3D Mobile Game Development using Unity Engine

Constantinos Constantinou

2520796

Supervised by:

Dr. Mario Kolberg

April 2020

Dissertation submitted in partial fulfilment for the degree of
Bachelor of Science with Honours *in Computing Science*

Computing Science and Mathematics

University of Stirling

Abstract

Today, video games have been one of the most common entertainment for a number of people. Nevertheless, it is not easy to develop such complex games as they require a combination of creative plot, compassion and communication abilities, in order to find a way to attract the audience's attention.

The purpose of the underlying thesis is to develop a 3D Mobile Game in Unity which would be playable and compatible across various android devices. Inspired by the *"Star Forces: Space Shooter", "Subdivision Infinity: 3D Space Shooter"* and *"Space Racing 3D – Star Race"* games, we created a Rail Shooter game with the viewpoint of 3rd person perspective where the player movements around the screen are limited as the game follows a specific route. The players' ultimate goal is to achieve the highest score by collecting points when destroying enemy spaceships.

In order to check whether the game is attractive to users, we released the game on Google Play and collected 96 responses regarding its compatibility, graphics quality, game performance and overall user experience. The results suggest that the overall user experience was positive. However, the user interface still lacks gameplay settings, and the game generally lacks extra features. More attention needs to be paid to user feedback and, in particular, to the professional suggestions made by Mr. Tim Gatland, the CEO of Rivet Games and his team.

Attestation

I understand the nature of plagiarism, and I am aware of the University's policy on this. I certify that this dissertation reports original work by me during my University project except for the sources mentioned in the reference section as well as assets downloaded from the Unity asset store.

Signature  **Date: 09 April 2020**

Acknowledgements

I would like to express my gratitude to all those who have helped and supported me through the whole dissertation process. First and foremost, I gratefully thank my supervisor Dr. Mario Kolberg who had patience and offered me numerous valuable suggestions during the dissertation writing. In the preparation of this study he has spent time reading my drafts and providing me with useful advice and feedback. Secondly, I would like to thank Mr. Tim Gatland and his team from Rivet Games for spending much time playing my game and providing me with their valuable feedback. Last but not least, I am grateful to my family and friends who have always been there for me and supported me through my studies without a word of complaint even though they had no idea how to help me with coding.

Table of Contents

List of Figures

1 Introduction

Mobile devices have become an integral part of our daily lives. The growing popularity of mobile technologies has led many companies to develop mobile apps, including mobile game apps based on consumer preferences and needs.

Mobile games are considered to be one of the most popular forms of entertainment to date, with a large number of mobile games covering different genres and preferences. According to TrustCrunch, mobile games account for 33% of all app downloads, 74% of consumer spending, and 10% of all time spent on apps.

1.1 Background and Context

Taking things from the beginning, video games date from the 50s, when researchers began to develop basic games and simulations. "OXO" and "Tennis for Two" are considered the world's first video games ever created. In 1952 Alexander S. Douglas originally coded a game called "OXO" on the Electronic Delay Storage Automatic Calculator (EDSAC), as part of his PhD in HCI (Human Computer Interaction) at the University of Cambridge. He created a game of noughts and crosses. A few years later, in 1958, the American nuclear physicist, William A. Higginbotham, launched "Tennis for Two" as a program aimed at reducing the frustration of the Brookhaven National Laboratory's guests. "Tennis for Two" was designed to create a bouncing ball trajectory on the Donner Model 30 analog computer. Ten years after the development of "OXO" a young computer programmer created a well-known "Spacewar!" in 1962 at the Massachusetts Institute of Technology (MIT). "A two-player game featuring warring torpedo firing ships." The interactive PDP-1 minicomputer has been built with a view of a cathode-ray tube and keyboard input. It was the first of its kind to allow several players to simultaneously share the machine, making it suitable for a two-player game.

By the 1970s, video games had become popular in most parts of the world. Over the years, we have experienced both rapid developments in video arcades and video consoles using joysticks and buttons.

1.1.1 Mobile Gaming in the era of feature phones

The first handheld telephone games were released in 1994 when the popular "Tetris" first appeared on a device named Hagenuk MT-2000. Nokia started a very popular game called SNAKE three years later in December 1997. By then, Snake and its clones, pre-installed on all of Nokia's handheld devices, have become one of the most popular video games available in over 350 million smartphones worldwide.

During this time, the most popular mobile games were single-player board games, word games and puzzles of all kinds. Since the gameplay was easy and the phones had limitations, the total time spent

on cell phones was about 30 minutes, which was different from the time usually taken to complete a console game.

1.1.2 Mobile Gaming in era of smartphones

With the launch of the first generation of smart phones and the provision of flat data broadband access, mobile gaming prospects improved in 2006-2007. The launch of the iPhone (late 2007) changed the dynamics of mobile gaming significantly. The introduction of new technologies (screen touch, motion sensor, accurate positioning system, extended monitor, heavy duty storage, HQ audio and embedded cameras, etc.) and widespread network connectivity have made a number of inventions possible, including app stores, online streaming, multiplayer games and games linked to device motion.

A variety of production companies have since 2007 planned to focus exclusively on mobile platforms rather than on the port of existing consoles or PC titles. Internet browsing and downloading from app stores was the main way to play video games.

1.2 Scope and Objectives

In the context of the previous discussion, the current project was inspired by the old game genre "3D Rail Shooter Games" which the player controls a vehicle on a stationary path directly through the environment, overcomes obstacles and shoots incoming enemies with laser guns. This type of games is mostly seen in old game consoles and we will try to develop one using a modern game engine.

1.2.1 Scope

A small team could take years to create a fully functional game, so the full development of this idea goes beyond the reach of this project. What needs to be achieved, however, is a functional game that includes the basic concepts of a game, with a possibility of future development. The focus was on creating a 3D rail shooter game in Unity Engine, the theme of which is a spaceship flying around space trying to destroy enemy spaceships. The player's experience will be recorded with a perspective to improving the game in future updates.

1.2.2 Objectives

The primary aim of this project is to create a virtual environment where players are exposed to a 3D shooter game genre, which will control a spacecraft that will fire laser beams at enemy spaceships and avoid environmental obstacles. Additionally, it covers the development of a single player native game application (or hybrid game that works on multiple platforms if it is applicable on the later stages), a storyline, characters, graphics, animations, sounds and final appearance. The agreed objective for this project is to simply develop a 3D Android Shooting Game from scratch using Unity Engine. The following priorities provide a description of the goals of the project:

- Plan how the game would work and what needs to be implemented.
- Design a game that is fun and keeps the user's attention.
- Design the user interface, the interactions, the story and the logic of the game.
- Use the design to develop the game.
- Test the final product before turning it to the participants to test it.
- Get feedback from the users on what needs to be improved for future updates.

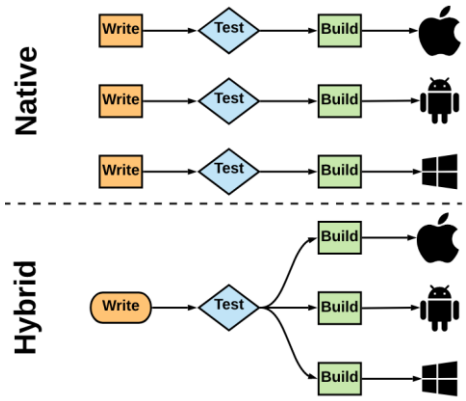


Figure 1. Presenting how Native and Hybrid Apps work

1.3 Achievements

Throughout this dissertation, I have learned and developed a better understanding of gaming development with a particular focus on the Unity Engine. I also learned about a programming language called C#, which was used for Unity scripts using Virtual Studio IDE.

Based on this knowledge, I have developed the Hydra – Rail Shooter Game. The creation of this game enabled further experimentation to gain a deeper understanding of the different optimization techniques and features of the game engine.

Rivet Games 'CEO and his staff reviewed the finished project and deemed it to be a completely working game that satisfies the basic conditions. Moreover, the final version of the game was reviewed by 96 other participants to verify that the game is easy to play, runs well and document each user experience.

1.4 Overview of Dissertation

The remainder of the underlying study is organized as follows. Section 2 goes through relevant state of the art whereas Section 3 captures the requirements the game much have. Section 4 provides the architecture of the game and introduces the tools and concepts used in detailed. Section 5 illustrates how these concepts and features are captured and incorporated in the game whereas in Section 6 we present the results of internal and external testing. Finally, Section 7, concludes, provides overall evaluation and feed for future work.

2 State-of-The-Art

Game Engines provide the architecture to developers to build their games. Developers are enabled to use the physics simulation, terrain editor, object collision detection, advanced lighting, VR support, animations, artificial intelligence and many more without the need to program them. A number of Game Engines are used in the development of games by both professionals and non-professionals. Unity and the Unreal are among the best Game Engines. Both engines provide the basis for programming a game and reflect an ideal place in ease of use and efficiency, as they are perfect for professionals, developers of commercial games and beginners with little coding experience. Both are easily accessible for simple use without cost and support a wide range of operating systems, making them perfect for cross-platform development.

2.1 Unity Engine

Unity was founded in 2005 and created by Unity Technologies in Denmark. Initially, only Apple devices (Mac) supported Unity until 2009, when they switched to Windows. Unity combines the Nvidia PhysX engine with Mono (the open source of Microsoft's .NET libraries) with a custom rendering system [1]. This section discusses what we consider as the main characteristics of Unity which makes it an outstanding game engine, as well as its disadvantages.

2.1.1 Advantages and Features

To begin with, Unity offers comprehensive documentation and examples for its entire API. This is its greatest advantage in improving efficiency compared to other engines, such as Unreal, which only offers limited documentation for non-premium users. Furthermore, an engaging online community of developers will also provide resources for novice users through books and tutorials.

Moving to the technical part, as opposed to Unreal Engine, Unity's editor is the simplest to use. As seen in the Figure 2, content is displayed in the Project window (green area) which the developer can drag and drop them to the game world. The Hierarchy window (cyan area) displays all the objects that are used in the game environment, each of which can be attached to a number of scripts written in C#, Boo (Python-inspired), or JavaScript, as well as rendering and physics properties. Scripts can provide interactive actions to game-objects, construct user interfaces, or simply manage information.

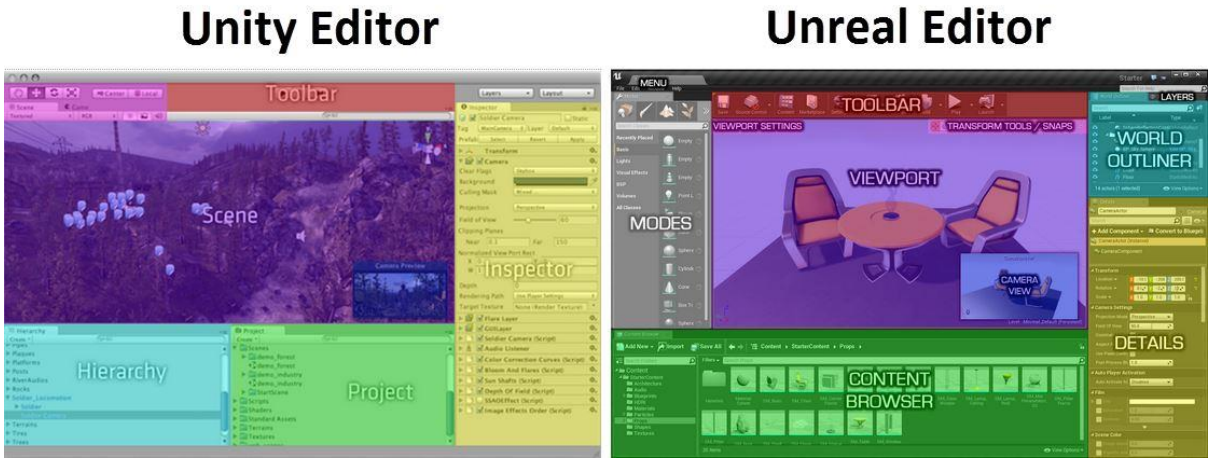


Figure 2. Interfaces of Unity 3D and Unreal 4 [2]

By using physics properties, objects can be assembled by a variety of joints, as well as, by mass, drag, springiness, bounciness and collision detection. Nvidia's PhysX technology simulates the physical properties of several commercial AAA titles. In addition, the rendering characteristics include shared and texture assignments that affect the appearance of visible objects. The Custom Rendering Engine from Unity uses a condensed shader language that is translated into DirectX 11 or OpenGL 3.2 shaders to OpenGL 4.5 based on the target platform.

Further to the above, Unity by default has faster development cycle and much faster build times in comparison with Unreal engine e.g. importing assets or exporting the game. Moreover, it has its own simplified User Interface and well-developed 2D tools when it comes to the development of 2D games. A 2D game in Unreal may be rendered with the Paper 2D system, but Unreal is not completely optimized for 2D games [3]. Compared to Unity's better workflow and its refined final product, it will add needless effort and complexity. This basic fact led to the widespread adoption of Unity by mobile developers – after all, most mobile games are 2D in nature.

2.1.2 Disadvantages

On the other hand, the documentation for some features is often outdated and, in some cases, completely inexistent. In addition, technologies such as Texture Rendering Engines, Profilers and Stencil Buffer support come with essential features at no cost and the price starts with £1.300, if a developer wants access to premium features. These technologies are commonly fully provided for free amongst other game engines. Last but not least, the "out of the box" visual quality is not considered as strong as Unreal's and most of the times it comes with performance issues.

2.2 Unreal Engine

The Unreal Engine is a game development engine created by Epic Games. Unreal Engine made its first reveal with the successful Unreal FPS game released for PC in 1998 [4]. It was originally created to

develop first-person shooting games but over time, it has been adapted to support any genre of game. In terms of practical gameplay programming, there are two main programming choices for Unreal Engine: C++ and Blueprints. Unreal Engine approaches to make C++ feel more like a scripting language and uses a technology called Live++ that does not allow the gameplay logic to be recompiled. On the other hand, blueprints is a dynamic virtual programming framework capable of compiling C++ code (see Figure 3). The blueprint code can be compiled 10 times faster than a dedicated C++ code.

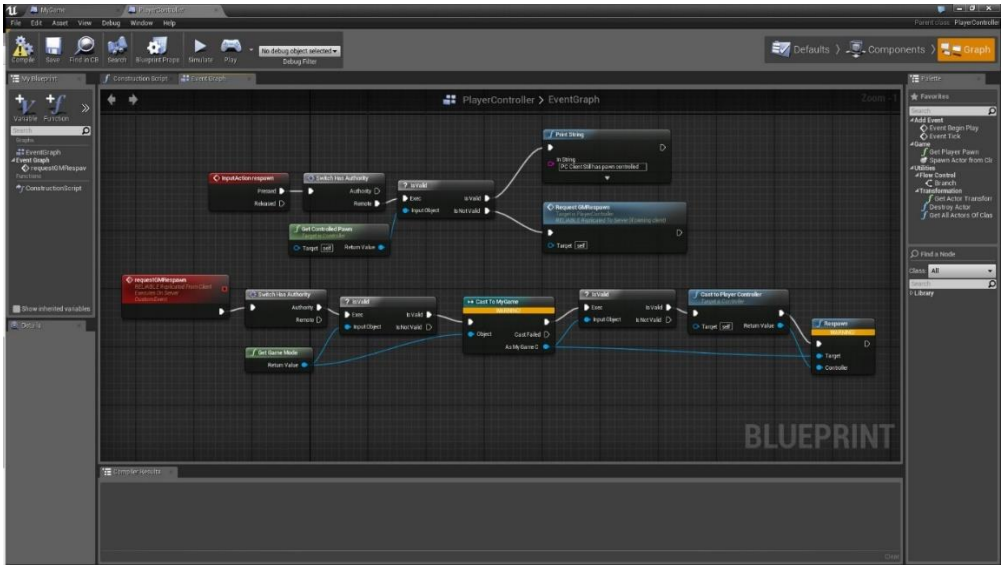


Figure 3. Unreal Engine 4 Blueprint Sample [5]

2.2.1 Advantages

Unreal Engine has a number of benefits. First, the graphics have better characteristics and are known for making High Tech AAA games. Secondly, it has a great tool called Blueprints that works as building blocks so that even designers without programming experience can use the engine to set up and modify fundamentals elements, such as, player movement mechanics, object collision, etc. This feature is ideal for prototyping, however without C++ knowledge it would be difficult to complete serious tasks.

Thirdly, Unreal is an open source that makes game development simpler and more effective. A developer, for example, can repair a game engine flow from its source code and continue developing their game rather than waiting for an official update. Additionally, on every patch update most of the bug fixes on the engine are done by the community. Another advantage of Unreal is its performance, as it uses less memory and great tools for game optimization. Finally, Unreal can access visual debugging more efficiently as the developer can control the entire process and track the time spent on a particular aspect.

2.2.2 Disadvantages

However, Unreal also has few drawbacks. Let's start by mentioning that this engine is not the best tool for developing small games. Even if a game is just a scene and a few scripts, the game build contains

lots of data. Therefore, small games use more space than they should. In addition, as mentioned above, the Unreal engine supports 2D, but the clear focus of the engine is on 3D graphics and shaders.

Further to the above, the underlying engine can be frustrating for one person as it is more suited for teams than a single person, making the learning curve even steeper. In addition, Unreal can be incredibly slow to import and build assets, and, it has a much smaller developer community with learning materials (books, tutorials, etc.).

2.2.3 Conclusion

After spending some time exploring each editor (Unity and Unreal's Editor), it was found that Unity Engine was easier to use as switching to and from Java and C# language is far easier than C++. Therefore, for beginners like myself with prior Java background, it would be more suitable to use the Unity Engine. Basic stuff like camera movement, physics system, third person character movement are easier to work with as well it has a clearer API, very large store and tons of tutorials to find help which is also great for beginners.

To conclude, for the purposes of the underlying project and for the reasons mentioned above, we will use the Unity Engine!

2.3 Similar Projects

After going through Game Engines and selecting the most appropriate one for the development of our Hydra – rail shooter game, we will now discuss about three mobile 3D games with similar theme.

2.3.1 Star Forces: Space Shooter

The most relevant game, that tries to achieve something similar is the "Star Forces: Space Shooter" which is a 3D Spaceship Action simulator game developed by Crescent Moon Games. In this game, the player controls a ship whose goal is to team up with other players to destroy enemies, complete missions and mine gold and crystals. The player is also able to choose from over 100 weapon types and all kinds of modules and equipment as well as upgrade his spaceship with power ups such as boosting weapon damage per second and reducing recharging time.

Some of its strongest aspects are its light weight (115MB app size), it is very well optimised and delivers outstanding performance and graphics on all kinds of devices, as well as a variety of features such as multiplayer mode, public game chat, settings menu, daily rewards and flexible controls (see Figure 4).

Nevertheless, when we come across the negative reviews on Google Play store, it reveals many connection problems and that AI enemy ship behaviour is not intelligent.

Figure 4. Screenshots of the Star Forces game

2.3.2 Subdivision Infinity: 3D Space Shooter

"Subdivision Infinity" is an immersive Sci-Fi 3D space shooter game that has also been developed by Crescent Moon Games. The main difference from their previous game is that the previous one is a Multiplayer game, while this one has only a Single Player option with over 50 missions in the main storyline. "Subdivision Infinity" is a game of emergence, as players need to find strategies to defeat each stage boss in order to advance on to the next level. The goal of the company is to make their game attractive by including features, such as, extra functionality in the settings menu, unique bosses and side-quests, so that the user can take a break from the main story and engage in some rewarding side activities, e.g. exploration with the aim of finding ship blueprints to equip and upgrade their ships, hunting enemy spaceships in an open world space, mining asteroid for rare minerals and diverse selection of weapons and ships available for purchase and upgrade (see Figure 5).

One of the negative aspects of this game relies on the Installation Size as the game file size reaches 400MB. Additionally, most of the negative feedbacks were concerned about the users' obligation to buy every other mission when they finished the first one.

Figure 5. Screenshots of the Subdivision Infinity game

2.3.3 Space Racing 3D – Star Race

"Space Racing 3D – Star Race" is another relevant game made by 3DGames. As stated in the title, it is not a shooter game, but it is a unique space racing game. The player is not moving into an open world in this game. This means the camera is attached to the player and moves along a restricted and predetermined route with the player (see Figure 6). This feature is the key feature that is most likely to be used in our game.

Figure 6. Screenshots of the Space Racing 3D game

2.3.4 Conclusion

In light of the above, we aim to develop a rail shooting game under a third-person perspective which is required for this genre of game (as shown to all of the similar games). Our game shall follow a fixed route as used in "Space Racing 3D" and not an open world as the game will follow a specific storyline. There would be animations of player and enemy spaceships and only a single player mode with a story would be include. Based on the "Subdivision Infinity", our game will also include a final boss that the player must defeat in order to make progress. In addition, taking into account the first two games explained above, our player will be able to navigate the spacecraft through a virtual joystick and shoot the enemy with a fire button.

The ultimate goal of the player is to achieve the highest score by destroying enemy spaceships. Although, due to limited time and resources, our game would include only one level without supporting online multiplayer mode or any monetizing settings, such as buying coins for upgrade.

3 Requirements

To structure this game effectively at the level of implementation, adequate design requirements had to be identified before development. This section outlines the general game criteria.

3.1 Stretch Goals

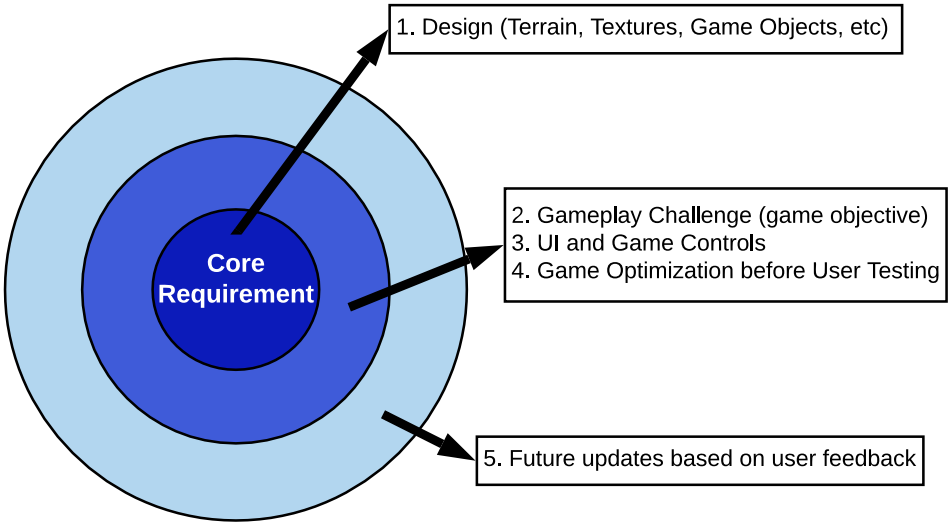


Figure 7. Onion Design for Hydra

For this project, Figure 7 shows an Onion Model to consider a) what is the minimum viable product, b) what is the most important priority that needs to be done to make the game playable, and c) what should be done after the game is released and people are really enjoying the game.

3.2 Design and Playability Requirements

The core and most important requirement of the underlying project is the creation of terrains with appropriate textures, game objects including rocks, bridges, buildings and avatars such as player and enemy spaceships. Gameplay and design are the main components of the formula for a successful game. What makes a game addictive is whether or not it is fun to play. Therefore, the storyline should be interesting, the scenery and graphics attractive, the controls easy to learn, and the objective must be a bit challenging to encourage the user to play again and again.

3.3 Gameplay Challenge

The gameplay should have some progression and not just the same pace the whole time. It should not have the same moments of more than 10 seconds and should have different intensity levels. Otherwise, it will become boring and very bland. We are taking a lot of control away from the player, by putting them on a rail, thus, we have to entertain them in a more cinematic way.

