TecnoCampus

Centre universitari adscrit a la

upf. Universitat Pompeu Fabra Barcelona

Grado en Diseño y Producción de Videojuegos

Development of a Multi-Genre Third-Person Character Controller

Memoria Final

Veronika Bolshakova

Tutor: Rafael González Fernández

2023-2024

TecnoCampus Mataró-Maresme

Special Thanks

To my mom, Olga, who has always supported me throughout this long and difficult process.

To my amazing partner, Joan, for keeping my spirit up and helping me save the remains of my sanity.

And finally to Adso Fernandez, for guiding me when it seemed like there was no light at the end of the tunnel.

Abstract

The objective of this project is to develop a multi-genre third-person character controller that has different presets for various game genres and can also be modified by the user in order to adapt better to the needs of their prototype. To do so, different game genres that use a Third-Person Character Controller will be researched and analyzed to determine their main characteristics and find out what affects each genre's *game feel*.

Resumen

El objetivo de este proyecto es desarrollar un controlador de personaje en 3a persona multi-genero que tiene diferentes *presets* para varios géneros de juegos y que también puede ser modificado por el usuario para adaptarse mejor a las necesidades de su prototipo. Para hacerlo se estudiarán y analizarán diferentes géneros de juegos que utilizan un controlador de personaje en tercera persona para determinar las características principales y determinar qué afecta el *game feel* de cada género.

Resum

L'objectiu d'aquest projecte és desenvolupar un controlador de personatge en 3a persona multi-gènere que té diferents presets per a diversos gèneres de jocs i que també pot ser modificat per l'usuari per adaptar-se millor a les necessitats del seu prototip. Per fer-ho s'estudiaran i s'analitzaran diferents gèneres de jocs que utilitzen un controlador de personatge en tercera persona per a determinar les característiques principals i determinar què afecta el *game feel* de cada gènere.

Table of Contents

List of Figures

List of Tables

1. Introduction

Character controllers are essential to every video game that involves the interaction between a player and an in-game character (Swink, 2009). The feeling that they provide sets the tone of the whole game since it is the interaction that is happening constantly throughout the gameplay. Whether it is a realistic *Third-Person Shooter* or a cartoony *3D Platformer*, the character controller has to be polished and perfected as much as possible to make the player's experience the best.

There are different types of character controllers, but this project will be focused on *Third-Person Character Controllers* that are widely used across games in genres such as *Third-Person Shooter*, *3D Platformer*, *Role-playing Game (RPG)*, or *Stealth*. These genres require very complex and varied character controllers since the characteristics and needs of each genre are very different.

The creation of a character controller is a key part of developing a game, and usually, most video game studios, whether they are Triple A (AAA) or Independent (Indie), have one or several programmers working on the character controller. However, there are solo Indie developers who sometimes do not have the knowledge or time to create a character controller that would match the needs of their game. This leads to many abandoned or poorly executed games.

The main goal of this project is to create a Third-Person Character Controller that can be adapted to different game genres and modified depending on the needs of the developer using it. It is oriented towards solo developers who prefer to focus on the design and art of the game rather than programming. The controller will be created with Unity 3D, a very popular video game engine. It will be available as a package in Unity Store, a digital platform that contains different kinds of assets available to all the developers.

In order to create a character controller that can be adapted to several game genres, these genres will be analyzed to determine their main characteristics and what creates a good game feel [1] (Steve Swink, 2008) in these genres. Different games will

[1] This term will be defined and analyzed later in the paper.

be studied to determine the best and the worst parts of each controller. From each genre, one game that is the most popular and has the best reviews on the game feel will be taken as a main reference.

After the research, based on its results a character controller will be created to match the needs of each genre in the best way possible. The character controller will include several presets for each genre, but all of them will be fully accessible to the user and easy to modify. The presets will be easy to understand and will have all the components exposed, allowing the users to adapt them better to the needs of their projects.

2. Objectives

In this chapter, the main and secondary objectives for this project are set and explained.

The main objective of this project is to **create a multi-genre character controller asset pack that would allow designers to test their levels and prototypes**.

The secondary objectives represent the building blocks of the final product and are the following:

- Analyze the character controller, the camera, and the mechanics of three game genres: Third-Person Shooter, 3D Platformer, and Adventure.
- Define the common mechanics that will be present in all the presets.
- Define the genre-specific mechanics for each preset.
- Implement a base character controller that would serve as a template for the three presets.
- Implement three presets for three game genres with common and specific mechanics.
- Implement animations for each preset.
- Implement the functionality of the presets for different input devices, such as keyboard and gamepad.
- Create documentation for the character controller to make it easier for the users to understand and make use of it.
- Create an Asset Pack for the Unity Asset Store.

The final product would allow designers to easily implement and test their levels and prototypes without the need to create a character controller themselves or collaborate with somebody. It will be user-friendly and open code, which means that the people who want to add their own features will be able to do so.

3. Referents

In order to better understand the needs of this project, several games that use third-person character controllers will be analyzed from the perspective of movement, game feel, and use of the camera. These game genres are intended to be used for the character controller presets.

3.1 Mass Effect 3

Mass Effect 3 (BioWare, 2012) is an Action RPG that has third-person shooter combat. It is the third part in the Mass Effect series and the last part in the original trilogy of games. It was developed by *BioWare* and published by *Electronic Arts* in 2012. The camera in this game is displaced to the right, leaving an unobstructed view of the center and also making the player character visible at all times. The player character and the camera are moved using a keyboard and a mouse or left and right joysticks on a gamepad. The player moves forward in the direction where the camera is looking, although they can move around in different directions too.

