Application of Unity Netcode for GameObjects in the production of an online multiplayer fighting game in 3D

Aurelio José Trigueros Miravalls

Final Degree Work
Bachelor's Degree in
Video Game Design and Development
Universitat Jaume I

April 2, 2024

Supervised by: José Vicente Martí Avilés

For those who supported me all these years with infinite patience, I appreciate it from the bottom of my heart.

Acknowledgments

I would like to thank the friends I made during my studies and with which I have shared unforgettable moments and experiences that have made me the person I am now. To Franc for motivating me to try to be a better programmer every day and to María for making me see that I can do the things I impulsively set out to do, even if they are sometimes too optimistic.

To my flatmates and close friends, who participated in the creation process helping me to choose design issues, giving me feedback and support at all times. I would also like to thank some of my professors, who have made me see how interesting and deep the field of videogames is and from whom I have learnt many things. I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.

To the companies that have motivated me to follow this path as a developer and to think about making a living from this, such as From Software, Team Cherry or Extremely OK Games, among many others. Their games have made me see what is achieved by working with love for your own work.

Last, but most important of all, I would like to thank my family. Without them I wouldn't be here and I always keep that in mind. Thank you for trusting me all these years and for understanding my vocation with pride. Especially my mother, who is the best person in the world, thank you for everything.

Abstract

Video games are, without a doubt, an endless source of memorable experiences. Multiplayer video games, in my experience, have been the purveyors of many of these moments. Sharing those moments with friends allows you to have a good time with them. It has always been a personal challenge for me to one day make an online game because of what I owe to these types of games.

This project consists of developing a fighting video game with 3D graphics in **Unity** [31] using its new native package, **Netcode for GameObjects** [22]. The game uses **matchmaking** [17] to connect players who are looking for an opponent at the same time and uses a **dedicated server** [15] to increase security. Additionally, the movement is **client-authoritative** [28] to preserve fluency. It is also interesting to implement **local multiplayer** [1] in this genre. The game is intended to be adapted at some point to the **Rollback Netcode** [7] method, an efficient way used by professional games to reduce the impact of lag, although it is costly to implement due to the use of AI and customization of the engine's **render pipeline** [29]. This report will summarize both the creative process of the game itself and the technical requirements that had to be researched to achieve it.

Keywords: Unity, Netcode for GameObjects, Fighting video game, Matchmaking, Dedicated server, Client-Authoritative, Local multiplayer, Rollback netcode, Render Pipeline.

Contents

CHAPTER 1

INTRODUCTION

Contents

This chapter shows how did the original idea of the work started, the main goals to achieve and the things that I already knew before looking deeper in the topic.

1.1 Work Motivation

The idea for the game originated from a conversation I had with my friend and classmate, Adrià. We both thought it would be interesting to create a game with a black magic and African voodoo mythology atmosphere. Inspired by this idea, I further developed a **Game Design Document** (GDD) as part of the *Conceptual Design for Videogames* course (VJ1222). To my delight, my final paper received the highest grade from *Professor Emilio Saéz*.

Throughout my life, I have always been an avid multiplayer game player, and I have been particularly intrigued by the complexities involved in creating fighting games, especially those designed for competitive play. I didn't want to undertake an easy project for my **TFG**; I wanted something that I could proudly showcase to prospective employers.

When I discovered that the highest-rated fighting games in the community utilized a technique called **Rollback Netcode** to enhance the game's responsiveness, I saw this

as an opportunity to delve into the challenging world of netcode and investigate its intricacies. While it may have been an ambitious undertaking, I was determined to prove myself and explore this new subject.

1.2 Objectives

As stated by ***Dan Fornace*** in his article [9], the keys to a good fighting game lie in its fluidity and readability. The objectives of this work are as follows:

- Create a game that is **satisfying to play** and can be competitive at the same time, striking a good balance between balance and fun. Prioritize readability in animations and effects. The focus will be on providing clear visual and sound cues that allow players to react appropriately, rather than overwhelming animations with excessive visual effects that could lead to confusion. For instance, it is important for players to clearly recognize when a character is taking damage during the fight.
- Maintain **fluidity** when playing online by implementing **Rollback Netcode**, a highly regarded technique in the fighting game community. This method is crucial for reducing the impact of high latency in international connections. Players should experience responsive input reactions and feel that the outcome of a match is determined primarily by skill, with an element of luck to make it more enjoyable.
- Implement a **robust multiplayer foundation** that is easily modifiable and allows for the straightforward addition of new features. The game will utilize a **dedicated-server** structure, where both clients send their character inputs to enable the other client to recreate the behavior of the opponent on their own machine.
- Achieve an **appealing art style** with a dark yet comic atmosphere, setting it apart from other games in the genre. The game's graphical user interface (GUI) and overall art should maintain consistency to create a visually appealing and cohesive atmosphere. Characters should be visually distinct from the background, enabling players to easily locate their own character.

1.3 Environment and Initial State

It was necessary to have some experience with **Unity** and a little knowledge about graphics to achieve an acceptable result in this matter.

There were some tutorials that helped out with the netcode implementation, specially with **Unity Netcode for GameObjects** [22], but there is few information for a rollback implementation, in general. But there was even fewer information for this implementation in Unity. It seems to be a new topic and people have not tested it too much.

The modeling skills were enough to model some environment inanimated objects, but they weren't capable to model and rig characters in a professional way, and some time would be destined to learning more about this.

Finally, at least the experience with shaders was quite enough to make the desired visuals that the game would have.

CHAPTER 2

Planning and resources evaluation

Contents

This chapter will explain where the hours spent have been allocated, the planning that was proposed at the beginning compared to what was actually followed due to various technical complications.

2.1 Planning

The past experience in making games helped in estimating the appropriate hours for each section involved in this creative and technical process. However, the lack of experience in making fighting games and multiplayer games necessitated a change in this planning, as well as a reduction in some parts that required extensive post-TFG work. The goal was to create a functional multiplayer game with appealing mechanics and visuals.

The estimated time was divided into the following aspects: game design, technical research, 3D models and animations, programming dedicated to mechanics and physics, graphics-oriented programming, sound design and implementation, and writing this dissertation.

More hours were used than the initially established time frame, totaling approximately 325 hours. In Figure 2.1, it shows the initial distribution of hours and how it evolved throughout the project (see Figure 2.2). It can be observed that, in the end,

more time was spent on research and programming compared to everything else.

2.2 Resource Evaluation

This section will list the different resources, both hardware and software, that have been used to carry out this work.

Software used:
- **Unity 2021.3.17:** more stable version with new multiplayer system, being this the engine of the game.
- **Blender:** used to make the 3D models.
- **Visual Studio 2022**: used to edit the C-Sharp scripts that compose the main code.
- **Krita:** used for character concepts and user interface design.
- **OverLeaf:** used to write this memory.
- **Trello:** helped with task planning and bug fixing.
- **Lucid.app:** for making the game flowchart.

In terms of hardware:
- OMEN by HP Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz 3.70 GHz
- 32 GB RAM
- NVIDIA GeForce 1080Ti
- Wacom Intuos S graphics tablet
- Razer Kraken X Headset
- XBOX 360 Gamepad

Activity	**Percentage of time (of 300 hours stipulated)**
Investigation (About fighter games and netcode, in most part)	5%
Game Design	5%
Modeling and GUI design	15%
Coding mechanics, animations and options	40%
VFX	20%
SFX and music	5%
Writing the memory	5%

Figure 2.1: Initial Planification (canva.com)

Activity	Percentage of time (of 325 hours)
Investigation (About fighter games and netcode, in most part)	10%
Game Design	5%
Modeling and GUI design	10%
Coding mechanics, animations and options	50%
VFX	15%
SFX and music	5%
Writing the memory	5%

Figure 2.2: Final Hour Destination (canva.com)

CHAPTER 3

System Analysis and Design

Contents

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design.

3.1 Requirement Analysis

It is important in a project of considerable dimensions to take into account the indispensable aspects that define it, both functional and non-functional. These are listed below to clarify the main objectives of the work.

3.1.1 Functional Requirements

List of functional requirements:

- The user can **move the character** with ease as well as jump and cancel jump.
- The user can **attack his opponent** with a fast attack or a slow charged attack.
- The user can **block** and counter attacks at the right moment.

- The user can **choose between local multiplayer, online multiplayer and training mode**.
- The user can **choose between two characters** in a selection screen.
- The user can **pause** the game if not in an online mode.
- The user can change the **volume** of the game.
- The user can **fast forward** to his opponent with a skill.
- The user can use his character's own skills in exchange for **magic**.
- The user can **throw his opponent** off the stage to win the fight.
- The user can **return** to the title screen and exit the game.

3.1.2 Non-functional Requirements

List of non-functional requirements:

- The game will have **local and online** multiplayer.
- The game will only be available on **PC**, although it will be crossplay oriented in the future.
- The game will be in **3D**.
- The controls will be simpler than a 2D fighting game but not very easy, because players expect a learning curve.
- The frames that occupy the **animations** will be very rigorous, always fulfilling some frames of preparation, action and recovery in each one of them.
- The interface will have a lighthearted and **comic look**, as well as an aesthetic of **patches** for clothes and seams.
- The graphics will be comic book style and will have a violent as well as comical feel.
- The sounds used will be unrealistic and will enhance the important actions that occur.
- The game will be playable with **both controller and keyboard**.
- The game will have a **matchmaking** system to find matches with opponents of similar level.
- The game will use **dedicated servers** to make cheating more difficult.

3.2 System Design

Attached is the flowchart of the game at the moment (see Figure 3.1)

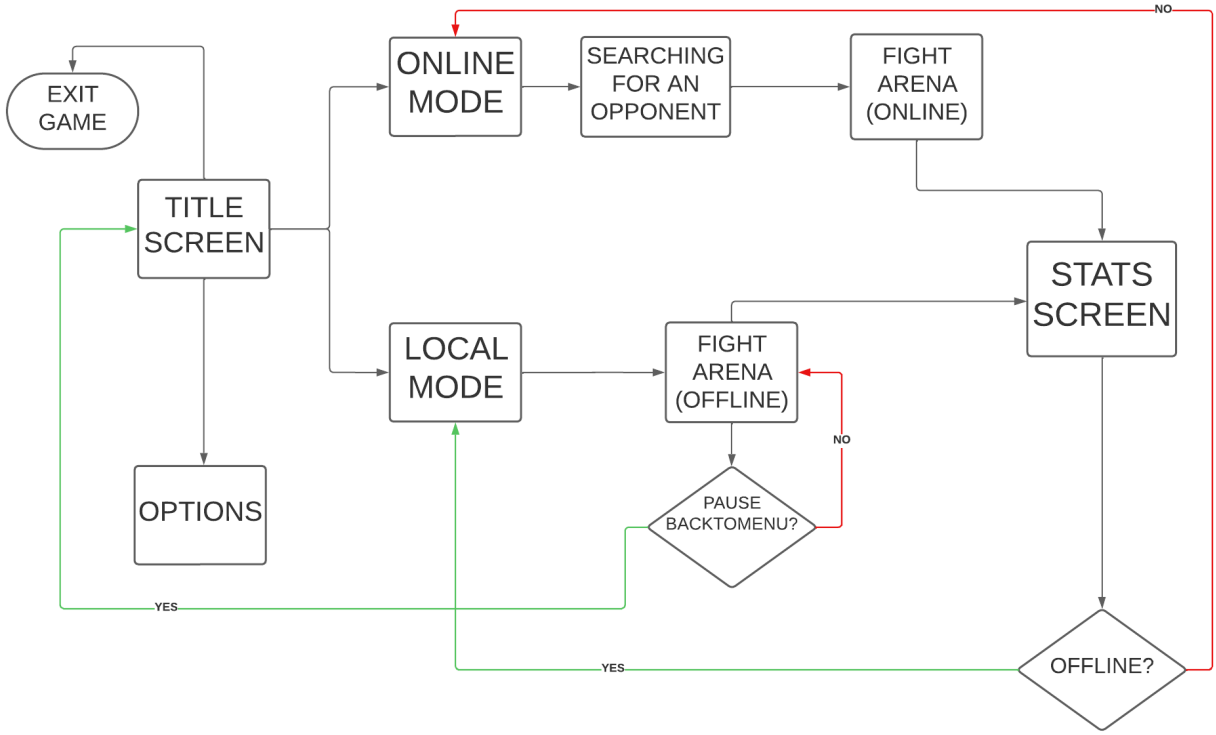

Figure 3.1: Flowchart of the game (lucid.app)

3.3 System Architecture

The game is designed to run on any mid-range computer, and can at least ensure the 60 *FPS* (Frames Per Second) [14] that fighting games need to remain smooth and reactive to inputs. For the development, the most stable version of Unity's Netcode for GameObjects, version **2021.3.17**, was used.