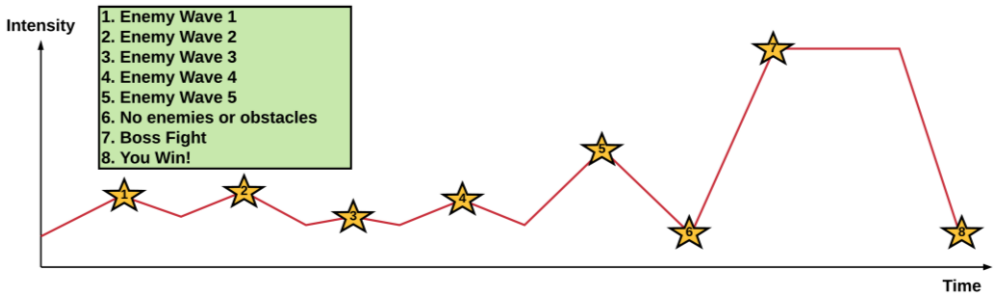


Figure 8. Intensity Level

At the beginning of our level, we start with a low intensity, with the user just flying along, avoiding some obstacles for the first few seconds just to get a feel for the controls. Then, as shown in Figure 8, the intensity increases to the first star in which the player is approaching the first wave of enemies. The same process is repeated until Star 6, which the intensity level decreases significantly to a calming moment that no enemies are coming or any obstacles to be avoided, leaving the player in a suspiciously big empty area. This causes a feeling that something is going to happen which in fact the boss arrives (at Star 7) for the final battle.

Last but not least, if the boss is defeated, the player must proceed to the end of the level which is actually the beginning, a loop that never ends and the player will fight the same enemies again and again. Any time an enemy ship is destroyed, the player will be rewarded with a few points depending on the level of the enemy. Bigger enemies will give more points, while smaller and easier enemies will give less points. The game ends when the player spaceship is destroyed by a crash with any other game object (environmental or enemy).

3.4 User Interface and Controls

The next stage of the onion design is the implementation of the User Interface Canvas, which will include the options section that will enable the player to customise the game settings (e.g. Volume, Resolution and Sensitivity Controls), a pause button that will pause the game, the controls such as a virtual joystick and Fire button, and a score counter that will show the score in real time and increase whenever the player destroy enemy spaceships.

3.5 Optimization Requirements

After the game is considered playable and ready to be deployed, it is important to test the game on different android devices to evaluate performance and compatibility. If the performance is very poor (<30 FPS) then some optimisation would be required in order to achieve smooth performance in combination with good quality. For example, reducing the mesh resolution in order to minimise the Verts and Tris of each frame or to bake lighting instead of using real-time lighting. Another example is the usage of the occlusion culling tool and to ensure that the vertex count does not exceed 100K and 800K per frame.

3.5.1 Compatibility

Nowadays, fewer than 10% of people worldwide are purchasing Android high-end devices. That means that most people use Android devices of the previous generation. As a result, the game should be made available and playable through various Android generations.

3.5.2 Device Displays

There are many different display sizes, resolutions and aspect ratios to consider in mobile game development. The solution must be able to easily view and play the game across a variety of devices by making the game resizable. The default aspect ratio of our game shall be 16:9 Landscape.

3.5.3 Optimization Checklist

Here we present a checklist to be followed in order to optimize the game before building:

- The number of materials shall be kept to a minimum. This makes batching simpler for Unity.
- Texture atlases (large images containing a set of sub-images) are to be used instead of a number of single textures. Atlases are quicker to load, have fewer state switches and are easy to batch.
- Light mapping (Baked GI) shall be used whenever possible instead of Real-Time lights.
- Pixel Light count shall be balanced.
- Numerous lights that illuminate the same object should be prevented.
- The total amount of shaders shall be minimized (shadows, pixel lights, mirrors)
- Post processing shall be avoided as it shows down the game performance on mobile devices.
- Usage of complex shader and particle overdrawing shall be prevented.
- Math functions (con, sin, tan, pow, etc) in shader code costs performance and must be avoided.
- It is recommended that the lowest possible precise number format (float) be used for better results.
- The “Static” property on non-moving object is suggested to allow internal optimizations such as static batching.

3.6 Future Updates

Last but not least, the exterior layer of the onion design it will occur sometime after user testing and based on feedback from the users some features will be added, for example, pick-up items. However, the last layer could be considered as a future upgrade, not a priority.

4 Design

In this Section, we are putting down the architecture of our game. We describe the features and the theory of functionality in the game.

4.1 Game Flow and Screens

We begin our design process by drafting a diagram of the game flow of the game.

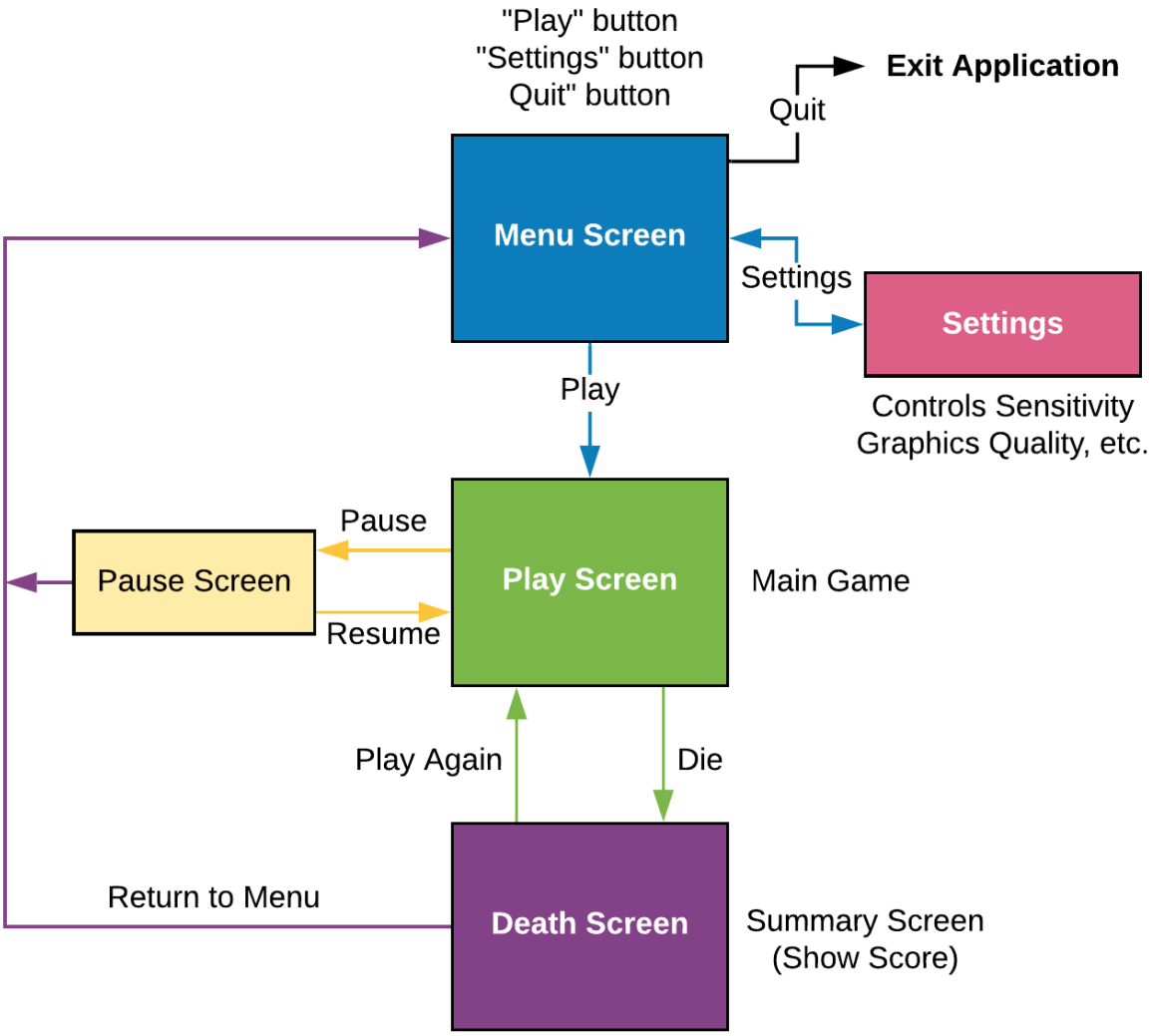


Figure 9. Menu System

Figure 9 demonstrates the entire game flow and menu system of the game. A screen with title graphics must appear immediately after the start of the game and provide a small menu of "Play," "Settings" and "Leave" options. Through clicking the "Settings" button, another menu shall appear with all of the possible options, e.g. Sound Volume, Controls Sensitivity and View Field of the camera. Therefore, pressing the "Quit" button will close the game and eventually pressing the "Play" button will switch the game from Scene 0 to Scene 1, which is the actual Play Screen (Main Game). In addition, on the same Scene, when the Player dies, the Death Screen must appear with the "Play Again" option, which will

reset Scene 1 and start the game from the start. This will reset the score and position of the enemy and players. Last but not least, at any point in the gameplay time, there should be a Pause Screen with Pause and Resume button, which shall make this system have the full functionality of the menu system.

4.2 Terrains

Terrain is the surface or simply the structure and composition of the game world. This can vary from numerous surface types, including sand, asphalt, grass, etc. The terrain in a game engine is basically a height map. In the Terrain Editor, height maps can be controlled by various types of brushes that can lift or lower the terrain in several ways. The terrain in a game engine is basically a height map which in the Terrain Editor can be controlled with various types of brushes, which can lift or lower the terrain in several ways.

4.2.1 Texture

A texture is an image file that can be applied to 3D models to give a textured surface (see Figure 10). In addition, textures may be also used for interface components which are used as normal maps, bump maps, heights maps, custom cursors, icons, splash screens etc [7].

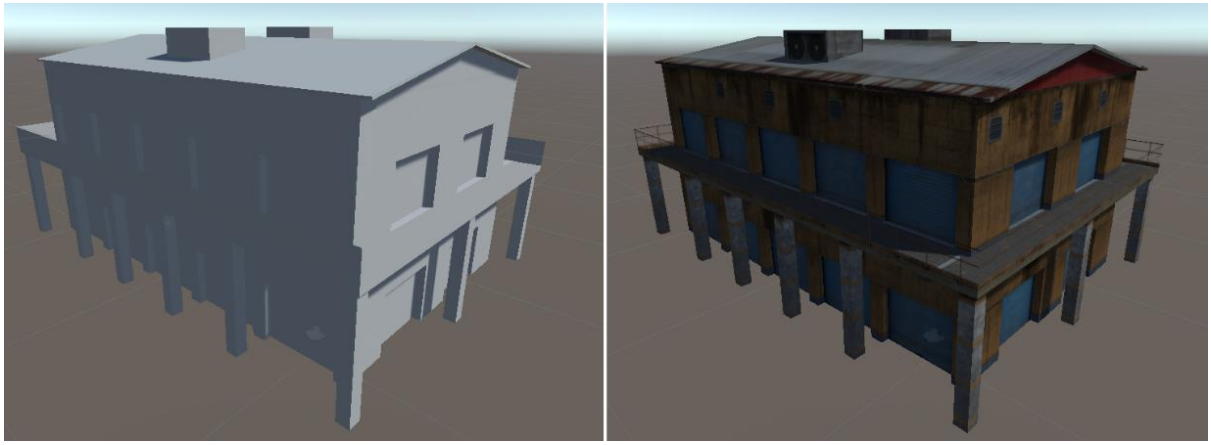


Figure 10. Before and after textures

Compatible textures and components can be found in Unity Assets Store, which is very convenient for accessing all types of assets that can be automatically imported from the store to the project. There are 2 types of textures to be imported when creating a new Texture Layer: Albedo (Standard) and Normal Maps.

An Albedo Map is an image texture without any shadows or highlights [8]. The reason why 3D Artists want to get rid of the default shadows and highlights is that in Unity, the light source can be placed elsewhere and shadows can be cast in the opposite direction, giving them full control over where shadows and highlighting appear (see Figure 11).

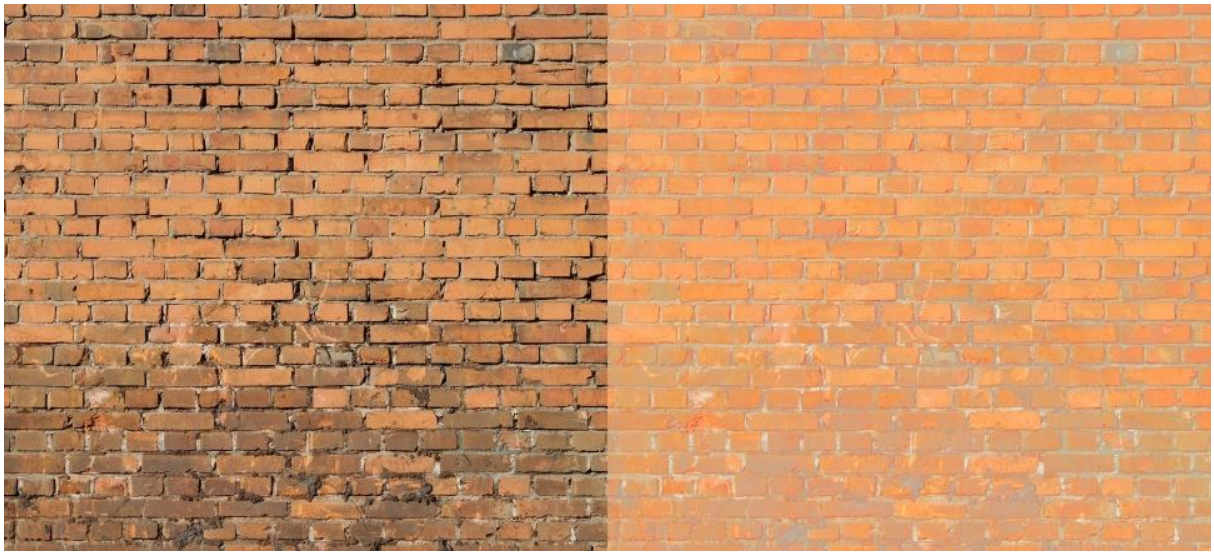


Figure 11. Left: Brick wall with shadows and highlights - Right: Shadows and highlights are missing [8]

The Height Map uses a white to black scale to indicate height. White makes it look higher, black makes it look lower. For example, as shown in Figure 12, it can be exported as a Raw Heightmap, which is basically what Unity reads and transforms into a nice-looking 3D representation using that height map.

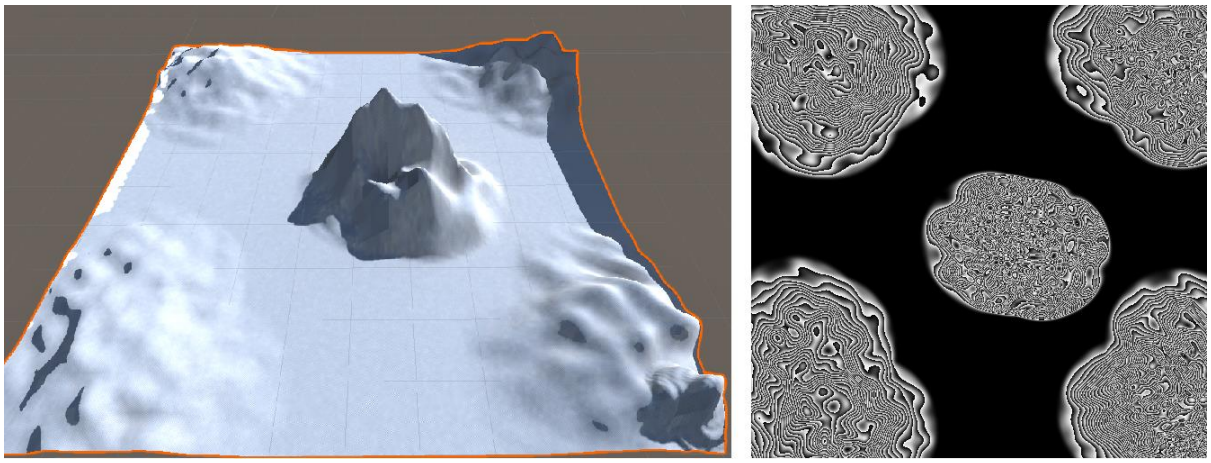


Figure 12. Example of Height Map

On the other hand, the Normal Map, also known as a Bump Map, uses the RGB value to indicate the direction of pixels or points x, y and z. It also refers to a group of textures with details on how the light captures the bump and makes the shape appear to be raised or lowered. As shown in Figure 13, the very bright blue-type colour faces upwards and the red colour faces in the right direction, etc. Therefore, in Unity Engine, a normal map above the texture indicates where the light should be captured and what kind of light should be added to each dimension.

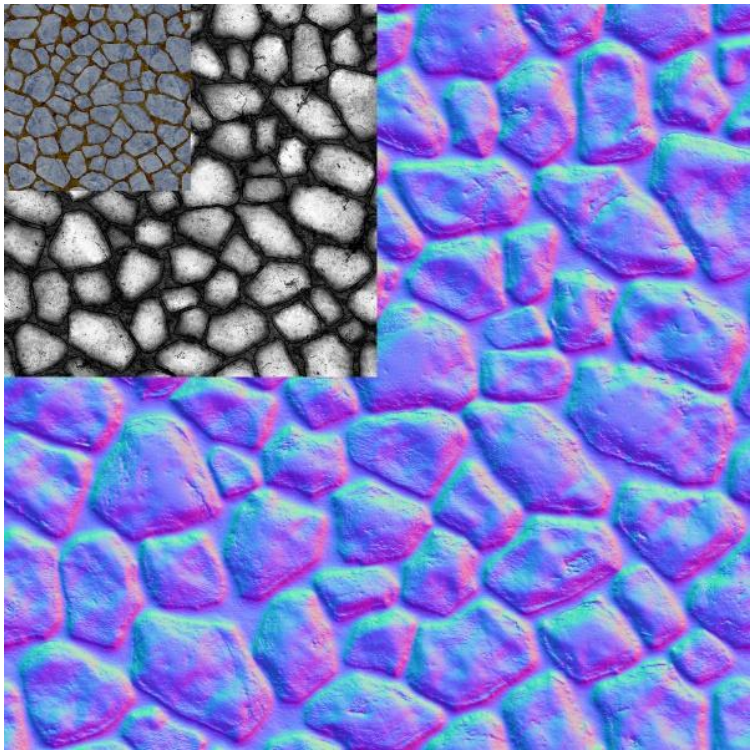


Figure 13. Height map and Normal map example [9]

The Metal and Smoothness values of the Texture Settings are another way to show how the light is going to get to the texture. If the Metallic value is up to 1, the texture will have a very dark and shiny metallic appearance, while the smoothness will make it look very flat if the value is close to 0 and very oily if the value is up to 1.

4.3 Skybox

A skybox is a way to build environments that makes the stage of the game appear larger than it is. When utilizing a skybox, the level is embedded in a cube shape. Using a method called *cube mapping*, the horizon, faraway mountains and other unattainable objects are placed on the cube's faces, generating the impression of a distant 3D surrounding [10]. Rather than using computational 3D graphics processing that pushes the game to render such three-dimensional surroundings in real time by making them as game objects, we can use skyboxes that are basic cubes with 6 separate texture faces (see Figure 14). In the very centre of the skybox, the observer can experience the impression of a true 3D universe made up of the six sides (see Figure 15).

Figure 14. 6 sides of Cube-map Skybox

As the player travels around the 3D world, it is normal for the skybox to stay stationary with the character. This perspective leads to the belief that the objects inside the skybox are extremely far away from the world as they have no parallax movement, through other 3D objects nearest to the player tend to be moving. Last but not least, any type of texture, including pictures or prerendered 3D geometry can be a source of a skybox.

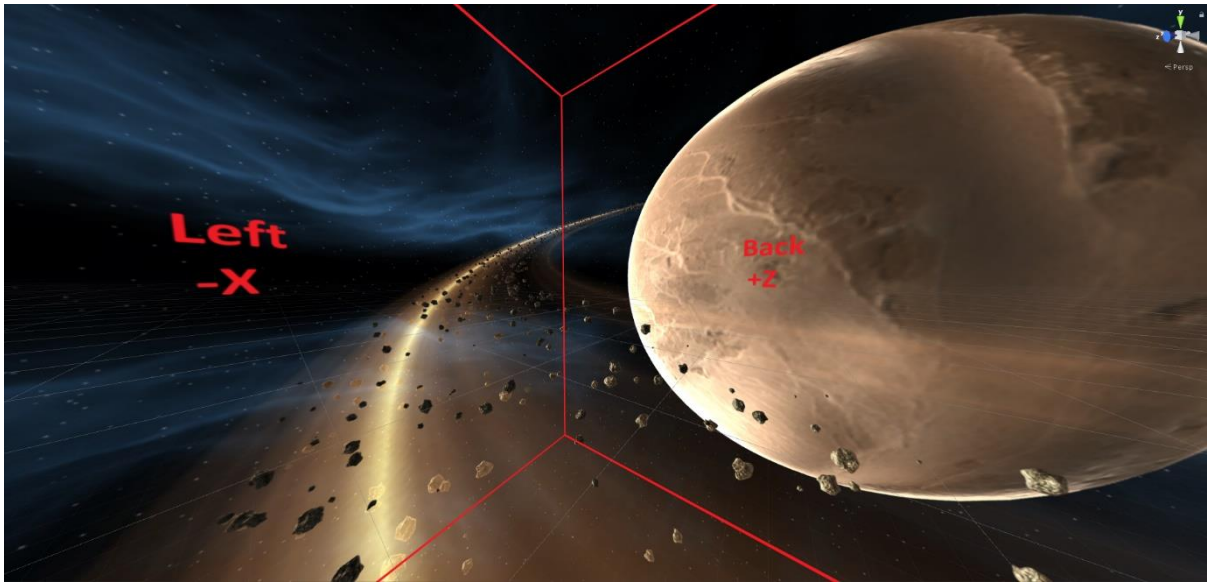


Figure 15. Cube-map in game

4.3.1 Cube-map

A cube-map consists of six square textures representing environmental reflections. The six squares are the sides of an abstract cube surrounding an object (see Figure 16). Every side represents a perspective of the direction of the world axes (forward, back, right, left, up, down). This approach is commonly used in video games as it helps developers to attach complicated and unexplorable worlds to the game as a skybox [3], with almost no performance cost, which is ideal for including a cube map in our mobile game.

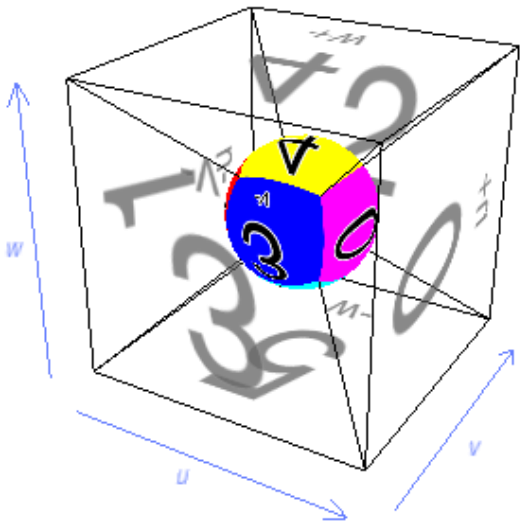


Figure 16. Representation of Cube-Map [11]

4.4 Cross Platform Input Manager

Cross Platform Input is an asset package mostly used for mobile controls which is part of Unity Standard Assets and contains motion scripts, joystick sprites and various types of joysticks [12]. The movement script contains direction X and direction Y variables that are used to pass values depending on the position of the virtual joystick (CrossPlatformInputManager) and then multiply the value by the speed of movement. As a result, when the user moves the joystick up or down left or right, it generates values from -1 to 1 for each axis, depending on its position. The code from the Standard Assets below shows an example of how the value is assigned to each axis:

```
Void Update(){
    dirX = CrossPlatformInputManager.GetAxis("Horizontal") * moveSpeed;
    dirY = CrossPlatformInputManager.GetAxis("Vertical") * moveSpeed;
}
```

Figure 17. CrossPlatformInputManager Code

On the other hand, the **Input.GetAxis** function is very useful for accessing the input of the keyboard and joystick, since the horizontal and vertical axis are mapped to the arrow keys by default. This functionality, however, makes no sense to mobile platforms that rely on touchscreen input [12].

Unity's default controls are the **Input.GetKey** or **Input.GetKeyDown** functions, in which the code is talking directly to the keyboard (see Figure 18). However, these functions create two problems. First, the player may not be able to remap the keys. Normally, in a game, a player should have some flexibility over remapping the keys. Secondly, the code will only work for keyboard keys and not for any other platform controllers, such as a gamepad or a mobile touch screen with a virtual joystick [12].