Since *Mass Effect 3* (BioWare, 2012) is a shooter, most players find it much more comfortable and convenient to play it with a mouse and keyboard setup, especially if they are focusing on using weapons more than skills. But it does not make the game unplayable with a gamepad. The variation of player character skills allows players to enjoy the game without the need to use weapons for every combat.

Even though the combat gameplay might feel a bit worse and uncomfortable on a gamepad, many users mention that the UI is definitely much more convenient for a gamepad rather than a mouse and keyboard setup. Like most shooters, the controls feel sturdy and reliable. They are very responsive and provide a great game feel.

Mass Effect 3 (BioWare, 2012) is a good example of a third-person shooter because its shooting mechanic is very polished and has a lot of variety to it. There are many weapons available that make it more interesting for the player, allowing to create a personalized setup of the player character. Apart from shooting, there are other mechanics that are commonly used in shooters, such as crouching, taking cover, and sprinting. The combination of these mechanics makes *Mass Effect 3* (BioWare,

2012) a good reference for the creation of a good character controller for a third-person shooter.

Figure 1: A screenshot of gameplay in *Mass Effect 3* (Source: YouTube)

3.2 Lil Gator Game

Lil Gator Game (MegaWobble, 2022) is a 3D platformer game developed by *MegaWobble* and published by *Playtonic Games* in 2022. In this game, the player controls a small alligator that explores the world. The main movement mechanics are running, jumping, climbing, swimming, and gliding. These are used in order to reach and discover new areas.

The camera is positioned in the center at eye level of the player character. It follows the player character at all times. The player can rotate it but does not have the possibility to move it separately from the player character. The camera avoids collisions with the walls, some objects turn semi-transparent in order to make the player character visible and the characters also have an outline. The player character moves forward in the direction of the camera, but can also move around the game world in other directions.

There are some items in the game that allow shooting, and for shooting mode, the camera is displaced to the left, leaving the character on the right side of the screen. This camera positioning is unusual since in most games it is displaced to the right.

The game can be played both with a mouse and keyboard setup, and a controller. Despite having a better game feel with a controller, it feels very nice and polished with the mouse and keyboard too.

Figure 2: A screenshot from *Lil Gator Game* (Source: IGN)

3.3 Stray

Stray (BlueTwelwe Studio, 2022) is an adventure game developed by *BlueTwelwe Studio* and published by *Annapurna Interactive* in 2022. It is a story about a cat that lives in a futuristic world. One day it falls into an underground city, gets separated from its mates, and needs to find its way home. The player controls the cat from a third-person perspective and can interact with the world in several ways, like scratching something or dropping or carrying items. The main objective is to find a way out of the city by discovering paths, running and jumping around, climbing obstacles, and solving environmental puzzles.

The camera is positioned in the center, behind, and slightly above the player character. The player can rotate the camera around the player character but does not have the freedom to move the camera separately. There is a minimum and a maximum distance between the player character and the camera. The player moves forward in the direction of the camera and can move in other directions too, independently from the camera.

The game advises the players to use a gamepad for a better experience, but it is completely playable with a mouse and keyboard setup. The game feel with this setup is good, except for one challenge, but the overall experience is great. However, using the gamepad feels more natural and provides a smoother gameplay.

The controls feel very responsive and give a lot of freedom to the player to move around and explore the in-game world. However, in *Stray* (BlueTwelwe Studio, 2022), the player cannot make the player character jump on their own will. The jump button activates only when the player character approaches a surface that it can jump on. This feature may seem limiting but in reality, it is the opposite. Most games allow jumping at all times meaning that the player has to decide when to jump, *Stray* (BlueTwelwe Studio, 2022) does the opposite. It has a limited jump which takes the responsibility of deciding when to jump off of players' shoulders giving a smoother gameplay with fewer interruptions. A game without the possibility of free jumping at the beginning may seem a bit uncomfortable, but after the player gets used to it, it feels just as natural as walking. This feature will not work for all games, but for this game in particular it works perfectly and only makes it better.

Figure 3: A screenshot of gameplay in *Stray* (Source: Reddit)

3.4 A Short Hike

A Short Hike (Adam Robinson-Yu, 2019) is an adventure game created by an indie game designer, *Adam Robinson-Yu*, and first released in 2019. It is a short game where the player controls a human-like bird character and the main objective is to get to the top of the mountain to get phone reception. The main mechanics are walking, running, flying, and climbing up the mountains. It could also be considered a 3D platformer since there are a lot of platformer elements.

The camera is centered, positioned above the character, and slightly inclined. There is a minimum and maximum position between the player character and the camera, so the player character is not always in the center of the screen but is always close enough. The camera is almost completely controlled by the game itself, the player can only rotate it slightly in two directions. It can feel limiting sometimes because there are moments when the player cannot see the player character and cannot move the camera enough. The player character moves in different directions in the world, independently from the camera's direction.

The game recommends using a gamepad but unlike *Stray* (BlueTwelwe Studio, 2022), this game is completely unplayable with a mouse and keyboard, so using a gamepad is more a requirement than a recommendation. The player can move around and try to play with the mouse and keyboard, but it is very difficult and the controls feel very limited.

Figure 4: A screenshot of the game *A Short Hike* (Source: A Short Hike).

4. Theoretical Framework

In this section, the main components of the final project will be analyzed and explained in order to understand their functioning. There are five main parts: the character controller itself and how it works, the camera and its positioning, common mechanics that are present in each genre used for the project, genre-specific mechanics, and finally the game feel of each genre. The game feel will also be divided into the analysis of mouse and keyboard game feel, and gamepad game feel.