These are the version system requirements for the editor [32]:
- Windows 7 (SP1+), Windows 10 and Windows 11, 64-bit versions only.
- CPU: X64 architecture with SSE2 instruction set support
- Graphics API: DX10, DX11, and DX12-capable GPUs

3.4 Interface Design

The user interface should have a **cartoon look**, cheerful and comical appearance. The overall look should reside in the theme of stitching and dolls, therefore the interface is based on **patches overlapping** everything else, with fabric texture (see Figure 3.2).

At the same time, the buttons will contain colloquial language with which the functionality of the game will be understood. All language used will end in an exclamation mark, as will the title of the game, as a way of differentiating the interface from that of other games. Sound feedback will be given to the player when pressing buttons, as well as visible evidence of which button is currently selected. The pointer will be hidden at all times as the pointer position is not used at any point in the game.

Figure 3.2: Cat patch (left) and band aid (right) patches. Pinterest.

Chapter 4

Game Design

Contents

This section will look at the design of the game, both mechanically and technically, as well as artistic decisions.

4.1 Game Summary

- **Title:** Voodoo Wars!
- **Platform:** PC
- **Genres:** Fighting Game, Action, Online
- **Artistic style:** 3D models, cartoon style, violent atmosphere
- **Similar games:** "*Super Smash Bros Ultimate*" [12],"*Brawlhalla*" [4] or "*Naruto: Ultimate Ninja Storm*" [19]

4.1.1 Setting

The game recreates a fictional sport with a mythological religious basis, in which people fight for the favor of a god. Only the best at the contact sport of ***Voodoo Wars!*** will reach paradise in this universe. Therefore, believers in this faith prepare lethal voodoo dolls to fight against other dolls, inspired by today's robot fights but with a macabre twist. This drive to be the best is one of the core elements of the game, which promotes competitiveness in the online mode. In fact, the best real game player of the year will have the opportunity to participate in designing a character of their choice.

As a sport, it features a boxing-style announcer and a stadium where the crowd can be heard roaring with every direct hit.

Figure 4.1: Concept art made with IA

4.1.2 Target audience

The target audience for this game consists of teenagers and young adults between 16 and 30 years old, usually male, which is the most common gender in fighting games. It seeks both people who want a game to play with their friends and competitive people who have no ceiling when it comes to improving their skills.

4.2 Gameplay

The opposing dummy will be controlled by another person, both locally and online. The implementation of an AI using behaviour trees is intended for future work (see Figure 4.2).

Figure 4.2: Fighting between two characters

The combat system is the core of all gameplay. The main objective is to **knock the opponent out of the ring**, thereby eliminating one of his chances of rebirth. The one who runs out of rebirth opportunities will die permanently, with the dummy left in the ring becoming the winner.

The blows have different thrust forces, plus they depend on a global variable that is different from any other fighting game: there is an on-screen representation of a scale that simulates **who the god prefers to win**. The less favour the god provides, the more the opponent will have. This will boost hits with more favour, making the player with less favour fly further more easily.

Hits, however, will have the possibility of **being blocked** or even countered, stunning the opponent and gaining the advantage of anticipation. Something important is that characters can only do one action at a time, and this can be interrupted if they are hit (see Figure 4.3).

In addition to all this, there is a kind of mana called ***Voodoo magic***, which is a resource used by some skills that characters cast with powerful results. This resource can be obtained either by landing blows or by collecting energy pellets that appear in

Figure 4.3: Blocking example

the ring from time to time. The fight will always start with an energy ball in the middle of the characters, encouraging aggression and action at all times, as whoever gets close to the other character first will be rewarded first.

The ring consists of a **large circular island** surrounded by smaller islands orbiting it. These islands can be dangerous to access, but will contain magic spheres that will help to charge special attacks sooner (see Figure 4.4).

There will be a time limit for combat, and the player with the fewest lives will lose in the event of a draw. In the event of a draw, the fight will enter sudden death mode, where the favour between the two opponents swings wildly and either can win with a good hit.

The aim is to make the combat system frenetic and entertaining, fluid and easy to learn. However, the real goal is to add even more mechanics such as combos and jumping attacks that steepen the learning curve to make the game more difficult to master.

4.3 Mechanics

The mechanics will be the same for all fights and will only differ in the different weapons and abilities that the various characters possess. Although the game can be played with either keyboard and mouse or controller, it is strongly recommended to play with a controller because it is much more comfortable for fighting games.

Figure 4.4: Fighting between two characters

- **Movement:** WASD keys in PC, Left Joystick in gamepad.
- **Fast Attack:** Left Click Mouse in PC, Left action button in gamepad
- **Strong Attack:** Right Click Mouse in PC, Right action button in gamepad
- **Block:** Control key in PC, Right Trigger in gamepad
- **Parry:** Q key in PC, Right Shoulder in gamepad
- **Jump:** Spacebar in PC, Down action button in gamepad
- **Dash:** Shift key in PC, Left Joystick in gamepad
- **Voodoo Power:** E key in PC, Left Shoulder in gamepad
- **Taunt:** T key in PC, Up arrow in gamepad

4.4 Game Art

4.4.1 Visual References

Visual references were sought through both *Pinterest* searches and drawing *AI* (Artificial Intelligence) description results (see Figure 4.6), all in an effort to come up with a charismatic art style. The art style draws from both traditional comics and games like ***Darkest Dungeon*** [18], with a much darker style and a strong black line. It had to derive from the Voodoo religion, so symbols of death, masks or other cultural references to this are often used.
The playful aesthetic and the ring was based on ***Brawlhalla***. See an example of this aesthetic in figure 4.5.
The image of the voodoo doll holding a nail was the key visual reference and the one that originated the idea of using a similar aesthetic throughout the game (see Figure 4.6).

The **3D Low Poly** [13] style goes well with a less serious and realistic gameplay and saves a lot of performance in this aspect.

Figure 4.5: Brawlhalla

4.4.2 VFX

VFX (Visual Effects) are very important in fighting games, accompanying all incoming inputs and making it easier to visualise what the opponent is doing. Most of them have been taken from the *Unity Asset Store* [23], but modified within the engine to suit the

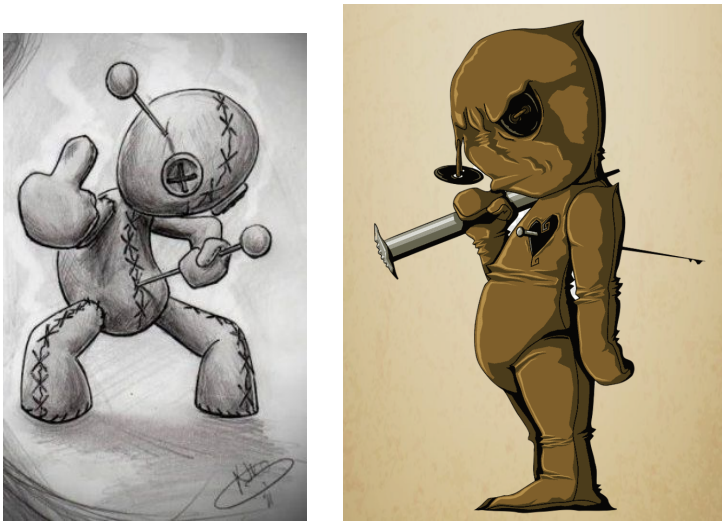

Figure 4.6: Pinterest references and original idea.

game and its appearance. They are accompanied by controller vibration if it is played with one, further increasing the player's awareness of what's going on. There is a slash effect for the attacks, a magic shield covering the character when it is blocking and a shining when a player is ready to counter attack, among other examples (See Figure 4.7)

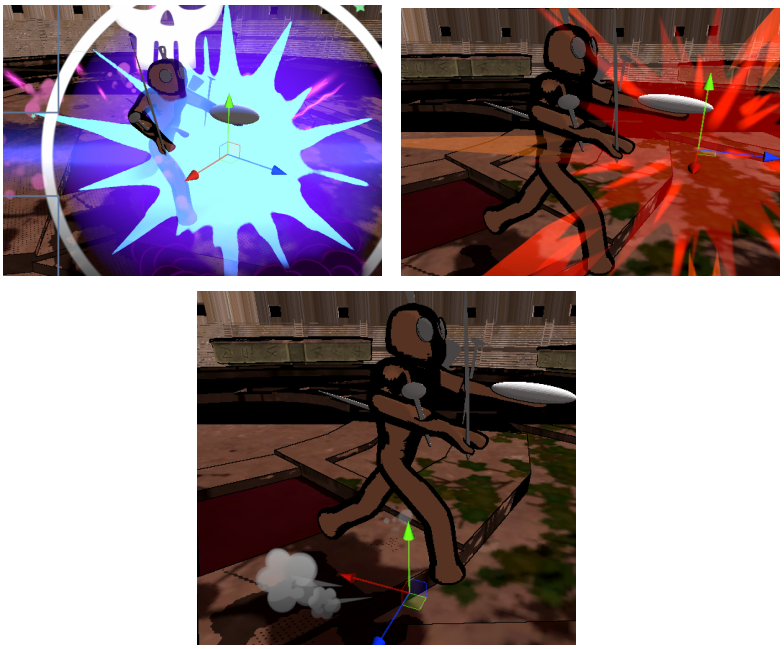

Figure 4.7: Blocking (left), getting hit (right) and run (bottom) VFX.

4.4.3 Sound Design

The sound of the game tries to be as reactive as possible to what is on screen, reinforcing important actions that occur. During the fight, soft combat music plays in the background. Actions such as punching, counterattacking, using the shield, sliding towards the opponent... Everything has important sound feedback to give the opponent a clue to react.

CHAPTER 5

Work Development and Results

Contents

Here it will presented the concluded results of the research process that was done first, both on the fighting game genre itself and on the technical challenge of making a multiplayer game. It will also discuss the work done during these months in the different areas of development.

5.1 Related Research

Before development started as such, there was a period of time where getting information about all the technical complications and best practices in fighting games was key. Especially because it was important to take everything that others were doing very well and bring it to a type of fighting game that was more neglected by fans: **3D fighting games**.