Fortunately, Unity Engine is a cross platform engine that provides the ability to deploy a game to multiple platforms. Whether a game is made for single or multiple platforms, everyone can benefit from using the **CrossPlatformInputManager** function, as the code talks to the middle layer, as shown in Figure 18, which in turn communicates to any platform input controller. This means that the developer can switch the game to any platform, and they should all work with a few caveats. In respect to the development of our game, we will use the **CrossPlatformInputManager** function which will give us the flexibility to expand the game to more platforms in the future.

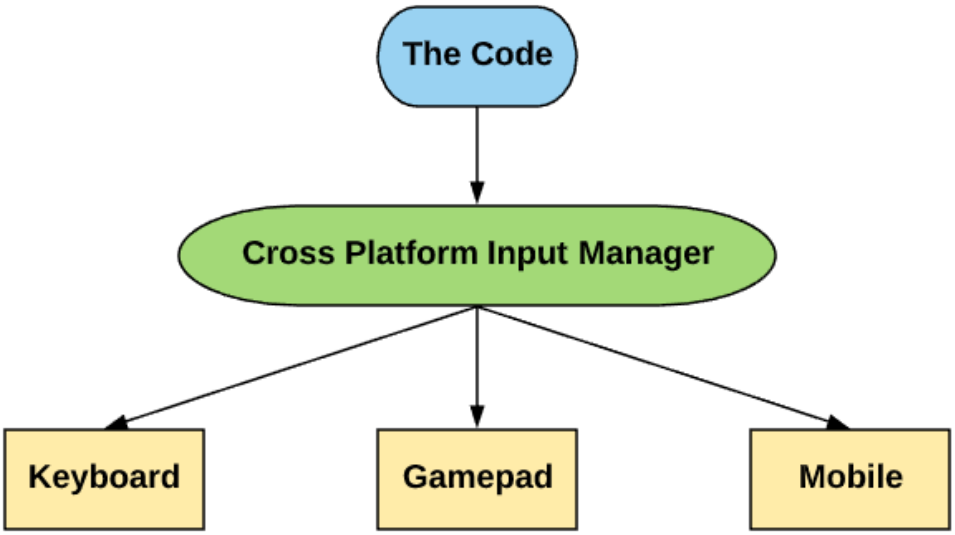


Figure 18. Cross Platform Input Manager

4.5 Pitch, Yaw, Roll

The central topic of this section is the inclusion of the extra rotation of the player spaceship when moving around the screen using the three Euler angles: Pitch (X), Yaw (Y) and Roll (Z). Every object in Unity has a Position XYZ and a Rotation XYZ. Accordingly, the X rotation of the player ship is known as the angle of Pitch because it pushes the spaceship up and down. The angle of Y is defined as the Yaw, since this is the orientation of the head of the spaceship [13]. Finally, the Z angle is the Roll that rotates along the axis of the spaceship itself (see Figure 19).

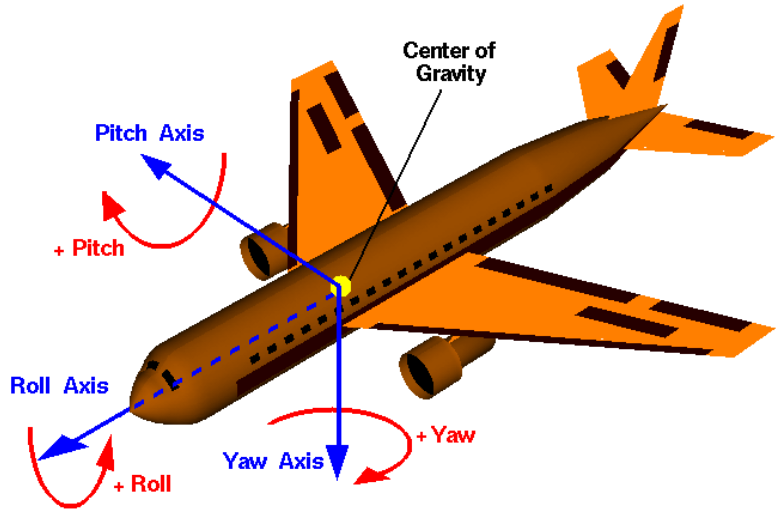


Figure 19. Aircraft Rotations [14]

4.6 Particle System Component

In a predefined environment, a particle system typically emits particles in random locations, and may have a structure like a circle or cone. The system decides the lifespan of the particle itself and kills the particle when it expires. The particle system consists a vast number of separate modules that manage various aspects of the system (see Figure 20). A module needs to be enabled in order to access its settings.

It must be clear that the whole Particle System and each particle are not game objects. The Particle System can be applied to a game object composed of Particles and the Emitter. In addition, modules such as Duration, Shape, Collision, Trails, etc. will be used to control the particles behaviour.

Figure 20. Particle System and Modules

4.7 Triggers & Collisions in Unity

There are two types of collisions: *collisions that process physics* as shown in the Figure 21 with yellow, and *triggers* as shown with green. When a box is added to a game object, the option to enable the Trigger feature appears. If the "Trigger" option is disabled, the physics will be processed and bounce off other objects, otherwise, if it is enabled, the physics will not be processed. In order to have some kind of collision, we need to add a ***Rigidbody*** to the player ship. ***Rigidbody*** is a property that allows the object to interact with a lot of basic physical behaviour, such as forces and acceleration [15].

	Static Collider	Rigidbody Collider	Kinematic Rigidbody Collider	Static Trigger Collider	Rigidbody Trigger Collider	Kinematic Rigidbody Trigger Collider
Static Collider		Collision			Trigger	Trigger
Rigidbody Collider	Collision	Collision	Collision	Trigger	Trigger	Trigger
Kinematic Rigidbody Collider		Collision		Trigger	Trigger	Trigger
Static Trigger Collider		Trigger	Trigger		Trigger	Trigger
Rigidbody Trigger Collider	Trigger	Trigger	Trigger	Trigger	Trigger	Trigger
Kinematic Rigidbody Trigger Collider	Trigger	Trigger	Trigger	Trigger	Trigger	Trigger

Figure 21. Collision Action Matrix [15]

When there is a collision between two objects, there will also be a variety of script occurrences, depending on the settings of rigid objects. Many configurations allow a collision to impact only one of the two objects, but the general law is that physics cannot extend to an object without a Rigidbody component applied to it.

If either of them does not have trigger, we will either have the following:

- Static Collider, a collider with no fixed body on it.
- A Rigidbody Collider that can absorb forces and torque in order to physically move the object.
- Kinematic Rigidbody Collider which allows us to move an object within a script. However, it does not respond like a non-kinematic rigid body to collisions and forces.

4.8 Timeline

Timeline is a tool of Unity that is used to produce video materials, gameplay animations, audio sequences and dynamic particle effects by visually organizing tracks and clips connected to scene Game Objects [16]. The Timeline Editor demonstrates how Timeline Assets and Timeline Instances are created, simple animations are captured, humanoid animated and other timeline features are employed (see Figure 22).

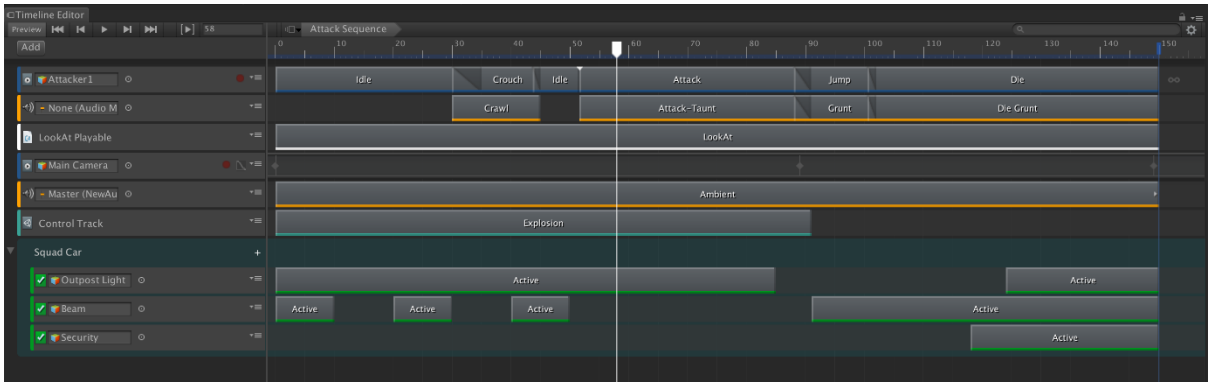


Figure 22. Timeline Editor [17]

The way the movement or animation will be made to all enemies of our game is through the recording system in the Timeline tool. First, the enemy shall be positioned at the beginning of the "A" position and then the key frame with all the Transform Information (Position XYZ, Rotation XYZ) of the enemy object will be added at the beginning of the Timeline. After that, we shall move the enemy to the desired "B" position, which means, for example, that this is the final position that we want our enemy to be after 5 seconds.

4.9 Lighting and Rendering

In any game, it is important to create the right atmosphere and mood. Light plays a major part in how players communicate with a game environment. Lighting in games is usually created by various built-in game engine tools, one of which has the ability to bake lights into materials to save rendering time. This gives the impression that light is being reflected, but it is really just the light that has been blended into the material [18].

4.9.1 Baked GI Lighting

When a lightmap is "baked," light effects on the static objects are measured and the results are written to overlaid textures over the scene structure to produce an illuminating effect, as illustrated in Figure 23.

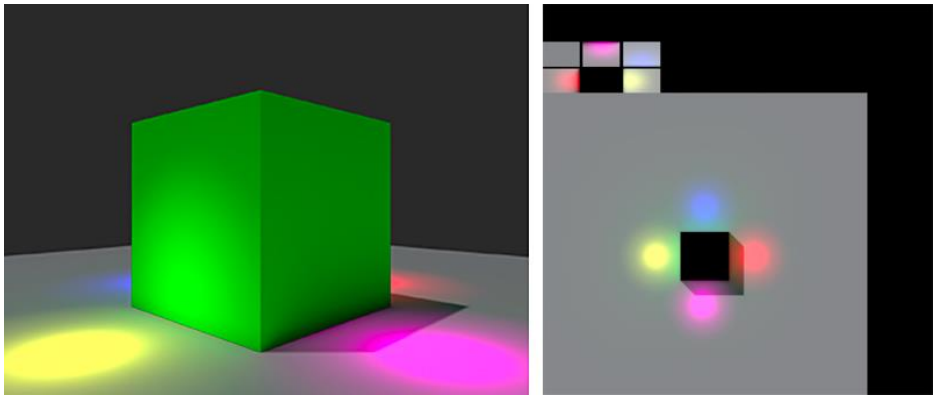


Figure 23. Left: A lightmapped scene. Right: A lightmap texture of the scene [19]

These lightmaps may contain both direct and indirect light, which scatters from other objects or surfaces in the area. This texture of illumination can be used along with details on the surface such as the Albedo and Normal shader maps, linked to the material of an object. Baked lightmaps (light textures) cannot be changed during gameplay and are considered “static”. On the other hand, there is Realtime Lighting which can be overlaid and used on an illumination-mapped scene additively, but we cannot alter the lightmaps interactively [18].

4.9.2 Real Time or Baked Lighting?

By definition, Real Time Global Illumination is running in real time, and it takes several precious CPU cycles to run. The issue of sacrificing CPU cycles to a real time GI may be dependent on whether the game appears to be constrained by a CPU or a GPU. On the other hand, Baked GI does not cost CPU cycles, but all baked lightmaps can be added fast. For example, if the bake GI produced 400 MB worth of lightmaps, for PC or Console should not be a big deal, but for mobile platform, 400 MB lightmap textures is a lot. Although, this can be fixed by reducing the resolution of the lightmap. The biggest drawback for baking the global illumination is that it could take from 2 hours to even days [20].

Like so many aspects of game development, optimizing Scene lighting is about finding the right balance between the desired visual result and the cost of performance. In many cases, it is worth sacrificing a small amount of lighting fidelity in exchange for reducing pre-compute times and improving run time performance. Further to the above, the use of Baked GI Lighting is the answer to our game. With this solution we will trade the opportunity to render lights in real time for potential efficiency improvements appropriate for less powerful devices e.g. mobile platforms.

4.9.3 Types of Light

There are three types of Light that we will use in our game: Point Lights, Spotlights and Directional Lights.

- **Point Lights:** The point light is positioned at a point in space and emits light evenly in all directions. With the distance from light, the intensity decreases to zero in the defined region. The strength of light is inversely proportional to the distance square of the source. This is called 'inverse square law' which describes how light in the natural world behaves [21]. Point lights are useful to mimic lamps and other nearby light sources in a scene (see Figure 24).

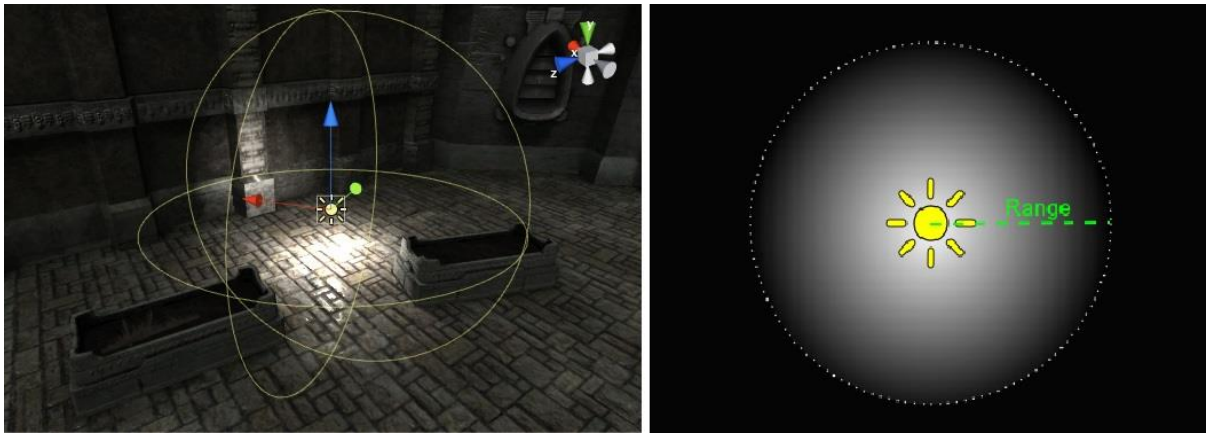


Figure 24. Effect of Point Light [21]

- **Spotlights:** A spotlight is similar to the Point Light; it has a certain location and range over which the light drops. However, this type of light is confined to an angle that contributes to a cone-shaped lighting area. Expanding the angle increases the diameter of the cone [21]. In general, spotlights are commonly used for artificial light like torches (see Figure 25).

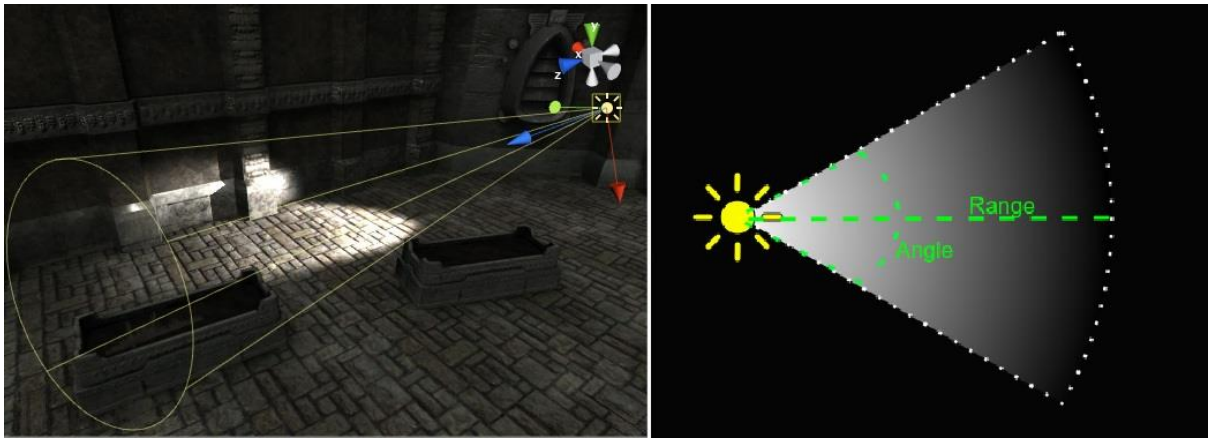


Figure 25. Effect of Spotlight [21]

- **Directional Lights:** Directional lighting is incredibly helpful to produce things like sunlight in a scene. Looking in many respects like the sun, directional lights can be perceived far apart as distant light sources. There is no clear source point for directional light, as a result, this type of light can be positioned anywhere in the environment. Every object in the scene is illuminated because the light always comes from the same direction. In addition, the light distance from the target object is not defined and therefore the light is not reduced [21]. Directional lighting represents a large light source from a location outside the game world that can be used to represent the sun (see Figure 26).

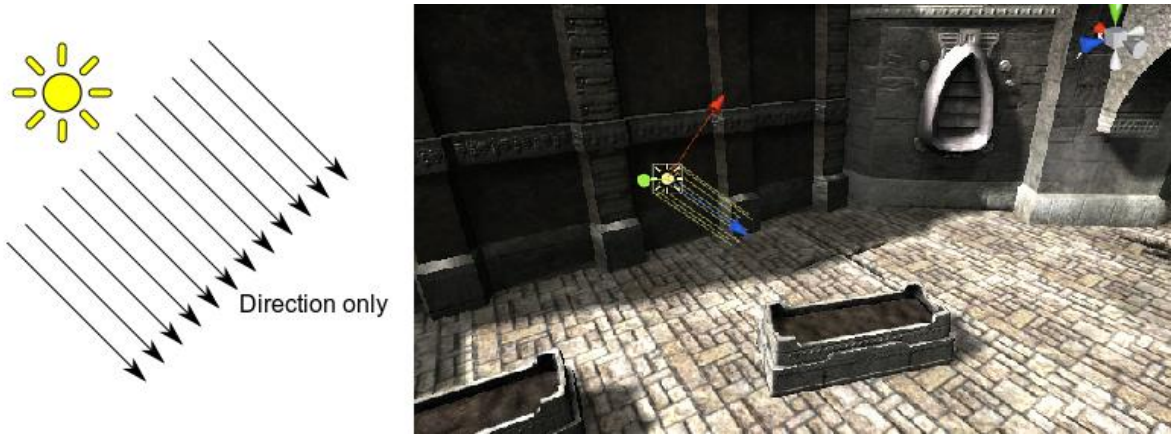


Figure 26. Effect of Directional Light [21]

4.10 Switching Platforms

Switching platforms sets the build target of the current project to any platform (see Figure 27). This game is going to be built for Android, which means that Unity is going to build an .apk game file. Switching platform also forces Unity to re-import all project assets that could take a very long time if the project is large. As with PCs, mobile devices can vary in level of hardware specifications and performance, for example, a mobile device can be multiple times more efficient for gaming than other devices. This leads to the conclusion that the final product should be tested on both low and high-end devices in order to monitor the performance of the game.

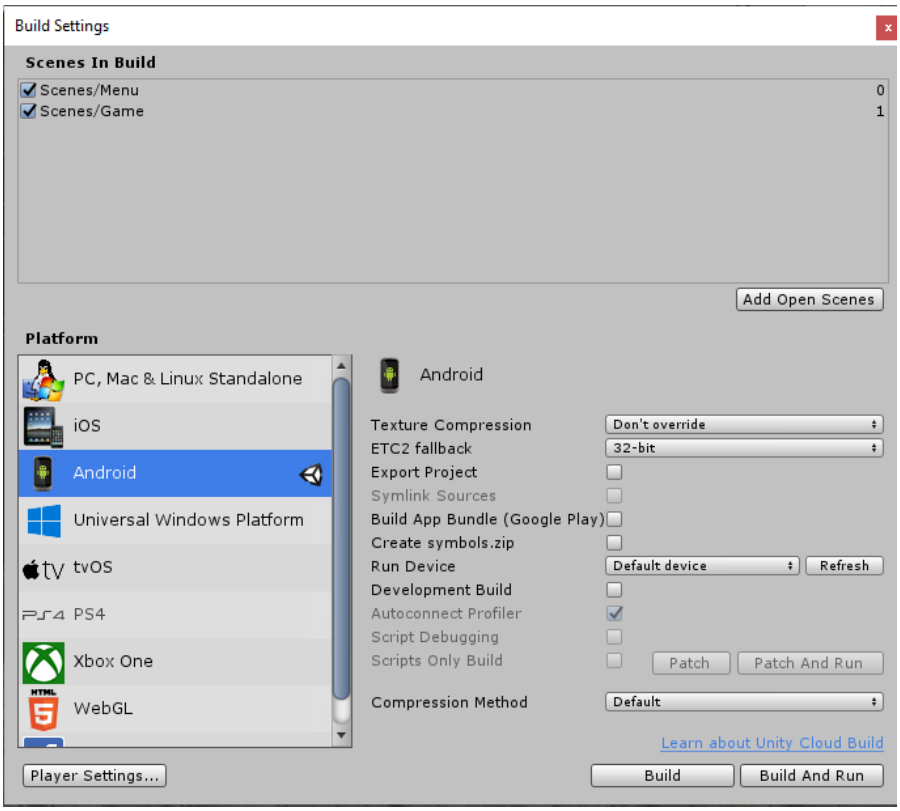


Figure 27. Build Settings / Switching Platform

4.11 Focus on GPUs

To start with, depth intensity, pixel and geometric complexity (vertex count) is the foundation for graphical efficiency. These three can be minimized by removing more renderers. If more renderers can be removed, these three can be minimized. Occlusion Culling Tool will assist us to this situation as it decapitates objects outside the viewing angle of the camera as defined in Sections 4.12 and 5.18.

Reducing texture resolution in the Quality Settings section will help the game to run faster even on low-spec devices, as it will require less graphical processing power to load the textures of the game. This means that compressing textures, using Mipmaps, lowering texture resolution and reducing LOD (Level of Detail) it could be a wise decision. LOD clarifies or removes objects as they move away from the camera.

In addition, the disadvantage on mobile GPUs is primarily heat, power consumption and noise. Thus, mobile GPUs have considerably less bandwidth, poor ALU efficiency and texturizing capacity compared to the computer components.

4.12 Occlusion Culling

Occlusion culling is a mechanism that prohibits Unity from executing GameObject calculations that are entirely covered from sight (occluded) by certain GameObjects. Every frame is used to view the renderers in the scene and delete those which do not need to be drawn. In addition, cameras conduct a frustrum culling procedure to exclude objects from being rendered that are entirely outside the viewing angle of the camera. However, the culling process does not check that the renderer is covered by other GameObjects, which is why Unity will spend the rendering time of the CPU and GPU on renderers that are not visible in the last frame. As a result, Occlusion Culling prohibits Unity from carrying out such inefficient activities [22].

Occlusion culling creates scene data in the Unity Editor and then, at runtime, uses the data to decide what the camera will see. The data collection process is known as baking (see Figure 28), which when performed, Unity splits the scene into cells and produces data that defines the structure of the cells and the distance between the neighbouring cells. Unity then combines cells to reduce the size of the data produced, where appropriate (see Figure 29).