4.1 Character Controller

A character controller is an instrument that allows the user to control the player character (Westecott, 2009) in the game. A character controller can be kinematic or dynamic, meaning that it works with input displacement vectors (not simulated physics), or input velocities or forces (*Character Controllers — NVIDIA PhysX SDK 3.4.0 Documentation*, n.d.). The dynamic character controllers simulate real-world physics to create a more realistic experience for the player.

Depending on the engine, the character controller can either be created manually by the programmer or using an existing preset provided by the engine (*Unity - Scripting API: CharacterController*, n.d.).

In every game that has a player character, the character controller is one of the essential parts of programming since it is a part the player will be interacting the most with during the gameplay.

Usually, a character controller consists of several parts: the physical body, which can be a simple capsule or a rigged 3D model, the colliders that help detect the collisions between the player character and the objects in the game world, the programmed behavior, which is the core of functioning of the player character, and the animations.

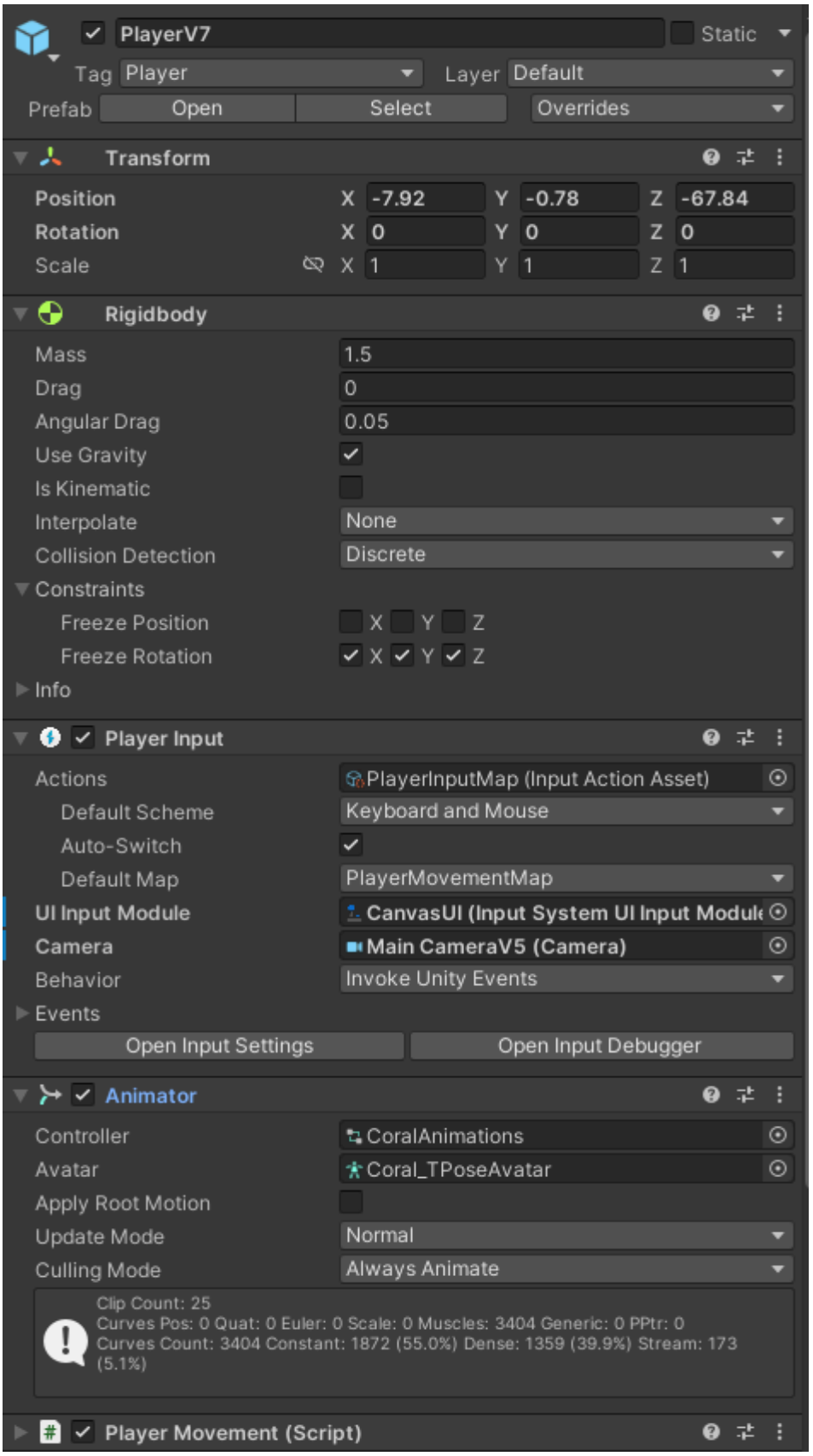


Figure 5: A screenshot in Unity 3D that shows an example of a complete character controller (own image).

Figure 5 shows an example of a complete character controller created with Unity 3D. It consists of several parts: *Rigidbody*, a component that allows a game object, in this case, the player character, to be influenced by Unity's physics engine (*Unity - Scripting API: Rigidbody*, n.d.), *Player Input* (*The PlayerInput Component | Input System | 1.5.1*, n.d.) component determines which input map should be used, *Animator* (*Manual: Animator Component*, n.d.) component is responsible for the animations of the 3D model, and *Player Movement*, which is the main script that is responsible for the player character's movement.