5.1.1 Genre Research

The first thing that was investigated was the main inspiration for this TFG, a famous game within this genre, ***Naruto: Ninja Storm Revolution 3*** (see Figure 5.1). Looking at the user reviews, there are several aspects that are liked among these players. Above all, they ask that if it is a 3D fighting game, the space should be used to add **verticality** and gameplay within the map. ***Dragon Ball Budokai Tenakichi 3*** [2]

(see Figure 5.2) was taken as an example of this, which took advantage of its 3D gameplay to recreate impressive fights in the air and on the ground.

Figure 5.1: Naruto Ninja Storm Revolution

Figure 5.2: Dragon Ball Tenkaichi 3

The research continued, and after a while the first key ideas of a well-received fighting game among users were synthesised: **fluidity, verticality, taking advantage of the scenery, easily executable combos and accountability to the player at all times**. If anyone should win in a fighting game, it is the player who has made the best use of his or her skills to take advantage, without unfair moments in the game that break the experience of participating in a real sport.

Physics was also an important issue, and the intention of the work was not to make a 3D ***Smash Bros*** [3] (see Figure 5.3), but to find the good things about this game and apply them to one with different mechanics. And it turns out that what ***Smash Bros*** does very well is physics [5]. They delved deeper into the subject and there are little technicalities that without them fighting games wouldn't be what they are now. Stun times, cancelling animations, taking control away from the player when thrown.... Fighting games are quite technical at bottom.

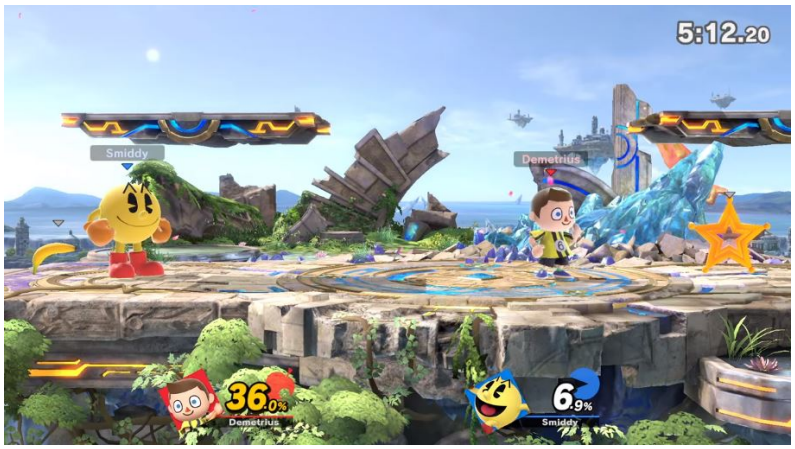

Figure 5.3: Smash Bros Ultimate

The research helped to approach the code architecture in a different way. The characters first had to have a **Finite State Machine** [21] to ensure that they are always in a state and cannot leave it until the animation ends. Each animation is divided into three distinct time slots: **preparation, action and recovery** [8]. When an input is pressed, the action does not occur until that point in the animation is reached. While in an animation and another input is pressed, it is possible to save it in an input buffer to execute it right after this animation ends. This is something that many acclaimed fighting games do and is something that is standardised and rare not to have (see Figure 5.4)

It was also important for the execution to learn the difference between **hitbox, hurtbox and collider** in this kind of games. The hitbox is the collision area (trigger) that an **attack** occupies. The hurtbox is the collision area (also trigger) of the **character** itself. If the hitbox collides with another character's hurtbox, they receive the corresponding effect. The collider (a collision area that each physical model has) however only prevents players from overlapping and is active all the time. [6]

Finally, as these are games where attacks last for specific frames and strategies depend on this, it is important to normalise the frame rate for all players to exactly 60 fps.

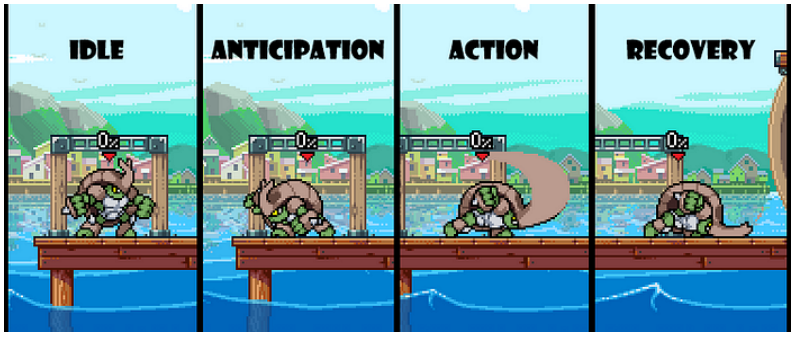

Figure 5.4: Animation explanation

And this will also be important for synchronising players, as it will be seen in the next section. [14] [9]

5.1.2 Multiplayer Research

Making a fighting game was no more than a fairly manageable challenge at the local level. However, the focus of the work had to be on making a multiplayer game as well as a multiplayer game in this genre. To this end, an easy-to-implement multiplayer solution for Unity was investigated, namely **Netcode for Game Objects** [22], their recently released native solution. However, after research into the methods used by online fighting games to compensate for poor connections, it was decided that **Rollback Netcode** would be implemented, a method that consists of not rendering one player's frame without having both player's frames. If a player's frame is missing due to a packet loss or a delay, the rollback simulates by means of artificial intelligence which is the most likely move the person would make. If that input arrives late and does not correspond to the simulated one, it goes back in the simulation, so to speak, and is rendered with the player's input. It is a way of taking advantage of the fact that the game can run at many fps, but only 60 fps are used for rendering, so the remaining fps are used to make corrections to the ones that really matter. As it may seem, this is an expensive technique to implement in Unity, not to say that there is almost nothing researched on the subject. However, even if it was not in this work due to lack of time, the game had to be oriented to take this multiplayer model at some point.

One of the requirements of Rollback, and certainly the topic on which research has been most extensive, has been how to make the game a **fully deterministic model**. A deterministic model consists of a system that receives an input and the same thing always happens when given the same outputs. This is very important, because then the rollback would only have to **send inputs** from one client to another, so that each player receives the input from the other and simulates it at the same time. The problem is that it is harder than it seems to ensure that something is deterministic. It turns out

that **computers are not deterministic** with each other, and this is mostly due to the **floating point** problem that different types of processors have. If one computer does not have the same processor as another, no one cannot be sure that after a while it will not be out of phase with the other player because of the accumulated error in processing floating commas. One processor may see 0.1 as 0.10000003 and the other as 0.10000004, and this accumulated difference over time causes the two players to become out of sync, causing chaos and absolute lack of control of the game.

Unity is going to provide a native solution for this problem, but they say they will release it sometime in 2023. Therefore, a way had to be found to use the benefits of **Netcode for Game Objects** without using the **Rigidbody** system, which uses normal floating numbers. One of the solutions seems to be to use a physics system outside Unity with **FixedFloatPoints** [10], which would ensure a deterministic system. In the meantime, multiplayer should be secured first and the game should be oriented in that direction, little by little.

Finally, thanks to the services provided by the new **Unity** people, dedicated servers, matchmaking and instant rematch would be available completely free of charge. This made this task much easier and it is thanks to this that online was implemented without much trouble. Also, a voice chat service would be a very good idea, because the essence of fighting games is to be able to comment the game while you play, as if you were next to the player. This would be optional for players and easy to implement with Unity's free ***Vivox*** [35] service.

5.2 Work Development

This section will explain the different aspects that have been developed in this final work: programming, 3D modelling and animation. The bulk of the time was spent in these three areas, and time would be saved in the VFX, SFX and textures. Some time would be used for learning about character modelling techniques, as well as the multiplayer library documentation.

5.2.1 Coding

The code developed in the work can be divided into two parts: the one that controls the **mechanics and behaviour** in general (the vast majority, written in C-Sharp) and a **custom lighting shader** for the game (written in ***HLSL*** (High Level Shader Language) [25] and **Unity's *ShaderGraph***).

As an organising and scripting entity, we can find the *GameManager*, which is present in every scene of the game. It is mainly in charge of changing the game mode, coordinating local multiplayer, changing global game variables and controller vibration.

Change Play Mode

```
public void ChangePlayMode(PlayMode playMode)
{
    activePlayMode = playMode;

    if (activePlayMode == PlayMode.ONLINE)
    {
        GetComponent<PlayerInputManager>().enabled = false;
        networkManager.SetActive(true);
        serverManager.SetActive(true);
        approvalManager.SetActive(true);
        SceneManager.LoadScene("OnlineRing");

    }
    else if (activePlayMode == PlayMode.OFFLINE)
    {
        GetComponent<PlayerInputManager>().enabled = true;
        networkManager.SetActive(false);
        serverManager.SetActive(false);
        approvalManager.SetActive(false);
        GetComponent<PlayerInputManager>().playerPrefab = charactersArrayOffline[0];
        SceneManager.LoadScene("OfflineRing");
    }
}
```

This function is responsible, with the help of the Input System Manager, for assigning the various controls to the individual prefabs so that each player can control her o his character.

On Player Joined

```
 void OnPlayerJoined(PlayerInput playerInput)
{
    if (playerInput.playerIndex == 0)
    {
        playerOne = GameObject.FindGameObjectWithTag("Player");
        playerOne.GetComponentInChildren<PlayerController>().playerOne = true;
        playerOne.tag = "PlayerOne";
        GetComponent<PlayerInputManager>().playerPrefab = charactersArrayOffline[playerTwoCharacterId];
        try
        {
            InputSystem.SetDeviceUsage(Gamepad.all[0], "Player1");
            playerTwoGamepad = InputSystem.GetDevice<Gamepad>(new InternedString("Player2"));
        }
        catch
        {
            InputSystem.SetDeviceUsage(Keyboard.current, "Player1");
        }
    }
    else
    {
```

```
            playerTwo = GameObject.FindGameObjectWithTag("Player");
            playerTwo.GetComponentInChildren<PlayerController>().playerOne = false;
            playerTwo.tag = "PlayerTwo";
            playerOne.GetComponentInChildren<PlayerController>().enemy = playerTwo;
            playerTwo.GetComponentInChildren<PlayerController>().enemy = playerOne;
            playerOne.GetComponentInChildren<PlayerController>().enemyController
            = playerTwo.GetComponentInChildren<PlayerController>();
            playerTwo.GetComponentInChildren<PlayerController>().enemyController
            = playerOne.GetComponentInChildren<PlayerController>();

            try
            {
                InputSystem.SetDeviceUsage(Gamepad.all[1], "Player2");
                playerTwoGamepad = InputSystem.GetDevice<Gamepad>(new InternedString("Player2"));
            }
            catch
            {
                InputSystem.SetDeviceUsage(Keyboard.current, "Player2");
            }
            mainCam.Follow = playerOne.transform;
            mainCam.LookAt = playerTwo.transform;
        }

    }
```

This changes the speed of the controller motor to cause vibration in the controller, only if the controller has a controller assigned to it and if it is therefore possible.