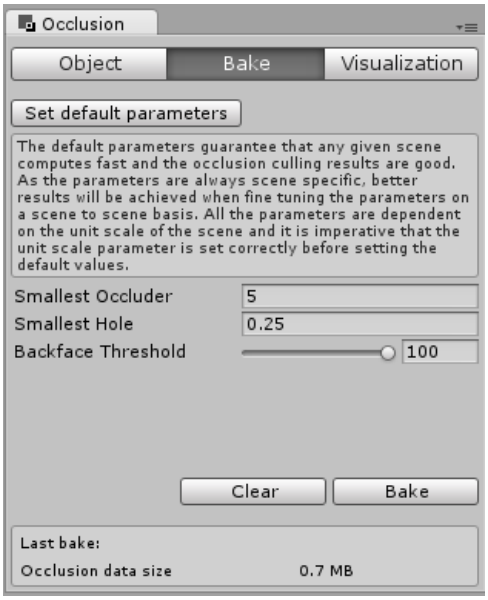


Figure 28. Occlusion Culling inspector bake tab

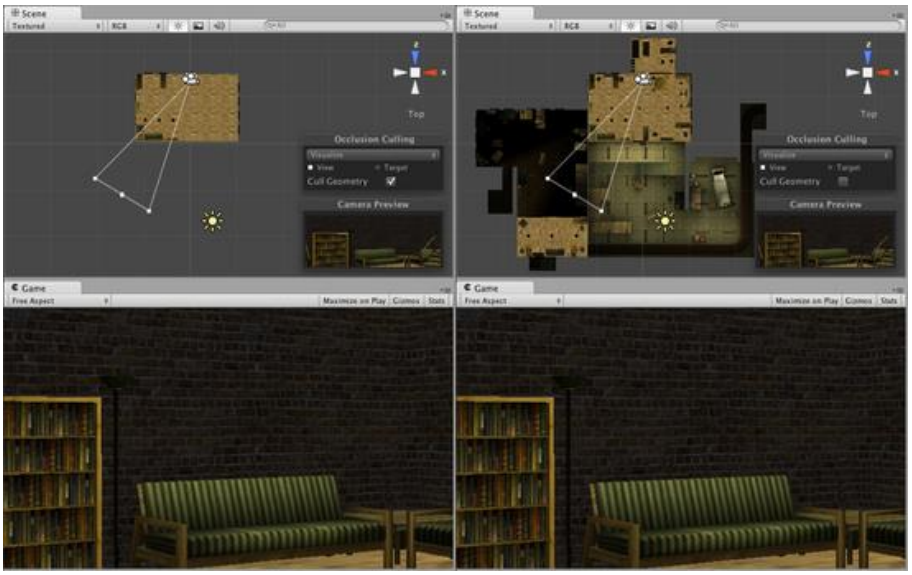


Figure 29. Left – Occlusion Culling: On, Right – Occlusion Culling: Off [22]

4.13 First-Person and Third-Person Perspective (FPP-TPP)

4.13.1 First Person Perspective

The *First-Person* in video games is any graphical perspective rendered from the player's character point of view, or from the front seat of a character-driven vehicle. In addition, it is usually based on the avatar, where the game shows what the player would see with the eyes of the avatar. Thus, typically the player cannot see the body of the avatar (see Figure 30 and Figure 31). Several genres, including adventure games, action games, driving and flying simulators provide *First-Person Perspective.*

Figure 30. Third-Person VS First-Person Perspective (Screenshot taken from Overwatch)

4.13.2 Third-Person Perspective

On the other hand, the *Third-Person* refers to a visual perspective from a fixed distance behind and slightly above the character of the player (see Figure 30 and Figure 31). This perspective allows players to see their entire avatar and is most popular in action and shooting games. Our game will be developed under a *Third-Person Perspective* structure, as the main element of this game genre is the movement of the ship in a restricted space (4 corners of the screen) on a predetermined route. This gives the player the advantage of better visualization of the ship's surroundings.

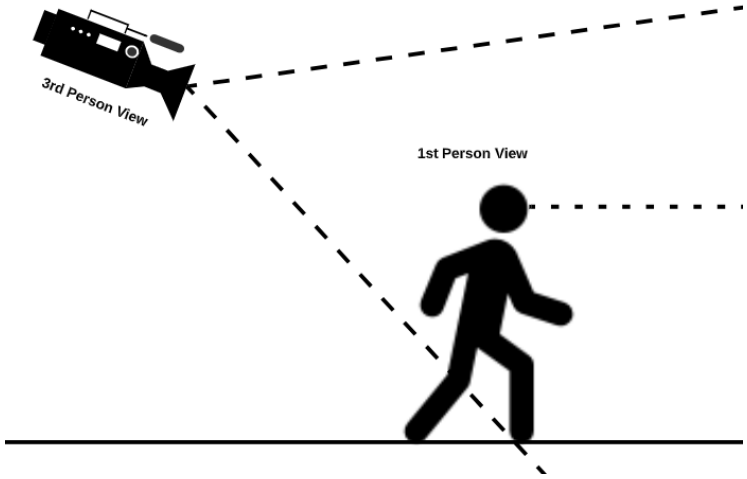


Figure 31. Third-Person View VS First Person View

4.14 Statistics – Profiler

Unity Engine includes a tool called “Statistics”, which shows real-time statistics and is extremely useful for optimizing game efficiency. The statistics window includes a lot of details and is divided in two sections, Audio and Graphics.

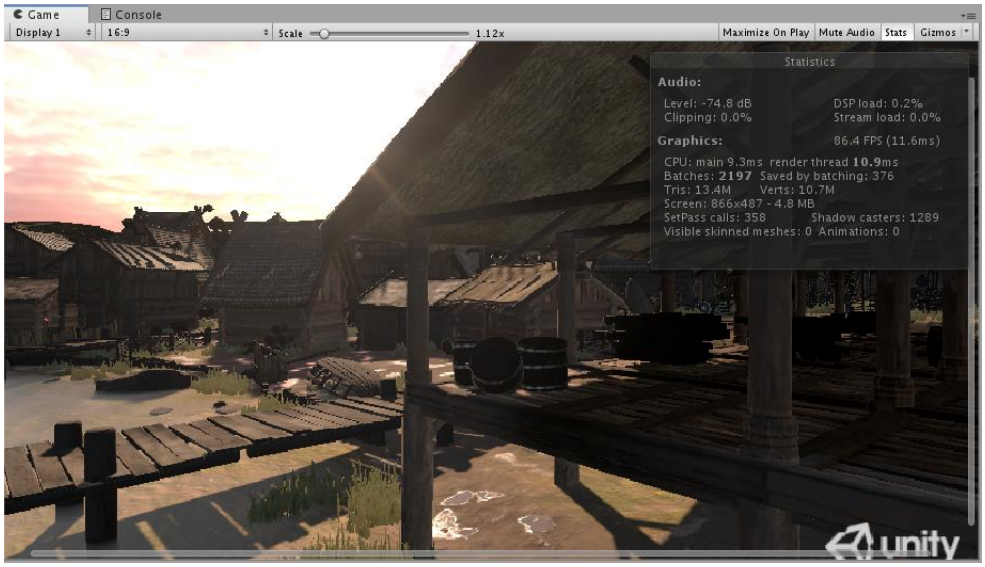


Figure 32. Rendering Statistics Window [23]

The Audio section includes four essential information parts about the current scene frame. The first one is *Level* indicating the current sound decibel. If the audio is muted, next to the decibel level will be displayed a "MUTED" text. Then we have *Clipping* that shows the percentage of audio distortion within the frame. *Stream Load* reflects the whole audio source load while the *DSP Load* calculates the DPS buffer's maximum capacity.

Moving on to the Graphics section, it also contains details about the given frame. To begin with, *FPS* provides the time it takes for one frame to be processed, while *CPU* provides the time it spends on the CPU and its processing thread. *Batches* are data packets which will be send to the GPU. The fewer packages there are, the higher the efficiency. In addition, *Saved by Batching* shows how many batches were paired. Generally, "Batching" is where the processor tries to merge the representation of several items into a piece of memory to minimize latency of the CPU due to resource transitions.

Furthermore, *Tris* and *Verts* indicate the number of triangles and the number of vertices drawn which are important numbers for low-end hardware optimisation. The *Screen* aspect indicated the screen size and memory usage, while the *SetPass calls* represent the number of rendering passes.

Each pass requires Unity runtime to add a new CPU overhead shader. Next aspect, the *Shadow Casters* represent the number of shadow-casting objects within the frame. Finally, the *Visible Skinned Meshes* and *Animations* display the number of skinned meshes in the frame and the number of animations performed.

The Statistics overlay offers feedback and project information in real time. It provides a brief look at what is going on in the current frame and allow us to recognise areas within the project which may trigger efficiency issues, and which can be analysed further within the Profiler window.

Likewise, when the Rendering Profiler tool is used, statistics about our scene are obtained every time it is performed. A seen in Figure 33, the profiler window itself is divided into 3 parts. The red area reflects the profiler segment in which the different measurements can be tracked. Next, in the green area, we have an outline of where each profiler region can be described. Finally, we have the pre-object display in the blue area in which we can get information on the objects in our description. By default, the profiler measures CPU Usage, GPU Usage, Rendering Performance, Memory Usage, Audio and Physics for both 3D and 2D.

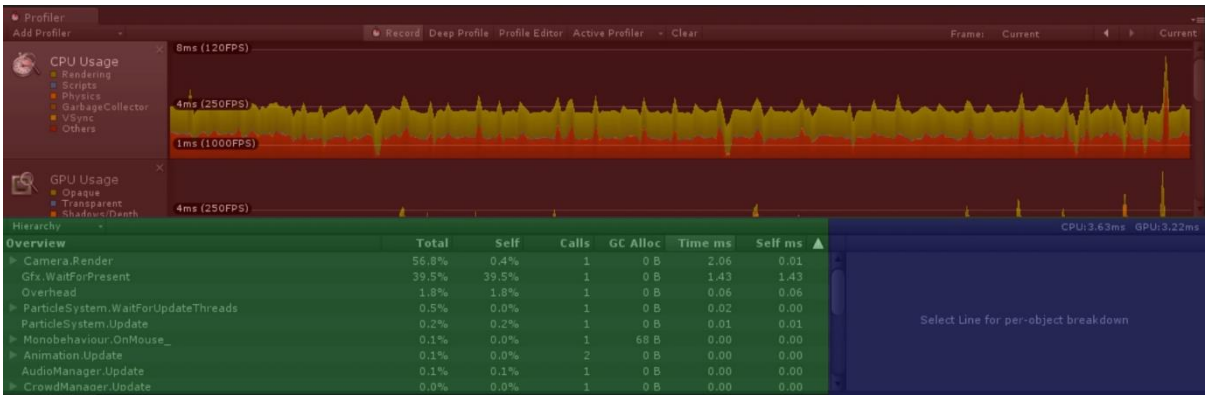


Figure 33. Rendering Profiler [6]

Both Statistics and Profiler tools will help us both optimise our game and achieve the maximum performance. For example, if the colour graph demonstrates high values, this means that our game performs multiple tasks at the same time, making it slower. If high values exist only on specific frames (see Figure 33) then, the developer applies the optimization process on that specific part of the game instead of the whole, which saves valuable time without changing current quality and performance. At the end of the day, we want to keep the values of the graph as low as possible.

5 Implementation

This section explores the game that was built in the previous section. It provides a detailed description of core mechanics and how each stage of the process is implemented.

5.1 Script Lifecycle

In general, when we have a script in Unity, it is going to be a Monobehaviour. A Monobehaviour is a basic class, since basically all the other scripts are going to be written inside Unity to inherit. In addition, the Monobehaviour has a lot of different methods that Unity knows how to interact with and call them as they cycle from frame to frame.

When all the scripts are running inside of Unity, all the methods are executed from top to bottom and get called at different times. As shown in Figure 34, the Initialization of the GameObjects in the Editor always takes place first. The Unity's build-in Physics Engine will then supply the components that handles the physical simulation. Next, the Game Logic works almost exactly the same way as the Physics Engine, except that it waits for the User Input in order to start any process first. The next in line is the Scene Rendering, which is the process of adding physical attributes and details, such as colour, shading, texture, and lamination to a grid frame, which in turn creates realistic images on the screen. Last but not least, the Pausing and Decommissioning sections only operate when they are called [24]. All of these methods are part of the Script Lifecycle, but only the following will be used for the scripts of this game, the rest will remain as default:

First Scene Load	
Awake	This function is always called before any Start functions and also just after a prefab is instantiated.
OnEnable	This function is called when the object is enabled. This occurs if an instance of MonoBehavior is established, such as loading a level or instantiating a GameObject with a script component.
Editor	
Reset	Reset is called to initialize the script properties when the object is attached and when the reset command is used.
Before the first frame update	
OnApplicationPause	It is called at the end of the frame where the pause is observed between regular frame updates. After OnApplicationPause is called, an extra frame will be provided to allow the game to display graphics indicating pause status.

Update	This function is called once per frame.
When the Object is destroyed	
OnDestroy	This function is called after the last frame of the object's life.
When Quitting	
OnApplicationQuit	All game objects have this function called before the application is stopped.
OnDisable	This function is called when the behaviour becomes disabled or inactive.

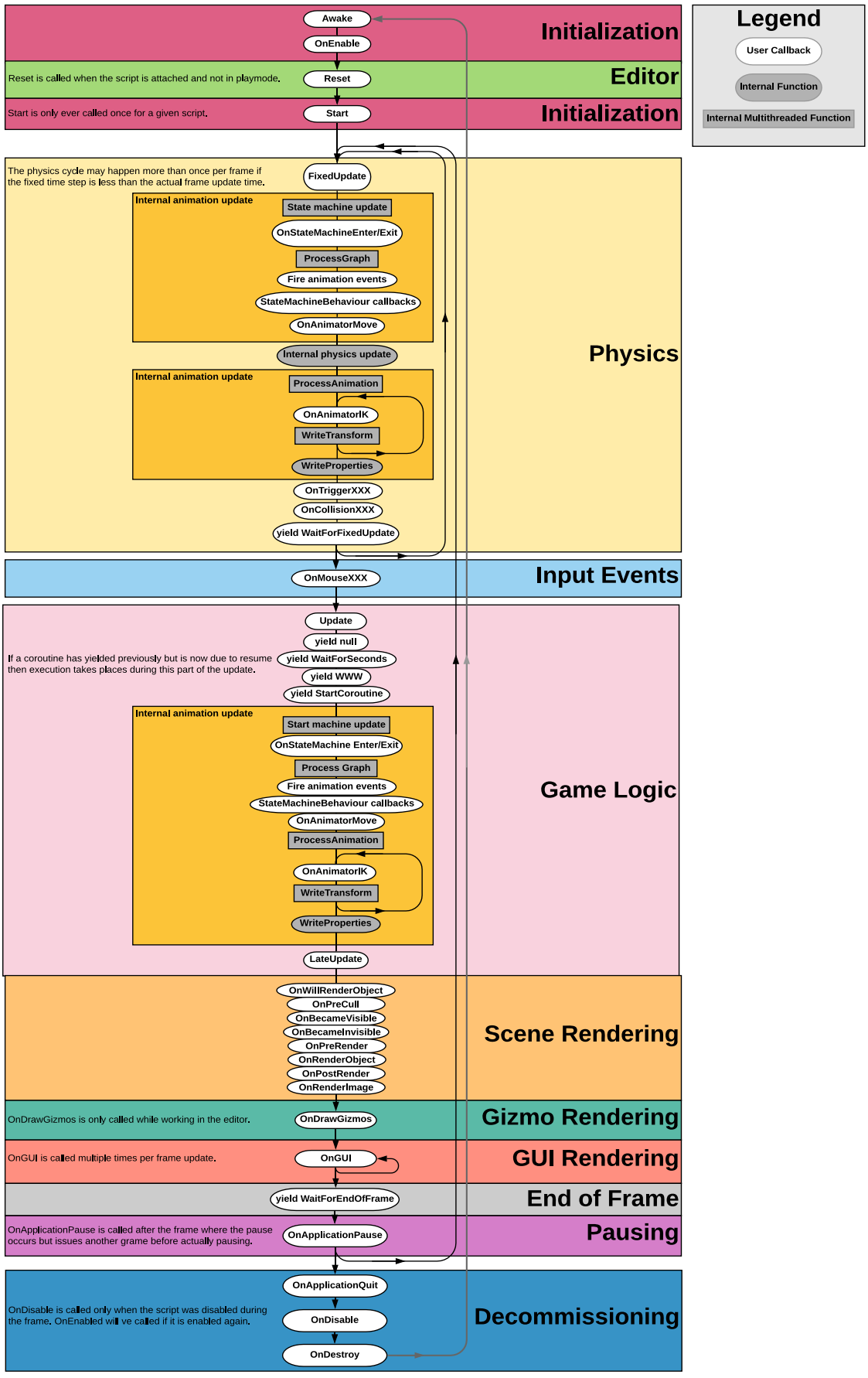


Figure 34. Script Lifecycle Flowchart [24]

5.2 Menus

As shown in Figure 35, Scene 0 is the main menu of the game that loads and directs the player to the next scene, the Game Scene (Scene 1). This happens when the user presses the "Play" button, which is implemented with a scene management script.

Figure 35. Scene 0 and Scene 1 Menus

5.3 Terrains

5.3.1 Shaping the terrain

Here is the starting point for creating a new flat terrain using the Unity Terrain Editor. The resolution on the ground is a large area of 2000 width by 2000 length that might cause some performance problems and we will discuss how to tweak some settings to improve performance in a later section.

The Terrain Editor can also be used to paint terrain textures, set the overall height of the terrain, line the rough edges, paint trees and details, e.g. grass and flowers. For the underlying project, the current height is 0, so there is no way, for instance, to build a valley below it. In this case, the Terrain Height Tool is used to lift the whole terrain by 100. Now, with the Lower Terrain tool, it is possible to build a valley that goes down with a maximum depth of 100 as it was all moved up in terms of rising.

Unity comes with a build-in Brushes collection. They range from simple circles for fast design to random scatter shapes that are good for creating detailed terrains [25]. The following figure shows what was used to create the terrain of the game.

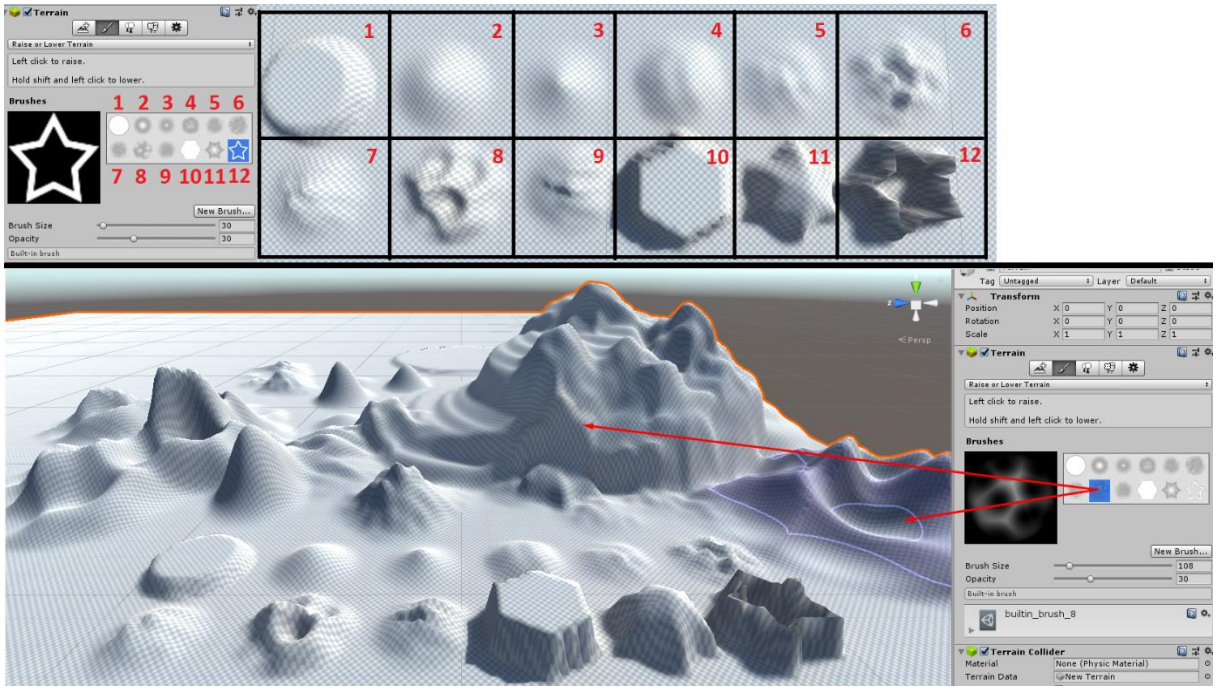


Figure 36. Demonstration of how the game terrain has been created (Top). Even a single brush can be used to create the whole terrain (Bottom).

5.3.2 Texture the Terrain

Once the terrain is shaped with some mountains and valleys, the texture can be applied to it. As shown in the figure below, on the right is the actual 3D representation of the textured terrain that will be used as our game world. On the other hand, as mentioned in Section 4.2.1, the other two images (middle and left) represent the Raw Height Map produced at a depth of 16 bits, which is what Unit actually reads and decodes to a 3D map.

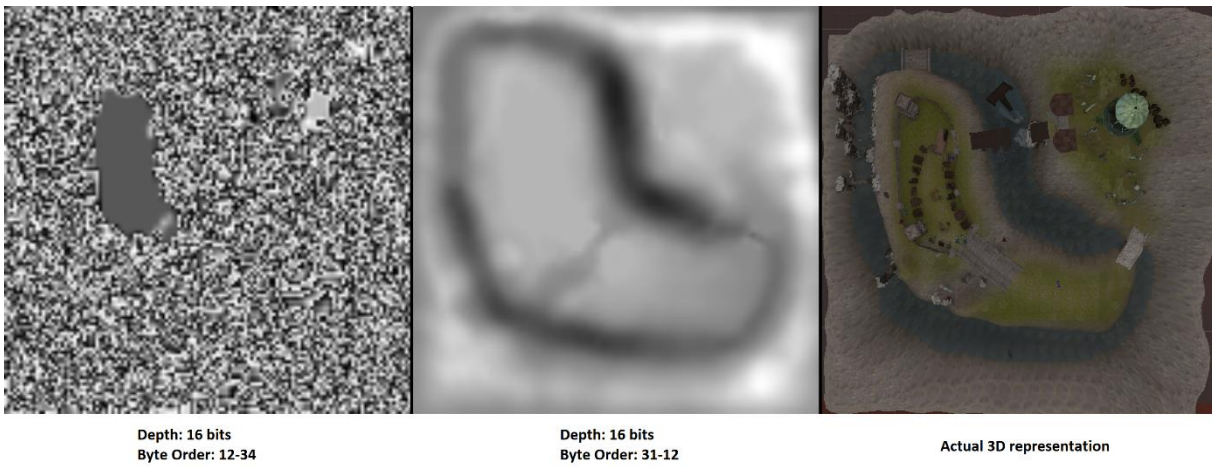


Figure 37. Side by side Raw Height Map and 3D Representation

5.4 Music between Scenes

Music and general objects that reside in a scene no longer exist when the next scene is loaded, or the scene is reset. This is usually a good effect if there's a big scene with a lot of objects and stuff in it, and we want to load a new scene. It is not efficient to drag all that stuff into the next scene, so by default, they all destroyed, which is the whole kind of definition of loading a scene or level in a game with a fresh start.

The figure below shows the concept of how a music player on the build index 0 will be inserted into the game loop of the scenes while the music player should continue to exist when there is a switch between the scenes.

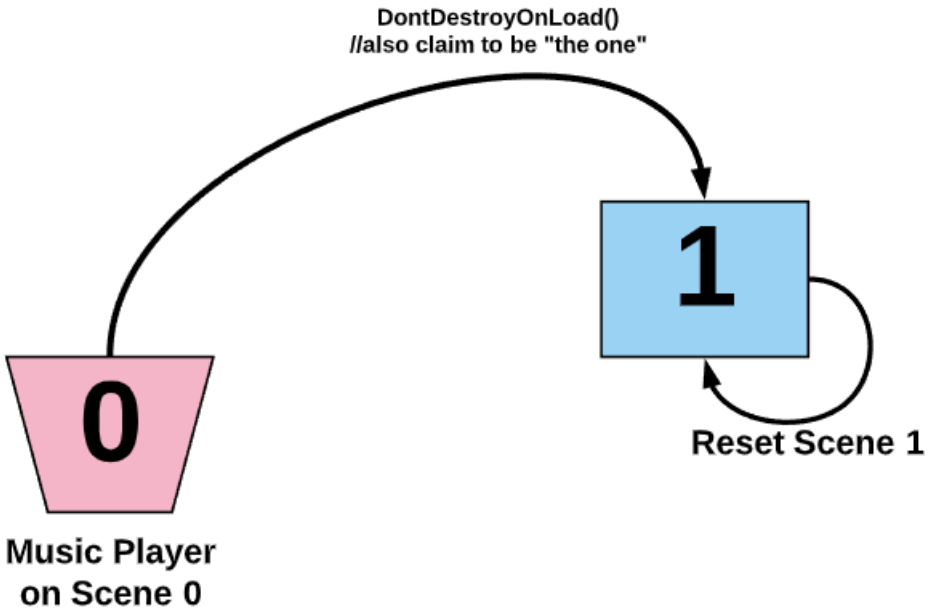


Figure 38. Injection Patent

Recalling the Unity Script Lifecycle Flowchart (Figure 34), there is something called *Awake* in the Initialization Segment, a message called *OnEnable* and a *Start* that can be called once for a specific script. According to the Script Lifecycle, *Awake* only happens a little before *Start*, then the game goes to the Game Logic segment that repeatedly calls *Update*. Accordingly, the code *"DontDestroyOnLoad (gameObject)"* is added to the script's *Awake* function to prevent the attached Music Player object from being destroyed during the scene transitions. Although the capital "G" *GameObject* is a type just like an integer, the small "g" *Game Object* is an instance or variable of a game type object and applies to the attached object.

