming = false;
93      }
94
95      private IEnumerator Aim()
96      {
97          Vector3 targetPos;
98          Vector3 targetDir;
99          Quaternion targetRot;
100         float angle;
101
102         do
103         {
104             targetPos = targets[destinationPoint].position;
105             targetDir = targetPos - playerCamera.transform.position;
106
107             // Determine the rotation needed to look at the target
108             targetRot = Quaternion.LookRotation(targetDir);
109
110             // Computes the angular difference between the current orientation and the target
                    orientation
111             angle = Quaternion.Angle(playerCamera.transform.rotation, targetRot);
112
113             // Smoothly rotate the camera towards the target over time
114             playerCamera.transform.rotation = Quaternion.Slerp(playerCamera.transform.rotation,
                    targetRot, Time.deltaTime * cameraRotateSpeed);
115
116             // Allowing the aiming animation to update smoothly over
117             yield return null;
118
119         } while (angle > ERROR_MARGIN_ANGLE);
120     }
121
122     private void MoveToNextWaypoint()
123     {
124         destinationPoint = (destinationPoint + 1) % waypoints.Length;
125         agent.destination = waypoints[destinationPoint].position;
126     }
127 }
```

5.4 Unreal Engine

In Unreal Engine, the implementation relies on the automatic generation of Navigation Meshes that allow precise navigation in complex environments. NPC behaviour logic is created exclusively through Blueprints, providing an intuitive and efficient visual toolset to design autonomous control without traditional programming. Version **4.27.2** is used to take advantage of navigation improvements and performance optimisation.

5.4.1 Navigation mesh implement

In Unreal Engine, a `Nav Mesh Bounds Volume actor` must be created to generate the navigation mesh. By default, this volume automatically creates another actor named `RecastNavMesh`, which is responsible for handling all necessary parameters related to navmesh generation. Unlike Unity, Unreal's navigation mesh considers only the colliders (collision geometry) of placed objects during generation. An overview of the key parameters involved in this setup is as follows:

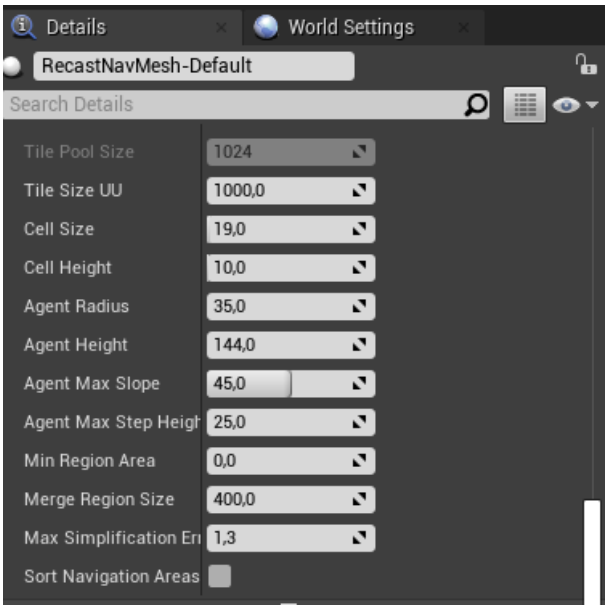

Figure 5.4: Navigation mesh Properties in Unreal

After the necessary configurations have been applied, Unreal Engine automatically generates the navigation mesh. However, it is also possible to manually trigger its generation by selecting `Build Paths` from the `Build` menu at the top of the viewport. This results in the creation of the navmesh, which appears in the editor as a green surface, indicating walkable areas for AI agents:

Figure 5.5: Navigation mesh result in Unreal

5.4.2 NPC implement

For the NPC implementation, two new Blueprints have been created to represent independent actors, along with a modification to the existing `BP_GameModeFPS` blueprint.

BP_NPC_AIController

This Blueprint is responsible for handling the NPC's movement along the navigation mesh. As usual, the internal workings are not of concern, since the Blueprint itself automatically calculates the path to the defined waypoints.

MoveToNextWaypoint

The `Move to Actor` node is used to move the actor towards a target, in this case, the next waypoint. The `Acceptance Radius` parameter defines a proximity threshold: if the NPC is within this radius of the target actor's position, the destination is considered reached. Once the movement is complete, an arrival event is delegated to the `MovementFinished` event, where the result is checked to determine whether the path was successfully completed. If so, the `StopMovement` method of the BP_NPC blueprint is called to indicate that the NPC can now aim at the target.

The image showing this is as follows:

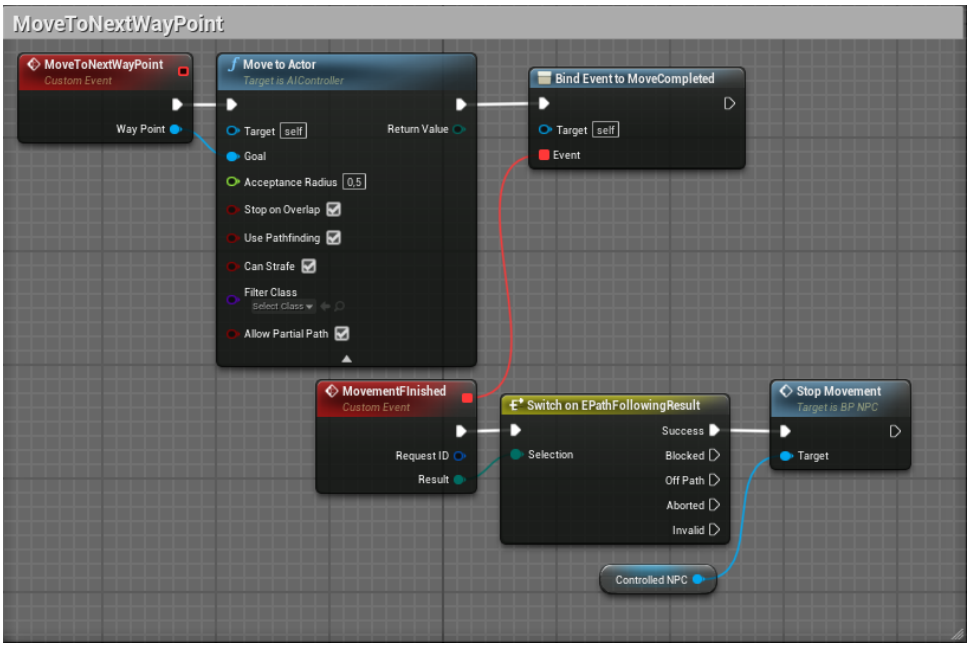

Figure 5.6: Move to next waypoint in Unreal

BP_NPC

The NPC Blueprint is essentially similar to the previously defined BP_Player, ith the exception that it does not include keyboard or mouse input handling. The new additions involve assigning waypoints and shooting panels to the NPC. On every game tick, a check is performed to determine whether the character is currently moving and aiming. If neither condition is met, the `Aim` method is called, which will be described in detail shortly. If the NPC is neither aiming nor moving, it proceeds to fire using the previously defined shooting method. After firing, the next waypoint is assigned, ensuring that the array bounds are respected and the path continues in a circular manner if no immediate target is found.

The image showing this is as follows:

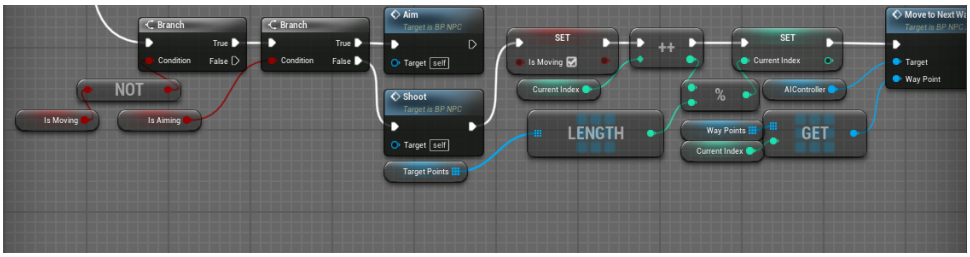

Figure 5.7: AimShootAndMove method in BP_NPC in Unreal

Unlike other engines, the `Aim` method in Unreal Engine is computed explicitly using vector mathematics and the dot product to determine whether the NPC is already aligned with its target. First, the positions of the camera and the target are retrieved. Then, the forward vector of the camera is obtained (typically via the `GetForwardVector` node), and a target direction vector is calculated by subtracting the camera's position from the target's position. This direction vector is then normalised, resulting in the desired direction for aiming.

Once both vectors are available (the camera's forward vector and the target direction), a dot product is computed. Since both vectors are normalised, the result will always be:

- **1** if they are perfectly aligned (the target is directly in the line of sight),
- **0** if they are perpendicular (not aiming at all),
- **-1** if they are pointing in opposite directions (target is behind the camera).

If the dot product is close to 1 (e.g., greater than a threshold like 0.999), the system considers the NPC to be properly aimed. The aiming state is disabled, and the graph flow resumes. Otherwise, the Get Rotator Camera To Target function is invoked (to be described later), and the camera is smoothly rotated towards the resulting rotation.

The images showing this is as follows:

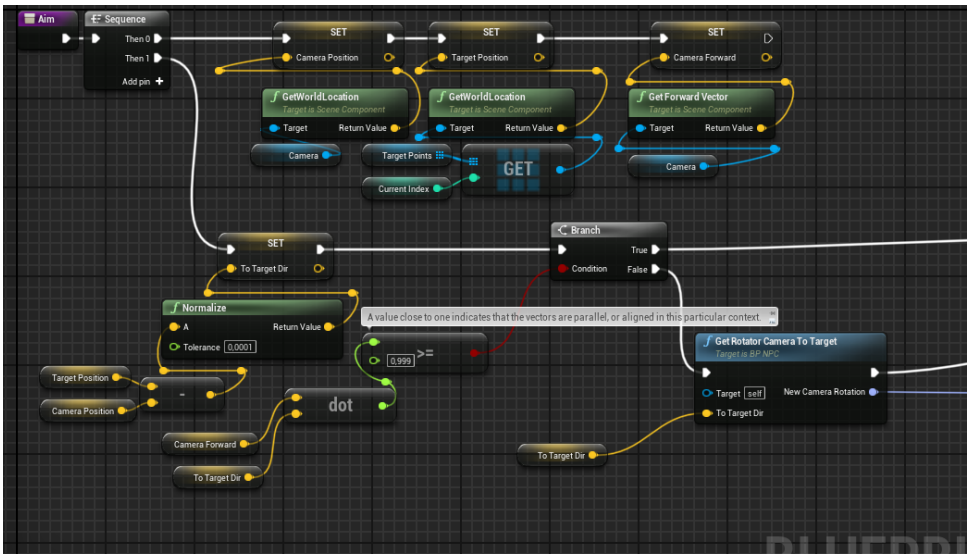

Figure 5.8: Aim function in Unreal (part 1)

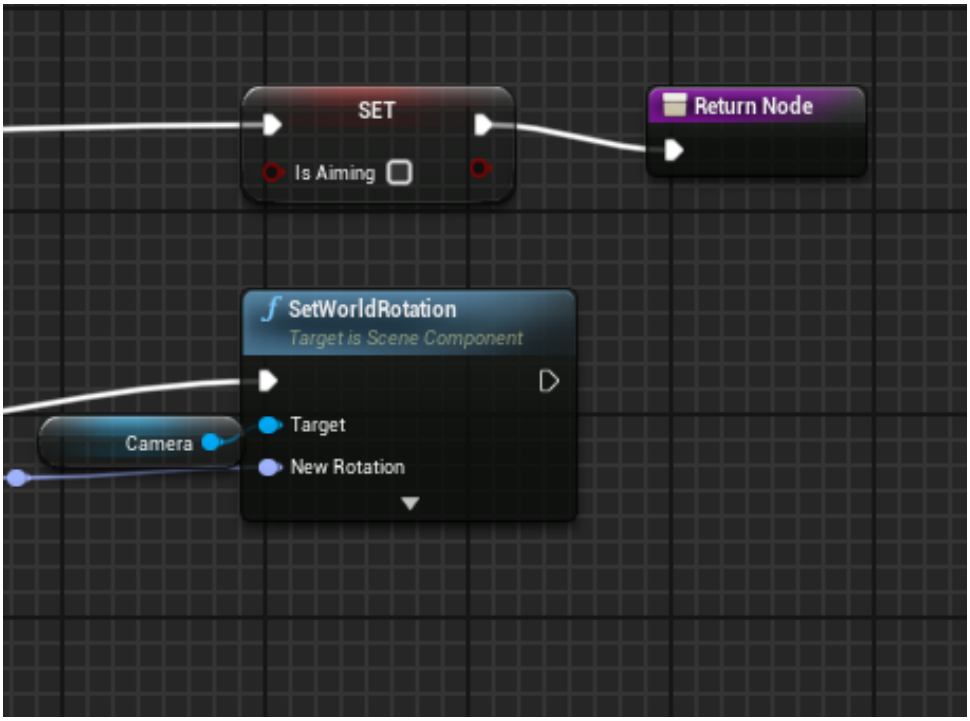

Figure 5.9: Aim function in Unreal (part 2)

With the direction vector pointing to the target (already normalised), its corresponding rotator is obtained relative to the X rotation axis using `Make Rot from X`. This rotator represents the desired camera orientation. The current camera rotation is then smoothly interpolated towards this target rotator each frame using `RInterp To`.

`RInterp` To takes the current rotation, the target rotation, the frame's `DeltaTime`, and an interpolation speed (`Interp Speed`). It returns a new rotation closer to the target, ensuring smooth, frame-rate independent rotation.

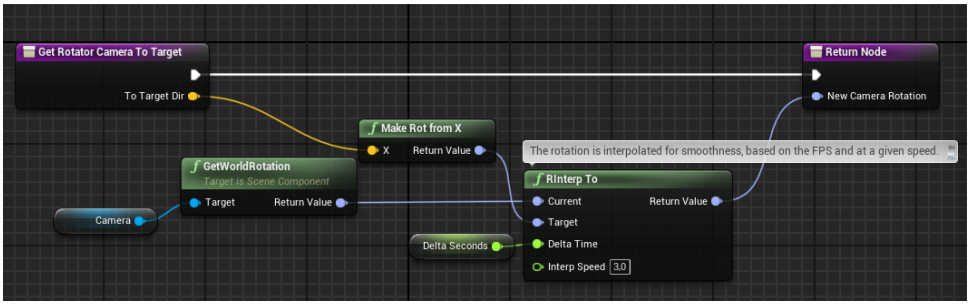

Figure 5.10: Aim function in Unreal (part 3)

BP_GameModeFPS

The blueprint has been modified to incorporate a choice between spawning the Player character or the NPC. This decision is governed by a Boolean variable named automaticMode, analogous to its use in Unity.

- If `automaticMode` is true: the NPC character is spawned. Subsequently, the player's viewpoint is switched to the NPC's camera using the `Set View Target with Blend` node, which ensures the player cannot directly control the NPC.
- If `automaticMode` is false: the Player character is spawned instead, and the player controller possesses this character using the `Possess` function, enabling direct control.

The spawn point is obtained from a waypoint by retrieving its transform to position the character correctly.

The image showing this is as follows:

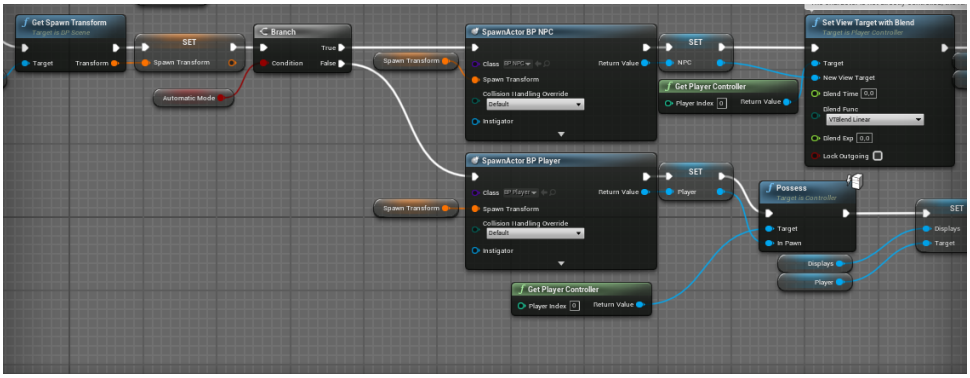

Figure 5.11: GameModeFPS modified for NPC in Unreal

5.5 Godot

The implementation in Godot uses its navigation system to generate NavMeshes guiding NPCs within the 3D environment. Logic is programmed in GDScript, enabling straightforward and agile integration. **Version 4.4.1** is applied, which includes new features enhancing navigation and overall system performance, making it a viable choice for lightweight and flexible projects.

5.5.1 Navigation mesh implement

In Godot, the *Navigation Mesh* is generated using the `NavigationRegion3D` node, which serves as a container for the `NavigationMesh` resource. This resource defines the navigable area within a 3D scene and determines how agents move through the environment. Proper configuration of its parameters is essential to ensure efficient and precise navigation.

Agent Parameters

These settings determine how the navigation mesh accommodates the agent's size and movement capabilities:

- **Radius**: specifies the agent's collision radius. This ensures the agent maintains a safe distance from obstacles and walls. If set too low, the agent might attempt to pass through unrealistically narrow gaps.
- **Height**: minimum vertical clearance required for the agent to pass beneath objects.
- **Max Slope**: the steepest incline (in degrees) that the agent is allowed to climb. Slopes exceeding this value are treated as impassable.
- **Max Climb**: the maximum height of vertical obstacles (e.g., steps or ledges) that the agent can climb over automatically.

Geometry Parameters

The *Geometry* section determines which parts of the scene are considered during NavMesh generation:

- **Use Geometry**: specifies the data source for generating the NavMesh. This can be visual meshes, physics colliders, or both.
- **Source Geometry Group Name**: filters the nodes to be considered based on group membership. For example, using the group `‘‘walkable’’` ensures only nodes in that group are included.

Cell Parameters

These parameters define how the environment is sampled and discretised for navigation purposes:

- **Cell Size**: the horizontal resolution of the navigation mesh. Smaller values yield more accuracy but increase computational load.
- **Cell Height**: the vertical sampling resolution. Lower values provide finer height details, at the cost of increased build complexity.

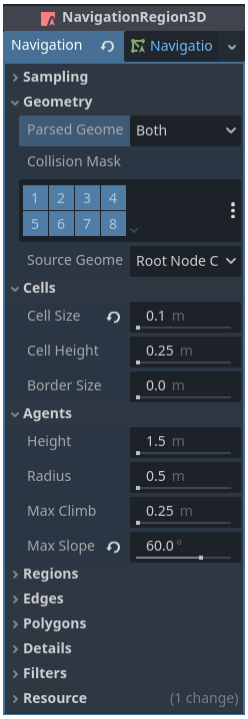

Figure 5.12: Navigation mesh Properties in Godot

which resulted in this:

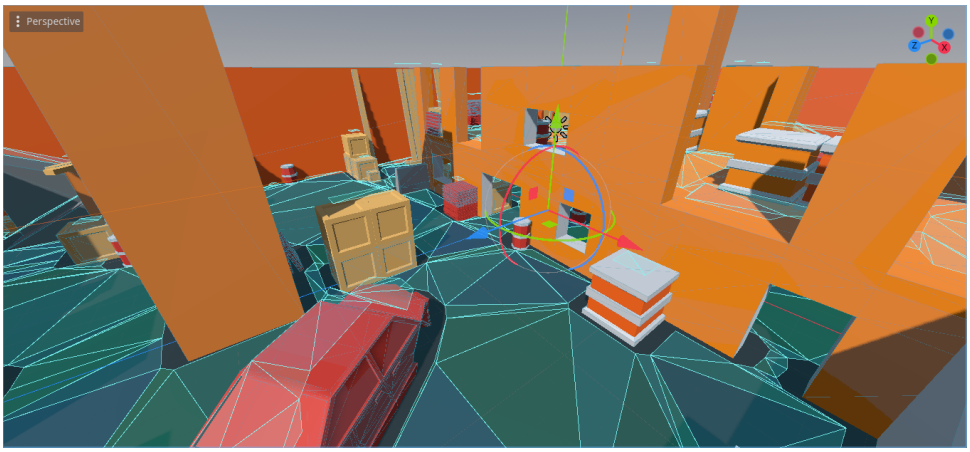

Figure 5.13: Navigation mesh result in Godot

5.5.2 NPC implement

The NPC implementation will be carried out in a way similar to how it would be done in Unity, with the main difference being that, in Godot, the calculation of directions to the next points on the Navigation Mesh (NavMesh) is not automatic. It is the developer's responsibility to manage and update these paths through code. The following script explains how this logic is implemented.

Patrol.gd

1. In `_ready()`, nodes are validated and the initial patrol destination is set. If no waypoints or targets are present, an error is thrown.
2. In `_physics_process(delta)`, In each frame:
   - If the destination has not yet been reached, the function `_move_towards_next_position(delta)` is called. This function calculates the direction the NPC should move in, based on its current position and the next point provided by the navigation mesh. If the NPC is close to the waypoint, it is considered to have arrived. In any case, the `_rotate_camera_towards_(direction_normalised, delta)` function is called first; this will be explained later. After that, the new velocity is applied and the agent is moved.
   - Otherwise, the `aim_and_shoot(delta)` function is called. This function is responsible for calculating the aiming direction towards the target and performing the shot. Its logic is similar to that implemented in Unity: from the camera, a gradual rotation is made towards the target's direction. If the angle between both directions is smaller than a threshold defined by `ERROR_MARGIN_ANGLE`, the target is considered correctly aimed at. At that point, the shot is executed and the destination is updated to the next point.
3. The `_rotate_camera_towards` function is responsible for smoothly rotating the NPC's camera towards a target direction, ensuring a fluid transition. To maintain rotational stability during movement, rotation is constrained to the Y-axis. Based on the normalised vector towards the target, the desired rotation is calculated as a quaternion. This is then interpolated between the camera's current rotation and the desired one using slerp, with the frame's delta and a configurable speed. Finally, the new rotation is applied to the camera, and the NPC's reticle is updated to reflect whether it is aiming at a valid target.

The implemented code is as follows:

```
extends CharacterBody3D

var utils = load("res://assets/scripts/Utils.gd")

const MIN_DISTANCE_WAYPOINT: float = 0.1
const ERROR_MARGIN_ANGLE: float = 0.03  # The maximum allowed angular deviation (in degrees) when comparing orientations.

var speed: float = 8.0
var cameraRotateSpeed: float = 5.0

var _waypoints: Array[Node3D] = [] # The points that define the predefined path the agent follows.
var _targets: Array[Node3D] = []
var _destination_index: int = 0
var _is_aiming: bool = false

@onready var agent: NavigationAgent3D = $NavigationAgent3D
@onready var camera: Camera3D = $Camera
@onready var weapon: Node3D = $Camera/Weapon
@onready var raycast_npc: RayCast3D = $Camera/RayCastNPC
@onready var hud_manager: CanvasLayer = get_node("/root/MainStage/HUD")

func _ready():
    if _waypoints.is_empty() or _targets.is_empty():
        push_error("Waypoints or targets are empty. Patrol wont start.")
        return

    utils.check_node_validity(agent, "agent")
    utils.check_node_validity(camera, "camera")
    utils.check_node_validity(weapon, "weapon")
    utils.check_node_validity(raycast_npc, "raycast_npc")
    utils.check_node_validity(hud_manager, "hud_manager")

    for waypoint in _waypoints:
        utils.check_node_validity(waypoint, waypoint.name)
    for target in _targets:
        utils.check_node_validity(target, target.name)

    agent.target_position = _waypoints[_destination_index].global_position

func _physics_process(delta):
    if _is_aiming:
        return

    if !agent.is_navigation_finished():
        weapon.set_walking_animation(true)
        _move_towards_next_position(delta)
    else:
        weapon.set_walking_animation(false)
        _aim_and_shoot(delta)


func _move_towards_next_position(delta):
    var next_pos = agent.get_next_path_position()
    var agent_pos = global_position

    if agent_pos.distance_to(next_pos) < MIN_DISTANCE_WAYPOINT:
        return
    var direction = next_pos - agent_pos
    var direction_normalized = direction.normalized()

    _rotate_camera_towards(direction_normalized, delta)

    velocity = direction_normalized * speed
    move_and_slide()

func _rotate_camera_towards(direction_normalized: Vector3, delta: float, is_moving := true):
    # Movement only involves rotation around the Y axis, as it provides greater stability
    if is_moving:
```

```
69          direction_normalized.y = 0
70
71      # Determine the rotation needed to look at the target
72      var target_rotation_in_quaternion = Quaternion(Basis.looking_at(direction_normalized
            , Vector3.UP))
73      var current_camera_rotation_in_quaternion = camera.global_transform.basis.
            get_rotation_quaternion()
74
75      # Smoothly rotate the camera towards the target over time
76      var new_camera_rotation = current_camera_rotation_in_quaternion.slerp(
            target_rotation_in_quaternion, delta * cameraRotateSpeed)
77      camera.global_transform.basis = Basis(new_camera_rotation)
78
79      _update_npc_crosshair()
80
81  func _aim_and_shoot(delta):
82      _is_aiming = true
83
84      var target = _targets[_destination_index]
85      var target_pos = target.global_position
86      var camera_pos = camera.global_position
87
88      var angle = 1.0
89
90      while angle > ERROR_MARGIN_ANGLE:
91          var direction_normalized = (target_pos - camera_pos).normalized()
92
93          # Full and precise rotation during aiming
94          _rotate_camera_towards(direction_normalized, delta, false)
95
96          var forward = -camera.global_transform.basis.z.normalized()
97          # Computes the angular difference between the current orientation and the target
                orientation
98          # Returns the unsigned minimum angle to the given vector, in radians.
99          angle = forward.angle_to(direction_normalized)
100
101         # Allowing the aiming animation to update smoothly over
102         await get_tree().process_frame
103
104     weapon.shoot()
105     _go_to_next_waypoint()
106     _is_aiming = false
107
108 func _go_to_next_waypoint():
109     _destination_index = (_destination_index + 1) % _waypoints.size()
110     agent.target_position = _waypoints[_destination_index].global_position
111
112
113 func set_waypoints(waypoints: Node3D):
114     _waypoints.clear()
115     for child in waypoints.get_children():
116         if child is Node3D:
117             _waypoints.append(child)
118
119 func set_targets(targets: Node3D):
120     _targets.clear()
121     for child in targets.get_children():
122         if child is Node3D:
123             _targets.append(child)
124
125 func _update_npc_crosshair():
126     var is_target_panel_shooting: bool = false
127     if raycast_npc.is_colliding():
128         var target = raycast_npc.get_collider()
129         is_target_panel_shooting = target.is_in_group("panel_shooting")
130     hud_manager.update_crosshair(is_target_panel_shooting)
```

Main.gd

This script is responsible for managing the scene's initialisation and the instantiation of the character, whether it be the player or an NPC. A boolean variable named automatic_mode is defined: if set to true, the NPC is instantiated and assigned patrol waypoints and shooting targets. If false, the player is instantiated instead, and the scene's scale is adjusted accordingly. Afterwards, the validity of the character's weapon is verified, the instantiated character is added as a child of the main node, and it is placed at the defined spawn point in the scene. The structure and control logic are analogous to the system previously demonstrated in Unreal.

```
extends Node3D

var utils = load("res://assets/scripts/Utils.gd")

@export var automatic_mode := true
@export var npc: PackedScene
@export var player: PackedScene