RumblePulse

```
     public void RumblePulse(float lowFrequency, float highFrequency, float duration, bool playerOne)
    {
        if (playerOne)
        {
            if (playerOneGamepad != null)
            {
                playerOneGamepad.SetMotorSpeeds(lowFrequency, highFrequency);
                StartCoroutine(StopRumble(duration, playerOne));
            }

        }
        else
        {
            if (playerTwoGamepad != null)
            {
                playerTwoGamepad.SetMotorSpeeds(lowFrequency, highFrequency);
                StartCoroutine(StopRumble(duration, playerOne));
            }

        }
    }
```

Finally, the *GameManager* deals in the Update method with the behaviour of the camera, which switches from Target to the nearest player at all times, and the Look at to the opposite player. See 5.5 to see more details of the **Cinemachine**[16] configuration.

CameraBehaviour

```
    if (Vector3.Distance(playerOne.transform.position, mainCam.gameObject.transform.position)
    < Vector3.Distance(playerTwo.transform.position, mainCam.gameObject.transform.position))
           {
               if (mainCam.Follow == playerTwo.transform)
               {
                   mainCam.Follow = playerOne.transform;
                   mainCam.LookAt = playerTwo.transform;
               }
           }
           else
           {
               if (mainCam.Follow == playerOne.transform)
               {
                   mainCam.Follow = playerTwo.transform;
                   mainCam.LookAt = playerOne.transform;
               }
           }
```

As for multiplayer, we can find 3 key scripts: *ConnectionApprovalHandler*, *MatchmakerClient* and *ServerStartUp*.

ConnectionApprovalHandler handles user authentication, which in the case of the project has been left as anonymous ID to make things easier, although it is not expensive to implement the API of each authentication service such as Google or Steam. This script only allows players to connect until the lobby maximum is reached, in this case two.

ApprovalCheck

```
   private void ApprovalCheck(NetworkManager.ConnectionApprovalRequest request, NetworkManager.ConnectionApprovalRes
   {
       response.Approved = true;
       response.CreatePlayerObject = true;
       response.PlayerPrefabHash = null;
       if (NetworkManager.Singleton.ConnectedClients.Count >= maxPlayers)
       {
           response.Approved = false;
       }
       response.Pending = false;
   }
```

ServerStartUp what it does in general is to assign an allocation when the desired number of players in a game is reached. First it checks if the run belongs to a server or a client. If it belongs to a server, then it starts the server functionality and waits for players until the game can be started. It assigns the given IP and port to each client.

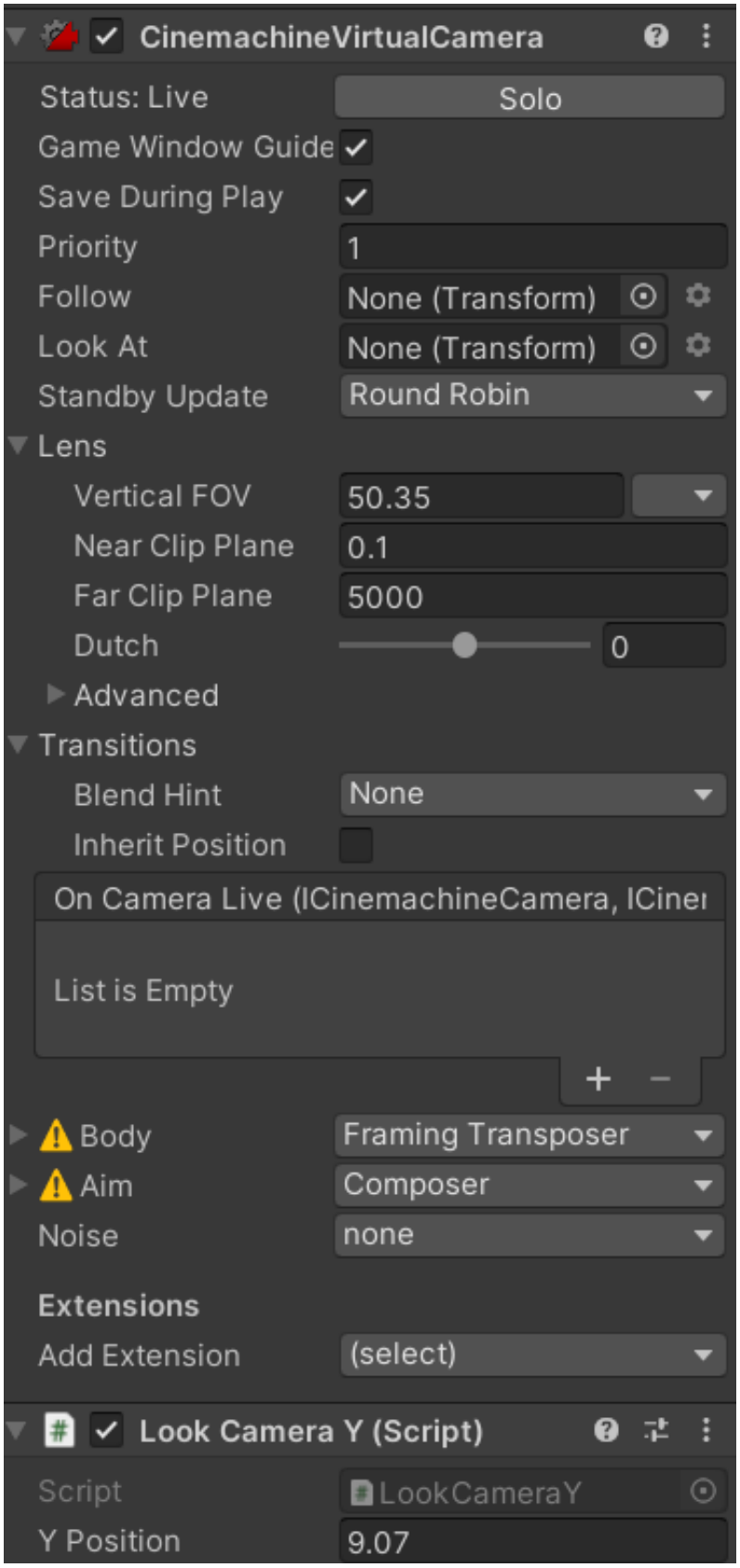

Figure 5.5: Camera Configuration

It also performs **backfill**, which means that the players do not have to search at the same time, but a player is assigned an allocation and waits there for another player to connect. This is the server configuration in **Unity Services** [20] (see 5.6) The build running on the Unity servers is a Linux build prepared to be a dedicated game server. It does not render graphics in order to make better use of the computer performance provided by Unity. These servers are always available and are only switched on when a new allocation is needed, in order to reduce the cost.

Start

```
1     private async void Start()
2    {
3        bool server = false;
4        var args = System.Environment.GetCommandLineArgs();
5        for (int i = 0; i < args.Length; i++)
6        {
7            if (args[i] == "-dedicatedServer")
8            {
9                server = true;
10           }
11           if (args[i] == "-port" && (i + 1) < args.Length)
12           {
13               serverPort = (ushort)int.Parse(args[i + 1]);
14           }
15
16           if (args[i] == "-ip" && (i + 1 < args.Length))
17           {
18               externalServerIP = args[i + 1];
19           }
20       }
21       if (server)
22       {
23           StartServer();
24           await StartServerServices();
25       }
26       else
27       {
28          ClientInstance?.Invoke();
29       }
30   }
```

Last but not least, *MatchmakerClient* is in charge of managing the ticket that is sent to the servers if it is a client, the ticket can be in four states: pending, accepted, failed or timeout. When two tickets are pending and meet, the server assigns them an allocation and they become accepted. The game therefore begins.

PollTicketStatus

```
1    private async void PollTicketStatus()
2    {
```

Server ID	IP:Port	Location	Fleet	Active build configuration	Status
59810933	35.228.179.128:9000	europe-north1-b	FleetTestVoodoo	Dev B build ServerLinuxTest	Available
59810942	35.228.179.128:9100	europe-north1-b	FleetTestVoodoo	Dev B build ServerLinuxTest	Available
59810951	35.228.179.128:9200	europe-north1-b	FleetTestVoodoo	Dev B build ServerLinuxTest	Available
59810957	35.228.179.128:9300	europe-north1-b	FleetTestVoodoo	Dev B build ServerLinuxTest	Available
59810960	35.228.179.128:9400	europe-north1-b	FleetTestVoodoo	Dev B build ServerLinuxTest	Available
59810963	35.228.179.128:9500	europe-north1-b	FleetTestVoodoo	Dev B build ServerLinuxTest	Available
59810972	35.228.179.128:9600	europe-north1-b	FleetTestVoodoo	Dev B build ServerLinuxTest	Available

Figure 5.6: Servers Available

```
3         MultiplayAssignment multiplayAssignment = null;
4         bool gotAssignment = false;
5         do
6         {
7             await Task.Delay(TimeSpan.FromSeconds(1f));
8             var ticketStatus = await MatchmakerService.Instance.GetTicketAsync(ticketId);
9             if (ticketStatus == null) continue;
10            if(ticketStatus.Type == typeof(MultiplayAssignment))
11            {
12                multiplayAssignment = ticketStatus.Value as MultiplayAssignment;
13            }
14
15            switch(multiplayAssignment.Status)
16            {
17                case StatusOptions.Found:
18                    gotAssignment = true;
19                    TicketAssigned(multiplayAssignment);
20                    matchmakingPanels.SetActive(false);
21                    fadePanel.SetActive(false);
22                    Debug.Log("Ticket␣found");
23                    break;
24                case StatusOptions.InProgress:
25                    Debug.Log("Waiting␣for␣ticket");
26                    break;
27                case StatusOptions.Failed:
28                    gotAssignment = true;
29                    Debug.Log("Failed␣to␣get␣ticket␣status");
30                    matchmakingPanels.SetActive(false);
31                    fadePanel.SetActive(false);
32                    break;
33                case StatusOptions.Timeout:
34                    gotAssignment = true;
35                    Debug.Log("Time␣out␣to␣get␣the␣ticket.");
36                    matchmakingPanels.SetActive(false);
37                    fadePanel.SetActive(false);
38                    break;
39                default:
40                    throw new InvalidOperationException();
```

```
41
42              }
43          } while (!gotAssignment);
44      }
```

If the ticket is assigned, then the script assigns a port and an IP to the client and start it.

TicketAssigned

```
1   private void TicketAssigned(MultiplayAssignment assignment)
2    {
3        NetworkManager.Singleton.GetComponent<UnityTransport>().SetConnectionData(assignment.Ip, (ushort)assignment.P
4        NetworkManager.Singleton.StartClient();
5    }
```

In terms of the overall code architecture of the game, the bulk of the lines are in the *PlayerController* and *NetworkController*, the script that holds all the characters in the game and controls all the character mechanics, behaviour, animations and particles. It is considered to eventually split this script into three separate scripts to further isolate the tasks it performs. **Unity's Input System** [27] (see Input System configuration in 5.7) is used to centralise the inputs and their calls, which are inside the **Player Controller**. This system is very versatile and can be used to adapt to any type of controller and add local multiplayer easily.