Some engines provide a pre-built functioning character controller that can be used for testing prototypes or as a base for further development, but most developers prefer to create the character controller from scratch to better cover the needs of their project.

A well-made character controller that provides a good game feel can significantly improve the sense of immersion the player feels.

4.2 Camera

The camera defines what the player will be seeing and how they will be seeing it. A camera that is shaky, too slow, too fast, or stuttering can completely ruin the gameplay experience and make the player abandon the game. That is why it is essential to create a well-functioning camera for the game.

There are several basic characteristics that define the way the player will see the in-game world. The first one is projection, which can be orthographic or perspective. If the projection is orthographic, the 3D objects are shown uniformly, by using different 2D views of these objects, as if they all were in the same plane (*Orthographic Projection Definition & Meaning*, n.d.). If the projection is in perspective, the objects are shown the way they would be seen in real life. The type of projection depends on the game genre, but the most common is perspective.

The example is shown in Figure 6 below.

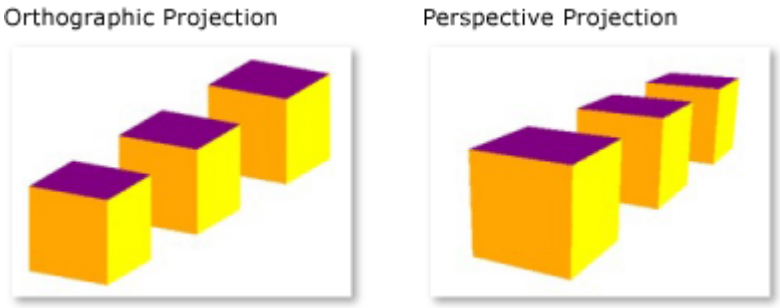


Figure 6: An example of orthogonal projection and perspective projection (Source: Comparing Orthographic and Perspective Cameras - Questions & Answers, 2016).

The next characteristic is the camera's Field Of View, which is the camera's view angle in degrees that defines what will be captured by the camera. The lower the FOV is, the smaller the angle of view.

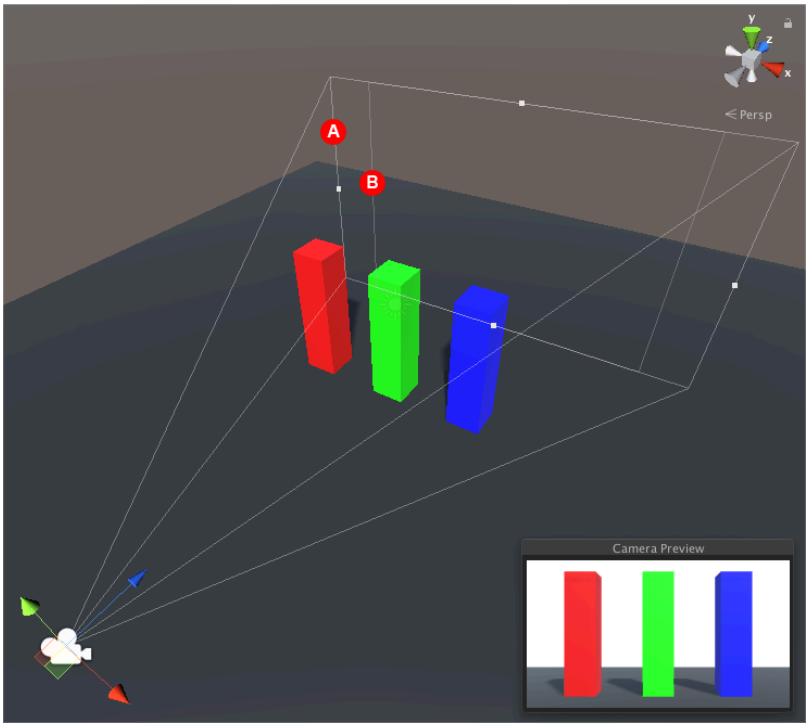


Figure 7: An example of the camera's FOV (Source: FOV Visual Representation Script. Rotation Causes Offset - Questions & Answers, 2016).

The last characteristics that are essential are the camera's Clipping Planes. The Near Clipping Plane defines the closest to the camera point where the objects will be seen. If the object is beyond the Near Clipping Plane (even closer to the camera) it will not be seen. The Far Clipping Plane defines the farthest from the camera point

where the object will be seen. If there are objects beyond this point, they will not be seen. The example is shown in Figure 8 below.

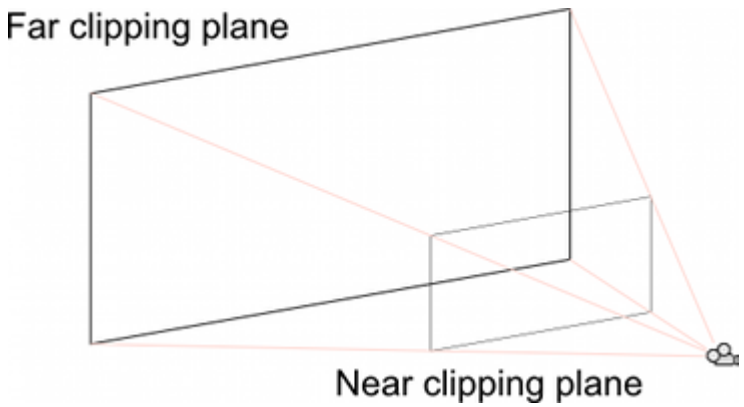


Figure 8: A graphical representation of the Near clipping plane and the Far clipping plane (Source: Understanding the View Frustum - Unity Manual).

There are more characteristics of the camera and rendering settings, but for this project, there is no need to get into all the details of rendering.