@onready var spawn_point : Node3D = $NavigationRegion3D/Scene/scene/SpawnPoint
@onready var patrol: Node3D = $NavigationRegion3D/Scene/scene/Patrol
@onready var targets: Node3D = $NavigationRegion3D/Scene/scene/"Panels Shooting"
@onready var scene: Node3D = $NavigationRegion3D/Scene

var character_instance: Node = null

func _ready():
    utils.check_node_validity(spawn_point, "spawn_point")
    utils.check_node_validity(patrol, "patrol")
    utils.check_node_validity(targets, "targets")
    utils.check_node_validity(scene, "scene")

    if automatic_mode:
        character_instance = npc.instantiate()
        character_instance.set_waypoints(patrol)
        character_instance.set_targets(targets)
    else:
        character_instance = player.instantiate()
        character_instance.scale_scene = scene.scale.length()

    var weapon = character_instance.get_node("Camera/Weapon")
    utils.check_node_validity(weapon, "weapon")

    add_child(character_instance)
    character_instance.global_transform.origin = spawn_point.global_transform.origin
```

This chapter has presented the implementation of the autonomous navigation system using Unity, Unreal Engine, and Godot. Equivalent prototypes were developed in each engine to allow for a fair comparison under consistent conditions. The focus was placed on the generation and configuration of navigation meshes, alongside the control logic required to manage the autonomous behaviour of NPCs.

These implementations provide not only a functional basis for navigating virtual environments without player input, but also a structured platform for assessing each engine's capabilities in terms of usability, adaptability, and development overhead. The manual control system developed in the previous chapter has been retained to allow for targeted performance testing; however, the automated system proves more suitable for systematic evaluation within controlled scenarios.

The next chapter will address the deployment of these prototypes across the three engines. It will outline the project build procedures specific to each platform and define the testing methodology employed to evaluate performance. This transition marks the beginning of the final technical analysis phase, in which metrics such as frame rate and computational efficiency will be measured to assess each engine's suitability for the complete system.

Chapter 6

Project deployment

Following the implementation of the autonomous navigation system in Unity, Unreal Engine, and Godot, it became necessary to prepare each project for performance testing. This chapter presents the process of building and deploying the prototypes across the three engines, with a focus on replicating consistent deployment conditions to ensure fair comparative analysis.

Although each engine provides its own set of tools and procedures for building executable versions of a project, the aim has been to maintain equivalence across platforms by using default configurations and minimal optimisation, unless required for functionality. This approach guarantees that the performance metrics obtained later will reflect the baseline capabilities of each engine rather than artefacts of manual tuning.

The deployment process is detailed individually for each engine, highlighting any relevant challenges, workarounds, or limitations encountered when preparing the builds for subsequent analysis.

6.1 Introduction

A build is the process of packaging the entire project (scenes, scripts, assets, configurations), along with the compiled code, into an executable or exportable application for a specific platform. n this case, for WebGL — that is, to run in a web browser using JavaScript — WebAssembly needs to be used.

WebAssembly is a compact binary code format that allows applications to run almost as fast as native. Thanks to WebAssembly, languages such as C++ (the language used by most game engines) have a way to compile their code to run in any web browser.[30]

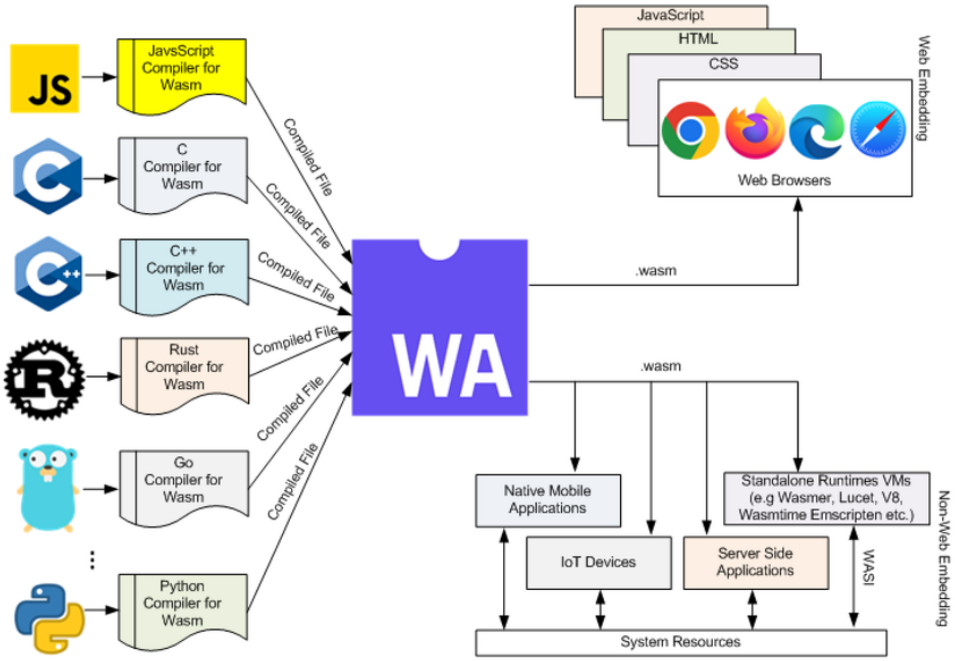

Figure 6.1: WebAssembly data flow

6.2 Unity

1. Open your project in Unity.
2. Go to `File -> Build Profiles`.
3. In the platform list, select **WebGL**.
4. Click **Switch Platform** to change if it's not already selected.
5. Click **Player Settings** (under **Player Settings Overrides**).
6. In the **Player Settings** window, locate the **Publishing Settings** section.
7. Configure the **Compression Format** option and select:
   - `Disabled` for local testing (recommended if you only want to serve locally without hassle).
8. Close **Player Settings**.
9. In the **Build Settings** window, click **Build**.
10. Select the `BuildWebGL/Unity` folder as the build destination.
11. Wait for the build to finish and verify that the files are in the folder.
    (*The first build may take between 40 and 60 minutes.*)

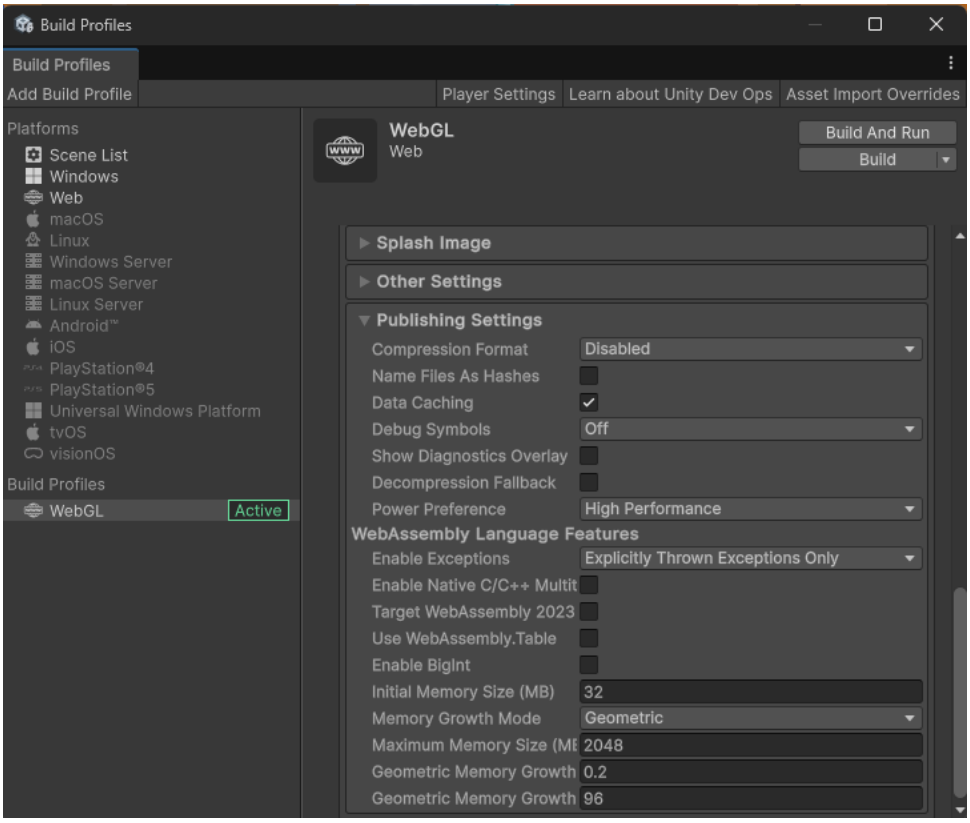

Figure 6.2: Building properties in Unity

6.3 Unreal Engine

By default, Unreal Engine 4.27 does not include support for WebGL. A fork of Unreal 4.27 that retains this support has been used instead, with credit given to *SpeculativeCoder*[1].

After downloading and compiling the engine, follow these instructions:

1. Open your project in Unreal.
2. If a message appears stating that the versions do not match your project, simply ignore it and proceed.
3. Go to `File -> Package Project -> HTML5`.
4. Select **HTML5**.
5. Select the `BuildWebGL/Unreal` folder as the build destination.
6. Wait for the build to finish and verify that the files are in the folder.
   (*The first time may take between 60 and 80 minutes.*)

If any changes are to be made during the project build, these should be done via:
`Edit -> Project Settings -> Platforms -> HTML5`

[1] `https://github.com/SpeculativeCoder/UnrealEngine-HTML5-ES3`

6.4 Godot

1. Open your project in Godot.
2. Go to `Project -> Export`.
3. Click the `Add` button in the top-left corner and select `Web`.
4. You may be prompted to install the export template; if so, go to `Project -> Install Export Templates`.
   - Click on `Online` mode and select the GitHub option.
   - Wait approximately 3–5 minutes. Once the process is complete, the window can be closed.
5. Select **Export Project**.
6. Select the `BuildWebGL/Godot` folder as the build destination.
7. Wait for the build to finish and verify that the files are in the folder.
   (*The first time may take between 2 and 3 minutes.*)

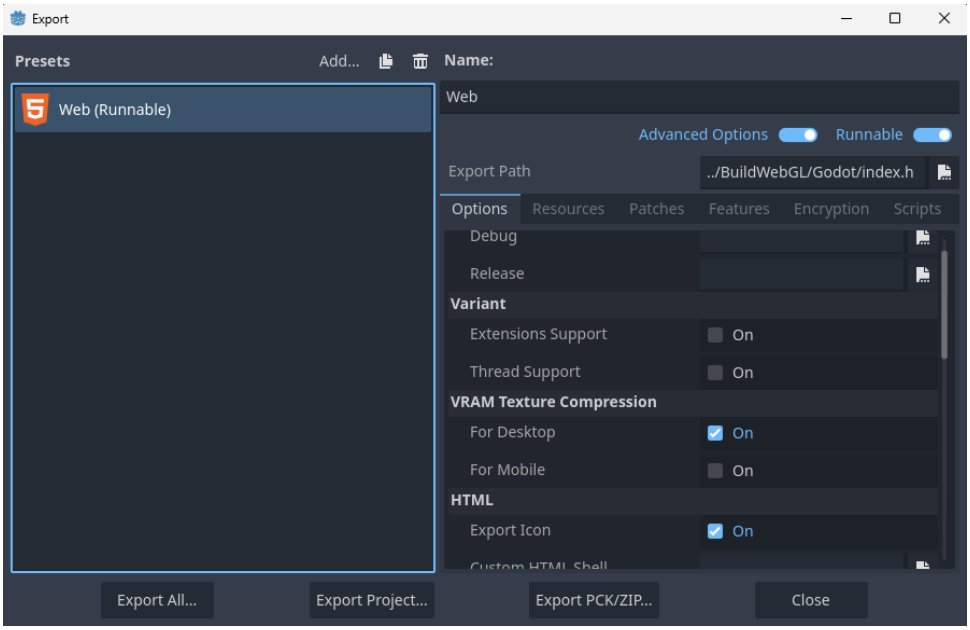

Figure 6.3: Building properties in Godot

This chapter has outlined the deployment procedures followed for Unity, Unreal Engine, and Godot, explaining how executable builds were generated for each engine under equivalent conditions. These deployments ensure that the performance of each engine can be fairly assessed, using exactly the same content and logic, without influence from platform-specific optimisations.

With functional versions now prepared, it is possible to proceed to the next chapter, which will focus on performance analysis. The testing methodology and tools employed to measure key metrics such as FPS and computational load will be presented.

Chapter 7

Performance testing

This chapter details the methodology employed to evaluate the performance of the prototypes developed in Unity, Unreal Engine, and Godot. It describes the tools and techniques used to record key metrics such as FPS and computational load during project execution.

Furthermore, the results obtained on two reference platforms — a personal computer PC and a Raspberry Pi — are presented. These data enable a comparative analysis of each engine's performance across different hardware environments, which is essential for determining the feasibility and efficiency of each solution in real-world scenarios.

Understanding these performance characteristics is essential for selecting the most suitable engine for the final system, ensuring that it can deliver smooth and reliable operation in both development and deployment environments.

7.1 Methodology

For automated testing, a JavaScript script will be created to simulate an incognito window environment and execute the full NPC test N times. Afterwards, both the trace for Chrome's profiler and a set of logs capturing the game's frame rate will be saved. For this purpose, the `Puppeteer` library will be used, as it allows for this type of automation[31].

7.1.1 FPS trace

Since it is difficult to observe the overall frame rate of the application using profilers, it has been decided to record FPS log traces for each of the engines. The case of Unity is shown below; however, in practice, the process works identically for the other two engines.

```
using System.Collections.Generic;
using System.Globalization;
using UnityEngine;

[System.Serializable]
public struct FrameInfo
{
    public int frame;
    public string timeStampInSec;
    public string frameTimeInMs;
}

[System.Serializable]
public struct FrameStats
{
    public string meanFrameTimeInMs;
    public string averageFPS;
    public string stdDevInMs;
    public int totalFrames;
    public int badFramesCount;
}

public class FPSStats : MonoBehaviour
{
    private List<float> frameTimes = new List<float>();
    private int totalFrames = 0;

    public float stdDevThreshold = 2.0f;  // Threshold multiplier for identifying outliers (bad frames)
    public int logFrequencyInFrames = 10; // Frequency (in frames) to output log entries

    void Start()
    {
        if (logFrequencyInFrames <= 0) logFrequencyInFrames = 1;
        if (stdDevThreshold < 0) stdDevThreshold = 0;
    }

    void Update()
    {
        float frameTimeInSec = Time.unscaledDeltaTime;
        frameTimes.Add(frameTimeInSec);
        totalFrames++;
    }

    void OnApplicationQuit()
    {
        if (totalFrames == 0) return;

        float mean = CalculateMean(frameTimes);
        float stdDev = CalculateStdDev(frameTimes, mean);

        // Define bounds for filtering outliers based on standard deviation threshold
        float lowerBound = mean - stdDevThreshold * stdDev;
        float upperBound = mean + stdDevThreshold * stdDev;

        float timeAccumulator = 0f;
        List<float> goodFrames = new List<float>();
        int badFramesTotal = 0;

        // Iterate over all recorded frames and classify them as good or bad
        for (int i = 0; i < totalFrames; i++)
        {
            bool isBadFrame = frameTimes[i] < lowerBound || frameTimes[i] > upperBound;
            if (isBadFrame)
                badFramesTotal++;
```

```
65              else
66                  goodFrames.Add(frameTimes[i]);
67
68              timeAccumulator += frameTimes[i];
69
70              if (i % logFrequencyInFrames == 0)
71                  LogFrame(i, timeAccumulator, frameTimes[i], isBadFrame);
72          }
73
74          if (goodFrames.Count == 0)
75          {
76              Debug.LogWarning("[FPSStats] No good frames detected to calculate statistics.");
77              return;
78          }
79
80          float goodMean = CalculateMean(goodFrames);
81          float goodStdDev = CalculateStdDev(goodFrames, goodMean);
82          float avgFPS = 1f / goodMean;
83
84          FrameStats summary = new FrameStats
85          {
86              meanFrameTimeInMs = (goodMean * 1000f).ToString("F2", CultureInfo.InvariantCulture),
87              averageFPS = avgFPS.ToString("F2", CultureInfo.InvariantCulture),
88              stdDevInMs = (goodStdDev * 1000f).ToString("F2", CultureInfo.InvariantCulture),
89              totalFrames = totalFrames,
90              badFramesCount = badFramesTotal
91          };
92
93          string summaryJson = JsonUtility.ToJson(summary, true);
94          Debug.Log("[STATS] " + summaryJson);
95      }
96
97      // Logs information about a single frame, tagged as good or bad
98      private void LogFrame(int frameIndex, float timeAccumulator, float frameTime, bool isBad)
99      {
100         FrameInfo frameInfo = new FrameInfo
101         {
102             frame = frameIndex,
103             timeStampInSec = timeAccumulator.ToString("F2", CultureInfo.InvariantCulture),
104             frameTimeInMs = (frameTime * 1000f).ToString("F2", CultureInfo.InvariantCulture)
105         };
106
107         string json = JsonUtility.ToJson(frameInfo);
108         Debug.Log(isBad ? $"[BAD_FRAME] {json}" : $"[FRAME_DATA] {json}");
109     }
110
111     private float CalculateMean(List<float> data)
112     {
113         float sum = 0f;
114         foreach (float d in data)
115             sum += d;
116         return sum / data.Count;
117     }
118
119     private float CalculateStdDev(List<float> data, float mean)
120     {
121         if (data.Count <= 1)
122             return 0f;
123         float sum = 0f;
124         foreach (float d in data)
125             sum += (d - mean) * (d - mean);
126         return Mathf.Sqrt(sum / (data.Count - 1));
127     }
128 }
```

Since profilers often make it difficult to observe the application's overall frame rate, an alternative logging system has been implemented to track frame times directly. Each individual frame time is collected during execution. From this data, the **mean** and **standard deviation** are calculated. Initial loading phases may produce abnormally high frame times, which can skew the results. These are treated as *outliers* and excluded from the final statistics. A frame is classified as an outlier if its duration exceeds the mean by more than **two standard deviations**.

Every `N` frames (default: `10`), frame information is printed to the log to avoid overwhelming the output. The following structure is used:

```
public struct FrameInfo
{
    public int frame;
    public string timeStampInSec;
    public string frameTimeInMs;
}
```

Each entry is prefixed with a tag:

- `[FRAME_DATA]`: if the frame is considered valid
- `[BAD_FRAME]`: if the frame is considered an outlier

At the end of the execution, a summary log is printed with the tag `[STATS]`, using the following structure:

```
public struct FrameStats
{
    public string meanFrameTimeInMs;
    public string averageFPS;
    public string stdDevInMs;
    public int totalFrames;
    public int badFramesCount;
}
```

7.1.2 Performace test script

The complete logic is encapsulated in the following script:

```
const puppeteer = require('puppeteer');
const fs = require('fs');
const path = require('path');

// Simple argument parser
const args = process.argv.slice(2);
const url = getArgValue('--url') || 'http://localhost:8000';
const repetitions = parseInt(getArgValue('--repetitions')) || 1;
const TIMEOUT_MS = (parseInt(getArgValue('--timeout')) || 45) * 1000;
const noTimeout = args.includes('--no-timeout');
const engine = getArgValue('--engine') || 'unity';

function getArgValue(option) {
    const index = args.findIndex(arg => arg === option);
    return index !== -1 ? args[index + 1] : null;
}

const TRACE_CATEGORIES = [
    '-*', // Exclude all categories by default, so only specified ones are collected
    'devtools.timeline', // General Chrome DevTools timeline events (loading, scripting,
        rendering)
    'v8.execute', // JS execution events from the V8 engine
    'disabled-by-default-v8.cpu_profiler', // CPU profiling info for JS
    'disabled-by-default-v8.runtime_stats', // Runtime stats for V8
    'blink.console', // Console API calls like console.log
    'disabled-by-default-devtools.timeline', // More detailed timeline events
    'disabled-by-default-devtools.timeline.frame', // Frame-level timeline events
    'toplevel', // Top-level event markers
    'benchmark', // Benchmarking events
    'blink.user_timing', // User-defined performance marks and measures
    'disabled-by-default-memory-infra' // Memory infra events for memory profiling
].join(',');

// Function that waits until "done" appears in the console or the timeout expires
function waitForDoneSignal(page, logs) {
    return new Promise((resolve, reject) => {
        const logRegex = /\b!?done!?\b/i; // i = case-insensitive
        // Listen to console messages for "done"
        const onConsole = msg => {
            const message = `[${msg.type()}] ${msg.text()}`;
            logs.push(message);

            if (logRegex.test(msg.text())) {
                if (timeoutId) clearTimeout(timeoutId);
                page.off('console', onConsole);
                console.log('Detected "done" signal from console output');
                resolve();
            }
        };

        page.on('console', onConsole);

        // Set timeout for the maximum wait time (unless --no-timeout was used)
        let timeoutId;
        if (!noTimeout) {
            timeoutId = setTimeout(() => {
                page.off('console', onConsole);
                reject(new Error(`Timeout: the game did not complete within ${TIMEOUT_MS
                    / 1000} seconds.`));
            }, TIMEOUT_MS);
        }
    });
}

async function runTest(i, browser) {
    console.log(`Execution ${i} started`);

    // Create a new incognito browser context for clean state
    const context = await browser.createBrowserContext();
```

```
    const page = await context.newPage();

    // Enable Chrome DevTools Protocol (CDP) session for performance tracing
    const client = await page.createCDPSession();

    // Array to collect trace data chunks
    const traceChunks = [];
    client.on('Tracing.dataCollected', event => {
        traceChunks.push(...event.value);
    });

    const tracingComplete = new Promise(resolve => {
        client.once('Tracing.tracingComplete', resolve);
    });

    // Start tracing with specified categories and options
    await client.send('Tracing.start', {
        categories: TRACE_CATEGORIES,
        options: 'sampling-frequency=10000', // 10 kHz sampling frequency for high-res
            CPU profiling
        bufferUsageReportingInterval: 1000 // Report tracing buffer usage every 1 second
    });
    console.log('Tracing started');

    await page.goto(url);

    // Set viewport to match the screen size of the page
    const screen = await page.evaluate(() => ({ width: window.screen.width, height:
        window.screen.height }));
    await page.setViewport({ width: screen.width, height: screen.height });

    // Wait for the fullscreen button to appear and have an onclick handler
    switch (engine.toLowerCase()) {
        case 'unity':
            await page.waitForSelector('#unity-fullscreen-button', { visible: true,
                timeout: 10000 });
            await page.waitForFunction(() => !!document.querySelector('#unity-fullscreen
                -button').onclick);
            await page.click('#unity-fullscreen-button');
            console.log('Clicked Unity fullscreen button');
            break;

        case 'unreal':
            await page.waitForSelector('#fullscreen_request', { visible: true, timeout:
                10000 });
            await page.waitForFunction(() => {
                const btn = document.querySelector('#fullscreen_request');
                return btn && btn.offsetParent !== null;
            });
            await page.click('#fullscreen_request');
            console.log('Clicked Unreal fullscreen button');
            break;

        case 'godot':
            await page.waitForSelector('#canvas', { visible: true, timeout: 10000 });
            await page.click('#canvas');
            console.log('Clicked Godot fullscreen button');
            break;

        default:
            console.log(`Engine "${engine}" no soportado.`);
            break;
    }

    // Array to store all console log messages
    const logs = [];

    await waitForDoneSignal(page, logs);

    await client.send('Tracing.end');
    await tracingComplete;
    console.log('Tracing stopped');
```

```
    // Save console logs and performance trace data to files
    const dir = path.resolve(`./output/${engine.toLowerCase()}/run_${i}`);
    fs.mkdirSync(dir, { recursive: true });

    fs.writeFileSync(`${dir}/console_logs.txt`, logs.join('\n'));
    fs.writeFileSync(`${dir}/performance_trace_${i}.json`, JSON.stringify({ traceEvents:
        traceChunks }, null, 2));

    await context.close();
    console.log(`Execution ${i} complete. Output saved to ${dir}\n`);
}

// Main
(async () => {
    const browser = await puppeteer.launch({
        headless: false,
        args: ['--autoplay-policy=no-user-gesture-required']
    });

    for (let i = 1; i <= repetitions; i++) {
        try {
            await runTest(i, browser);
        } catch (e) {
            console.error(`Execution ${i} failed:`, e.message);
        }
    }

    await browser.close();
    console.log('All test runs completed.');
})();
```

Explication

1. The event categories that Chrome DevTools will collect are defined, focusing on performance, JavaScript execution, memory, and timeline events.
2. The function `waitForDoneSignal(page, logs)` listens to the page's console messages to detect the keyword "done" (case-insensitive, with possible exclamation marks). If detected, the promise resolves and the script proceeds. If the timeout expires (and is enabled) without detection, the promise is rejected with an error. All console messages are also collected into a `logs` array.
3. The function `runTest(i, browser)` receives an index `i` and an open browser instance to execute a complete test. It opens a new incognito context to ensure a clean state, initiates a CDP session to trace performance, and navigates to the target İt adjusts the viewport size to the detected screen dimensions. Depending on the engine, it detects whether a fullscreen button generated by the engine's build is present and activates it to enter fullscreen mode. Then, it calls `waitForDoneSignal` and saves the results.
4. All of this is performed within an asynchronous main loop that launches the browser in visible mode and closes browser instances once all runs are complete.

Execution

The script can be executed with the following command:

```
node performance-profile.js
  --url <URL>
  --repetitions <N>
  [--timeout <seconds> | --no-timeout]
  --engine <engine>
```

where:

- `--url`: URL of the page to be tested. **Default:** `http://localhost:8000`
- `--repetitions`: number of times to run the test. **Default:** `1`
- `--timeout`: maximum duration in seconds allowed for each test run to complete. **Default:** `45`
- `--no-timeout` (optional): disables the timeout limit. Each test run will be allowed to complete regardless of duration.
- `--engine`: engine used to run the game. Required for handling full-screen mode during testing. **Default:** `unity`

Output

After each run, a folder will be created at `./output/<engine>/run_X/` containing:

- `console_logs.txt` → Browser console output during execution
- `performance_trace_X.json` → Performance trace in JSON format (compatible with Chrome DevTools Performance)

7.2 Results

As previously established, two systems will be analysed: a low-end one based on a Raspberry Pi 4, and a modest PC — in this case, the workstation in use. Five test runs have been performed with each engine, and various technical aspects will now be examined.

7.2.1 PC results

The specifications of the PC are as follows:

1. **CPU**: Intel i3 550 3.33 GHz
2. **RAM**: 8 GB
3. **GPU**: NVIDIA GTX 960

Following this, we will analyse each of the key aspects in detail.

FPS

The following table shows the average frame times for each engine across five runs, as well as the average frame rate and the number of bad frames recorded, along with their percentage relative to the total number of frames, calculated from the previously mentioned logs:

In the graphs, the situation is observed as follows:

Engine	Mean Frame Time (ms)	Std Dev (ms)	Avg FPS	FPS Std	Total Frames	Bad Frames	Bad Frames %
Godot	16.776	0.0493	59.598	0.1782	2339.4	10.8	0.462%
Unity	18.886	0.0434	52.944	0.1203	1994.0	2.0	0.100%
Unreal	16.652	0.0110	60.052	0.0349	2361.0	11.2	0.474%

Table 7.1: FPS stats in PC

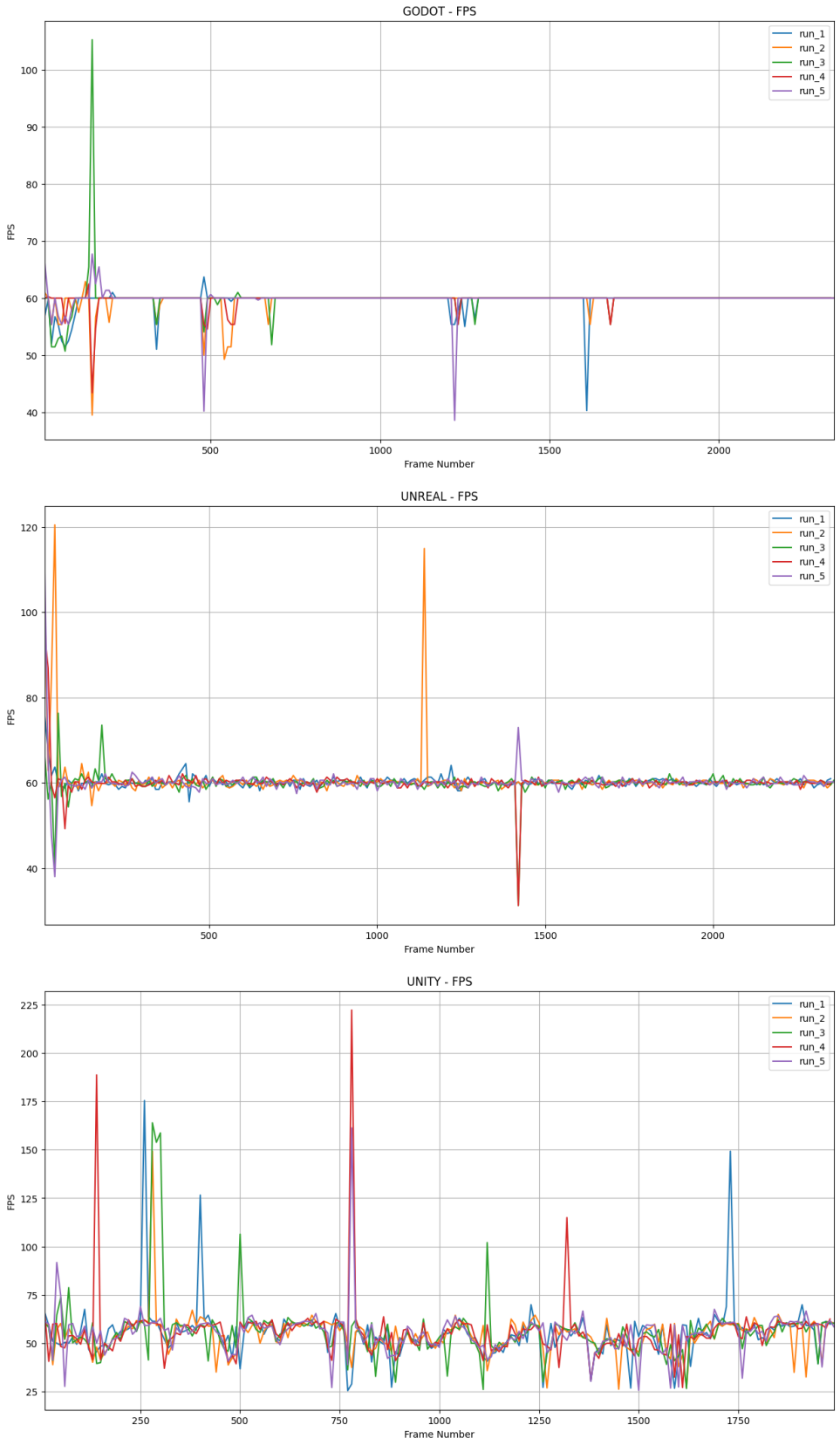

Figure 7.1: FPS graphics in PC

When comparing the FPS, clear differences are observed both in smoothness and stability. Unreal positions itself as the most powerful, achieving the highest average FPS and the lowest frame time, with practically no variation between runs. Godot follows closely, with similar figures, although it shows slightly greater instability which could result in occasional minor stutters. Unity, on the other hand, exhibits the lowest performance in terms of FPS and frame time, although it maintains reasonable stability. In summary, Unreal offers the smoothest and most consistent experience, Godot holds as a balanced option, and Unity, although slower, maintains stable execution within its limitations.

CPU

To calculate CPU usage, the trace provided by the CDP is analysed. This is done by dividing the time spent on scripting—which represents the CPU time exclusively dedicated to processing code— by the total duration of the test. This yields the actual percentage of CPU usage during execution.

The formula is expressed as:

$$\text{CPU Usage} = \frac{\text{Time spent on scripting}}{\text{Total test duration}} \times 100$$

All of this can be observed in the trace shown in the following figure:

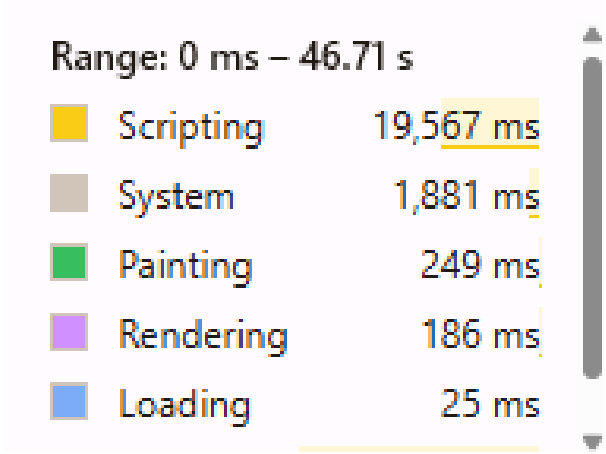

Figure 7.2: CPU Example in CDP

The test results are as follows:

Godot	CPU Time (s)	Scripting Time (s)
Run 1	48.08	39.66
Run 2	42.54	36.41
Run 3	42.40	35.06
Run 4	42.40	34.76
Run 5	42.37	36.42

Unity	CPU Time (s)	Scripting Time (s)
Run 1	44.53	12.91
Run 2	42.85	11.87
Run 3	42.82	11.58
Run 4	42.83	11.44
Run 5	42.81	11.74

Unreal	CPU Time (s)	Scripting Time (s)
Run 1	48.46	21.41
Run 2	46.84	19.60
Run 3	46.65	19.64
Run 4	46.60	20.03
Run 5	46.71	19.57

The following chart is obtained using the proposed calculation:

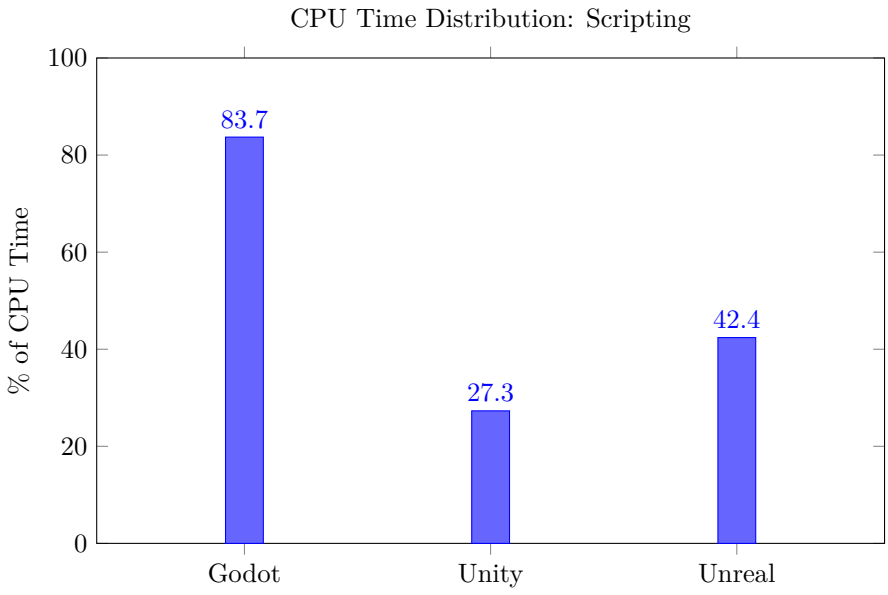

Figure 7.3: CPU Usage graphic in PC

Godot exhibits high usage in both total CPU time and scripting time, with scripting accounting for approximately **83–85%** of the CPU time, and scripting durations around 35–40 seconds. This indicates that most of the load is due to code execution during the test.

Unity is clearly the most efficient in scripting, with times between 11 and 13 seconds, equivalent to **27–30%** of total CPU time. Total CPU usage is slightly lower than in Godot and Unreal, reflecting an engine more optimised for script execution.

Unreal shows the highest total CPU usage (approximately 46–48 seconds), but scripting accounts for only **40–45%** of CPU time, suggesting that a significant portion of CPU consumption is devoted to non-scripting tasks such as rendering and system processing.

This pattern is consistent across all five runs, demonstrating stability in results and confirming these efficiency differences between engines.

Memory heap

The Memory Heap data is obtained by enabling the `memory` option in the CDP above the profiler. At the end of the profiling session, both the minimum and maximum memory allocations are displayed.

All of this can be observed in the trace shown in the following figure:

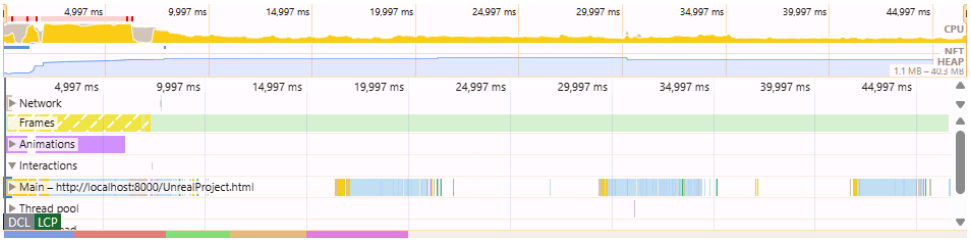

Figure 7.4: Memory heap Example in CDP

The test results are as follows:

Table 7.2: Godot – Sizes per Run

Run	Initial Size (MB)	Maximum Size (MB)
1	0.269	13.2
2	0.269	15.8
3	0.269	16.6
4	0.269	16.9
5	0.269	17.0

Table 7.3: Unity – Sizes per Run

Run	Initial Size (MB)	Maximum Size (MB)
1	1.1	16.8
2	1.2	16.8
3	1.2	17.8
4	1.2	17.8
5	1.2	16.6

Table 7.4: Unreal – Sizes per Run

Run	Initial Size (MB)	Maximum Size (MB)
1	1.1	40.2
2	1.3	40.4
3	1.3	40.4
4	1.3	40.4
5	1.1	40.3

The analysis of the Memory Heap reveals notable differences in memory management among the evaluated engines.

Godot stands out for its extreme efficiency in memory usage. In all runs, it maintains a consistently low minimum memory footprint of just 0.269 MB (269KB) and a maximum memory usage progressively ranging between 13.2 MB and 17.0 MB. This behaviour suggests a lightweight and highly optimised management model, especially suitable for resource-constrained environments.

Unity shows a higher, albeit stable, minimum usage (around 1.1–1.2 MB) and a maximum memory range between 16.6 MB and 17.8 MB. This consistency indicates a good balance between performance and efficiency, making it a solid and versatile choice.

On the other hand, Unreal Engine maintains a minimum memory usage similar to Unity's (1.1–1.3 MB), but its maximum memory consumption is considerably higher, reaching up to 40.4 MB. This elevated usage reflects its greater technical complexity and orientation towards more graphically demanding experiences.

In summary, in terms of memory consumption, Godot is clearly the most efficient engine. Nonetheless, Unity may represent an attractive alternative for those seeking a compromise between performance, portability, and moderate resource consumption.

All of this is summarised in the following chart:

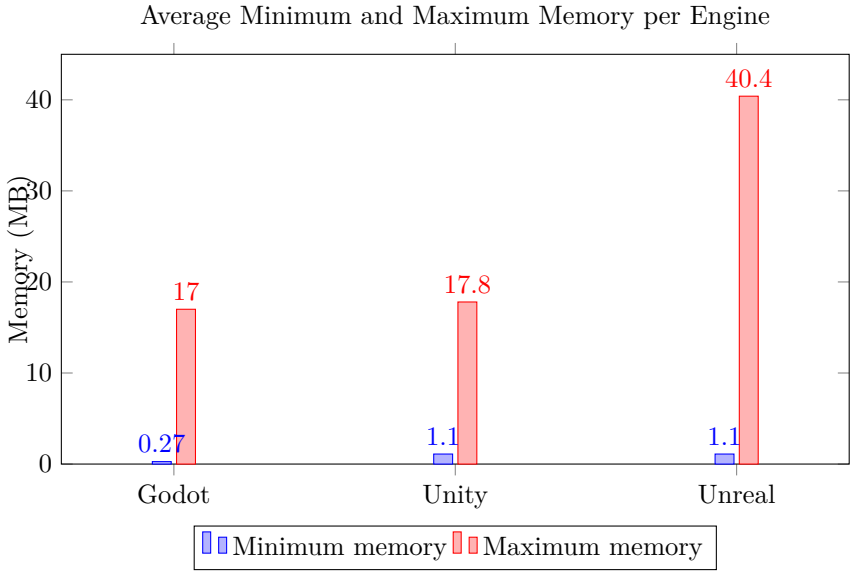

Figure 7.5: Memory Heap Graphic in PC

Load time

Load time is measured by the red segment at the top of the profiler timeline, representing the total duration for both the webpage to load and the game engine to initialise.

The test results are as follows:

Table 7.5: Godot – Load Time

Run	Load Time (s)
Run 1	7.35
Run 2	2.17
Run 3	2.11
Run 4	2.12
Run 5	2.09

Table 7.6: Unity – Load Time

Run	Load Time (s)
Run 1	6.54
Run 2	5.01
Run 3	5.00
Run 4	5.01
Run 5	5.10

Table 7.7: Unreal – Load Time

Run	Load Time (s)
Run 1	8.02
Run 2	6.52
Run 3	6.34
Run 4	6.38
Run 5	6.30

All of this is summarised in the following chart:

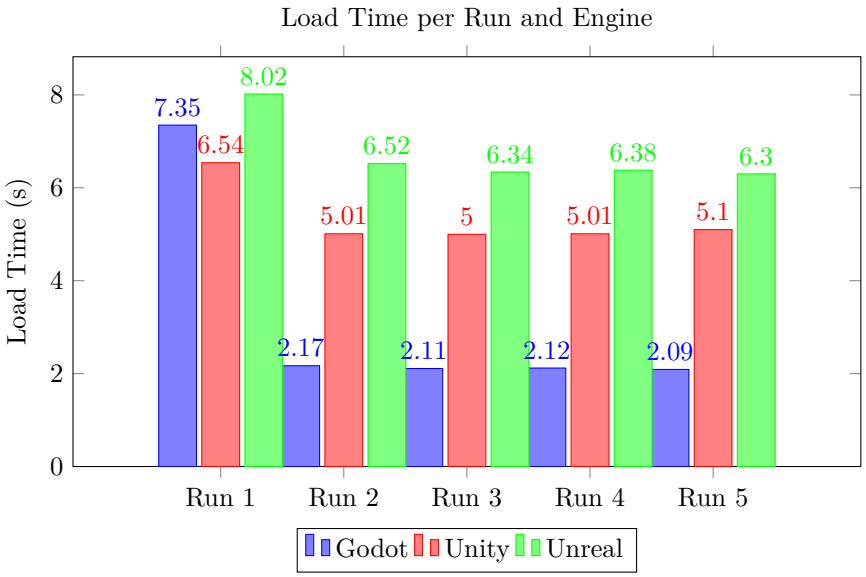

Figure 7.6: Load Time Graphic in PC

The load time varies significantly between engines. Godot exhibits a much slower first run (**7.35 s**) compared to subsequent runs (approximately **2.1 s**), which may indicate initialisation or caching processes that optimise later loads. Unity shows more consistent and slightly lower load times, around **5** seconds, indicating relative stability and speed. Unreal, on the other hand, has the longest load times overall, with values ranging between **6.3** and **8** seconds, suggesting a heavier or more complex loading process. These results suggest that for quick loads, Unity is the most efficient, followed by Godot after the initial run, and finally Unreal with heavier load times.

7.2.2 Raspberry Pi results

The specifications of the Raspberry Pi are as follows:

1. **CPU**: Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
2. **RAM**: 8GB
3. **Graphic interface**: OpenGL ES 3.1, Vulkan 1.0

Following this, we will analyse each of the key aspects in detail.

FPS

In the graphs, the situation is observed as follows:

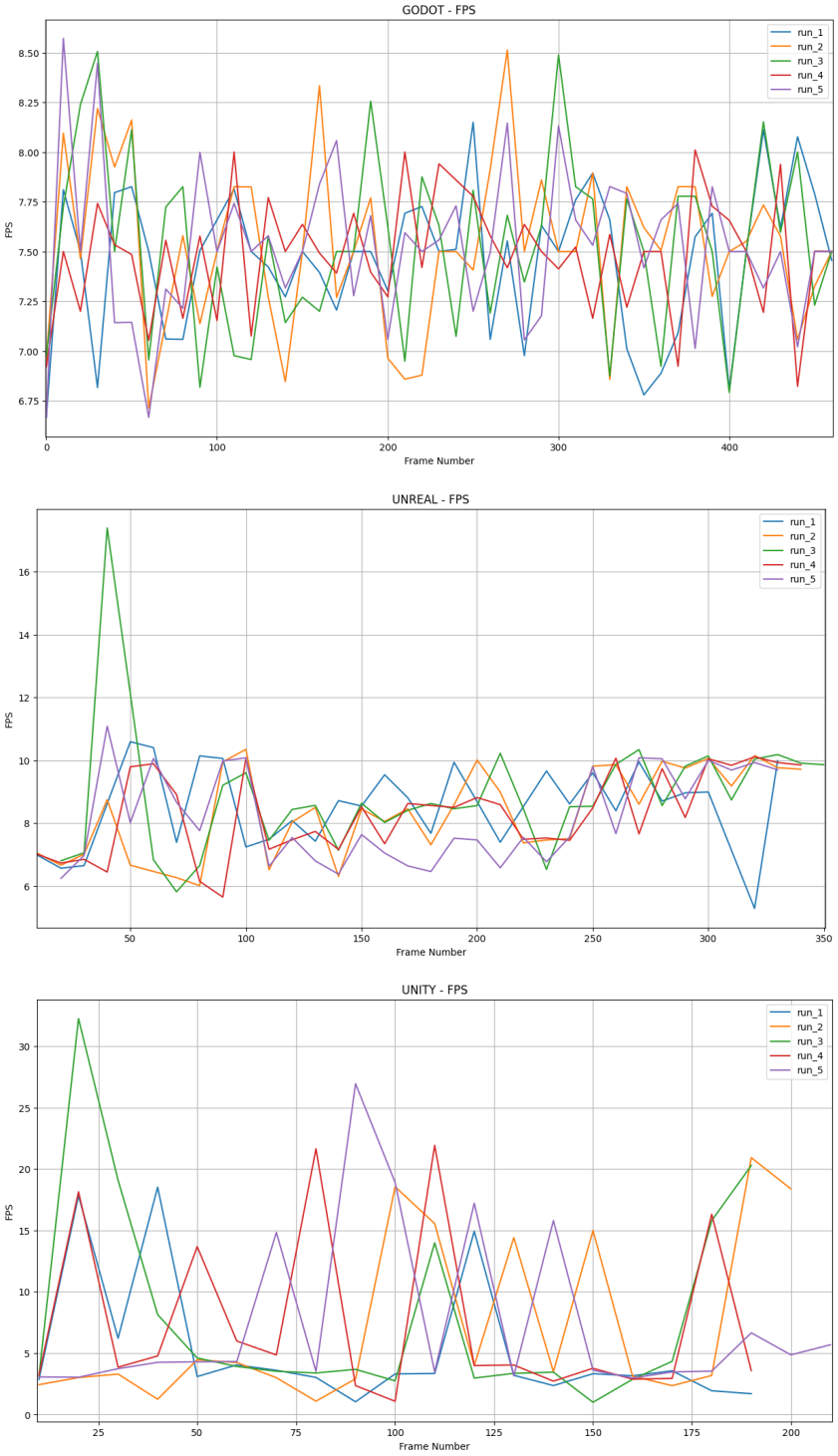

Figure 7.7: FPS graphics in Raspberry Pi

And the table resume:

Engine	Mean Frame Time (ms)	Std Dev (ms)	Avg FPS	FPS Std	Total Frames	Bad Frames	Bad Frames %
Godot	133.40	0.013	7.5	0.00	469.0	5.2	1.11%
Unity	292.55	1.79	3.42	0.02	202.4	2.2	1.09%
Unreal	122.77	3.11	8.15	0.21	341.2	14.4	4.22%

Table 7.8: FPS Stats in Raspberry Pi

Analysing these results, the following observations are made:

Unreal achieves the highest average FPS (8.15) and the lowest mean frame time (122.77 ms), although it exhibits the greatest variability in frame time and the highest percentage of bad frames (4.22%), suggesting occasional performance hiccups.

Godot performs slightly worse than Unreal in terms of average FPS (7.5) and mean frame time (133.40 ms) but maintains much lower variability and a significantly smaller percentage of bad frames (1.11%).

Unity shows the lowest overall performance, with an average FPS of only 3.42 and a mean frame time of 292.55 ms, indicating less smooth gameplay. Nevertheless, it keeps the percentage of bad frames similar to Godot's (1.09%).

CPU

The following table presents the results for the Raspberry Pi 4 system:

Engine	Total CPU Time (ms)	Scripting Time (ms)	CPU Usage (%)
Godot Run 1	113,166	107,912	95.36%
Godot Run 2	110,631	105,552	95.41%
Godot Run 3	110,560	105,389	95.32%
Godot Run 4	114,477	108,322	94.62%
Godot Run 5	111,068	106,095	95.51%
Unity Run 1	74,790	22,370	29.91%
Unity Run 2	81,514	22,117	27.14%
Unity Run 3	74,417	22,031	29.60%
Unity Run 4	75,097	21,757	28.97%
Unity Run 5	79,430	22,750	28.63%
Unreal Run 1	94,368	63,693	67.46%
Unreal Run 2	96,089	64,377	66.97%
Unreal Run 3	95,473	64,252	67.26%
Unreal Run 4	94,240	62,763	66.58%
Unreal Run 5	97,915	63,023	64.38%

Table 7.9: CPU Usage in Raspberry Pî

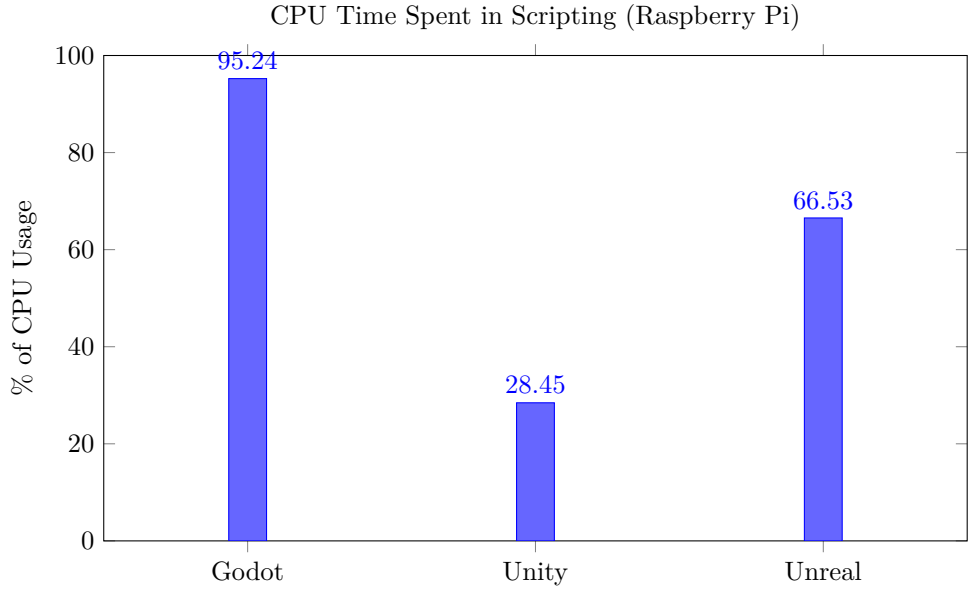

Figure 7.8: CPU usage graphic in Raspberry Pi

Godot exhibits extremely high scripting utilisation, consistently using around **95%** of CPU time for script execution. This indicates a very script-intensive execution model, which, while efficient in code handling, may saturate the CPU quickly in low-end environments.

Unity shows significantly lower CPU scripting usage (around **28%**), suggesting better delegation to other engine components or more efficient runtime code execution.

Unreal lies in between, with scripting accounting for **66.5%** of CPU time. This balance likely stems from Unreal's complex internal processes and heavier rendering pipelines that share CPU time with scripting tasks.

Memory heap