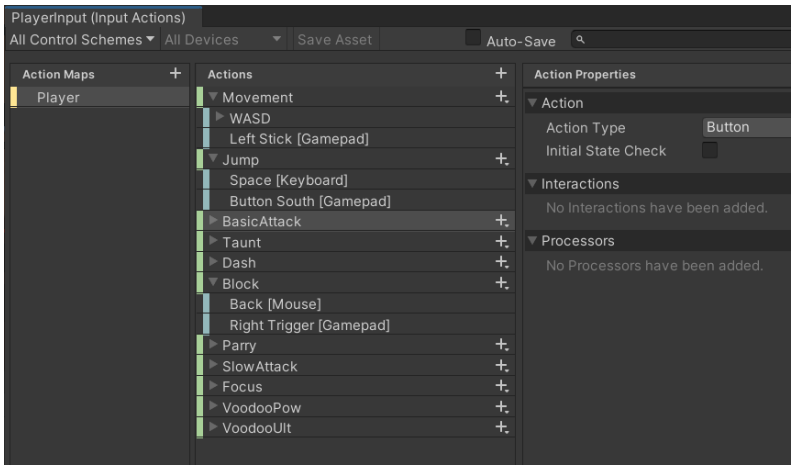

Figure 5.7: Player Input System

Inside this script is the **Finite State Machine**, which controls the states in which all the characters in the game can be, these are:

```
public enum PlayerState
{
```

```
        FAST_ATACK, SLOW_ATACK, BLOCK, PARRY, DASH, VOODOO_POW,
        MOVEMENT, IDLE, JOKE, HIT, STUNNED, KNOCKED_UP
    }
```

Except for *HIT*, *STUNNED* and *KNOCKEDUP*, which are reactions to receiving attacks, all the others are states that the player can access of his own free will through his controls. It contains an important variable called **currentState**, which contains the state the player is in at all times and cannot change to another state until the animation ends. Animations call **Unity events** [33] at certain frames, which are one-time functions. There are differences between the *PlayerController* and the *NetworkPlayerController* , which are mainly that the online script must always know whether it is the owner or not. IsOwner is a boolean that returns true or false depending on whether the script belongs to the local player or not. The functions that check the inputs do the action if the button is pressed or if the player is in a certain state when it is the other client. This way the behaviour of the other client is simulated on the client itself based on the state passed to it by the **RPC** call. [30] call, which we'll talk about later. Here is an example of how a script changes in an online game.

Taunting

```
1    if (IsOwner)
2        {
3            if (taunt && canChangeState && canTaunt)
4            {
5                jokeParticle.Play();
6                canChangeState = false;
7                canMove = false;
8                currentState = PlayerState.JOKE;
9                TransitionToStateServerRpc(currentState);
10               playerAnimator.SetBool("Taunting", true);
11               print("Taunt_Owner");
12               canTaunt = false;
13           }
14           else
15           {
16               if (!taunt)
17               {
18                   if (playerAnimator.GetBool("Taunting") == true)
19                   {
20                       playerAnimator.SetBool("Taunting", false);
21                   }
22
23                   if (!canTaunt) canTaunt = true;
24               }
25
26           }
27       }
28       else
29       {
```

```
30              if(currentState == PlayerState.JOKE)
31              {
32                  //DO THE SAME
33              }
```

As for RPC calls, it is also interesting to see how they act differently depending on whether it is a server or a client.

TransitionToStateServerRpc

```
1      public void TransitionToStateServerRpc(PlayerState newState)
2     {
3         if (!IsServer) return;
4         currentState = newState;
5         // Call client RPC to update the state on other clients
6         UpdateStateOnClientRpc(currentState);
7     }
```

While the client sends its status to the server, the server stores it on the respective other client.

UpdateStateOnClientRpc

```
1      public void TransitionToStateServerRpc(PlayerState newState)
2     {
3          if (IsOwner || IsServer) return;
4         print("Actualizo cliente id: ." + NetworkObjectId);
5         currentState = newState;
6     }
```

There is the function *TakeInputs()*, which contains all the methods of each input that are checked in the *Update()*. The jump and dash will be used as an example.

The jump for example is something that a lot of effort has been put into, because it can be cancelled in mid-air to quickly return to the ground, as well as control the height of the jump depending on how long you press its button.To cancel the jump, a downward force had to be applied at the moment when the Y velocity of the character was no longer positive and the lock input was pressed.
To achieve a time-dependent jump, gravity had to be removed from the character and this method had to be added, which added a base jump force and gradually a small force if the button was kept pressed.

FixJump

```
1         private void FixJump()
2     {
3         if (inAir)
4         {
```

```
            jumpTime += Time.deltaTime;
            if (jumped && jumpTime < jumpMaxTime)
            {
                playerRb.AddForce(Vector3.up * jumpForceContinuous, ForceMode.Impulse);
            }
            else
            {
                if(currentState != PlayerState.DASH) playerRb.useGravity = true;
            }
        }
    }
```

The functions that make up the dash are also worthy of attention, as they control the timing of the dash, cancel the animation when it collides with the opponent, can be done in the air, and keep you floating until you advance quickly. The dash consists of moving the character towards the opponent, but what is really interesting about this machine are the two corrutines it triggers, *DashDelay()* and *StopDash()*.
The first one orients the player towards the other and doesn't apply the force until later, to give the impression that it charges this power for a while.

DashDelay

```
      IEnumerator DashDelay()
    {
        Vector3 relativePos = enemy.transform.position - transform.position;
        transform.rotation = Quaternion.LookRotation(new Vector3(relativePos.x, 0, relativePos.z).normalized, Vector3.up);
        if (inAir)
        {
            playerRb.velocity = Vector3.zero;
            playerRb.useGravity = false;
        }

        yield return new WaitForSeconds(dashDelay);
        relativePos = enemy.transform.position - transform.position;
        playerRb.velocity = (relativePos.normalized * dashSpeed);
        dashParticle.Play();
        playerAudio.loop = false;
        playerAudio.clip = dashStartSound;
        playerAudio.Play();
        if (!isDashing)
        {
            StartCoroutine("StopDash");
            isDashing = true;
        }
    }
```

The dash can end in two ways: either you don't hit anything for a few seconds or you hit your opponent. For when you collide with your opponent, there is this routine that resets the dash early.

StopDash

```
    IEnumerator StopDash()
{
    yield return new WaitForSeconds(dashDuration);
    playerRb.velocity = Vector3.zero;
    canMove = true;
    canChangeState = true;
    StartCoroutine("DashReset");
    if (playerAnimator.GetBool("Dashing") == true)
    {
        playerAnimator.SetBool("Dashing", false);
    }
}
```

It is also important to note the damage-receiving function, which applies the necessary forces to the characters as well as state changes.

ReceiveDamage

```
    internal void ReceiveAttack()
{
    Vector3 enemyPos = enemy.transform.position - transform.position;
    transform.rotation = Quaternion.LookRotation(new Vector3(enemyPos.x, 0, enemyPos.z).normalized, Vector3.up);
    if (isParrying)
    {
        enemyController.currentState = PlayerState.STUNNED;
        if (enemyController.playerAnimator.GetBool("Stunned") == false)
        {
            enemyController.playerAnimator.SetBool("Stunned", true);
            parryStunnedParticle.Play();
            parryDoneParticle.Play();
            gameManager.RumblePulse(1.0f, 2.0f, 0.8f, playerOne);
            gameManager.RumblePulse(1.0f, 2.0f, 0.8f, !playerOne);

        }
    }
    else if(currentState == PlayerState.BLOCK)
    {
        if (playerAnimator.GetBool("BlockedHit") == false)
        {
            playerAnimator.SetBool("BlockedHit", true);
            enemyController.blockEffectParticle.Play();
        }
    }
    else
    {
        Vector3 relativePos = (transform.position - enemyController.gameObject.transform.position).normalized;
        if(enemyController.currentState == PlayerState.SLOW_ATACK)
        {
            currentState = PlayerState.HIT;

```

```
33                  if (playerAnimator.GetBool("Receiving") == false)
34                  {
35                      playerAnimator.SetBool("Receiving", true);
36                      enemyController.slowAtackImpactParticle.Play();
37                      playerAudio.clip = slowAttackImpactSound;
38                      playerAudio.loop = false;
39                      playerAudio.Play();
40                      gameManager.RumblePulse(1.0f, 2.0f, 0.8f, playerOne);
41
42                  }
43
44                  if (!playerOne)
45                  {
46                      gameManager.ChangeFavor(-strongAttackBaseFavorGain);
47                  }
48                  else
49                  {
50                      gameManager.ChangeFavor(strongAttackBaseFavorGain);
51                  }
52
53                  float totalDmgDone;
54
55                  if (playerOne)
56                  {
57                      if (gameManager.playerOneFavor > 50)
58                      {
59                          totalDmgDone = strongAttackBaseFavorGain + (strongAttackBaseFavorGain * (gameManager.playerOneFavor -
60                      }
61                      else
62                      {
63                          totalDmgDone = strongAttackBaseFavorGain - (strongAttackBaseFavorGain * (50 - gameManager.playerOneFa
64                      }
65
66
67                  }
68                  else
69                  {
70                      if (gameManager.playerOneFavor > 50)
71                      {
72                          totalDmgDone = strongAttackBaseFavorGain + (strongAttackBaseFavorGain * (gameManager.playerTwoFavor -
73                      }
74                      else
75                      {
76                          totalDmgDone = strongAttackBaseFavorGain - (strongAttackBaseFavorGain * (50 - gameManager.playerTwoFa
77                      }
78                  }
79
80                  enemyController.gameManager.ChangeMagic((int)totalDmgDone, !playerOne);
81
82
83                  relativePos.y = 1f;
84                  GetComponent<Rigidbody>().AddForce((relativePos * totalDmgDone * enemyController.currentChargedAttackForce) *
85              }
86              else if(enemyController.currentState == PlayerState.FAST_ATACK)
```

```
87              {
88                  currentState = PlayerState.HIT;
89                  if (playerAnimator.GetBool("Receiving") == false)
90                  {
91                      playerAnimator.SetBool("Receiving", true);
92                      enemyController.basicAtackImpactParticle.Play();
93                      playerAudio.clip = basicAttackImpactSound;
94                      playerAudio.loop = false;
95                      playerAudio.Play();
96                  }
97                  if (!playerOne)
98                  {
99                      gameManager.ChangeFavor(-basicAttackFavorGain);
100                 }
101                 else
102                 {
103                     gameManager.ChangeFavor(basicAttackFavorGain);
104                 }
105
106                 float totalDmgDone;
107                 if (playerOne)
108                 {
109                     if (gameManager.playerOneFavor > 50)
110                     {
111                         totalDmgDone = basicAttackFavorGain + (basicAttackFavorGain * (gameManager.playerOneFavor - 5
112                     }
113                     else
114                     {
115                         totalDmgDone = basicAttackFavorGain - (basicAttackFavorGain * (50 - gameManager.playerOneFavo
116                     }
117                 }
118                 else
119                 {
120                     if (gameManager.playerOneFavor > 50)
121                     {
122                         totalDmgDone = basicAttackFavorGain + (strongAttackBaseFavorGain * (gameManager.playerTwoFavo
123                     }
124                     else
125                     {
126                         totalDmgDone = basicAttackFavorGain - (strongAttackBaseFavorGain * (50 - gameManager.playerTw
127                     }
128                 }
129                 enemyController.gameManager.ChangeMagic((int)totalDmgDone, !playerOne);
130                 relativePos.y = 0.2f;
131                 print(relativePos);
132                 GetComponent<Rigidbody>().AddForce(relativePos * totalDmgDone, ForceMode.Impulse);
133             }
134         }
135     }
```

As for the custom lighting shader, it uses **Unity's URP** as a base and calculates the colour of the fragment differently. It calculates diffuse lighting, specular lighting, the *Fresnel* effect and even puts an outline on each edge of the objects to give a comic

book feel. All this with fully customisable values and it is a totally versatile shader to which you can apply both textures and normal maps without any inconvenience. See in the figure 5.8 all the values that can be modified.