A camera can be created in different ways. Some engines offer a prebuilt camera that works very well and is fast to understand and put to use, like *Cinemachine* in Unity 3D (*Cinemachine*, n.d.), but some developers may prefer to create their own script for the camera's behavior. Programming the camera's movement and behavior from scratch can be tedious but it allows the developers to create a camera that will suit their game the best way possible.

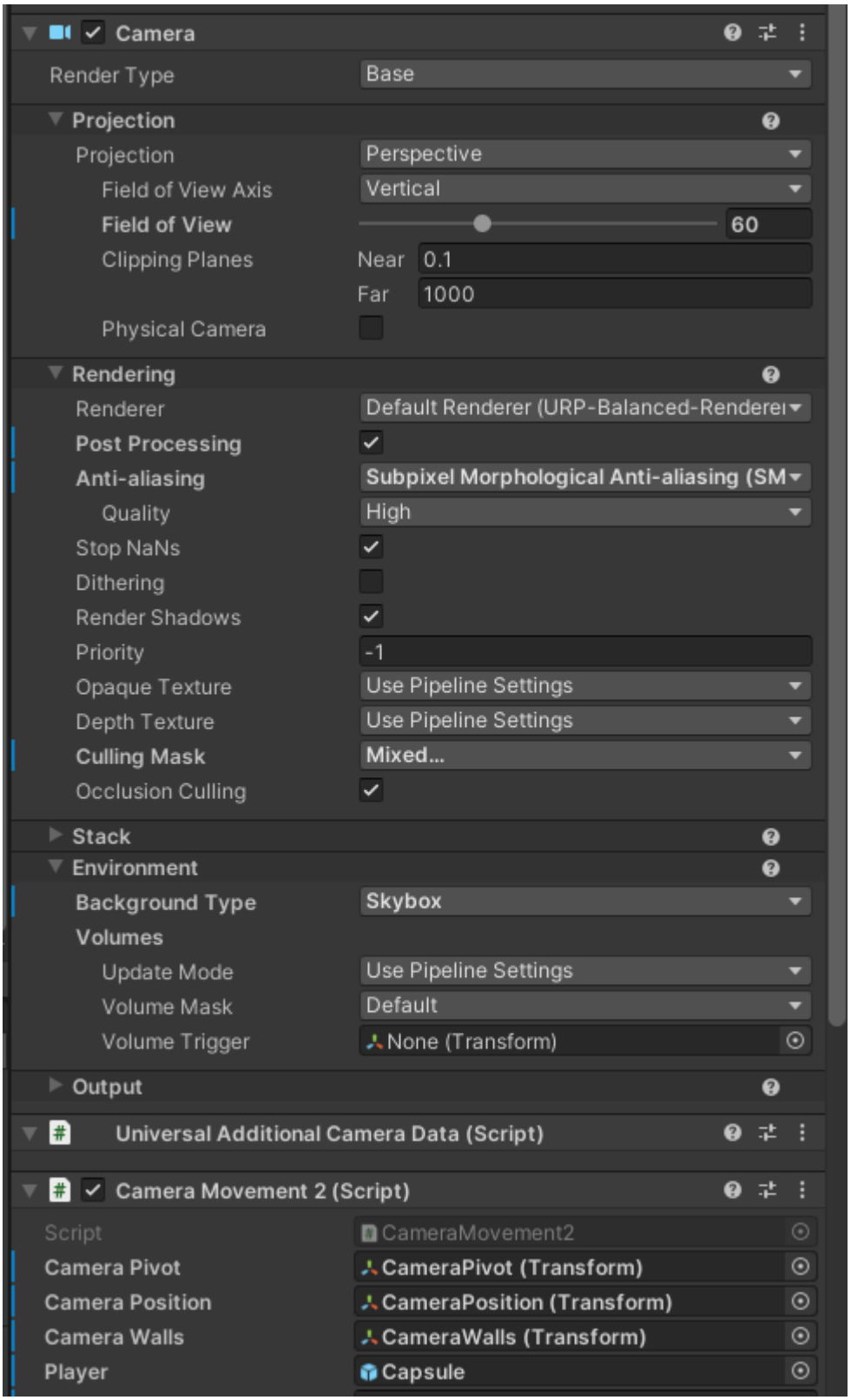


Figure 9: A screenshot in Unity 3D that shows an example of a camera functioning with a custom script (own image).

4.2.1 Camera Positioning

There are three main types of cameras in video games: First-Person Camera, Third-Person Camera, and Cinematic Camera. Only the Third-Person Camera will be described and analyzed for this project.

Figure 10: An example of a Third-Person Camera VS a First Person Camera (Source: Lynch, 2023).

A Third-Person Camera is a camera that is separated from the player character (Westecott, 2009), making the player character visible. There are two main ways of positioning the camera: behind and above the character, centered, or above the shoulder of the character, not centered, the character is usually on the left (Buehler, 2019). These two positions can also be combined in some games, e.g. having the camera centered during most of the gameplay and displacing it to the right when the player needs to shoot.

Fixing a camera in only one of these positions can be limiting in a way, since having the camera centered all the time may obstruct the view sometimes, and having the camera displaced may prevent the player from fully observing the player character.

Figure 11: An example of a centered camera positioned behind and above the character (Source: Chaotic Cat Studio, 2023).

Figure 12: An example of a camera displaced to the right, over the character's shoulder, for shooting (Source: Chaotic Cat Studio, 2023).