The test results for the Raspberry Pi system are as follows:

Table 7.10: Godot – Sizes per Run (Raspberry Pi)

Run	Initial Size (MB)	Maximum Size (MB)
1	0.269	11.4
2	0.269	11.3
3	0.269	11.5
4	0.269	11.4
5	0.269	11.8

Table 7.11: Unity – Sizes per Run (Raspberry Pi)

Run	Initial Size (MB)	Maximum Size (MB)
1	1.1	14.5
2	1.1	14.7
3	1.2	14.7
4	1.2	14.5
5	1.2	14.7

Table 7.12: Unreal – Sizes per Run (Raspberry Pi)

Run	Initial Size (MB)	Maximum Size (MB)
1	1.1	32.7
2	1.3	32.8
3	1.3	32.7
4	1.1	32.8
5	1.1	32.7

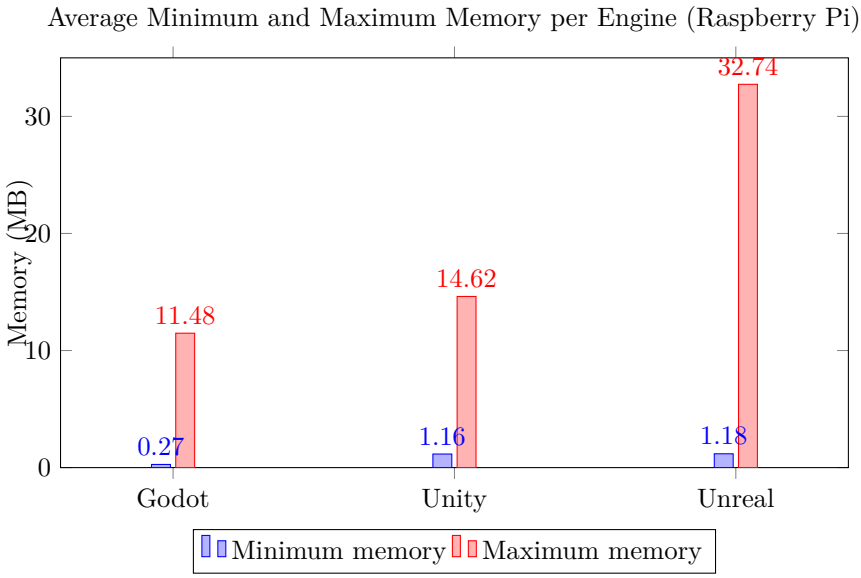

Figure 7.9: Memory Heap Graphic in Raspberry Pi

On the Raspberry Pi, no modifications were observed in the heap memory usage.

Load time

Table 7.13: Godot – Load Time

Run	Load Time (s)
Run 1	17.90
Run 2	16.75
Run 3	16.56
Run 4	17.10
Run 5	16.82

Table 7.14: Unity – Load Time

Run	Load Time (s)
Run 1	14.49
Run 2	13.65
Run 3	13.76
Run 4	13.59
Run 5	13.88

Table 7.15: Unreal – Load Time

Run	Load Time (s)
Run 1	52.89
Run 2	57.74
Run 3	53.39
Run 4	51.94
Run 5	52.64

All of this is summarised in the following chart:

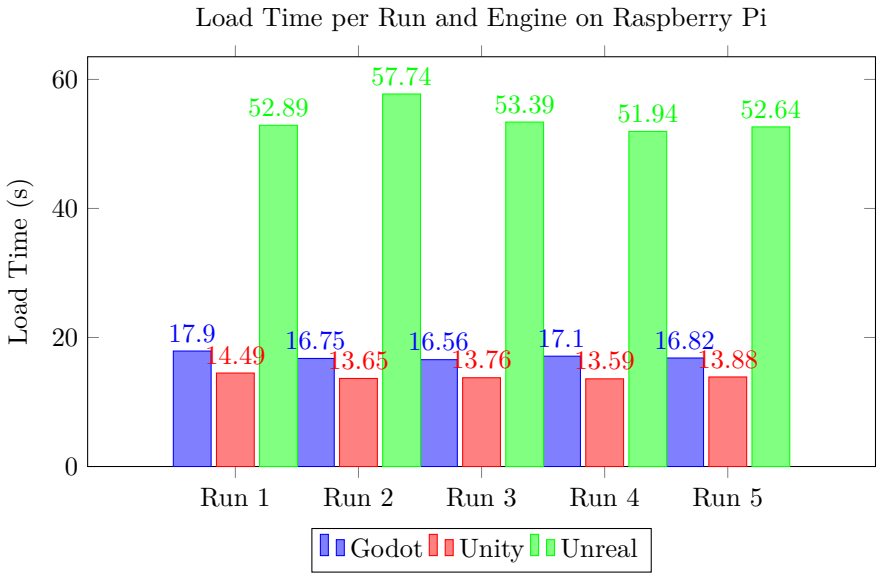

Figure 7.10: Load time graphic in Raspberry Pi

The load times on the Raspberry Pi show a clear difference between engines. Godot maintains a steady load time around **16.5–17.9 s**, Unity is consistently faster, around **13.5–14.5 s**, while Unreal exhibits significantly longer load times, ranging from **51.9** to **57.7 s**. These results highlight the greater resource demands of Unreal on this platform, while Godot and Unity perform more efficiently in terms of loading speed.

7.3 Performance Analysis

Following the tests carried out in both desktop and Raspberry Pi environments, the following conclusions have been drawn based on four key performance pillars:

FPS

- **Godot** achieved the highest average FPS on desktop, demonstrating smooth and stable graphical performance, particularly suitable for lightweight or 2D experiences.
- **Unity** remained competitive, albeit slightly behind Godot, with consistently high and stable performance.
- **Unreal** proved to be the most demanding engine; although it delivered decent performance, it did not reach the frame rates achieved by the other two engines.

CPU usage

- **Godot** exhibited a notably high CPU usage, particularly in scripting (approximately 83.7%), indicating that a significant portion of the workload is handled by the code logic rather than the graphics engine.
- **Unity** distributed the CPU load more efficiently, with scripting accounting for only around 27.3%, reflecting a well-optimised and modular internal structure.
- **Unreal** reported the highest total CPU usage, although scripting represented only approximately 42.4%, as most of the resources were allocated to rendering and advanced engine subsystems.

Memory heap

- **Godot** again stood out due to its extreme efficiency: starting at just 269KB and reaching a maximum of 17MB on Raspberry Pi, making it ideal for low-resource environments.
- **Unity** required slightly more memory (up to 17.8MB) but remained stable and predictable within acceptable limits.
- **Unreal** reached peaks of up to 40MB, indicating a clear prioritisation of visual and technical capability over memory efficiency.

Load time

- **Godot** demonstrated a slow initial load time (˜7.3s), but subsequent runs stabilised at around 2 seconds, likely due to internal caching or initialisation mechanisms.
- **Unity** provided the most consistent and fastest load times from the first execution (˜5s), making it suitable for use cases where short load times are critical.
- **Unreal** was once again the slowest in this aspect, with load times ranging between 6.3 and 8 seconds, reflecting the engine's complexity and resource requirements.

Performance on Raspberry Pi

On Raspberry Pi, none of the three engines delivered functionally acceptable performance. To contextualise these results, a basic WebGL performance test was conducted using a simple animation of a rotating cube with a black background, lighting, and colours. The trace recorded in Chromium is shown below:

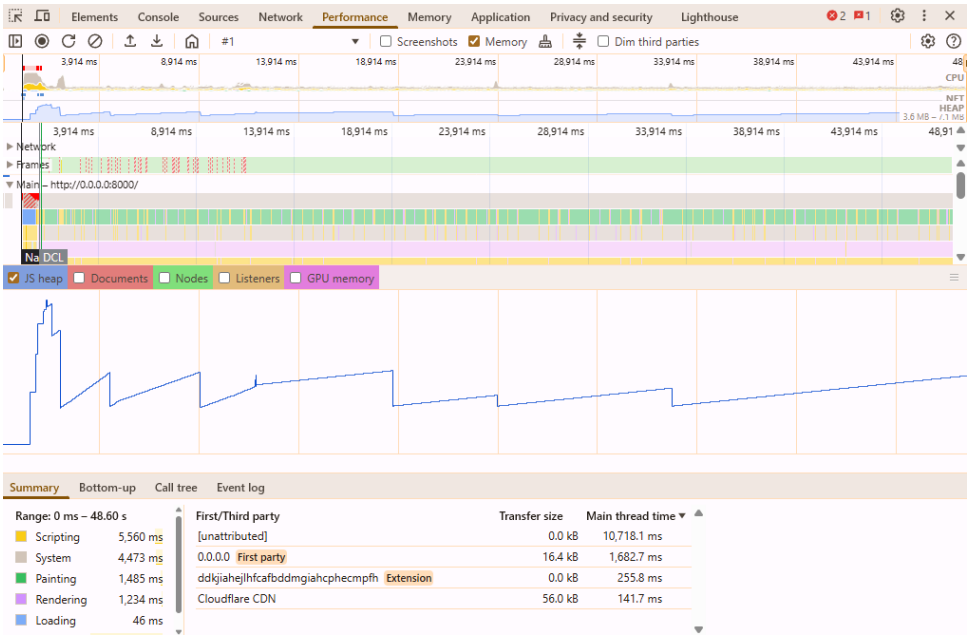

Figure 7.11: Performance about cube rotating in webGl with CDP

As observed, even such a simple scene exhibits frame rate fluctuations between 30 and 60 FPS. This is due to the fact that only the rendering and painting systems are active. A complete engine involves additional systems—physics, AI, networking, animations, etc.—which significantly increase the workload.

Storage

The final build sizes were also evaluated:

- **Unity** → 93.1 MB
- **Unreal** → 60.2 MB
- **Godot** → 70.6 MB

Unity produced the largest build size, primarily because it includes all internal libraries required for execution, even those unrelated to the specific implementation. This also accounts for the longer build times compared to Godot or Unreal.

It is also worth noting that the version used, **Unreal Engine 4.27.2**, does not officially support WebGL export. An *ad hoc* solution based on WebGL 2.0 was required to conduct the tests. Therefore, Unreal is effectively disqualified as a viable solution for modern WebGL development.

Final summary

For scenarios where **maximum efficiency and portability** are required, **Godot** is the optimal choice. However, it is important to acknowledge that Godot is a relatively modern engine compared to Unity and Unreal. Several of its features, such as the navigation mesh system, are still experimental or under development.

Unity, by contrast, represents an excellent balance between performance, compatibility, and stability. Its modular architecture, extensive community support, and consistent results make it the most robust alternative—particularly for projects seeking a middle ground between efficiency and capability.

Unreal, while technically impressive, is excluded from consideration in this context due to its high resource consumption, lack of official WebGL support, and excessive load times.

Following the presentation and analysis of the performance results across the different platforms, a clear understanding is established regarding the strengths and limitations of each game engine in terms of efficiency and execution capability.

These findings will form the basis for the final conclusions and proposed future work outlined in the subsequent chapter, where key insights will be summarised and potential improvements and optimisations will be suggested.

Chapter 8

Conclusions and Future work

8.1 Conclusions

During the course of this work, a comprehensive effort was made to compare and analyse the performance of three different game engines, alongside addressing numerous technical challenges. Below is a step-by-step breakdown of the work carried out:

- A functional mini-game prototype was developed in three leading industry engines: Unity, Unreal Engine, and Godot, adapted for export to web format.
- The prototype includes two game modes: manual, with direct player control, and automatic, where the system navigates and shoots at predefined targets.
- The game was exported and tested in resource-limited environments, including a modest PC and a Raspberry Pi running Raspberry Pi OS.
- Key performance parameters such as FPS, CPU, and memory usage were defined and measured using profiling tools integrated into web browsers, ensuring data comparability and objectivity.
- Each engine was tested through multiple runs to guarantee statistical reliability and eliminate bias or anomalies.
- Comparative analysis of the results identified the strengths and limitations of each engine in web environments and low-power hardware.
- It was found that engine choice should balance performance, graphical quality, and ease of export—factors that vary depending on the use case and target hardware.
- This work provides a technical and practical foundation for developers interested in optimising games for web platforms and resource-constrained devices such as the Raspberry Pi.

Of course, there were complications during the development of this project, including:

- **Hardware Compatibility:** one of the computers used had a GTX 260 GPU, incompatible with the drivers required for Unity 6, which necessitated waiting a week to receive a new PC that could run Unity without issues. In Unreal Engine, only version 4.27.2 could be used due to similar limitations.
- **Scene Import and Adjustment:** importing the scene from Sketchfab caused more problems than benefits. Manual rescaling and adaptation of the navmesh were required for it to function correctly across all three engines, a tedious and repetitive process.
- **Handling Three Different Engines and Languages:** mastering Unity (C#), Unreal (C++/Blueprints), and Godot (GDScript) simultaneously proved to be an odyssey, with learning curves and conceptual differences that slowed progress.
- **Reliable FPS Measurement:** the use of the CDP for measuring FPS was key, as the native tools of each engine did not offer homogeneous or reliable comparisons for web performance. Learning to use these profilers involved a steep curve.
- **Technical Limitations of Web Export:** exporting to the web from each engine posed various technical challenges, from build configurations to specific optimisations required to maintain

playability on limited hardware. For example, Unreal required using a community version available on GitHub, as the official version no longer exists or is maintained.

The learnings taken away as a computer engineer are also essential and important; these are:

- In-depth exploration of three major game engines (Unity, Unreal Engine, and Godot), gaining practical knowledge in C#, Blueprints, and GDScript respectively.
- Improved understanding and application of profiling and performance analysis techniques, particularly using the CDP to obtain reliable metrics in web environments.
- Applied software engineering methodologies to analyse, design, and document the development process, ensuring an organised and professional approach.
- Task management and submission were learned using Jira, which allowed clear organisation, rigorous tracking, and efficient prioritisation of daily work.
- Plastic SCM was used for version control, improving the ability to work as a team and handle code conflicts, something essential for collaborative projects.
- Corrections and feedback received during the process were integrated, developing a mindset of continuous improvement that will be key in the professional environment.
- The experience allowed familiarisation with agile workflows, progress reporting, and effective communication with the team, aspects that will facilitate adaptation to the professional daily routine.

8.1.1 Actual Project Cost

Although the original plan anticipated project completion by the end of May, development extended until 1st July, resulting in an approximate four-week delay. This extension was partly due to an additional two weeks required for assimilating key concepts, reflecting the necessary learning curve. Furthermore, risk R01 was activated for one week, and risk R07 occurred over three days, causing further delays. Despite these setbacks, the documentation was finalised and valuable conclusions were drawn to meet the project objectives.

The actual project cost has been calculated taking the following factors into account:

- **Engineer hourly rate**:
$$\frac{40,000\,€}{52 \times 40} = 19.23\,€/\text{hour}$$
- **Total hours worked** (22 weeks × 40 hours/week):
$$22 \times 40 = 880\,\text{hours}$$
- **Labour cost excluding risks**:
$$880 \times 19.23 = 16,922.40\,€$$
- **Amortisation cost of original hardware** (PC, 1 month out of 36 months):
$$\frac{1,000\,€}{36} \times 1 = 27.78\,€$$
- **Cost of replacement hardware** (due to incompatibility issues):
$$1,000\,€$$
- **Amortisation cost of Raspberry Pi 4**[32] (6 months out of 36 months):
$$\frac{82,50\,€}{36} \times 6 \approx 13.75\,€$$
- **Additional cost due to activated risks** (1 week for risk R01 and 3 days for risk R07):
$$(5 + 3) \times 8 = 64\,\text{hours}$$
$$64 \times 19.23 = 1,230.72\,€$$

It should be noted that the original estimated total cost of

$$\mathbf{18,873.68}\,€$$

was calculated assuming that all anticipated costs and risk contingencies would be fully realised.

Therefore, the actual project cost of

$$\mathbf{19,194.65}\,€$$

exceeds the original estimate by approximately **320.97 €**, mainly due to the extended development period and the necessity of acquiring replacement hardware.

8.2 Future work

Based on the results obtained in this study, two main lines of future work have been identified, both of which could provide deeper insight into the performance and viability of the evaluated engines:

- **Optimisation of minimal builds:** A key next step will involve identifying and removing unnecessary libraries that are included by default in Unity-generated projects. The aim is to reduce the final build size and assess the actual impact on load times and resource usage. This optimisation would make it possible to evaluate the engine's performance under more constrained and realistic conditions, particularly for resource-limited environments such as embedded systems or low-powered browsers.
- **Development of a custom basic engine:** The creation of a minimal, purpose-built graphics engine is proposed, specifically tailored for comparative performance testing. To avoid the complexities of working directly with WebGL, an abstraction layer based on the *Three.js* library will be used. This will allow a focus on the essential aspects of the rendering system. A direct comparison with existing engines will help identify which functionalities have a meaningful performance impact and what trade-offs are necessary in terms of scalability and maintainability.

Bibliography

[1] M. S. Saleem, "Top advances in gaming technology in 2025 – important for gaming entrepreneurs," https://www.tekrevol.com/blogs/top-advances-in-gaming-technology, accessed: Apr. 1, 2025.

[2] R. Kreese, "Zelda famicom, screaming in the microphone!" https://youtu.be/A2UtC_SwAfY?si=1kWT3-BdrP4cXL1U, 2008, accessed: Apr. 21, 2025.

[3] D. M. Waqar, T. S. Gunawan, M. Kartiwi, and R. Ahmad, "Real-time voice-controlled game interaction using convolutional neural networks," in *2021 IEEE 7th International Conference on Smart Instrumentation, Measurement and Applications (ICSIMA)*, 2021, pp. 76–81. [Online]. Available: https://doi.org/10.1109/ICSIMA50015.2021.9526318

[4] J. V. Moniaga, A. Chowanda, A. Prima, Oscar, and M. D. Tri Rizqi, "Facial expression recognition as dynamic game balancing system," *Procedia Computer Science*, vol. 135, pp. 361–368, 2018, the 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050918314741

[5] F. Dehghani and L. Zaman, "Facial emotion recognition in vr games," 2023. [Online]. Available: https://arxiv.org/abs/2312.06925

[6] Y. Zhu and B. Yuan, "Real-time hand gesture recognition with kinect for playing racing video games," in *2014 International Joint Conference on Neural Networks (IJCNN)*, 2014, pp. 3240–3246. [Online]. Available: http://www.cmap.polytechnique.fr/~nikolaus.hansen/proceedings/2014/WCCI/IJCNN-2014/PROGRAM/N-14190.pdf

[7] N. Alnaim, "Hand gesture recognition using deep learning neural networks," Ph.D. dissertation, Brunel University London, 2020. [Online]. Available: https://bura.brunel.ac.uk/bitstream/2438/20923/1/FulltextThesis.pdf

[8] G. Paizanis, R. Schonfeld, E. Pagano, and N. Schmidt, "Leveling up for the new reality," https://www.bcg.com/publications/2024/leveling-up-new-reality, accessed: Mar. 31, 2025.

[9] Anonymous, "The future of the global gaming industry: Opportunities amid industry challenges," https://www.bcg.com/press/12december2024-future-of-global-gaming-industry, accessed: Jan. 4, 2025.

[10] D. Peppiatt, "Video game music has arrived on the festival circuit – and it's only going to get bigger," https://www.theguardian.com/games/2025/mar/21/video-game-music-london-soundtrack-festival, accessed: Apr. 1, 2025.

[11] Anonymous, "Gaming is booming and is expected to keep growing. this chart tells you all you need to know," https://www.weforum.org/stories/2022/07/gaming-pandemic-lockdowns-pwc-growth, accessed: Jan. 4, 2025.

[12] E. Khasabo, "The rise of content creators in the gaming industry," https://www.vidovo.com/blog/the-rise-of-content-creators-in-the-gaming-industry, accessed: Jan. 4, 2025.

[13] V. Koski, "Benchmarking and comparison of open-source html5 game engine performance," 2024, accessed: Apr. 21, 2025. [Online]. Available: https://lutpub.lut.fi/handle/10024/168662

[14] A. M. Barczak and H. Woźniak, "Comparative study on game engines," *Studia Informatica. Systems and Information Technology. Systemy i Technologie Informacyjne*, no. 1-2, 2019. [Online]. Available: https://bazawiedzy.uws.edu.pl/info/article/UPH3c6a533b32d74ae89fb8273e94ff1c20/Publikacja+%25E2%2580%2593+Comparative+study+on+game+engines+%25E2%2580%2593+Uniwersytet+Przyrodniczo-Humanistyczny+w+Siedlcach?r=publication&ps=20&lang=en&pn=1&cid=5220

[15] Anonymous, "Mobile game app development cost: A detailed guide 2025," https://www.apptunix.com/blog/game-development-cost-how-much-does-it-cost-to-develop-a-game-app, accessed: Jan. 4, 2025.

[16] "Raspberry pi 4 tech specs," accessed: Apr. 22, 2025. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/

[17] Anonymous, "Broadcom videocore vi," accessed: Apr. 24, 2025. [Online]. Available: https://www.cpu-monkey.com/en/igpu-broadcom_videocore_vi

[18] W. contributors, "Playstation 5," accessed: Apr. 24, 2025. [Online]. Available: https://en.wikipedia.org/wiki/PlayStation_5

[19] "What is directx? why does every pc game need it?" accessed: Apr. 22, 2025. [Online]. Available: https://www.corsair.com/us/en/explorer/gamer/gaming-pcs/what-is-directx-why-does-every-pc-game-need-it/#:~:text=DirectX%20is%20a%20collection%20of,components%20in%20any%20given%20PC.

[20] W. contributors, "Unity (game engine)," https://en.wikipedia.org/wiki/Unity_(game_engine), accessed: Jan. 4, 2025.

[21] ——, "Unreal engine," https://en.wikipedia.org/wiki/Unreal_Engine, accessed: Jan. 4, 2025.

[22] ——, "Godot (game engine)," https://en.wikipedia.org/wiki/Godot_(game_engine), accessed: Jan. 4, 2025.

[23] ——, "Unity version control," accessed: May. 12, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Unity_Version_Control

[24] U. Technologies. (2025) Monobehaviour.start(). Accessed: May. 22, 2025. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Start.html

[25] ——. (2025) Monobehaviour.update(). Accessed: May. 22, 2025. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html

[26] ——. (2025) Monobehaviour.startcoroutine. Accessed: May. 22, 2025. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.StartCoroutine.html

[27] ——. (2025) Input.getaxis. Accessed: May. 23, 2025. [Online]. Available: https://docs.unity3d.com/ScriptReference/Input.GetAxis.html

[28] E. G. Developers. (2025) Blueprints visual scripting. Accessed: June. 04, 2025. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine

[29] G. Developers. (2025) Gdscript reference. Accessed: June. 13, 2025. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

[30] mdn web docs. (2025) Webassembly. Accessed: June. 29 2025. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly

[31] P. Developers. (2025) Puppeter. Accessed: June. 29 2025. [Online]. Available: https://pptr.dev/

[32] A. Logo. (2025) Raspberry pi 4 model b - 8 gb ram. Accessed: June. 29 2025. [Online]. Available: https://www.adafruit.com/product/4564?src=raspberrypi

[33] W. mathworld. (2025) Euler angles. Accessed: May. 23, 2025. [Online]. Available: https://mathworld.wolfram.com/EulerAngles.html

[34] U. Technologies. (2025) Quaternion.euler. Accessed: May. 23, 2025. [Online]. Available: https://docs.unity3d.com/6000.1/Documentation/ScriptReference/Quaternion.Euler.html

[35] W. van Toll, R. Triesscheijn, M. Kallmann, R. Oliva, N. Pelechano, J. Pettré, and R. Geraerts. (2016) A comparative study of navigation meshes. Accessed: June. 27 2025. [Online]. Available: https://www.cs.upc.edu/~npelechano/MIG2016_Wouter.pdf

[36] Anonymous, "What is cloud gaming?" https://www.microsoft.com/en-us/edge/learning-center/what-is-cloud-gaming?form=MA13I2, accessed: Apr. 16, 2025.

[37] ——, "What are convolutional neural networks?" https://www.ibm.com/think/topics/convolutional-neural-networks, accessed: Apr. 21, 2025.

[38] "Nvidia dlss," accessed: Apr. 22, 2025. [Online]. Available: https://developer.nvidia.com/rtx/dlss?sortBy=developer_learning_library%2Fsort%2Ffeatured%3Adesc%2Ctitle%3Aasc&hitsPerPage=6

[39] "Amd fidelityfx™ super resolution," accessed: Apr. 22, 2025. [Online]. Available: https://www.amd.com/en/products/graphics/technologies/fidelityfx/super-resolution.html#requirements

[40] Anonymous, "What is the k-nearest neighbors (knn) algorithm?" https://www.ibm.com/think/topics/knn#:~:text=The%20k%2Dnearest%20neighbors%20(KNN)%20algorithm%20is%20a,regression%20classifiers%20used%20in%20machine%20learning%20today., accessed: Apr. 21, 2025.

[41] ——, "Preface: What is opengl?" accessed: Apr. 24, 2025. [Online]. Available: https://openglbook.com/chapter-0-preface-what-is-opengl.html

[42] W. contributors, "Snake (video game genre)," https://en.wikipedia.org/wiki/Snake_(video_game_genre), accessed: Apr. 21, 2025.

[43] Anonymous, "What are support vector machines (svms)?" https://www.ibm.com/think/topics/support-vector-machine., accessed: Apr. 21, 2025.

[44] Vulkan, "What is vulkan?" accessed: Apr. 24, 2025. [Online]. Available: https://vulkan.lunarg.com/doc/view/1.4.304.1/mac/antora/guide/latest/what_is_vulkan.html

Appendix A

3D Rotation Theory — Euler Angles, Gimbal Lock, and Quaternions

A.1 Euler angles

Euler angles are a method of describing the orientation of a coordinate system in three-dimensional space by means of three successive rotations around axes, which may be either fixed or moving. There are a total of 12 possible sequences of Euler angle rotations [33]. For example, consider the ZXY convention, which is the one used by Unity [34]. In this case, the rotations are applied in the following order: first around the Z axis, then around the X axis, and finally around the Y axis.

Let:

- ϕ: **roll** angle (rotation around the Z-axis)
- θ: **pitch** angle (rotation around the X-axis)
- ψ: **yaw** angle (rotation around the Y-axis)

The elementary rotation matrices are as follows:

$$R_Z(\phi) = \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad R_X(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \quad R_Y(\psi) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix}$$