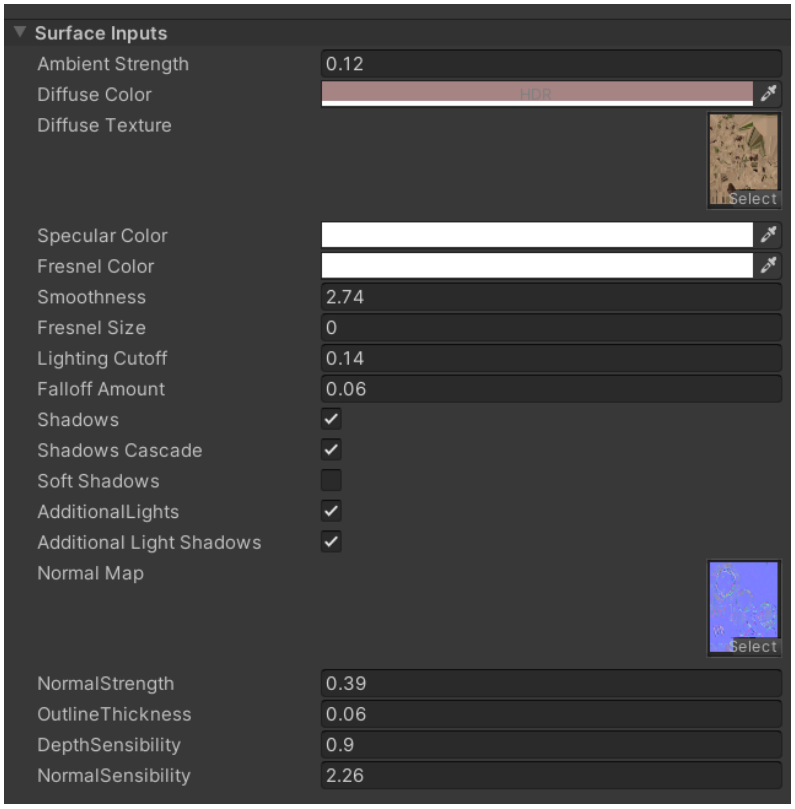

Figure 5.8: Material Settings

This is the *HLSL* script where the functions that get the values of the URP are. This could not be implemented in ShaderGraph because it requires loops to calculate the lighting of all the point lights in the scene and sum them up.
There is no cartoon shader on the internet that has this functionality of having all possible lights, so it is a proprietary implementation.

Lightning.hlsl

```
1     void MainLight_float(float3 WorldPos, out float3 Direction, out float3 Color,
2         out float DistanceAtten, out float ShadowAtten)
3 {
4 #ifdef SHADERGRAPH_PREVIEW
5     Direction = normalize(float3(0.5f, 0.5f, 0.25f));
6     Color = half3(1.0f, 1.0f, 1.0f);
7     DistanceAtten = 1.0f;
```

```
    ShadowAtten = 1.0f;
#else
    float4 clipPos = TransformWorldToHClip(WorldPos);
#if SHADOWS_SCREEN

    float4 shadowCoord = ComputeScreenPos(clipPos);
#else
    float4 shadowCoord = TransformWorldToShadowCoord(WorldPos);
#endif
    Light mainLight = GetMainLight(shadowCoord);

    Direction = mainLight.direction;
    Color = mainLight.color;
    DistanceAtten = mainLight.distanceAttenuation;
    ShadowAtten = mainLight.shadowAttenuation;
#endif
}

void MainLight_half(half3 WorldPos, out half3 Direction, out half3 Color,
        out half DistanceAtten, out half ShadowAtten)
{
#ifdef SHADERGRAPH_PREVIEW
    Direction = normalize(half3(0.5f, 0.5f, 0.25f));
    Color = half3(1.0f, 1.0f, 1.0f);
    DistanceAtten = 1.0f;
    ShadowAtten = 1.0f;

#else
    #if SHADOWS_SCREEN
        half4 clipPos = TransfromWorldToHClip(WorldPos);
        half4 shadowCoord = ComputeScreenPos(clipPos);
    #else
        half4 shadowCoord = TransformWorldToShadowCoord(WorldPos);
#endif

    Light mainLight = GetMainLight(shadowCoord);

    Direction = mainLight.direction;
    Color = mainLight.color;
    DistanceAtten = mainLight.distanceAttenuation;
    ShadowAtten = mainLight.shadowAttenuation;
#endif
}

#ifndef SHADERGRAPH_PREVIEW

Light GetAdditionalLightForToon(int PixelLightIndex, float3 WorldPosition) {
    int perObjectLightIndex = GetPerObjectLightIndex(PixelLightIndex);


    Light light = GetAdditionalPerObjectLight(perObjectLightIndex, WorldPosition);

    light.shadowAttenuation = AdditionalLightRealtimeShadow(perObjectLightIndex, WorldPosition);
    return light;
```

```
}

#endif

void AdditionalLight_float(float3 WorldPosition, float3 WorldNormal, float3 WorldView, float Smoothness, float3 SpecColor, ou

    #ifdef SHADERGRAPH_PREVIEW

        Diffuse = (0, 0, 0);
        Specular = (0, 0, 0);
    #else


    float3 diffuseColor = 0;
    float3 specularColor = 0;

#ifdef _ADDITIONAL_LIGHTS
    Smoothness = exp2(10 * Smoothness + 1);
    WorldNormal = normalize(WorldNormal);
    WorldView = SafeNormalize(WorldView);


    uint numAdditionalLights = GetAdditionalLightsCount();
    for (uint lightI = 0; lightI < numAdditionalLights; lightI++) {
        Light light = GetAdditionalLight(lightI, WorldPosition, half4(1, 1, 1, 1));
        float3 attenuatedLightColor = light.color * (light.distanceAttenuation * light.shadowAttenuation);
        diffuseColor += LightingLambert(attenuatedLightColor, light.direction, WorldNormal);
        specularColor += LightingSpecular(attenuatedLightColor, light.direction, WorldNormal, WorldView, float4(SpecColor, 0)

    }

    Diffuse = diffuseColor;
    Specular = specularColor;

#endif

#endif
}

void AdditionalLight_half(half3 WorldPos, int Index, out half3 Direction,
        out half3 Color, out half DistanceAtten, out half ShadowAtten)
{
        Direction = normalize(half3(0.5f, 0.5f, 0.25f));
        Color = half3(0.0f, 0.0f, 0.0f);
        DistanceAtten = 0.0f;
        ShadowAtten = 0.0f;

#ifndef SHADERGRAPH_PREVIEW
        int pixelLightCount = GetAdditionalLightsCount();
        if (Index < pixelLightCount)
        {
                Light light = GetAdditionalLight(Index, WorldPos);

                Direction = light.direction;
```

```
116                 Color = light.color;
117                 DistanceAtten = light.distanceAttenuation;
118                 ShadowAtten = light.shadowAttenuation;
119         }
120 #endif
121 }
```

It also supports point lights and shadows for each of the possible lights, as well as shadows cast on objects. It is a shader to which a lot of time has been spent, but a lot of importance was given to the graphical aspect of this work. Here (see **??**) is diffuse lighting, the key to which in the cartoon shader lies in the **Smoothstep** [34], to push the illuminated values to the extreme and not let there be a wide range of colours to illuminate an object.

After the specular, which is similar to the specular calculated in the URP, the Fresnel effect is calculated, which consists of putting a halo around the object when a light is right behind it (Figure 5.10).

Finally, the Outline [26] is multiplied to the final result of the shader, which is calculated with the position of the camera at all times (Figure 5.11).

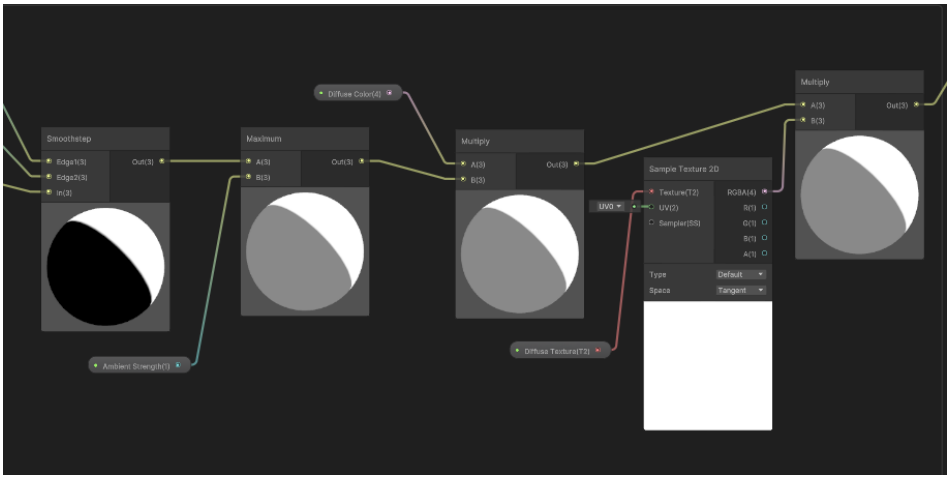

Figure 5.9: Diffuse Calculation

Este es el resultado final del shader in-game (figure 5.12).

5.2.2 Modeling

As for the 3D models, some time was spent on learning how to model characters and getting a bit of a feel for Blender. **"Skin"** Blender modifier was used, and starting from a vertex, it was extruded until it looked like the shape of the desired doll design.

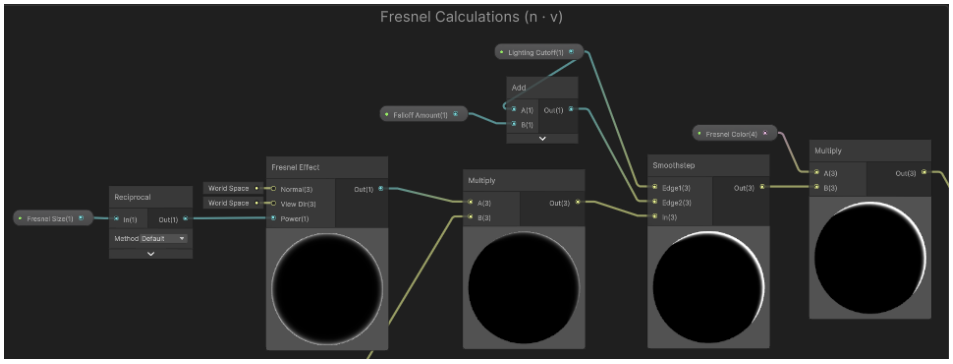

Figure 5.10: Fresnel Calculation

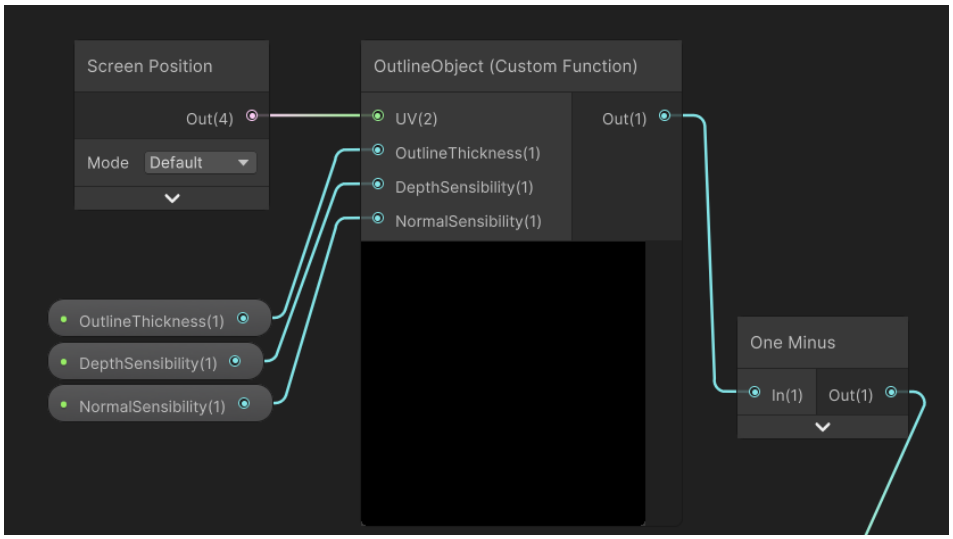

Figure 5.11: Outline Calculation

There wasn't enough time for texturing, so the models use free-use textures at the moment to atleast have a good appareance. The two characters, Rum and Xaka, are completily modeled for the game (see figures 5.13 and 5.14)