5.4.1 Singletons

Singleton does not like being around other things of its kind. For example, if a Music Player finds another Music Player in the scene, the audio with singleton will destroy itself. The Music Player is currently on Scene 0 and we do a *DontDestroyOnLoad* but we also want to have only one Music Player on the game scene and destroy any other copies that have been created. When the player dies and Scene 1 resets, Unity creates another Music Player every time the Scene resets. Thus, the *FindObjectOfType* function is used to get an array of all the music players that are in the scene and *Destroy(gameObject)* if the Music Player is more than 1 (see Figure 39).

```
1 reference
public class MusicPlayer : MonoBehaviour
{
        0 references
        private void Awake()
    {
        int numMusicPlayer = FindObjectsOfType<MusicPlayer>().Length;
        Screen.sleepTimeout = (int)SleepTimeout.NeverSleep;

        if (numMusicPlayer > 1)
        {
            Destroy(gameObject);
        }
        else
        {
            DontDestroyOnLoad(gameObject);
        }
    }
}
```

Figure 39. Destroy duplicated game Objects

5.5 Input Sensitivity & Gravity

The Figure 40 shows the horizontal axis with Time in seconds and the vertical axis with Throw while pressing the key on the keyboard, e.g. the "D" key, to go right for a period of time. The result is that the Throw reaches a maximum value of + 1 over time in a straight line. This means that the speed of the throw increases with the button is determined by the sensitivity (this is not applied to the gamepad or the virtual joystick). Once the throw hits 1, it ends capped at 1 and drops at 1 when the key is released, depending on the gravity setting.

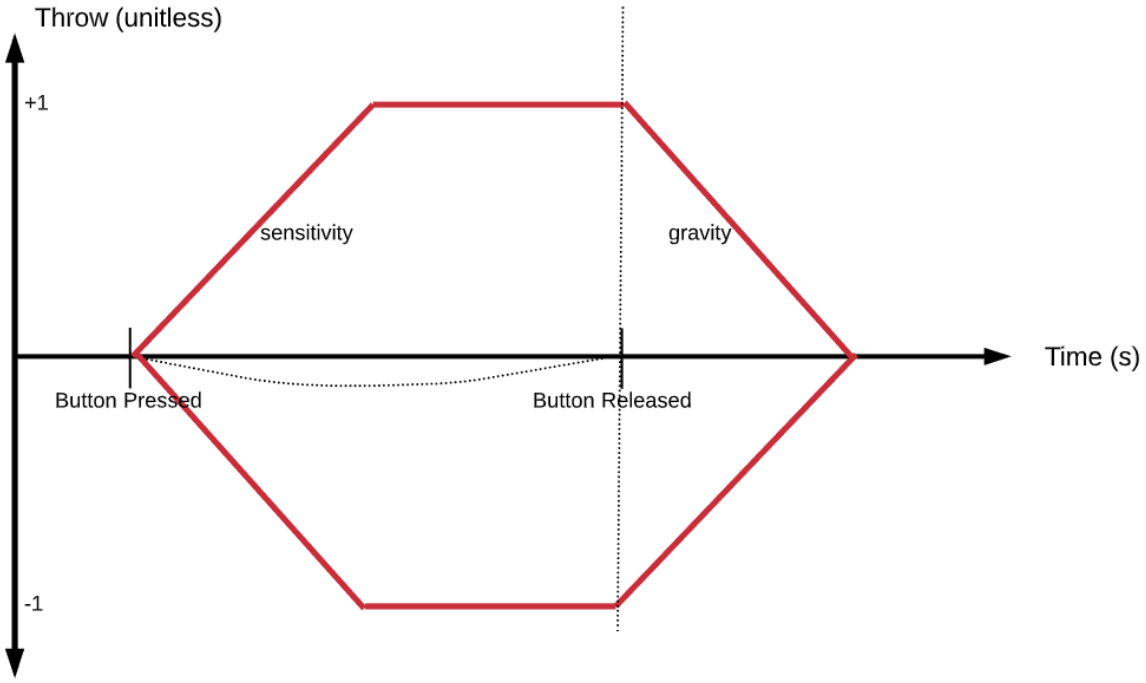


Figure 40. Sensitivity and Gravity when button is pressed and released

The next phase is to match the speed by multiplying the pitch by a certain speed and then correcting the frame rate (*Time.deltaTime*) as shown in the Figure 41. Then, if the key is still pressed (Right or Left), the speed of the ship (player) moving left and right on the screen will increase to a certain maximum, and then the speed will remain constant (see Figure 42). This means that the object is still whipping across the screen, but the speed it moves across the screen will actually end up capped when the throw is capped. It is therefore a simple multiplier that applies that value to the movement of the player ship combined with the *Mathf.Clamp* function [26], which restricts the output to a range of inputs to prevent the ship from flying out of the screen.

```
float xThrow = CrossPlatformInputManager.GetAxis("Horizontal")
float xOffset = xThrow * speed * Time.deltaTime;
float xPos = transform.localPosition.x + xOffset;
float clampedxPos = Mathf.Clamp(xPos, -xRange, xRange)
Transform.localPosition = newVector3(clampedxPos, transform.localPosition.y, transform.localPosition.z);
```

Figure 41. Function Mathf.Clamp() code

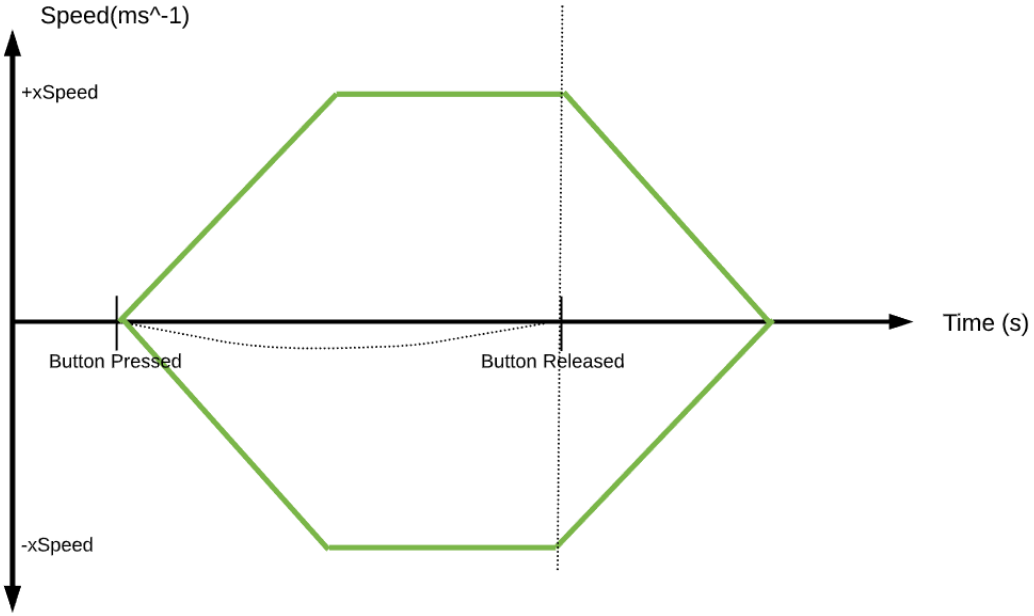


Figure 42. Speed is capped when Throw capped

5.6 Rotation Order Principles

The angles of Euler are added to this model in a certain order: Z, Y and X. As seen in the Figure 43, the reference frame shifts with each rotation as the object rotates around two or more axis. The spaceship goes up and down as the X axis is shifted. When the Y rotation is applied then the spaceship does not rotate around the world's Y axis, but instead around the plane identified by the X rotation. In other words, the Y rotation is related to the spaceship's frame of reference and not the world reference. The rotation of Z also adjusts clearly with respect to the spaceship's direction and flips it around its axis. Following these principles, we have successfully applied the extra rotation of the player spaceship as discussed in Section 4.5.

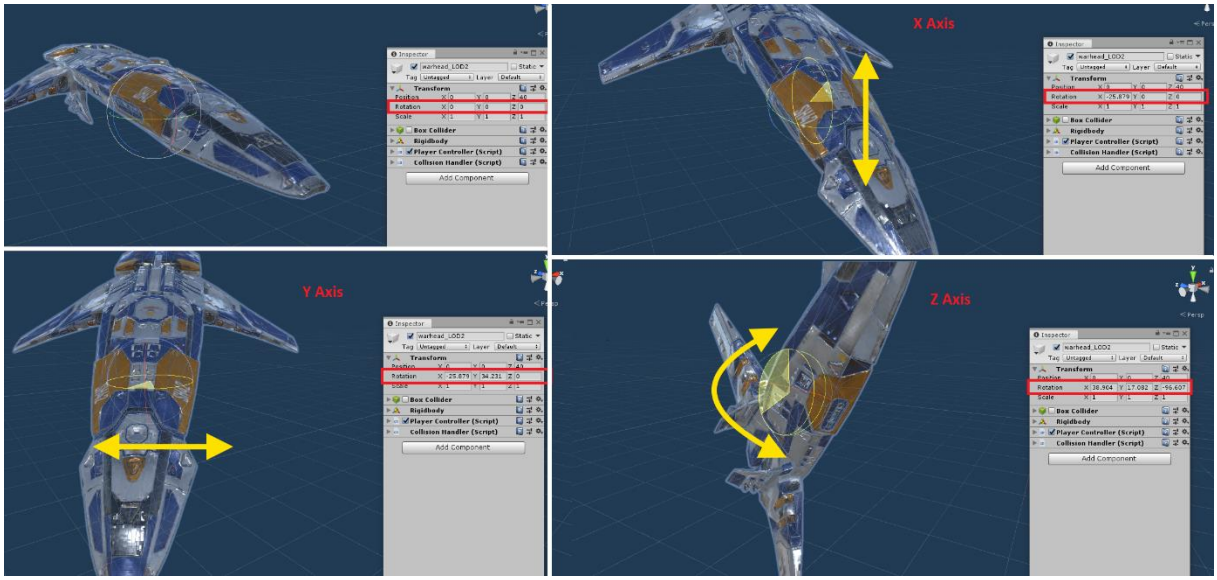


Figure 43. Principal axes of the player ship - Euler Angles

Taking the above into account, we need to think about how the ship should aim and hit things in a different position on the screen. When the ship moves around the screen, it always faces the centre of the screen, which means it will always shoot in the middle whenever it goes. In this paradigm, we are not actually going to be able to shoot anything unless it comes to us from the centre of the screen which is not practical. In order to fix this, we need to manipulate the pitch, yaw and roll of the ship. Furthermore, when the ship is on the left side of the screen, it should yaw slightly to the left (Y Axis) so that it can be directed to the left side of the screen. In addition, if the ship is low, then we would like to pitch it slightly downwards (X Axis) so that it is more aimed at the bottom of the screen. So, as the player moves the ship around the four corners of the screen, they should also aim at the four corners of the screen in a virtual target box as shown in the figure below.

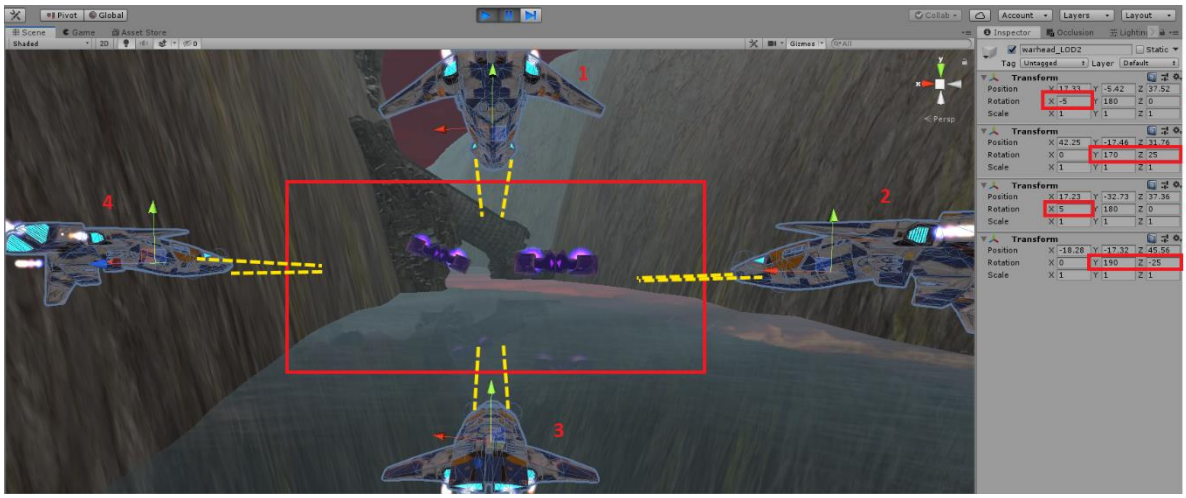


Figure 44. Using Euler angles to fix aiming

5.7 Tuning

Small tunings to the game are essential and incredibly timesaving at every stage of its development. The Figure 45 shows the tuning circle for the gameplay. When a feature is added, it is critical to fix the bugs associated with it. After that, the tuning of the game is essential, which will allow the developer to test some more values until the gameplay feels better than before, e.g. by tweaking the speed of movement on the rail, the distance of the camera and the field of view (FOV), the speed of the player, the rotations, the clamps and the obstacles to avoid. Finally, the playtest summarizes the overall experience and shows whether the gameplay needs to be tuned again.

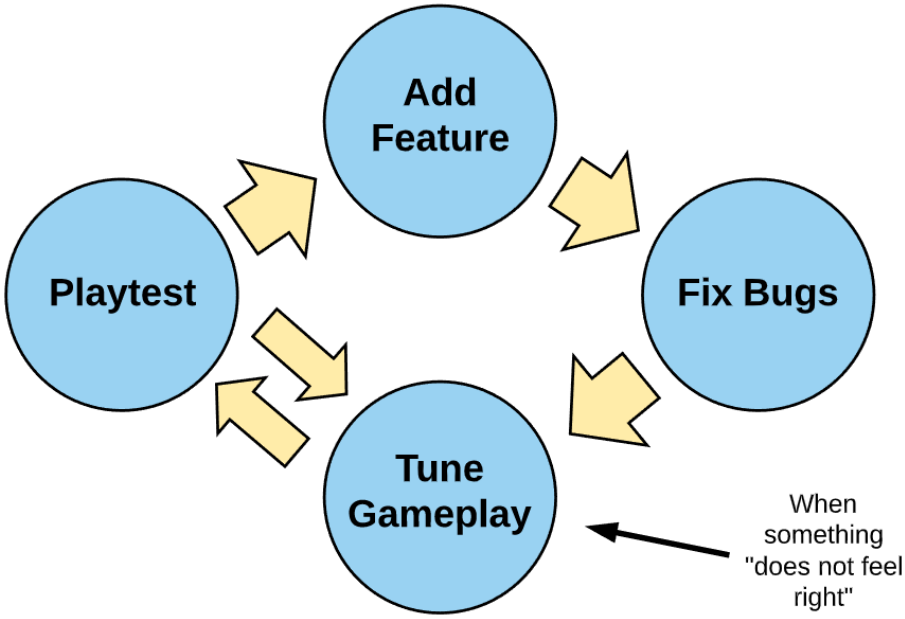


Figure 45. Game Testing Circle

5.8 Particle System

A particle system simulates and returns several tiny pictures, called particles, creating a visual effect. Each particle in the system is a graphical unit of the outcome. The system dynamically simulates each particle to generate a full effect impression. Particle structures are useful for generating realistic objects such as flames, smoke, liquids, or laser beams [27].

5.8.1 Create and child a Particle System to the Spaceship

For the purposes of the underlying project, we are going to create a Particle System which will be under the player ship as a child. The default output animation of the Particle System is just a few white tiny particles flying around the world with no purpose whatsoever. Our goal here, by using a Particle System, is to create projectiles like laser beams, and when a projectile collides with an enemy, it triggers an explosion and sends a message that the enemy should be damaged.

5.8.2 Shape of Particle System

The first step in creating the laser beam particles is to change the shape of the volume from which the particles can be emitted, the direction of the starting velocity to the cone and reduce the angle down to 0 in order for the particles to be emitted in one direction. If the shape is left as a sphere, the particles will be emitted in all sorts of directions (see Figure 46).

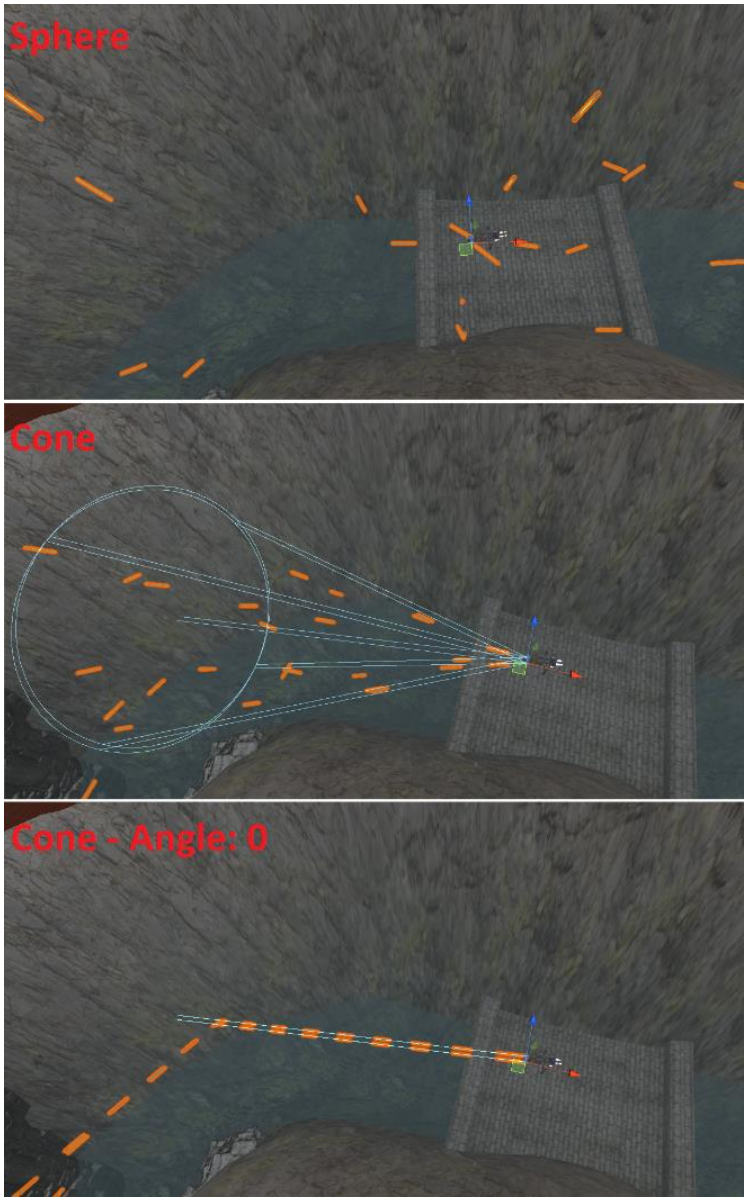


Figure 46. Shape of Particle System

5.8.3 Duration of particles

Next step is to change the duration of particles. The default duration is 5 seconds, which means that the particles are emitted for 5 seconds, and then they stop. We do not really need 5 full seconds of particles to be emitted. We want our system to say, "Emit something" and then "Stop" because when the user releases the fire button, we need to kill our particle system and stop the bullets.

5.8.4 Forming a laser beam particle

Although it looks like a huge laser beam is not going very far. This is because the Start lifetime value is 2 seconds, meaning that each of the particles will last 2 seconds. In addition, we increased the Start Speed to 300 to make it as fast as a bullet and change the starting colour to yellow to make it stand out

from the background. More importantly, the Particle Simulation Space needs to be Global instead of Local because, as it is now, the particle system is a child of the player ship and responds in local space. Thus, when the player ship moves up and down, the whole particle system moves up and down, the bullets tend to keep going where they were shot.

5.8.5 Trails

Obviously, as seen in movies, the laser beams leave a trail behind. Thus, in the Trails Module, the Lifetime value, which is the value that defines the duration of each trail relative to the life of the particle, is changed from 1 to 0.2 in order to create some empty gaps between each beam.

5.9 Core Game

When it comes to avoiding obstacles, we need to have a buffer of about half the screen of space for the player to move through. It is too challenging to have a tiny keyhole to thread the way through, which will frustrate the player and probably give up. In addition, the figure below shows the static environment of objects / obstacles, e.g. bridges, castles, buildings, rocks, caves and enemies that block the way, while providing extra space for players to enjoy the gameplay and have the opportunity to avoid obstacles very easily.

Figure 47. Obstacles and enemies that blocks the way

5.10 Triggers & Collisions

For the player ship, we just want to receive a trigger message and blow up the ship with an explosion when it collides with the terrain or enemies. Therefore, if the terrain always has a *Static Collider*, the ship will have a *Kinematic Rigidbody Trigger Collider*. As shown in Figure 48, we used *Box Collision* for the player ship and *Mesh Collision* for all of the static game objects.

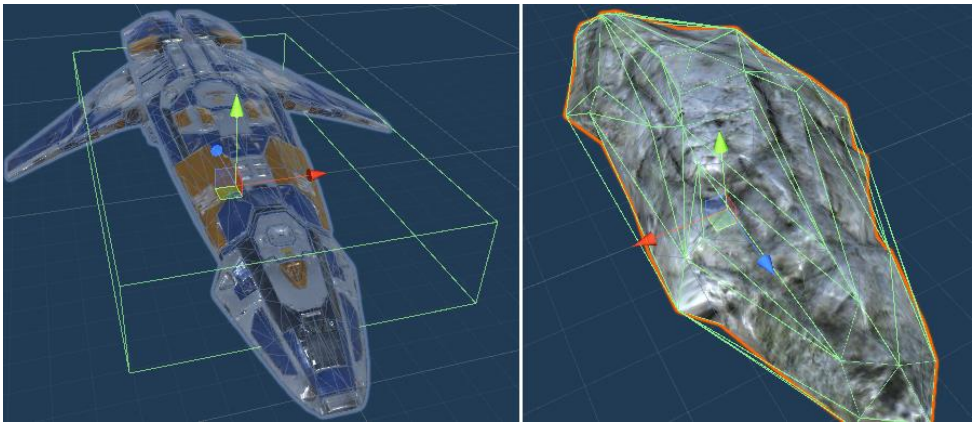


Figure 48. Left: Box Collision, Right: Mesh Collision

When a player ship collides with another GameObject, Unity calls *OnTriggerEnter* [28] as shown in Figure 49. *OnTriggerEnter* occurs on the FixedUpdate function when the two GameObjects collide and enables the *deathFX* which is the explosion sound of the ship when it dies. It then calls the *StartDeathSequence* function which deactivates the movement controls and finally resets the Scene 1.

```
void OnTriggerEnter(Collider other)
{
    deathFX.SetActive(true);
    StartDeathSequence();
    Invoke("ReloadScene", levelLoadDelay);
}

1 reference
private void StartDeathSequence()
{
    SendMessage("OnPlayerDeath");
}

0 references
private void ReloadScene() // string referenced
{
    SceneManager.LoadScene(1);
}
```

Figure 49. OnTriggerEnter Collision

5.10.1 Enable a Game Object During Play

We are effectively checking that box in the Inspector to enable a game object as shown in the Figure 50. In our game, we need some game objects to be enabled / disabled when a collision occurs. We made sure that the *"Play On Awake"* particles and audio were enabled, and then we referred them to a game object, e.g. an explosion (*deathFX*) that will be switched to True after a player collision*: deathFX.SetActive(true).*

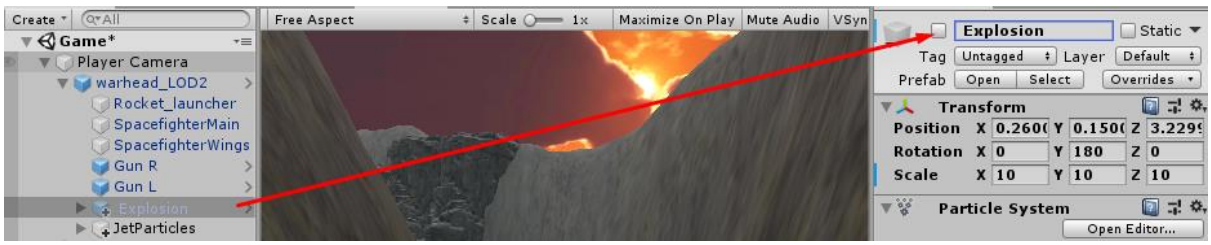


Figure 50. Disable/Enable gameObject

5.11 Detecting Particle Collisions

In this section, we are going to use the particle system as a stream of bullets that will cause collisions and destroy the enemy. First, we need to ensure that the Collision Module (under Particle System) and Collision Messages (under Collision Module) are switched on [29]. As shown in Figure 51, when the Collision Module is turned on, the laser beams bounce when they collide with the terrain. After examining the *Min Kill Speed* property, which was responsible for removing particles from the system when traveling below the default zero speed after a collision, we decided to keep the bounciness as it was easier for the player to accidentally kill more enemies with an unforeseeable bouncing laser beam. The next step is to add a non-trigger collider to the target that, in our case, targets are enemy spaceships and use the *OnParticleCollision* [30] message on the target.(see Figure 52).