Some may argue that first-person cameras provide a higher feeling of immersion since the player is not seeing the player character all the time and is perceiving the world through “their own eyes”, but third-person cameras give the player a wider FOV and show many more interactions of the player character with the in-game world (Haigh-Hutchinson, 2009).

On the downside, the games that use third-person cameras require complex sets of animations for the player character. The player character is fully visible at all times which focuses the attention of the player on it much more than in first-person games. Since the player’s attention is drawn to the player character and its interaction with the in-game world, the animations need to be much more complex and polished than the animations for a first-person game. If the animations do not look and feel good, the game feel may become worse and the player will get a bad experience from playing the game.

4.2.2 Camera Control

The camera can be controlled either by the player or by the game itself. Very few games use fully automated control of the camera since it gives the player a sense of helplessness and inability to control the situation. But a camera fully controlled by the player is not used in all games either. If the player has to move the camera too in order to follow the player character, it can become tedious and create an unnecessary amount of effort for the player. It also feels very unnatural when the player moves the player character but the camera itself does not move.

The solution to all these problems is a hybrid camera (Buehler, 2019). In a hybrid system, the camera follows the player character and most of the camera control is done by the game itself, but the player still has the ability to rotate the camera and move it to a certain extent. This combination provides a sense of control that the player needs but also does not overload the player with the obligation to move the camera every time they move the player character.

The freedom of the hybrid cameras can vary from giving the player the option to move the camera around to a certain distance from the player character and rotating

it to having the camera controlled by the game itself most of the time and only allowing small degrees of rotation, like in *A Short Hike* (Adam Robinson-Yu, 2019).

4.2.3 Camera Movement

The way the camera moves in third-person games can vary a lot, from simply following the player character and going through the in-game walls to changing perspectives depending on the situation and avoiding collisions with any objects.

There are several characteristics that define how the camera will be positioned and how the player will see the in-game world:

1. Distance from the player character.
2. Camera movement speed.
3. Rotational behavior.

The first characteristic affects the gameplay in a very important way. The camera cannot be too far from the player character to not confuse the player and make it more difficult for them to find the player character, but it cannot be too close since it would limit the perspective and not give enough space to see the in-game world.

A “perfect” distance does not exist, it is different for every game. In some games, the distance between the camera and the player character is fixed during the whole game, in other games it can change depending on the situation or the environment (e.g. if the player character enters a cave, the camera will get closer to simulate the limited space).

A fixed distance between the player character and the camera is not always completely fixed, sometimes it means that there is a certain threshold for the camera not to move. This means that there is a minimum distance and a maximum distance between the camera and the player character. There is also a zone in the center of the camera in which the player can move without triggering the camera’s following behavior. This aspect also depends on the game genre and the wanted game feel.

The second characteristic, which is the speed of the camera’s movement, is also very important: making the camera move very slowly also requires limiting the player

character's speed and making the camera move too fast can provoke dizziness for the player. Usually, there is a minimum and a maximum speed for the camera, it starts at the minimum and increases gradually until it reaches the maximum.

The third characteristic is the rotational behavior. The camera can be rotated in three axes: x, y, and z. In the video games industry, the names for these axes were taken from the aviation industry: Pitch, Yaw, and Roll Axis.

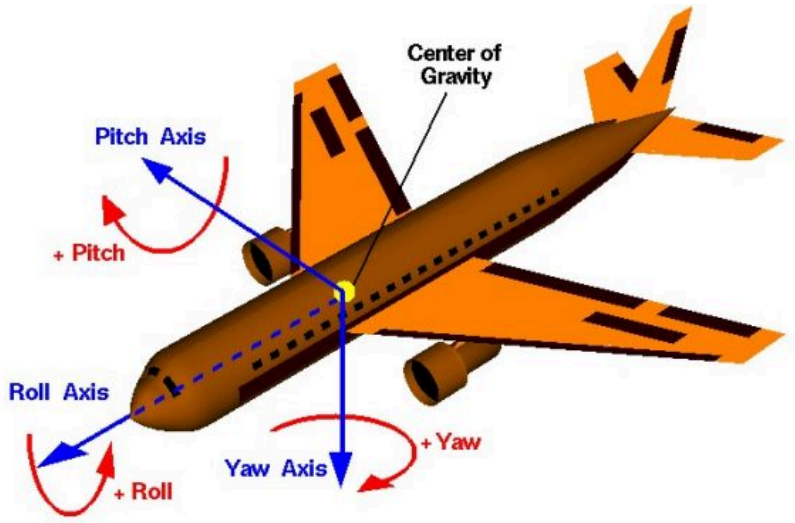


Figure 13: A representation of rotation axes in the aircraft industry (Source: Glenn Research Center)

In *Unity 3D* they are usually assigned this way: Yaw - Y, Pitch - X, Roll - Z. The rotation around the player character would be represented by yaw in the horizontal plane and by pitch in the vertical plane. Roll is not widely used for the camera's movement in a third-person controller.