5.2.3 Animating

Animation is one of the most delicate aspects of a fighting game and that's why it has to be an extensive but well synchronised **animation tree** [11]. It was a bit difficult to coordinate everything, but this way the animations look correct at all times.
The animations are imported from **Mixamo**, it would have been a very good point to do the animations manually with the help of the **Mocap Room**(motion capture), but it required an investment of time that couldn't be used in this area. This is the animation

Figure 5.12: Outline Calculation

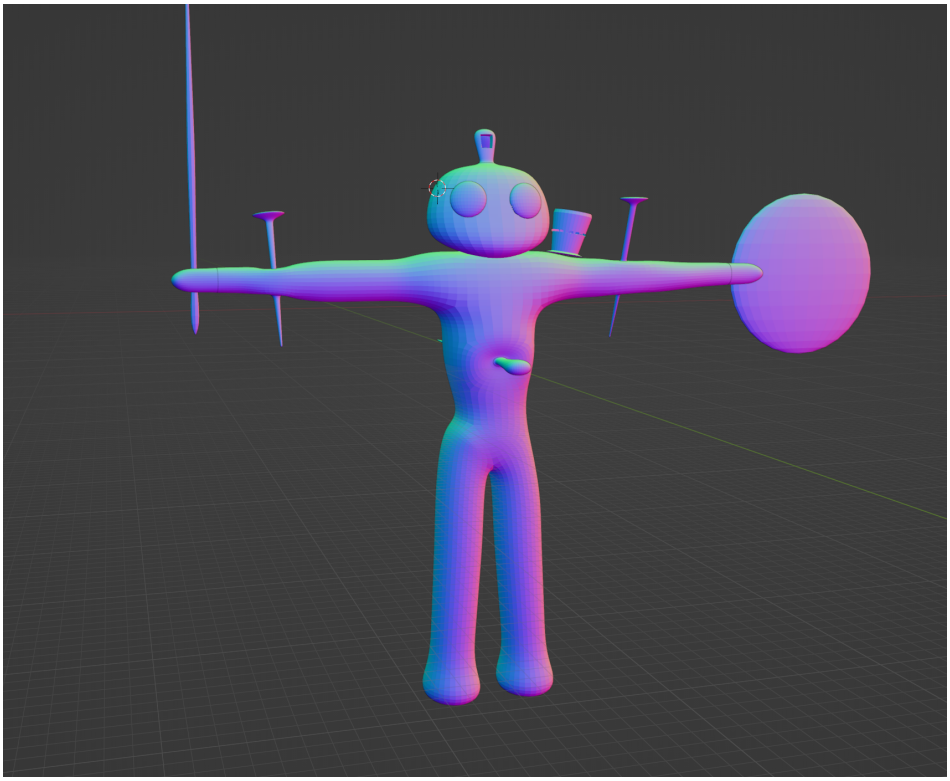

Figure 5.13: "Xaka", first character

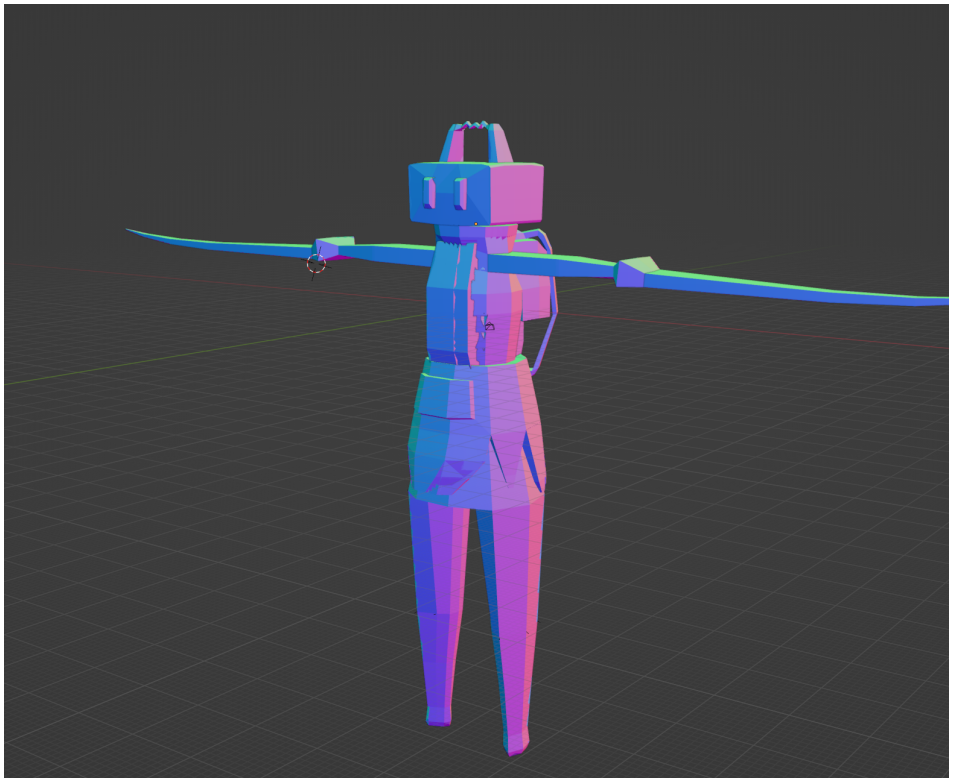

Figure 5.14: "Rum", second character

tree of a default character, see figure 5.15.

A **blend Tree** [24] was also used to merge the walking and running animations depending on the speed of the character (figure 5.16). Except for the running, jumping and static animations, all other animations can only switch from one to another until the animation is finished (**hasExitTime = true**). The animations call events at different frames, all to coordinate the gameplay and **hitbox** activations well.

5.3 Results

As a final result, the objectives set out in point 1.2 have been achieved. The game has been smooth, giving a good feeling to the people who have come to try it.
The multiplayer system is fully implemented, and only a little more effort is required to add functionalities that are fully expected by the player, such as Instant Rematch or Anticheat. Matchmaking has been a bit of a challenge, but it has been more challenging to code player control with both local and online multiplayer in mind, as well as the use of RPCs to communicate clients with each other. A lot of research time has been

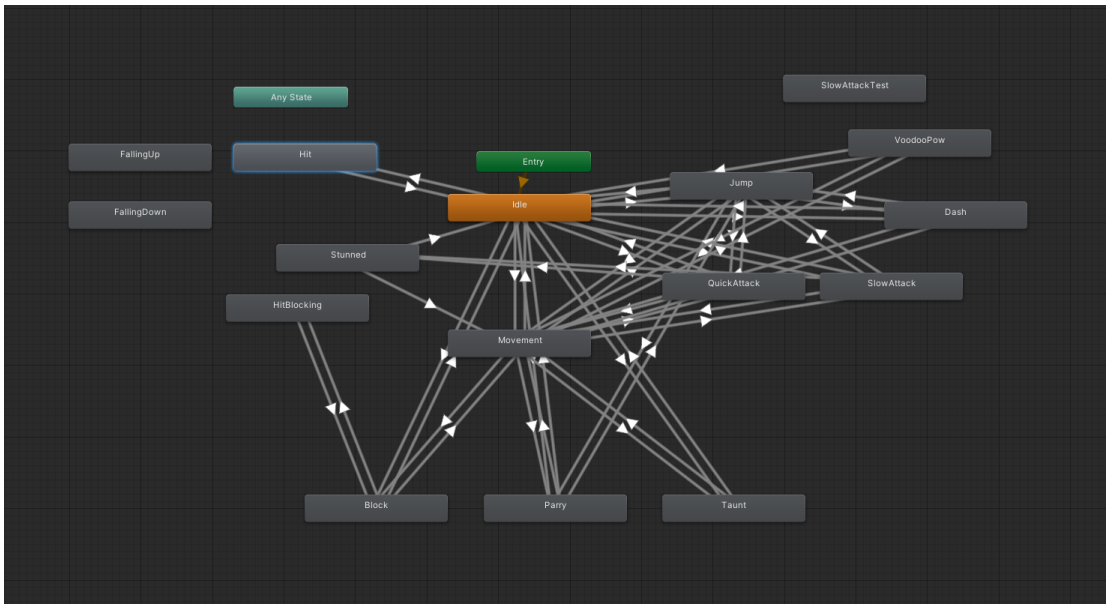

Figure 5.15: Animator controller of a character

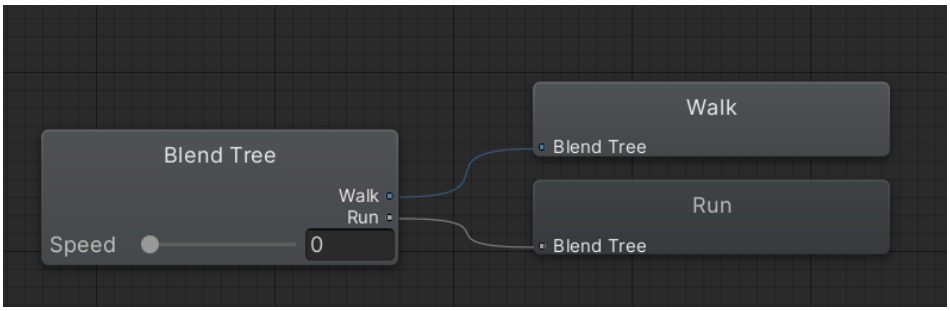

Figure 5.16: Blend tree movement

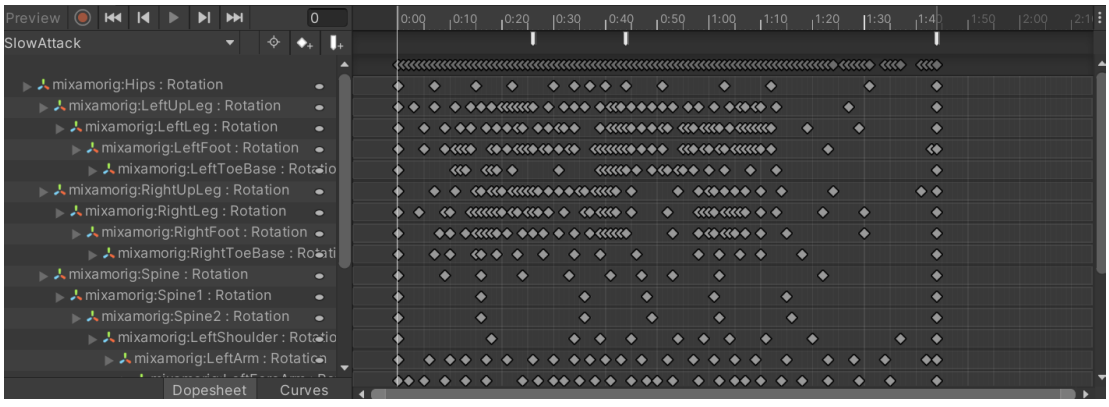

Figure 5.17: Animation with Unity Events

spent on the implementation of rollback, and it is something that has taken a lot of time away from implementing the basis of online and more game mechanics, however, good knowledge has been gained on the subject to establish a foundation to continue in that direction in the future.