The total rotation matrix, applying the rotations in the ZXY order, is:

$$\mathbf{R} = R_Y(\psi) \cdot R_X(\theta) \cdot R_Z(\phi)$$

Step 1: multiply $R_X(\theta)$ and $R_Z(\phi)$:

$$R_X(\theta)R_Z(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \cdot \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \cos\theta\sin\phi & \cos\theta\cos\phi & -\sin\theta \\ \sin\theta\sin\phi & \sin\theta\cos\phi & \cos\theta \end{bmatrix}$$

Step 2: multiply $R_Y(\psi)$ with the previous result:

$$\mathbf{R} = R_Y(\psi) \cdot (R_X(\theta)R_Z(\phi)) = \begin{bmatrix} \cos\psi & 0 & \sin\psi \\ 0 & 1 & 0 \\ -\sin\psi & 0 & \cos\psi \end{bmatrix} \cdot \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \cos\theta\sin\phi & \cos\theta\cos\phi & -\sin\theta \\ \sin\theta\sin\phi & \sin\theta\cos\phi & \cos\theta \end{bmatrix}$$

$$= \begin{bmatrix} \cos\psi\cos\phi + \sin\psi\sin\theta\sin\phi & -\cos\psi\sin\phi + \sin\psi\sin\theta\cos\phi & \sin\psi\cos\theta \\ \cos\theta\sin\phi & \cos\theta\cos\phi & -\sin\theta \\ -\sin\psi\cos\phi + \cos\psi\sin\theta\sin\phi & \sin\psi\sin\phi + \cos\psi\sin\theta\cos\phi & \cos\psi\cos\theta \end{bmatrix}$$

With this, any point in three-dimensional space can be rotated. Let $\boldsymbol{v} \in \mathbb{R}^3$, be a vector; then the rotated vector is given by:

$$\boldsymbol{v}' = \mathbf{R}\boldsymbol{v}$$

As can be observed, this results in a hierarchy of rotations, since each subsequent rotation is applied within a new coordinate system that has already been modified by the previous rotations. Specifically:

- The rotation about the Y-axis is applied in the original coordinate system.
- The rotation about the X-axis is applied within a system already rotated by Y.
- The rotation about the Z-axis is applied within a system that has already been rotated by both Y and X.

This leads to a common issue in computer graphics known as *gimbal lock*.

A.2 Gimbal lock problem

Before describing the problem, it is necessary to demonstrate how the Euler angles can be recovered from the rotation matrix. The total rotation matrix in ZXY order is then obtained:

$$\mathbf{R} = \begin{bmatrix} \cos\psi\cos\phi + \sin\psi\sin\theta\sin\phi & -\cos\psi\sin\phi + \sin\psi\sin\theta\cos\phi & \sin\psi\cos\theta \\ \cos\theta\sin\phi & \cos\theta\cos\phi & -\sin\theta \\ -\sin\psi\cos\phi + \cos\psi\sin\theta\sin\phi & \sin\psi\sin\phi + \cos\psi\sin\theta\cos\phi & \cos\psi\cos\theta \end{bmatrix}$$

Step 1: calculation of θ

From the element R_{23}:

$$R_{23} = -\sin\theta \implies \theta = \arcsin(-R_{23})$$

Step 2: calculation of ψ

Use of the elements R_{13} and R_{33}:

$$R_{13} = \sin\psi\cos\theta$$
$$R_{33} = \cos\psi\cos\theta$$

Dividing both:

$$\tan\psi = \frac{R_{13}}{R_{33}} \Rightarrow \boxed{\psi = \arctan 2(R_{13}, R_{33})}$$

Step 3: calculation of ϕ

Use of the elements R_{21} and R_{22}:

$$R_{21} = \cos\theta\sin\phi$$
$$R_{22} = \cos\theta\cos\phi$$

Dividing both:

$$\tan\phi = \frac{R_{21}}{R_{22}} \Rightarrow \boxed{\phi = \arctan 2(R_{21}, R_{22})}$$

Here lies the problem. This procedure assumes that $\cos\theta \neq 0$. if $\theta = \pm\frac{\pi}{2}$, then:

$$R_{21} = 0$$
$$R_{22} = 0$$
$$R_{13} = 0$$
$$R_{33} = 0$$

and therefore

$$\tan\psi = \frac{0}{0} \Rightarrow \boxed{\psi = \arctan 2(0,0) \Rightarrow indefinied}$$
$$\tan\phi = \frac{0}{0} \Rightarrow \boxed{\phi = \arctan 2(0,0) \Rightarrow indefinied}$$

This is the issue known as *gimbal lock* It causes the expressions for the elements of the rotation matrix to simplify in such a way that two of the rotation axes align, effectively eliminating one degree of freedom. That is, the ability to rotate independently around one of the axes is lost.

Example: if $\theta = \frac{\pi}{2}$, then:

$$R = \begin{bmatrix} \sin\psi\sin\phi + \cos\psi\cos\phi & \sin\psi\cos\phi - \cos\psi\sin\phi & 0 \\ 0 & 0 & -1 \\ \cos\psi\sin\phi - \sin\psi\cos\phi & \cos\psi\cos\phi + \sin\psi\sin\phi & 0 \end{bmatrix}$$

It is observed that the entries $R_{13}, R_{21}, R_{22}, R_{33}$ become zero, indicating insufficient information to distinguish between the angles ψ and ϕ: they have **collapsed into a single composite rotation**.

Graphically, the system loses the ability to rotate independently around one of the axes, which can lead to unexpected or undesired behaviour in simulations or animations. This is a common problem in representations using Euler angles.

To address this, another tool is used that resolves this issue: the quaternion.

A.3 Quaternions

Quaternions are an extension of complex numbers, a quaternion is defined as:

$$q = w + xi + yj + zk$$

where $w, x, y, z \in \mathbb{R}$ and $\{i, j, k\}$ are the imaginary units that satisfy the following relations:

$$i^2 = j^2 = k^2 = ijk = -1$$

Before addressing rotations using quaternions, it is appropriate to introduce some preliminary concepts.

The norm of q, denoted $\|q\|$, is given by:

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2}$$

The inverse of a quaternion q is given by:

$$q^{-1} = \frac{q^*}{\|q\|^2}$$

where $q^* = w - xi - yj - zk$ is the **conjugate** of q.

The demonstration is now presented.

Let $\boldsymbol{v} \in \mathbb{R}^3$, be a vector; Its representation as a quaternion would be the following:

$$\mathbf{v}_q = 0 + xi + yj + zk,$$

This is known as a pure quaternion. The rotation is defined by a unit quaternion ($\|q\| = 1$) that represents the axis and angle of rotation:

$$q = \cos\left(\frac{\theta}{2}\right) + (u_x i + u_y j + u_z k)\sin\left(\frac{\theta}{2}\right),$$

where:

- θ is the total angle of rotation,
- (u_x, u_y, u_z) is a unit vector indicating the axis of rotation.

To rotate the vector $\vec{v}$, the calculation performed is:

$$\mathbf{v}' = q\mathbf{v_q}q^{-1},$$

Since a unit quaternion is used, its inverse equals the conjugate, which simplifies the calculations. As can be observed, the rotation depends solely on a single operation with a unique angle, and therefore the gimbal lock issue does not occur.

Appendix B

Navigation Mesh Concept Overview

A navigation mesh is a simplified polygonal map of a virtual space that helps characters figure out where they can walk and how to get from point A to point B smoothly, avoiding obstacles and collisions[35].

To define a navigation mesh, it is necessary to first introduce the concept of free space, denoted as E_{free}:

$$E_{\text{free}} = \{x \in \mathbb{R}^3 \mid x \notin \mathcal{O}\}$$

where:

- $\mathbb{R}^3$ is the three-dimensional space in which the environment is defined.
- $\mathcal{O}$ denotes the set of obstacles (e.gwalls, solid objects, or scene boundaries).

Thus, E_{free} comprises all points in space where an agent can move freely without colliding with obstacles.

Once the free space $E_{\text{free}} \subset \mathbb{R}^3$ has been defined, a navigation mesh can be formally described as a tuple:

$$\mathcal{M} = (R, G)$$

where:

- $R = \{R_0, R_1, \ldots, R_n\}$ is a finite set of geometric regions in $\mathbb{R}^3$ that represent E_{free}. Each region R_i is *P-simple*, meaning that its projection onto the ground plane P does not self-intersect.
- $G = (V, E)$ is an undirected graph, where vertices V correspond to the regions R_i, and edges E indicate navigable connectivity between adjacent regions.

Thanks to this, points can be defined in space for the NPC to traverse without needing to know how it does so

Appendix C

Abbreviations and Acronyms

2D 2 Dimensional. 14, 16, 17, 71, 105, 167

3D 3 Dimensional. 14–17, 24, 26, 52, 54, 55, 69, 71, 105–107, 129

AI Artificial Intelligence. 11, 117, 124, 189

API Application Programming Interface. 17, 54, 58, 118, 189

AR Augmented Reality. 11, 54

CDP Chrome DevTools Performance. 147, 151, 153, 171, 172

CNN Convolutional Neural Networks. 11, 12

CPU Central Processing Unit. 14, 17, 52, 151, 153, 161, 162, 171

DLSS Deep Learning Super Sampling. 17

Famicom Family Computer. 11

FBX Filmbox. 54

FPS First Person Shooter. 20, 54, 57, 58, 63, 87, 105

FPS Frames Per Second. 14, 15, 52, 116, 139, 141, 142, 150, 171

FSR FidelityFX™ Super Resolution. 17

GLTF Graphics Library Transmission Format. 54

GNU GNU's Not Unix. 17

GPU Graphics Processing Unit. 14, 17, 52

HUD Heads-Up Display. 24, 27–29, 31, 38, 39, 41–45, 57–59, 69, 72, 76, 77, 80, 87, 94, 98, 100, 102, 105, 110, 114, 117

IDE Integrated Development Environment. 51

JSON JavaScript Object Notation. 24

KNN K-Nearest Neighbors. 12

LTS Long-Term Support. 22

NES Nintendo Entertainment System. 11

NPC Non Playable Character. 50, 116–118, 121, 124–126, 128, 131, 134

OS Operative System. 17

PC Personal Computer. 17, 18, 20, 141, 148

RAM Random Access Memory. 14, 17, 52

RGB Red Green Blue. 12, 84

SVM Support Vector Machine. 12

TFLOPS Tera Floating-Point Operations per Second. 17

URL Uniform Resource Locator. 148

VR Virtual Reality. 11, 12, 54

Appendix D

Glossary

Cloud gaming A type of online gaming that runs video games on remote servers and streams them directly to a user's device.[36]. 11

Convolutional Neural Network Type of neural network used for image classification and computer vision tasks uses three types of layers: the convolutional layer to apply filters to the images to obtain local patterns such as edges, textures, or different shapes; the pooling layer to reduce the dimension of the data and make the network more efficient; and one or more fully connected layers, where each neuron is connected to the neurons of previous layers, used for the final classification or prediction.[37]. 11

Deep Learning Super Sampling Technology developed by NVIDIA that, through the use of deep learning, reduces the game's resolution to increase GPU performance, and then upscales it to the native resolution of the display, making the resolution drop imperceptible.[38]. 17

FidelityFX™ Super Resolution Technology developed by AMD that uses upscaling techniques and temporal image reconstruction with the use of Artificial Intelligence (AI) for frame generation to enhance GPU performance and improve FPS fluidity.[39]. 17

k-nearest neighbors A supervised learning method that makes no assumptions about the shape of the data. This classifier is based on the proximity between data points to make predictions or classify new items, assigning them the most common category or value among these.[40]. 12

OpenGL Cross-language, cross-platform Application Programming Interface (API) that allows a programmer to communicate with graphics hardware.[41]. 17, 158

Snake A classic action video game in which the player controls a moving line — usually represented as a snake — that grows longer each time it collects an item (such as food or points). The objective is to survive for as long as possible without crashing into the edges of the playing area or into the snake's own body, which becomes increasingly difficult as the snake lengthens.[42]. 11

Support Vector Machine Supervised learning method used for classification and regression in machine learning. Its main objective is to find the hyperplane that best separates the different classes in a dataset, maximizing the margin between the classes.[43]. 12

Vulkan cross-platform API and open standard that conformant hardware implementations follow, it was intended to address the shortcomings of OpenGL.[44]. 17, 158