Figure 51. Particle System (Laser beams bouncing when collision happens)

```
0 references
void OnParticleCollision(GameObject other)
{
    Destroy(gameObject);
}
```

Figure 52. Destroy Enemy when Collide with Particle

5.12 Add Simple Score UI

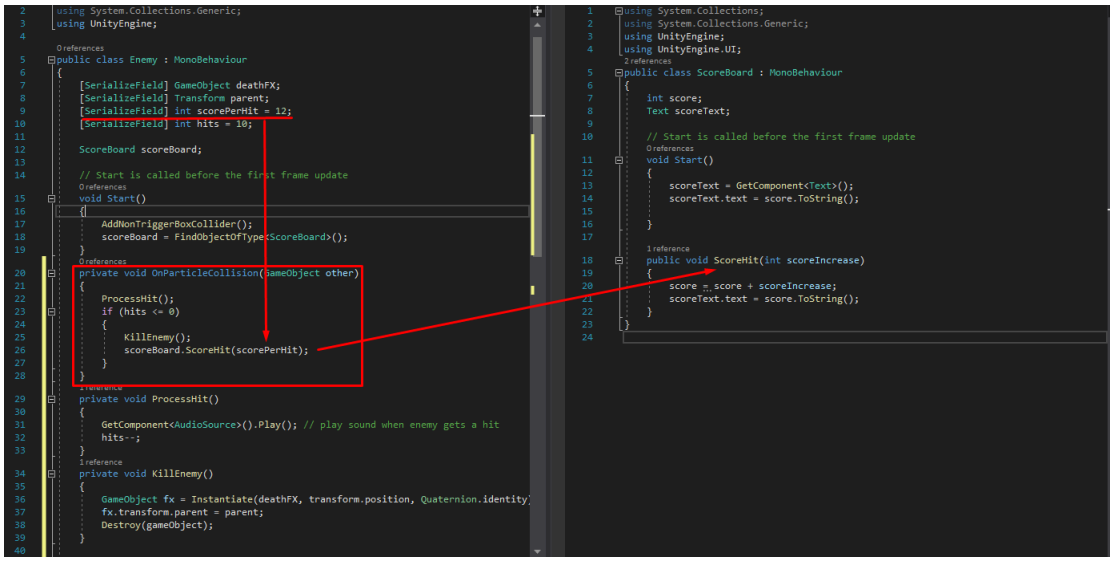
In this section, we are going to set up a scoreboard with a font on the top right of the screen (see Figure 53). When a UI element is added for the first time in a project, it generates something called "Canvas" [31] which is an area that all UI elements should be inside and a *"EventSystem"* [32] which is a way to send events to objects in an application based on input (keyboard, mouse, touch, etc.). The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of the Canvas. The type of Canvas used for this game is something called a Screen Space Overlay canvas, which overlays

the entire canvas over the top of the screen. There are also other types, where one that takes effect on the effects of the camera and anything that passes between the camera and the UI. And another that is in world space that can be used to float over enemy health bars.

Figure 53. Canvas Scoreboard

5.12.1 Get the Score Updating

In order to increase the score, we need to have any game object that has an "*Enemy*" script on it to be able to communicate with the game object that has the "*ScoreBoard*" script on it (see Figure 54). In addition, the enemy needs to refer to the ScoreBoard and call the "*ScoreHit*" function to update the score when the enemy is destroyed.



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

0 references
public class Enemy : MonoBehaviour
{
    [SerializeField] GameObject deathFX;
    [SerializeField] Transform parent;
    [SerializeField] int scorePerHit = 12;
    [SerializeField] int hits = 10;

    ScoreBoard scoreBoard;

    // Start is called before the first frame update
    0 references
    void Start()
    {
        AddNonTriggerBoxCollider();
        scoreBoard = FindObjectOfType<ScoreBoard>();
    }
    0 references
    private void OnParticleCollision(GameObject other)
    {
        ProcessHit();
        if (hits <= 0)
        {
            KillEnemy();
            scoreBoard.ScoreHit(scorePerHit);
        }
    }
    1 reference
    private void ProcessHit()
    {
        GetComponent<AudioSource>().Play(); // play sound when enemy gets a hit
        hits--;
    }
    1 reference
    private void KillEnemy()
    {
        GameObject fx = Instantiate(deathFX, transform.position, Quaternion.identity);
        fx.transform.parent = parent;
        Destroy(gameObject);
    }
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
2 references
public class ScoreBoard : MonoBehaviour
{
    int score;
    Text scoreText;

    // Start is called before the first frame update
    0 references
    void Start()
    {
        scoreText = GetComponent<Text>();
        scoreText.text = score.ToString();