The game perfectly simulates the opponent's behaviour only by receiving its current state generated by the input, and this is entirely intentional in order to reduce data sending to a minimum and to achieve a fast response.

Regarding the graphical shader, a lot has been learned about how the Shader Graph works and the calculation of lighting in a graphics engine. In addition, we have obtained a clean result, easy to modify parameters, and that can serve perfectly as the final shader of a commercial game.

The art developed by me resides in the user interface and the character models, which I think have acquired the desired personality for the project and form a totally valid atmosphere for a fighting game. The visual effects have been made totally acceptable thanks to good Unity assets and modifications to these to bring them more in line with the style of the game.

But above all, the satisfaction that the game can be played anywhere in the world and that anyone can find anyone else to play with is what stands out the most about the realisation of this project (see figure 5.18 for the final result).

This is the repository of the project: `https://github.com/aerisWay/VoodooWars`

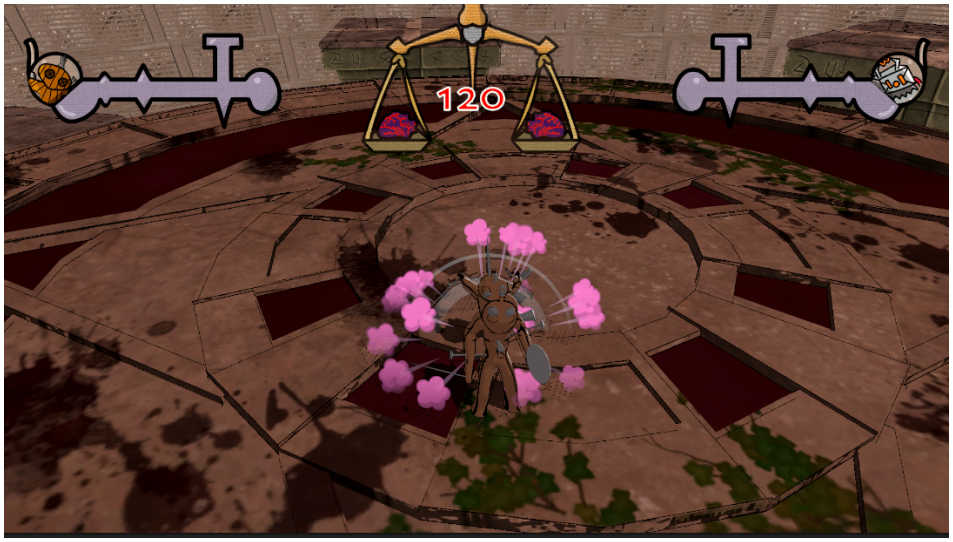

Figure 5.18: Final Result

Chapter 6

Conclusions and Future Work

Contents

In this chapter, the conclusions of the work, as well as its future extensions are shown.

6.1 Conclusions

I have really enjoyed making this project in many ways, the research has been very productive and because of this I feel the desire to make a game of this genre and publish it on Steam for everyone to enjoy. However, it has been frustrating at times, mostly due to technical circumstances and limitations that seemed simpler at first. Many features required implementation from scratch and networking knowledge that I did not have before this work. I was only familiar with various concepts, but had no idea how to implement them.

At one point I was very advanced in this work, because I started the project already at Christmas and I had a very clear idea of it, but these complications and the lack of time caused by the curricular internship just in the same space of time made that I had to cut back on some aspects, at least for this work. I feel it was too ambitious an idea, but I don't regret it, as it has made me try harder to learn new things and not stay in my comfort zone with things I already knew how to do. My knowledge in Blender and interface design has improved a bit in this project, so I have also done things I am not specialised in and I have increased the number of tasks that I could do in a professional environment.

In short, it has been a very profitable experience professionally speaking and a good starting point to get a game published on a platform, a requirement that is the order of the day in the profiles that companies ask for. In this way, I feel that the work has been put to good use and that it will help me professionally to achieve the goals I have set myself for the coming years.

6.2 Future work

Future work is undoubtedly the key to this project, which has only built a good skeleton waiting to be perfected. It is clear to me that I will continue the project for a while until I feel that the result is polished enough to bring it to market so that I can have a good flagship in my portfolio.

As for multiplayer functionalities, the first aspect would be to apply a matchmaking filter that matches players depending on their skill level and experience in the game. It would also be interesting to add Instant Rematch, a feature highly acclaimed by the community, as well as to establish a secure and fast client-server-client communication model. The Rollback Netcode would be available from the release of the game, and this would increase the good reception in the market. To build such an implementation a good AI using Behaviour Trees or Machine Learning is needed as long as it is good at predicting player inputs. It would also be important to make a non-competitive mode, so that players could practice new characters or techniques without the pressure of feeling ranked. In terms of mechanics, there is a long list of features thought of but not implemented at the moment: an Input Buffer to improve player feel, double jumping, throwing a chain that serves to return to the stage when you are far away, different special attacks for each character with their distinctive ultimate abilities, combos, a mechanic to match the players' favour called God's Mercy...

There are a lot of things to polish in terms of mechanics to make it a game complex enough to appeal to players of the genre, but they don't take too much effort to achieve in the short term with a little work, thanks to having a good foundation in character control.

It would also improve the code by using events for player interactions, as well as native animator events that achieve more accurate results when projecting hits and damage. The game would also feature more starting characters, so players wouldn't soon tire of them.

As for the visuals, it wouldn't hurt to have the help of a more artistic person to go over both the models and the interface and get a better finish on this aspect. I would like all the visual effects to be totally created by me, so that I feel that the whole product is mine in its totality. The sound effects would be one of the things I would polish

the most, because it is the aspect where I have lacked the most time, all to get an epic and sporty atmosphere in every game and good feelings when navigating through the UI.

Finally, I would start a weekly testing where I would write down every feeling the game gives to the testers and bugs that occur in the process in order to improve the game experience as much as possible.

Bibliography

[1] AllInGames. What is local co-op? https://www.allingames.com/what-is-local-co-op/.

[2] Bandai. Dragon ball z: Budokai tenkaichi 3. https://dragonball.fandom.com/es/wiki/Dragon$_B$all$_Z$:$_B$ udokai$_T$enkaichi$_3$.

[3] Alex Blackfrost. I made smash bros in unity. https://youtu.be/7E3N$_4$f9oLM.

[4] BlueMammoth. Brawlhall. https://www.brawlhalla.com/.

[5] David Cabrera. Super smash bros. ultimate basics: Life percentage, blowback and offense guide. https://www.polygon.com/super-smash-bros-ultimate-guide/2018/12/19/18149079/life-percentage-blowback-weight-blowback.

[6] Andrea "Jens" Demetrio. I wanna make a fighting game! a practical guide for beginners — part 4 (2021 update). https://andrea-jens.medium.com/i-wanna-make-a-fighting-game-a-practical-guide-for-beginners-part-4-2021-update-4c26f6964179.

[7] Andrea "Jens" Demetrio. I wanna make a fighting game! a practical guide for beginners — part 5. https://andrea-jens.medium.com/i-wanna-make-a-fighting-game-a-practical-guide-for-beginners-part-5-f049a78ddc5b.

[8] Dan Fornace. Anticipation, action, recovery. https://www.rivalslib.com/workshop$_g$uide/art/anticipation$_a$ction

[9] Dan Fornace. Dan fornace's 10 tips for making a fighting game. https://fornace.medium.com/dan-fornaces-10-tips-for-making-a-fighting-game-e2c982da2396.

[10] IShapeUnity. Fixfloat. https://github.com/iShapeUnity/FixFloat.

[11] James Marijeanne. Understanding the video game animation tree for interactive character animation. https://lesterbanks.com/2015/01/understanding-video-game-animation-tree/.

[12] Nintendo. Super smash bros. https://www.smashbros.com/es$_E$S/.

[13] Angela Palacios. Low poly: el arte de crear personajes y escenas con polígonos. https://www.crehana.com/blog/estilo-vida/low-poly/.

[14] Reddit. Do fighting games play differently above or below 60 fps? https://www.reddit.com/r/Fighters/comments/se1p9w/do$_f ighting_g ames_p lay_d if f erently_a bove_o r_b elow$

[15] Margaret Rouse. Dedicated server. https://www.techopedia.com/definition/4868/dedicated-server.

[16] samyam. Cinemachine third person controller w/ input system - unity tutorial. https://www.youtube.com/watch?v=ImuCx$_X VaEQ$.

[17] Sam Skinner. How matchmaking works. https://netduma.com/blog/how-matchmaking-works/.

[18] Steam. Darkest dungeon. https://store.steampowered.com/app/262060/Darkest$_D ungeon$/.

[19] Steam. Naruto: Ultimate ninja storm. https://store.steampowered.com/app/495140/NARUTO$_U ltimat$

[20] Tarodev. 9 tools for multiplayer game development ft. tarodev | unity gaming services. https://youtu.be/$_L JKIwmOyJA$.

[21] TechTarget. finite state machine. https://www.techtarget.com/whatis/definition/finite-state-machine.

[22] Unity. About netcode for gameobjects. https://docs-multiplayer.unity3d.com/netcode/current/about/index.html.

[23] Unity. Asset store. https://assetstore.unity.com/.

[24] Unity. Blend trees. https://docs.unity3d.com/es/530/Manual/class-BlendTree.html.

[25] Unity. Hlsl in unity. https://docs.unity3d.com/Manual/SL-ShaderPrograms.html.

[26] Unity. How to make a toon outline effect in unity 2019 lwrp! (tutorial). https://youtu.be/joG$_t mXUX4M$.

[27] Unity. Input system. https://docs.unity3d.com/Packages/com.unity.inputsystem@1.6/manual/index.ht

[28] Unity. Networktransform. https://docs-multiplayer.unity3d.com/netcode/current/components/network

[29] Unity. Render pipelines. https://docs.unity3d.com/Manual/render-pipelines.html.

[30] Unity. Sending events with rpcs. https://docs-multiplayer.unity3d.com/netcode/current/advanced-topics/messaging-system/.

[31] Unity. Unity. https://unity.com/es.

[32] Unity. Unity editor system requirements. https://docs.unity3d.com/2021.1/Documentation/Manual/sys requirements.html.

[33] Unity. Unity events. https://docs.unity3d.com/Manual/UnityEvents.html.

[34] Patricio Gonzalez Vivo. Smoothstep. https://thebookofshaders.com/glossary/?search=smoothstep.

[35] Vivox. In-game voice and text chat (vivox). https://unity.com/products/vivox.