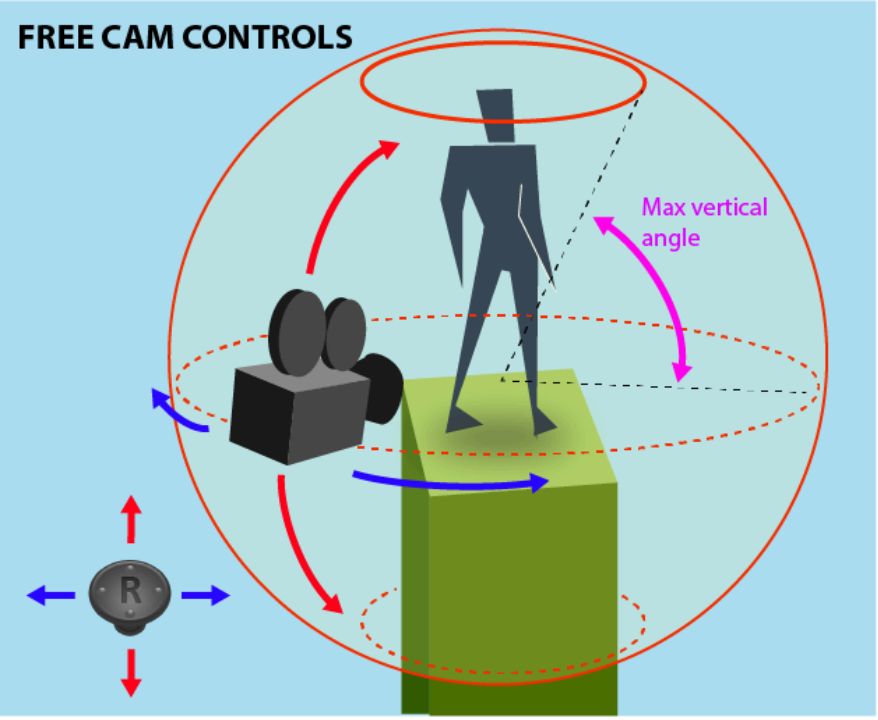


Figure 14: Free movement of the camera around the player character (Source: Pignole, 2015)

Figure 14 represents the movement of a free camera around the character. The distance between the camera and the player character is fixed, and there is also a restricted angle for the Pitch (x) rotation. It is usually fixed so the camera does not go below the ground or completely above the player character, forming a 90-degree angle. In this case, the Yaw (y) rotation is unrestricted, but some games allow a limited Yaw rotation.

In terms of the camera following the player character, there are also many ways to set it up. There can be a small zone around the player character where it can move without having the camera moved, the camera can move with every movement of the player character. The player character rotation can affect the camera rotation and vice versa, e.g. in some games, when the player is aiming, rotating the camera in Yaw also rotates the player character.

It is essential to make sure that the player can see the player character at all times, but also that the camera does not go through the ground or the walls. In order to avoid this, the cameras have colliders that help detect and avoid collisions with all in-game objects. Sometimes, to make the player character visible in narrow spaces,

the camera's angle is changed. Other ways to help the player locate the player character are to make walls invisible when the player character is near, make parts of the walls see-through, or use an outline shader to define the silhouette of the player character.

4.2.4 Camera Collisions

Depending on the game, the camera can detect and avoid collisions in many different ways. The functioning of a camera is similar to the functioning of an Artificial Intelligence element in the game since it has a basic behavior that the player does not affect (Haigh-Hutchinson, 2009). But unlike AI characters, a camera has many more restrictions. One of the most important constraints is collision detection and avoidance. Collisions can be detected in different ways, the first one is *Ray Casting*.

Ray Cast is an imaginary line projected from the camera in one or several directions that detects certain objects or types of surfaces. It also helps detect when a surface is covering the player character, which is very important if the player character needs to be visible at all times. If the player character is not visible, the camera can change its position or a certain graphic solution can be used in order to highlight the player character, for example, making the walls transparent or highlighting the player character with an outline. The problem with ray casting can be that this imaginary line can go through small openings in the game world, left there unintentionally, and in this case, the collision will not be detected.

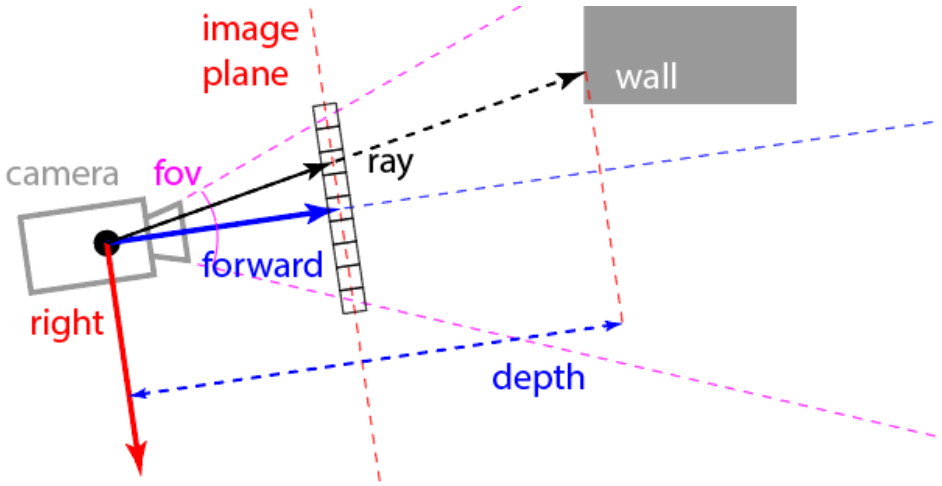


Figure 15: An example of a ray cast projected from a camera (Source: Stack Exchange)

Another similar way of detecting possible collisions is *volume projection*. Instead of projecting a single line (ray), an extruded geometric shape is projected, usually a capsule that can be rectangular or cylindrical. Volume projection is occasionally used to determine whether the camera will fit into small spaces. It is more demanding in terms of performance, so it can be imitated by projecting several rays in different directions.

A different approach to detecting collisions would be to put a cylindrical or spherical collider on the camera. A spherical collider is more widely used since sphere collisions are easier to detect and a spherical collider is less likely to “stick” to a surface. The volume of the sphere is determined by the camera’s near plane, it cannot be smaller than the near plane distance in order to avoid visual clipping. However, using a sphere collider for the camera implies that all the objects in the in-game world that can possibly collide with the camera need to have physical colliders.