    }

    1 reference
    public void ScoreHit(int scoreIncrease)
    {
        score = score + scoreIncrease;
        scoreText.text = score.ToString();
    }
}
```

Figure 54. Update Score

5.12.2 Enemy Health System

As identified by a quick gameplay test, the enemy is dying instantly when a laser beam hits them, making the game unfavourably easy without even a slight challenge. The enemies are now supposed to have a number of hits that they can take, and they are going to depend on the particle system and how

fast the player ship is firing particles. Additionally, some enemies should be harder to kill than others, e.g. the boss will take a lot of hits before it gets destroyed, but it will give the player a lot of points compared to the weaker enemies. To do this, we need to create two *SerializeFields* in the *Enemy* script, one for *Score Per Kill* and *Hits (maxHealth)* which each enemy will give different scores based on their maximum health. Figure 55 shows the types of enemies that will be included in the game, as well as their health and number of points. Every time the enemy is damaged by the player, the enemy's health is reduced until the enemy is destroyed and reward the player with a few points. In order to do this, we need to set a condition inside the *OnParticleCollision* function that triggers each time a particle collides with the enemy. As a result, this reduces the enemy's maxHealth by 1 on each hit until it drops to zero and destroys it.

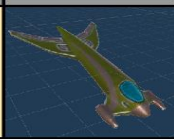
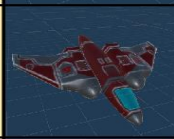
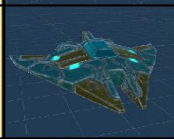
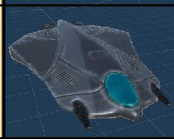
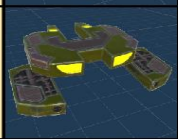
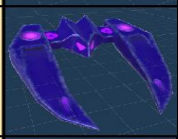
Type	Image	Hits to die	Points
Speedy		1	30
Level 1 Red		1	25
Level 1 Green		1	25
Level 1 Grey		1	25
Level 2		1	50
Level 3		5	100
Boss		600	10,000

Figure 55. Enemy Types

5.13 Fire Button

Usually in a lot of shooting games, there's a button that triggers fire. Similarly, in this game, we want to disable guns if the "Fire" button is not pressed and enable guns if the "Fire" button is pressed (see

Figure 56). We need to use the *CrossPlatformInputManager.GetButton("Fire")* and attach it to the texture button in the canvas UI while, at the same time, the *PlayerController* script detects if the "Fire" button is pressed and activates the guns using *guns.SetActive(true)*.

Figure 56. Fire Button (Left: Released, Right: Pressed)

5.14 Level Design Iteration

We give the player a taste of the story by giving them the opportunity to fly around the map and kill all enemies in order to save the island and prevent the evil Hydra from getting the crystal. The story has to match the design of the level. As a result, the appropriate textures, objects, lighting, paths the player will follow, and the overall scenery needs to match expectations. The Figure 57 shows some of the proper scenes of the prototype game.

Figure 57. Illustration of the small village, the magic crystal, some bridges, temples, the forest and the river

In addition, the route that the player will follow is quite unpredictable in order to give the player the opportunity to fly under bridges, over mountains and not just a straight path. As a result, the player can see a bit of normality in the world, and not just a terrain that has bumps and is random.

5.14.1 Enemies attacking

When the game starts, it will be unreal if the player sees the enemies spawning in front of them. It will be too obvious, and the player will end up losing interest, because that does not make any sense. The best way to avoid this mistake is to have the enemies spawning out of the line of sight of the player, which means that the enemy should spawn off-screen and approach the player from somewhere that the player does not see them, e.g. spawn the enemy behind a rock or a mountain as shown in Figure 58.

Figure 58. Enemies approaching from outside the line of sight of the player

5.15 Timeline

The following figure shows the Boss Battle Animation using the Timeline tool. At 0:00 the Boss spawns to the first key frame with location -362 (X), 361 (Y), 1267 (Z) and rotation -12 (X), -93.5 (Y), -5.7 (Z). The next key frame is after 7 seconds with location -323 (X), 761 (Y), 1103 (Z) and rotation 7.5 (X), -25 (Y), 5 (Z), which means it will take 7 seconds for the Boss to move from the first key frame to the second key frame and change its location and rotation values accordingly. In the same way, movements and animations applied to the rest of the enemy and the player ship, using the same procedure.

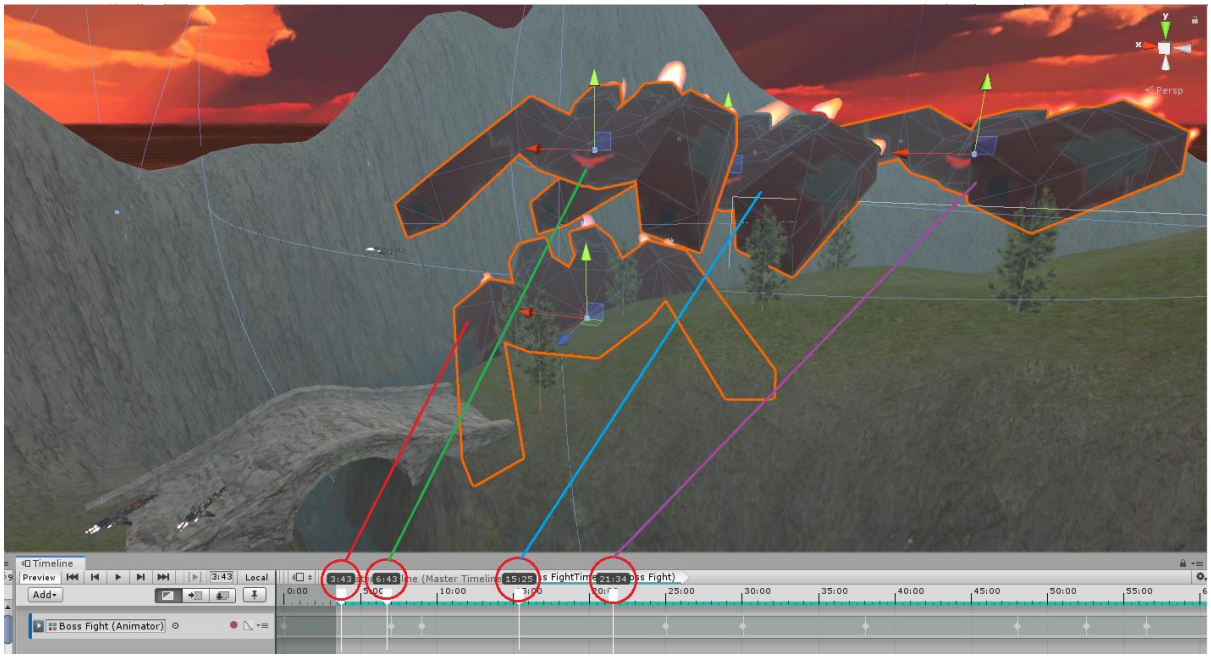


Figure 59. Boss Fight animation using Timeline tool

5.16 Nested Timelines on Control Tracks

The Figure 60 shows our Master Timeline, and for each enemy wave, we create individual timelines with animations for each enemy with their key frames represented as red diamonds. Then we create a Control Track, and within Control Track, there is a Control Playable Asset that will be the enemy wave timeline. This will allow us to control the "when" that specific timeline will occur in relation to the overall master timeline by moving it forward or backward.

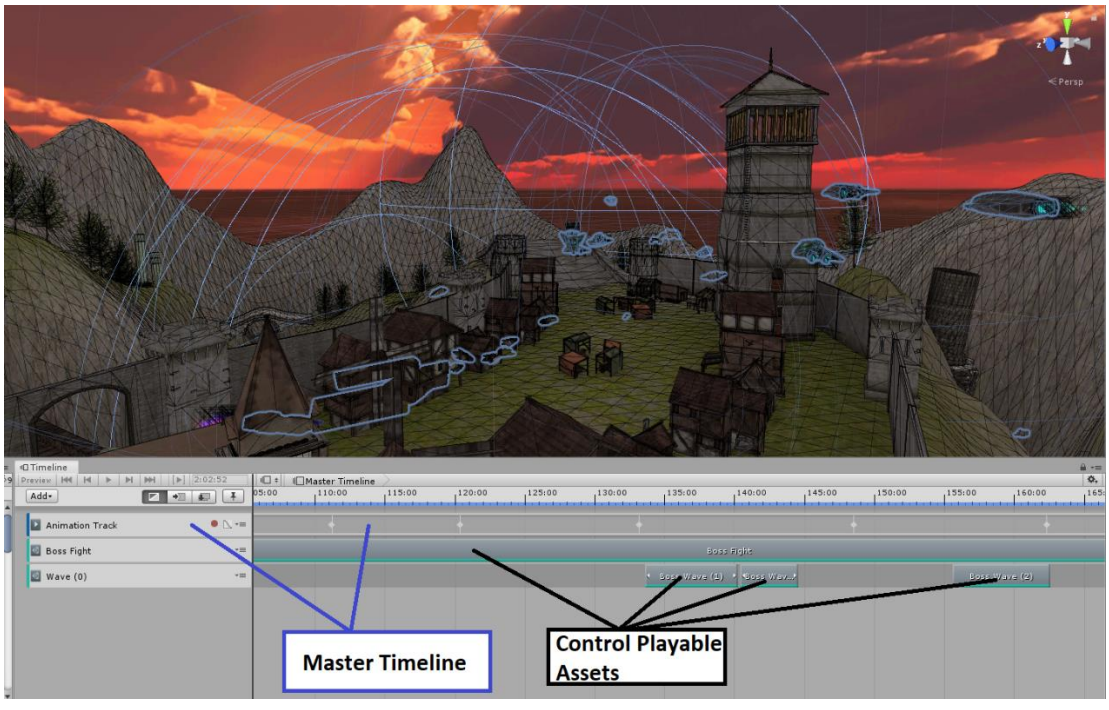


Figure 60. Control Playable Assets

5.17 Deferred Fog

Based on how far the game objects are from the camera, the fog effect overlaps the colour of objects. The Fog effect produces a screen-space fog based on the camera's depth texture. It simulates fog in outdoor conditions and also hides clipping game objects while the camera's virtual clip plane is progressing to performance [33], which is perfect for our game. The figure below shows the visual difference between the option Fog enabled and disabled.

Figure 61. Left: With Fog Enabled, Right: No Fog

5.18 Baking Occlusion Culling

Some manual configuration requires in order to use Occlusion Culling. The geometry of the floor must first be separated into sensibly large sections. We thus set the level in small, well-defined areas where large objects such as walls, bridges, buildings and so on occlude each other. The theory is that each mesh is turned on or off depending on the occlusion details. Upon baking, the Unity loads baked data to the memory at runtime and allows queries against the data for each camera that can see the Occlusion Culling object. The figure shows the difference between enabled and disabled Occlusion Culling during runtime of our game.

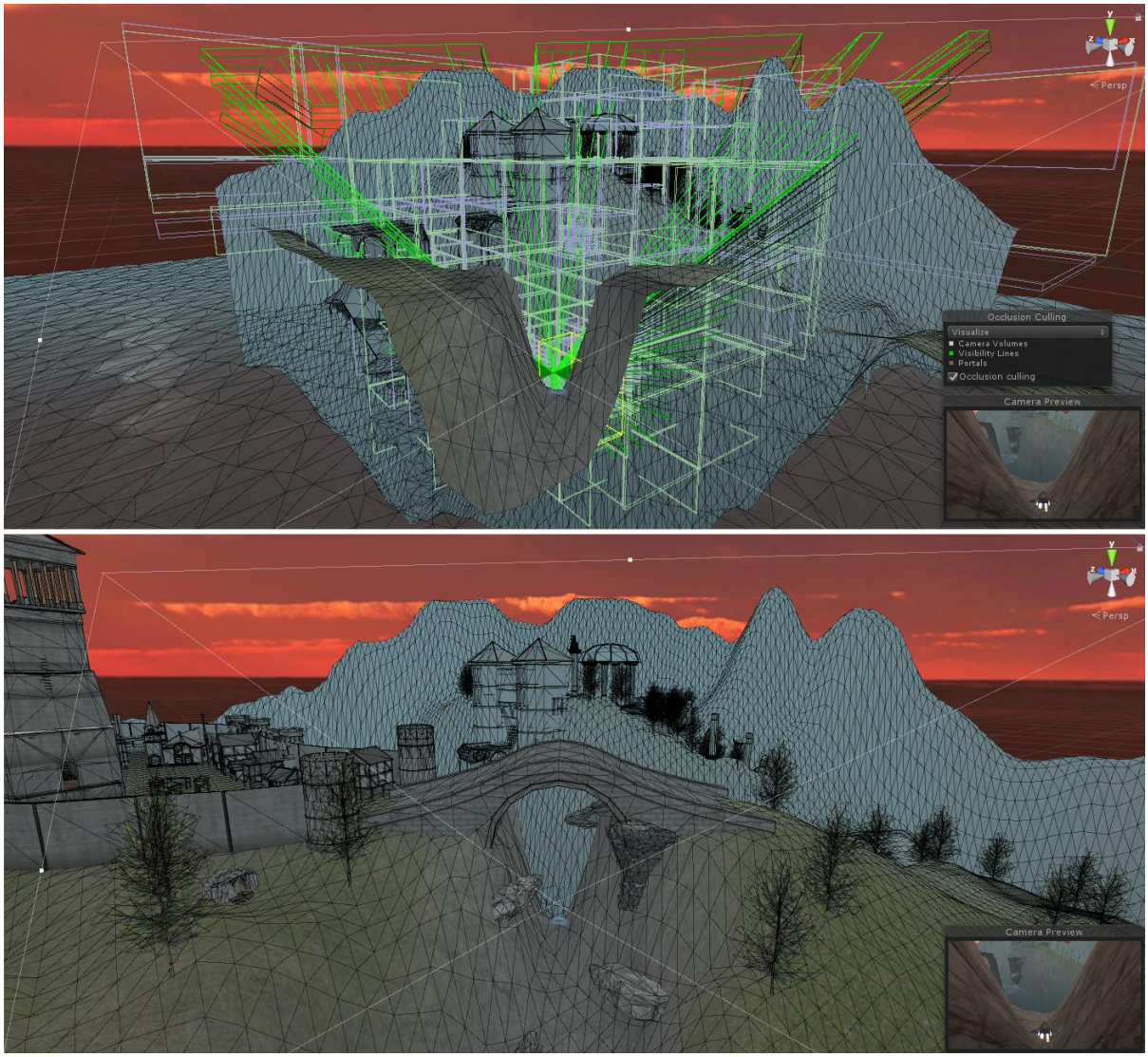


Figure 62. Unity renders only what the camera sees – Enabled Occlusion Culling (Top), Actual Scene - Without Occlusion Culling (Bottom)

The figures below (Figure 63 & Figure 64) demonstrate the performance statistics and the noticeable improvement when occlusion culling is enabled using the *"Statistics"* runtime tool and the *"Profiler"* more in-depth analysis tool. Batches, tris, verts and calls are significantly reduced.

Figure 63. Statistics – Left (Occlusion Culling Disabled), Right (Occlusion Culling Enabled)

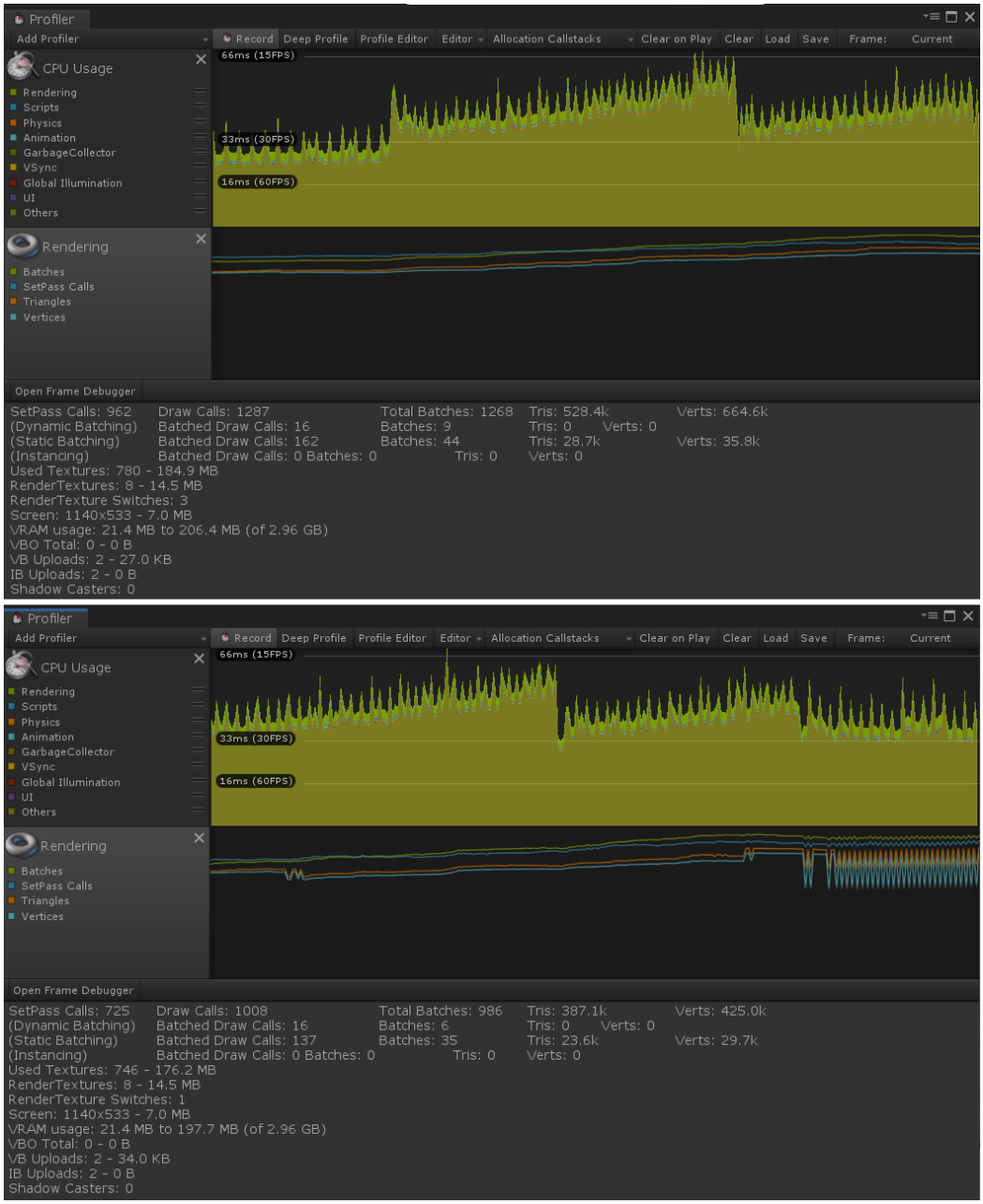


Figure 64. Profiler – Top (Occlusion Culling Disabled), Bottom (Occlusion Culling Enabled)

5.19 Touch Controls / Mobile Input

The first thing we need to do when working with mobile devices in general is to move our build target to the Mobile Platform under the Build Settings section. When the project is on the Mobile Platform, all UI elements will work by default with a touch without any additional code, e.g. press the button or tap the screen. For this game, we want the player to be able to move the spaceship around the screen using the virtual joystick and placeholder sprites from the asset store. All that needs to be done is to use the Fixed Joystick and make it our main Canvas as a child (see Figure 65). The script that Unity provides with its Standard Asset Pack handles everything as long as the joystick variable is declared in the PlayerController script and attached to the spaceship object.

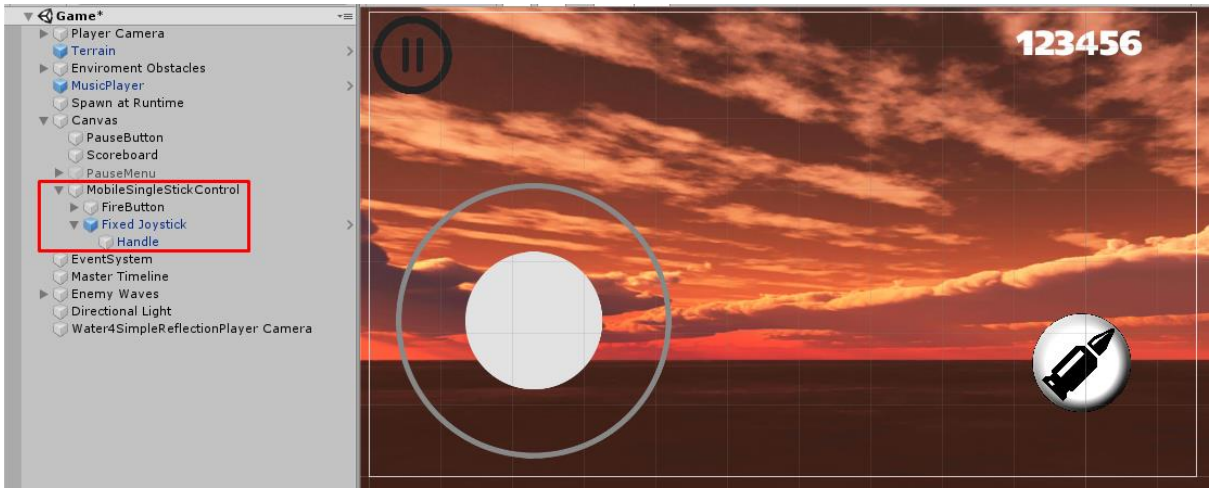


Figure 65. Virtual Joystick and Fire Button

5.20 Route of the spaceship

The figure below shows the starting point of the ship, the enemy waves and, in general, the entire route of the spaceship around the map. The route is predetermined as the game genre requires, while the goal of the player is to win a game by achieving the highest score. As a result, the route is infinite by connecting the end waypoint to the start waypoint, creating a never-ending loop.

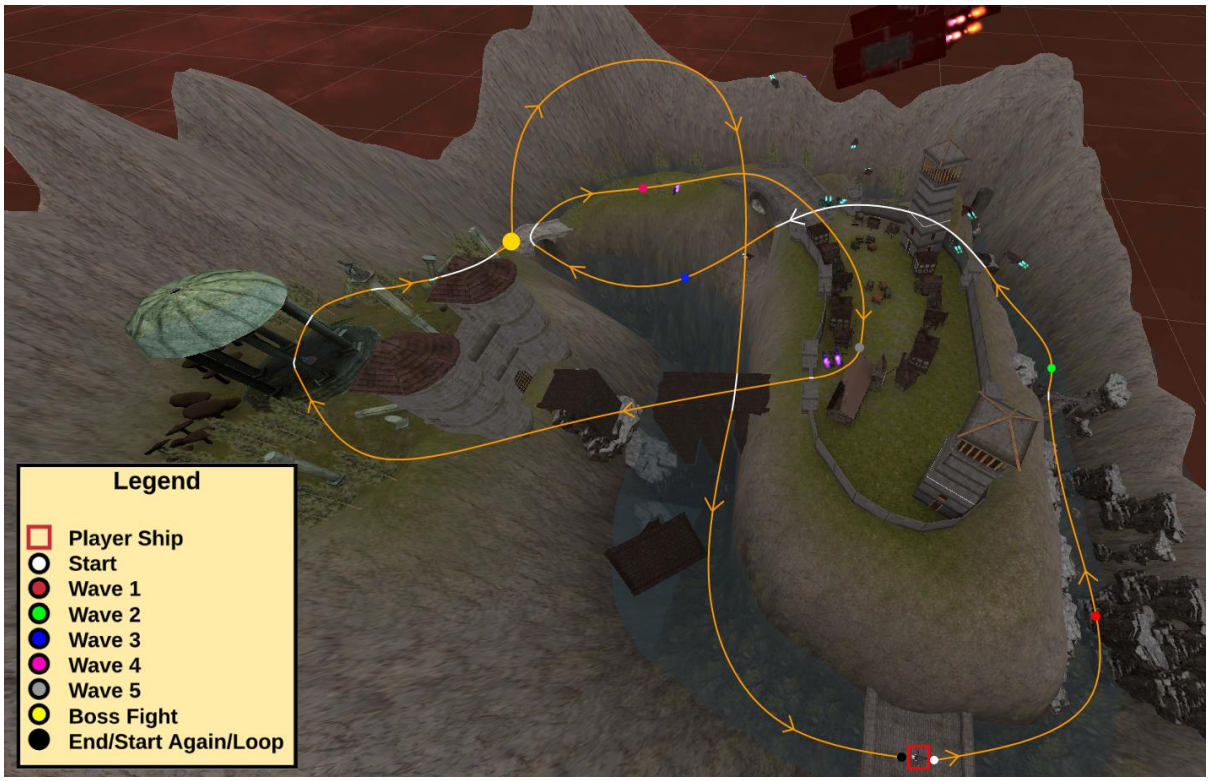


Figure 66. Route of the Player

5.21 Build Settings / Optimizations

The game as it is now running on any android device, however, due to performance issues, it runs with a maximum of 10 frames per second (FPS) which is unplayable even on higher end devices. In order to achieve above 60 FPS, which is the standard frame rate for games to be playable and supported on any smartphone screen, the following optimization has been made:

- Enabled Optimized Frame Pacing: Frame Pacing delivers frame rate consistency by enabling frames to be distributed with less variance between frames, creating smoother gameplay.
- Default Orientation: Landscape
- Game Screen Resolution: 16:9 (landscape)
- Target Architectures: ARMv7 and ARM64 (chosen both to support the game on older Android devices and also it was required by Google Play Store before publishing)
- Scripting Backend: IL2CPP (required in order to use ARM64 architecture)
- Enabled Prebake Collision Meshes
- Pixel Light Count: 0
- Texture Quality: Half Resolution
- Disabled Anisotropic Textures
- Disabled Anti-Aliasing
- Disabled real-time rendering of Reflection Probes
- Disabled Shadows
- Baked GI and Occlusion Culling
- Decreased Mesh and Texture Resolution on terrain data to the lowest (see Figure 67)

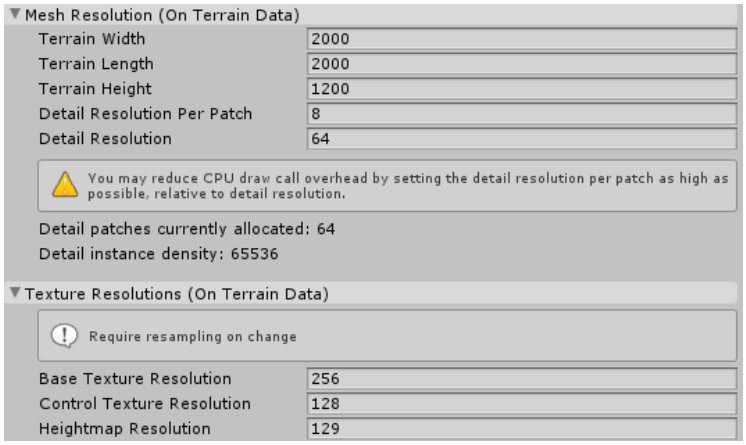


Figure 67. Mesh and Texture Resolution (On Terrain Data)

6 Testing

This section contains two separate game tests: The Personal Testing and the External Testing. The internal evaluation is my own truthful feedback, while the external evaluation depends upon user and professional responses. The testing is more concerned with an external evaluation of the game, rather than the composition of internal logic.

6.1 Personal Testing

Prior to the public release of the game for user testing, internal testing was applied to test its playability. The game was tested by my own Samsung Galaxy S10 Plus (high-end device), Samsung Galaxy A7 2015 (low-end device), Android Emulator (BlueStacks) and Unity Engine Editor. All testing methods suggested successfully compatibility.

The game's development was very focused on making it playable and accessible on different android devices without compromising on graphic quality and meeting the most demanding requirements. While all game problems could easily be overcome by changing the appropriate values by trial and error in the game engine. Despite sufficient time, there is one fundamental question that is still difficult to answer: is Hydra entertaining?

Thanks to my previous gaming experience, I found the game difficulty easy to play, but I enjoyed the production of the plot and the consistency of the game. However, I do think the game lacks the sparkling inspiring players want to play as a lot of features are incomplete, such as camera field for view settings, pickup items such as shields and powerups, more stages with various plot progression and most notably, a multiplayer mode e.g. co-op missions or a 4 vs 4 deathmatch.

To conclude this section with a more optimistic note, the game has several technical features, the majority of which can be described as complete, usable and ready for expansion.

6.2 External Testing

6.2.1 User Testing

In order to validate the playability of the game as well as the performance and compatibility, we made the game available to the Google Play Store along with a Questionnaire (see Appendix 1) asking participants about their experience. We managed to collect 96 responses, a representative number of responses which makes the results produced unbiased and accurate. Moreover, as we can see in the Figure 68, the most dominant age group was 21-25 with 56 participants out of which 29 are females and 27 are males. Overall, the survey obtained almost equal responses between genders.

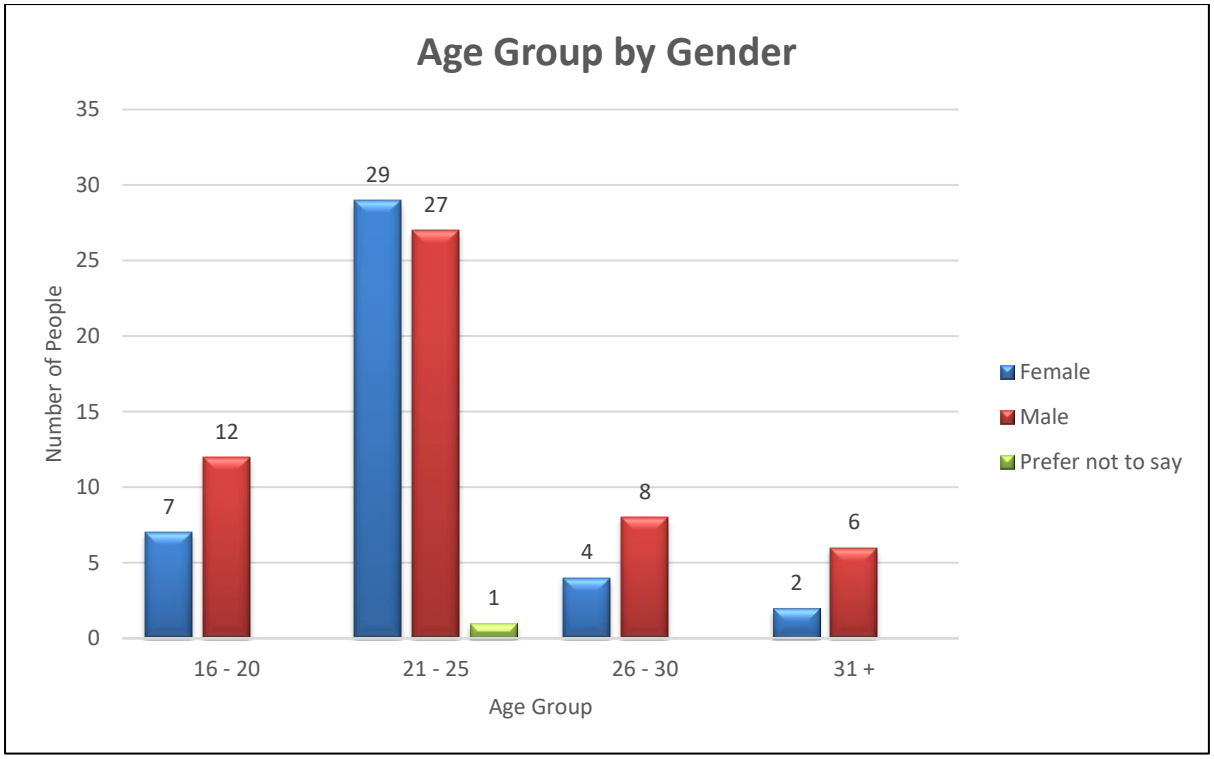


Figure 68. The demographics of our participants

Before asking participants about their experience on the game, we wanted to know their opinion on the following aspects:

- Preferred platform for gaming.
- Which of the following features is the most important: Graphics, Story or Performance?
- Mobile Game pros and cons.

As shown in the Figure 69, the majority (45%) of our participants prefer playing games on mobiles rather than computer and console platforms mostly due to portability access and low to zero costs to play games (see Figure 70). Moreover, the Performance and Story of a game are considered equally important features. The results here are much more useful as they give the developer a goal and a specific priority for the next update. Last but not least, we asked the participants to tell us what they dislike about mobile games. More than half of the responses are concentrated in Battery Drain (38%) and Advertisements (26%) as this tell us that people prefer lightweight games with no annoying advertisements (see Figure 71).

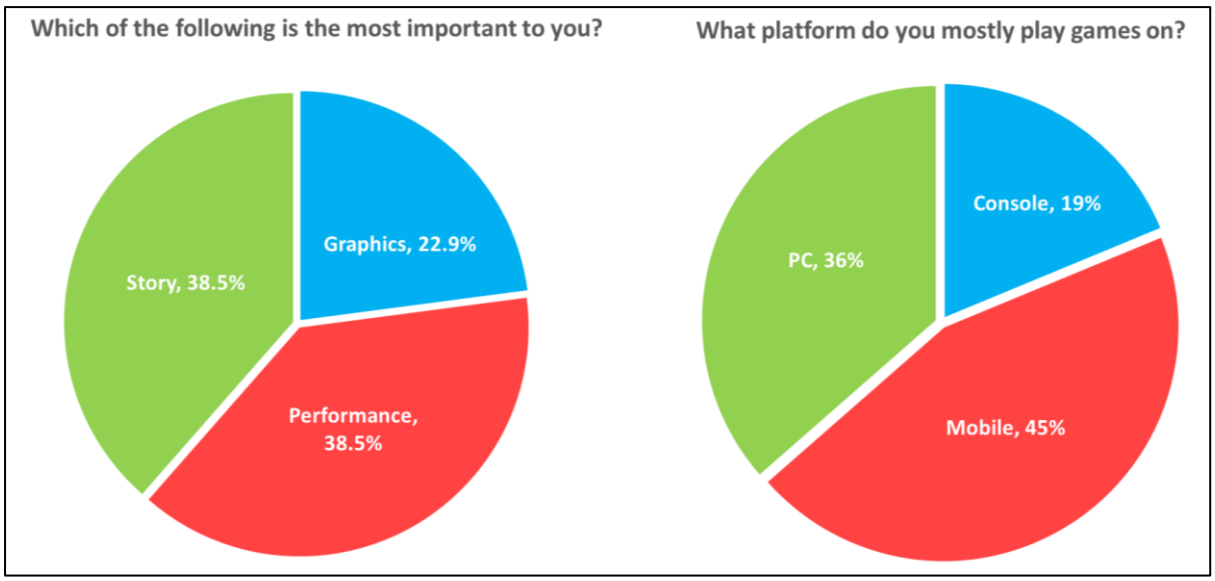


Figure 69. Preferable platforms and features

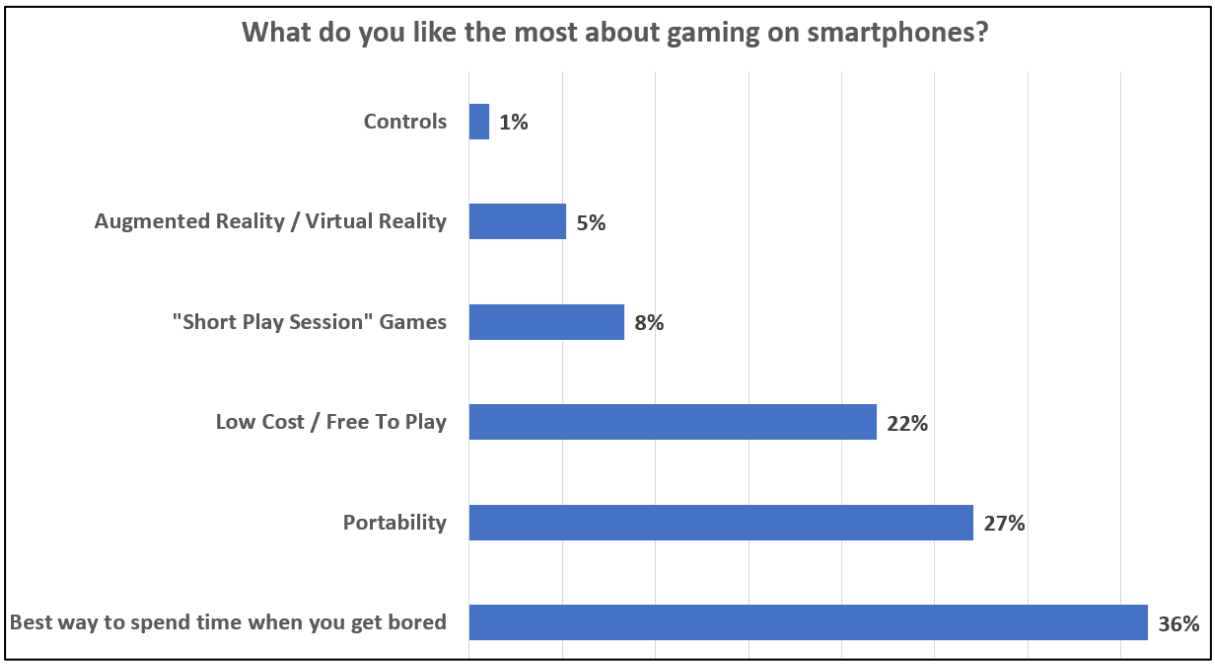


Figure 70. The advantages of playing video games on smartphones

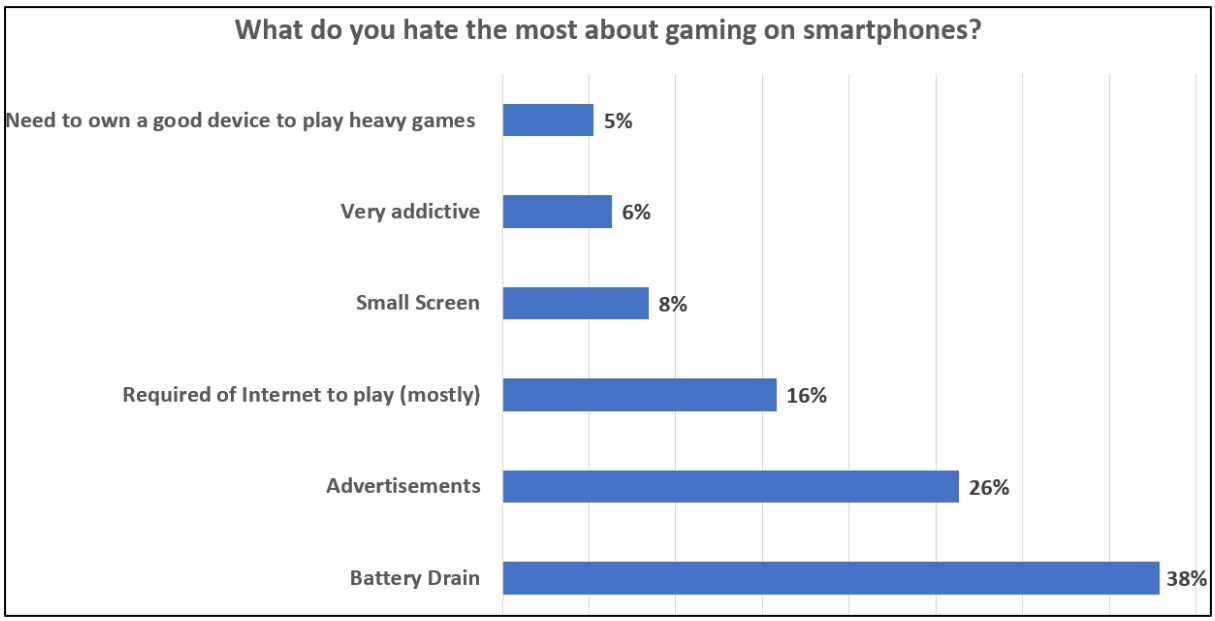


Figure 71. The disadvantages of playing video games on smartphones

We are now moving to the second part of the questionnaire which focuses on the user experience. To begin with, our game was tested in a variety of android smartphone brands (see Figure 72). In this respect, the game is successfully supported by all the android models tested, which fulfils the compatibility requirement as set in the Section 3.5.1.

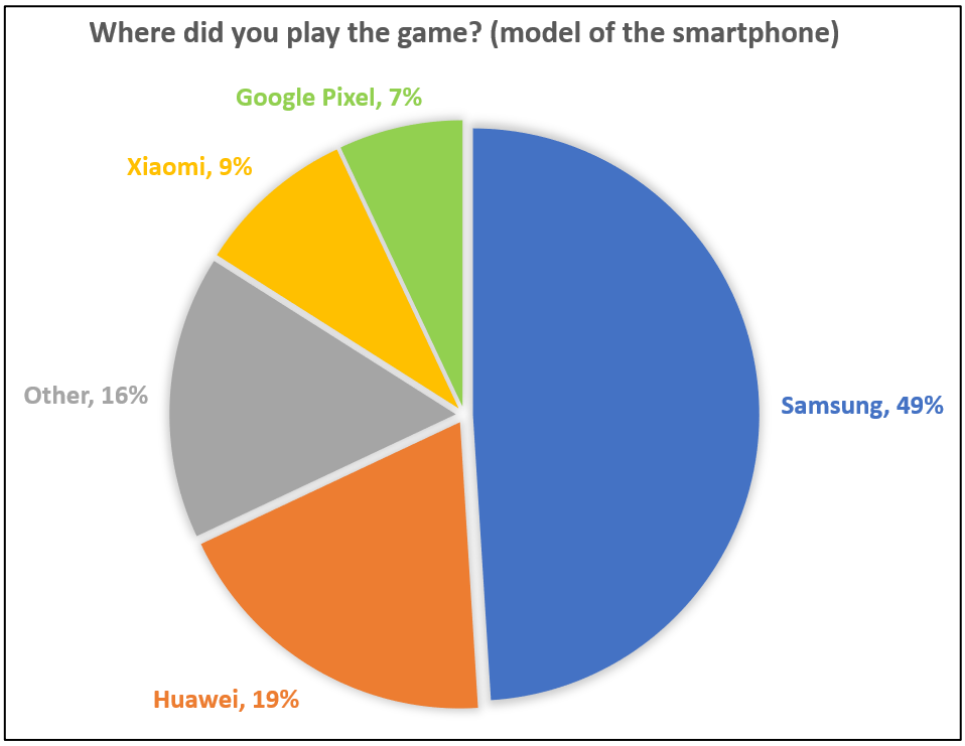


Figure 72. Android devices brands used for user testing

Secondly, we asked the respective players/users to rate in a scale from 1 to 5 of how well the game performed in their Android smartphone, how well the graphics were, how pleasant their experience was

and how responsive the controls of the game were. As seen in the Figure 73, no one had poor performance issues, however, some players expressed their unsatisfaction over the quality and controls, suggesting unpleasant experience of the game.

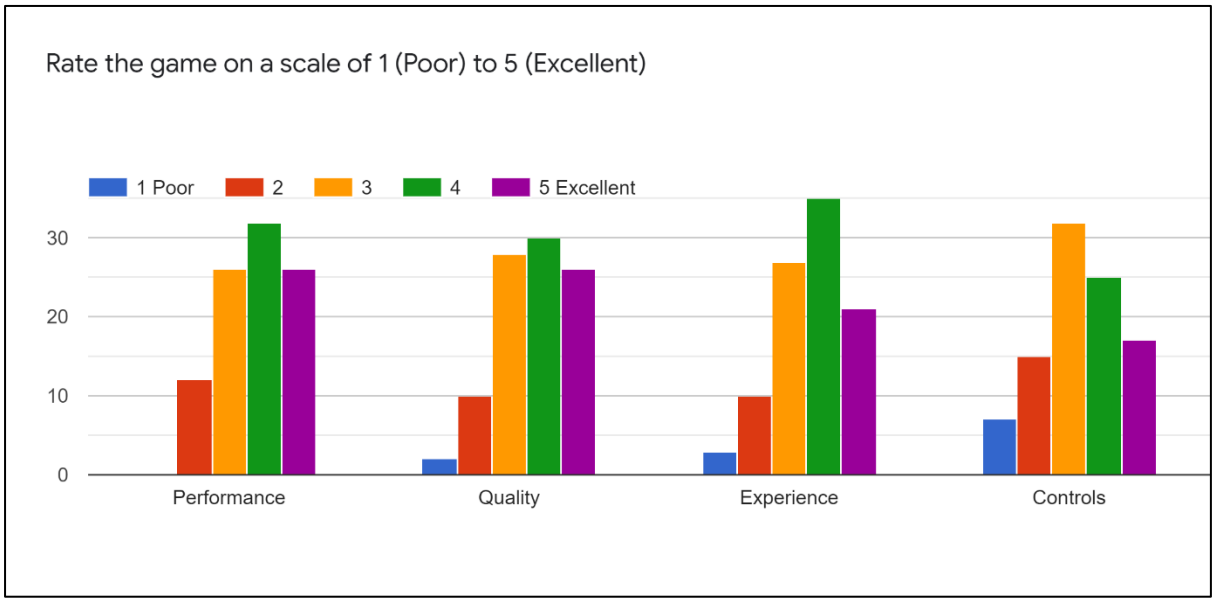


Figure 73. Ratings of the performance, quality and controls of the game and the overall experience of the user

Further to the above, we asked the participants to tell us what they liked and disliked about the game. As we can see in the Figure 74, the Gameplay was the most favored element. However, most of the "dislike" responses are concentrated in Controls, followed by Installation Size, Lack of Features and Difficulty. These results suggest that major changes should be done in Controls to be more responsive and user friendly (as 1/3 of the participants disliked controls). Moreover, our results suggest significantly reducing the size of the game as the current installation accounts for 400MB which is way to high in comparison of the average published 3D Android games of 25MB-100MB.

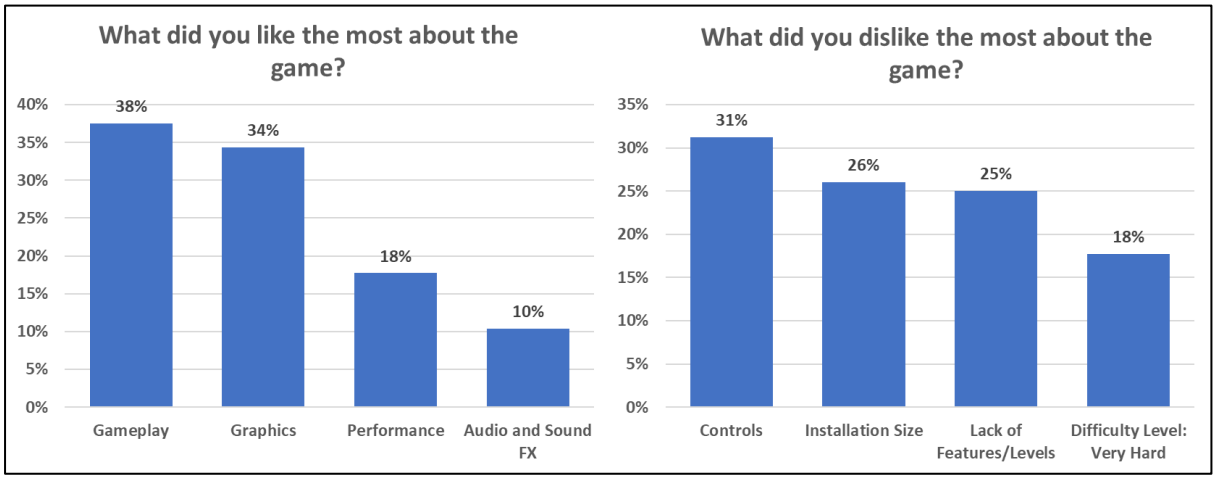


Figure 74. The advantages and disadvantages of Hydra

Finally, we asked the participants to provide us with suggestions. The majority of the participants outlined that improvement of controls and installation size is required. Other than that, the participants seem to be interested of the game and they proposed some features that could be added in the next update such as:

- More levels.
- Option to configure the controls sensitivity.
- Option to change the difficulty level (Easy, Normal, Hard).
- Option to change the camera angle (Field of View).
- Multiplayer option.
- Include pick-up items (Coins, power ups).
- Customize of spaceship (Spaceship upgrades in trade of coins).

6.2.2 Professional Testing and Evaluation

Further to user feedback, we also managed to obtain a professional one. Mr. Tim Gatland, the CEO of Rivet Games, and his team have spent time playing the game and provided me with an honest review according to their experience in the video gaming industry. Their overall feedback was positive and included technical advice on improving specific areas of the game. Specifically, Mr. Gatland, was surprised by the fact that the whole game was developed by only one person whose background does not rely on Gaming Development. He also, added that the game was enjoyable and addictive in a good way which this is the whole point of creating a game in the first place.

The technical advice concerns the improvement of the existing collision into a more detailed one as the player ship can very easily collide with environmental objects or enemies and die. This feature will help the amateur user to play and enjoy this game genre instead of focusing only to the most advanced ones. In addition, the team suggested the implementation of configuration settings such as Camera Field of View as this will improve the overall user experience. Lastly, we addressed the value of how a game can provide the developer or company with revenue. Monetization is one option where the game could be free with no annoying ads but include an in-app Store in which the player has the option to buy items such as Game Coins.

7 Conclusion

7.1 Summary

Today, video games have been one of the most common entertainment for a number of people. Nevertheless, it is not easy to develop such complex games as they require a combination of creative plot, compassion and communication abilities, in order to find a way to attract the audience's attention.

The purpose of the underlying thesis is to develop a 3D Mobile Game in Unity which would be playable and compatible across various android devices. Inspired by the *"Star Forces: Space Shooter"*, *"Subdivision Infinity: 3D Space Shooter"* and *"Space Racing 3D – Star Race"* games, we created a Rail Shooter game with the viewpoint of 3rd person perspective where the player movements around the screen are limited as the game follows a specific route. The players' ultimate goal is to achieve the highest score by collecting points when destroying enemy spaceships.

In order to check whether the game is attractive to users, we released the game on Google Play and collected 96 responses regarding its compatibility, graphics quality, game performance and overall user experience. The results suggest that the overall user experience was positive. However, the user interface still lacks gameplay settings, and the game generally lacks extra features. More attention needs to be paid to user feedback and, in particular, to the professional suggestions made by Mr. Tim Gatland, the CEO of Rivet Games and his team.

Summing up, all the project requirements outlined in Section 3 have been met and all design features discussed in the Section 4 have been successfully used. With all the limitations, Hydra is the best I have done in a given period of time, and it is definitely a step in the right direction to develop a proper 3D mobile game.

7.2 Evaluation

My own evaluation: As aforementioned, the primary purpose of the underlying project relies on the development of a 3D mobile game in Unity which will be playable and accessible on different android devices without compromising on graphic quality and meeting the most demanding requirements. In this respect, the game was tested on a) a high-end device, b) a low-end device c) an Android Emulator and d) Unity Engine Editor to ensure its playability and compatibility. All four testing methods along with both professional and user testing suggested successful playability and compatibility.

Further to the above, I successfully found some small user interface problems that were solved later when I reviewed the program myself. An example of this issue is light rendering, which relates with how Unity's default settings treat light sources in projects. The problem was that when the user switches between scenes the light of adjacent scenes would change immediately because of the default *"Automatic Bake"* light setting, which would influence the reflections brightness of the object and lead

to scenes with poor visibility, some of which were almost completely dark. To fix this problem, I looked for a way to manually manage light baking between scenes and discovered that Unity provides a game engine menu tool that lets the developer manually configure the *"Baked"* levels and, ultimately, after ensuring every scene is correctly handled (Main Menu, Game Screen), the problem was gone.

Another issue that was discovered during the tests performed by me was the issue of the file size of the build project ".apk". When I exported the project to Android Application, the size was surprisingly large (300 MB and above) with only 2 scenes, meaning that the game has a lot of unused assets, baked maps, and high-resolution textures. This, as suspected, has led to performance issues for low-end devices. To fix this problem, I manually lowered all of the scene polygons and general resolution to the "Lowest" menu in Unity Graphic Settings. The issue has been fixed, and the file size is now 135 MB without losing any overall graphic quality. Any differences are barely visible because the phone screens are so small in comparison to the computer screens. As a result, the differences between Version 1 (before optimizing the graphics) and Version 2 (after optimizing the graphics) are barely visible and the performance has improved significantly even to the lower end devices.

Since addressing the faults, I have done further checks to ensure that those who will test the product will have no issues on either the user interface, installation file size or performance and compatibility. The final output follows all of the core goal criteria.

Professional Evaluation: Mr. Gatland and his team were impressed by the fact that the whole game was developed by a student who has no background in Game Design and Development and highlighted that the game was enjoyable and addictive in a good way. The overall feedback received was very positive and encouraging, however, throughout the process, there were many aspects and details of the design that the Rivet Games professional team was unsure about. For example, it was not certain when and how the game would end as it automatically restarts every time the player dies. In terms of playability, they suggested using an online Leaderboard to save the score online so that each player would have a more solid goal, e.g. to beat the player with the highest score, as this is a never-ending game. In addition, they suggest some final touches and small tweaks on the controls and collision system e.g. make the controls bigger and the collision box of the player ship smaller in order to give the user more advantage not to die.

User Evaluation: The User Evaluation was collected through a survey published on "Survey Exchange" online groups and asked random people to play the game and complete the survey for me. After observing the participants' answers, I came to the conclusion that the overall user experience was positive with small negative feedback on the player's camera angle and the sensitivity of the controls.

Reflection: Overall, I am pleased with the final result, as all requirements listed in Section 3.5 were met. If I had to go over this process again, I would then try to give more control to the player by making the game an open environment so that the player can navigate the spacecraft wherever they want.

7.3 Future Work

Hydra 3D Shooting Game is essentially a proof of concept at its present level. Iteratively, the project can be taken to a more complete game by increasing the efficiency of the user interface and by increasing the features by adding more levels. The following features may be a significant change to the prototype:

Adding flavour to the story using Timeline: We are currently using the Timeline tool to manipulate the position of the player and enemy spaceships to create movement animations. Adding a voice over an object that represents something like a Chief that gives instructions to the player, or even explaining the back story of the game, would add more interest.

Settings: Right now, the "Settings" section is not complete, but it has room for expansion. It would be extremely useful for players to access this section and configure the game settings based on their satisfaction, such as FOV, Sensitivity Controls and Graphics Quality.

Extra features: As mentioned in Section 3.1, the outer layer of Onion Design consists of user feedback and evaluation for future updates. A lot of users suggested, the addition of more levels in order to have some progress and a reason to keep playing the game, the implementation of checkpoints in the game to save their progress when they die, the addition of a variety of weapons and spaceships to customize and, last but not least, the addition of pick-up items, e.g. power-ups, coins, abilities.

Assets: All of the existing objects used in the game are known to be "high poly" meaning that the number of polygons is fairly high, making them look realistic due to their complicated shapes. However, this leads to game performance issues and creates the need to reduce polygons manually while maintaining texture resolution in order to balance game quality and performance. This could be further improved by purchasing and using advanced, high-quality assets from the Unity Assets Store, optimized for and used by Android devices.

Monetizing: Developing a game is really time consuming and could be expensive by investing in premium quality assets and optimization tools. As a result, the game can be combined with an optional In-App store that the user may want to purchase extra coins for upgrades to spaceships.

References

[1] Takahashi, D., 2014. *Unity Technologies CTO Declares the Company Isn't Up for Sale*. [online] VentureBeat. Available at: <https://venturebeat.com/2014/10/16/unity-cto-declares-the-company-isnt-up-for-sale/> [Accessed 9 April 2020].

[2] *Valuecoders.com*, 2017. [Online]. Available: https://www.valuecoders.com/blog/wp-content/uploads/2017/03/Unreal-Engine-vs-Unity-3D-ease-of-use.jpg. [Accessed: 09- Apr- 2020].

[3] F. R and K. M. J., *The CG Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 2003, Chapter 7: Environment Mapping Techniques.

[4] Horvath, S., Oberhaus, D., Edelman, G., Martineau, P., Oberhaus, D., Greenberg, A. and Jonathan M. Gitlin, A., 2015. *The Imagination Engine: Why Next-Gen Videogames Will Rock Your World*. [online] WIRED. Available at: <https://www.wired.com/2012/05/ff-unreal4/> [Accessed 9 April 2020].

[5] *Answers.unrealengine.com*, 2020. [Online]. Available: https://answers.unrealengine.com/storage/temp/10554-playercontroller.jpg. [Accessed: 09- Apr- 2020].

[6] U. Technologies, "Unity - Manual: Rendering Profiler", *Docs.unity3d.com*, 2017. [Online]. Available: https://docs.unity3d.com/560/Documentation/Manual/ProfilerRendering.html. [Accessed: 09- Apr- 2020].

[7] U. Technologies, "Unity - Manual: Normal map (Bump mapping)", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/Manual/StandardShaderMaterialParameterNormalMap.html. [Accessed: 08- Apr- 2020].

[8] "The Albedo Map Mystery Revealed - CG Director", *CG Director*, 2020. [Online]. Available: https://www.cgdirector.com/albedo-map/. [Accessed: 09- Apr- 2020].

[9] Carter, A., 2019. [online] Available at: <https://www.artstation.com/artwork/Z501dG> [Accessed 8 April 2020].

[10] "Skybox Basics - Valve Developer Community", *Developer.valvesoftware.com*, 2018. [Online]. Available: https://developer.valvesoftware.com/wiki/Skybox_Basics. [Accessed: 09- Apr- 2020].

[11] "Cube Maps — Panda3D Manual", *Docs.panda3d.org*, 2019. [Online]. Available: https://docs.panda3d.org/1.10/python/programming/texturing/cube-maps. [Accessed: 09- Apr- 2020].

[12] U. Technologies, "Unity - Manual: Cross-Platform Considerations", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/Manual/CrossPlatformConsiderations.html. [Accessed: 09- Apr- 2020].

[13] G. Arfken, *Mathematical Methods for Physicists*, 3rd ed. Orlando: Academic Press, 1985, pp. 198-200.

[14] "Aircraft Rotations", *Grc.nasa.gov*, 2015. [Online]. Available: https://www.grc.nasa.gov/WWW/K-12/airplane/rotations.html. [Accessed: 09- Apr- 2020].

[15] U. Technologies, "Unity - Manual: Colliders", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/Manual/CollidersOverview.html. [Accessed: 09- Apr- 2020].

[16] "Timeline overview | Timeline | 1.2.14", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.timeline@1.2/manual/tl_about.html. [Accessed: 09- Apr- 2020].

[17] *Forum.unity.com*, 2020. [Online]. Available: https://forum.unity.com/attachments/timelineexample-dark-png.218796/. [Accessed: 09- Apr- 2020].

[18] "Introduction to Lighting and Rendering - Unity Learn", *Unity Learn*, 2020. [Online]. Available: https://learn.unity.com/tutorial/introduction-to-lighting-and-rendering#. [Accessed: 09- Apr- 2020].

[19] *Connect-prd-cdn.unity.com*, 2020. [Online]. Available: https://connect-prd-cdn.unity.com/20190130/011b0c3c-be8e-4c71-b051-3ace77a37a87_lightmap.jpg. [Accessed: 09- Apr- 2020].

[20] "Precomputed Realtime GI (Global Illumination) - Unity Learn", *Unity Learn*, 2020. [Online]. Available: https://learn.unity.com/tutorial/precomputed-realtime-gi-global-illumination#5c7f8528edbc2a002053b559. [Accessed: 09- Apr- 2020].

[21] U. Technologies, "Unity - Manual: Types of light", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/Manual/Lighting.html. [Accessed: 09- Apr- 2020].

[22] U. Technologies, "Unity - Manual: Occlusion culling", Docs.unity3d.com, 2020. [Online]. Available: https://docs.unity3d.com/Manual/OcclusionCulling.html. [Accessed: 09- Apr- 2020].

[23] *Connect-prd-cdn.unity.com*, 2020. [Online]. Available: https://connect-prd-cdn.unity.com/20190522/learn/images/576b9969-c902-4761-b887-6d67412a2708_Stats_Fig01.PNG. [Accessed: 09- Apr- 2020].

[24] U. Technologies, "Unity - Manual: Order of Execution for Event Functions", *Docs.unity3d.com*, 2019. [Online]. Available: https://docs.unity3d.com/Manual/ExecutionOrder.html. [Accessed: 09- Apr- 2020].

[25] U. Technologies, "Unity - Manual: Brushes", *Docs.unity3d.com*, 2019. [Online]. Available: https://docs.unity3d.com/Manual/class-Brush.html. [Accessed: 09- Apr- 2020].

[26] U. Technologies, "Unity - Scripting API: Mathf.Clamp", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/ScriptReference/Mathf.Clamp.html. [Accessed: 09- Apr- 2020].

[27] U. Technologies, "Unity - Manual: Particle systems", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/Manual/ParticleSystems.html. [Accessed: 09- Apr- 2020].

[28] U. Technologies, "Unity - Scripting API: Collider.OnTriggerEnter(Collider)", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/ScriptReference/Collider.OnTriggerEnter.html. [Accessed: 09- Apr- 2020].

[29] U. Technologies, "Unity - Manual: Collision module", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/Manual/PartSysCollisionModule.html. [Accessed: 09- Apr- 2020].

[30] U. Technologies, "Unity - Scripting API: MonoBehaviour.OnParticleCollision (GameObject)", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnParticleCollision.html. [Accessed: 09- Apr- 2020].

[31] "Canvas | Unity UI | 1.0.0", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/UICanvas.html. [Accessed: 09- Apr- 2020].

[32] "Event System | Unity UI | 1.0.0", *Docs.unity3d.com*, 2020. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.ugui@1.0/manual/EventSystem.html. [Accessed: 09- Apr- 2020].

[33] U. Technologies, "Unity - Manual: Deferred Fog", *Docs.unity3d.com*, 2019. [Online]. Available: https://docs.unity3d.com/Manual/PostProcessing-Fog.html. [Accessed: 09- Apr- 2020].

Appendix 1

Dissertation Questionnaire

Hello,

My name is Constantinos Constantinou, I am a 4th year student and I am pursuing a Bachelor of Science degree with Honours in Computing Science at University of Stirling. I am currently writing my thesis about the "Development of a 3D Mobile Game", in particular to examine how and why the performance can vary from different android devices running a game I developed. In this respect, and as part of my dissertation project I am conducting a survey which includes general questions and questions about the user experience of the game developed running on users' device.

Please search the word "Fireaxxe" in the Google Play store and download the game "Hydra-Rail Shooter Game" or click the link below as it is required to play for a few minutes in order to answer some of the questions:

https://play.google.com/store/apps/details?id=com.coc00031.Hydra&hl=en

The questionnaire consists of 22 questions. All responses will be kept anonymous and no one will be identifiable in the research.

For further questions please contact me at coc00031@students.stir.ac.uk

Thank you very much for your time.

Yours Faithfully,

Constantinos Constantinou

*Required

Or scan to download the game:

Section 1: General Questions

1. What is your gender? *

○ Male

○ Female

○ Prefer not to say

2. Which of the following best describes your current age? *

○ Under 15

○ 16 – 20

○ 21 – 25

○ 26 – 30

○ 31+

3. How many hours per week (on average) do you play games? *

	None	2	4	6	8	10+
Hours	○	○	○	○	○	○

4. What is your favourite genre of game? (Choose one or more) *

☐ Action / Adventure

☐ Simulation

☐ Strategy

☐ Sports

☐ Puzzle

☐ Card

☐ FPS

☐ Racing

☐ MMORPG / MMO / RPG

☐ Indie

☐ Other: _____

5. How much money on average do you spend for games per year? *

○ Nothing

■ £1 - £25

○ £25 – £50

○ £50 – £75

○ £75 - £100

○ £100+

6. What platform do you mostly play games on? *

○ PC

○ Console

○ Mobile

○ Other: _____

7. Which of the following is the most important to you? *

○ Graphics

○ Story

○ Performance

8. Multiplayer is important to you? *

○ Yes

○ No

9. How often do you play games on your smartphone? *

○ Never

○ Rarely (can't remember the last time)

○ Sometimes (not regularly)

○ Often (weekly)

○ Daily

10. What do you like the most about gaming on smartphones? *

○ Portability

○ Low Cost / Free to Play

○ Controls

○ Augmented Reality / Virtual Reality

○ "Short Play Session" Games

○ Best way to spend time when you get bored

○ Other: _____

11. What do you hate the most about gaming on smartphones? *

○ Battery Drain

○ Required of Internet to play (mostly)

○ Small Screen

○ Very addictive

○ Need to own a good smartphone for playing heavy games

○ Advertisements

○ Other: _____

Section 2: Questions Regarding the Project

Please answer the following questions after you play the game couple of times.

Please find below the link to the game:

https://play.google.com/store/apps/details?id=com.coc00031.Hydra&hl=en

12. Where did you play the game? (model of the smartphone) *

_____

13. Rate the game on a scale of 1 (Poor) to 5 (Excellent) *

	1 Poor	2	3	4	5 Excellent
Performance	○	○	○	○	○
Quality	○	○	○	○	○
Experience	○	○	○	○	○
Controls	○	○	○	○	○

14. Do you think this game is suitable for mobile devices? If no, why? *

○ Yes

○ Other: _____

15. How can you describe the difficulty of the game? *

	1	2	3	4	5	
Very Easy	○	○	○	○	○	Very Hard

16. How much time did you spend playing the game? *

○ 0 – 30 minutes

○ 30 – 60 minutes

○ 60 – 90 minutes

○ 90+ minutes

17. What was your highest score? *

○ 0 – 250

○ 250 – 750

○ 750 – 1000

○ 1000+

18. How many times did you die playing the game? *

○ 0 – 5

○ 5 – 10

○ 10 – 15

○ 15+

19. Have you found any bugs? *

○ No

○ Other: _____

20. What did you like the most about the game? *

○ Graphics

○ Performance

○ Gameplay

○ Audio and Sound FX

○ Other: _____

21. What did you dislike the most about the game? *

○ Controls

○ Installation Size

○ Lack of Features / Levels

○ Difficulty Level: Very Hard

○ Other: _____

22. If you were a Game Development, what would you change and / or suggest for improvement?

_____

_____

_____

_____