4.3 Game Feel

“Game Feel: A Game Designer’s Guide to Virtual Sensation” is a book written by Steve Swink in 2009. For this project, this book is used as one of the sources of information about the concept of *game feel*.

4.3.1 The Definition of Game Feel

Throughout the history of video games, the concept of *game feel* has been left out or under-explained by many authors and developers. The first book that mentions it is “The Art of Computer Game Design” by Chris Crawford. It is described as “The input structure is the player’s tactile contact with the game; people attach deep significance to touch, so touch must be a rewarding experience for them.” (Crawford, 1984). This definition mentions the importance of the input of the game and the player’s contact with it but still leaves out many aspects that are essential to the game feel.

The characteristics that form the game feel are varied and every game developer has a different opinion on what features should be considered the most important for the game feel. However, Steve Swink managed to describe the game feel in the most profound and complete way possible. There is no formal definition of the game feel since it cannot be described with just one phrase, but there are several essential parts to it.

4.3.2 The Three Building Blocks of Game Feel

In the first chapter of his book, Steve Swink defined the most important aspects of game feel that allow one to better understand what it really is and how to create a good game feel.

4.3.2.1 Real-Time Control

Real-time control is the first aspect analyzed by Swink. He defines it as a loop of interaction between two participants, the user and the computer. The user produces an action, the computer receives this action as an input, analyzes it, and provides an output for the user. The user then analyzes the received output and provides the next action based on the information received. This forms a loop of interaction during

which the user and the computer exchange information and provide actions based on it. Steve Swink mentions Chris Crawford again, referencing his definition of this interaction: “a cyclic process in which two active agents alternately (and metaphorically) listen, think and speak” (Crawford, 1984).

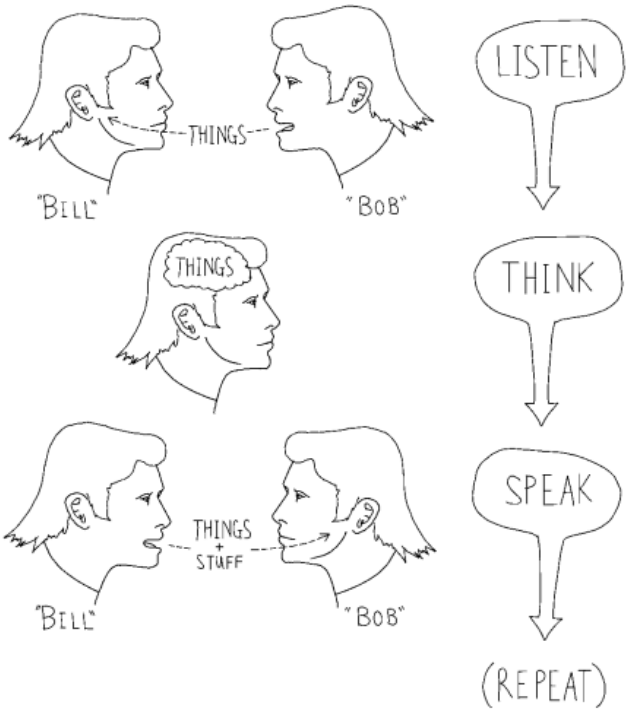


Figure 16: Interactivity as a conversation (Source: Swink, 2009)

The interaction between the user and the computer can be viewed as a conversation between two people. Both participants provide information, process the received information, and give a response afterward. Crawford represents the computer as one of the speakers, that receives information via the player’s input and “talks” to the player via any output device, like a screen, speakers, or even a gamepad.

The main difference between a normal conversation and a user-computer interaction is the time of response. When two people are having a conversation, one of them waits for the other one to provide information, processes it, and then replies. But a user-computer interaction happens much faster, the user does not have to “wait” for the response, and the feedback is usually immediate. Swink compares this type of interaction to driving a car: controlling a character in a game can feel similar to driving a car. The rapidness of response from the computer makes this interaction

fluid and gives the sense of perceiving the result of an action at the exact moment it is produced.

This is the starting point for the definition of game feel: “Real-time control of virtual objects” (Swink, 2009). However, it is impossible to perceive the movement of an object if there are no visual references around it. If a person sitting on a train has no static visual references and sees another train moving, they may get the feeling that their train is moving. If the surroundings in a game are just a solid color with no difference and no points of reference, the player will not be able to see that the player character is moving. This leads to the following fundamental part of game feel.

4.3.2.2 Simulated Space

Simulated space is the world that surrounds the player character in the game. The simulated space is perceived by the player and allows them to situate themselves in the game world. It also includes collisions and interaction with objects and characters. The objects in the simulated space serve as points of reference for the player and give sense to the movements of the player character. They create virtual interactions that simulate real-world interactions. The player character works as a “tool” for experiencing the game world on the same level as the player would experience the real world.

It is important to note that the in-game world is perceived by the player in an active form. For example, movies are perceived in a passive way since the spectators have no effect on what is happening on the screen. Games are the opposite, the player explores the game world via real-time control of the player character, which makes game feel an active perception. This adds to the definition of game feel started previously: “Real-time control of virtual objects in simulated space” (Swink, 2009).

But there would be no game feel without simulated responses from the objects that are provided with effects. This leaves the last fundamental part of game feel.

4.3.2.3 Polish

Polish can be defined as “any effect that artificially enhances interaction without changing the underlying simulation” (Swink, 2009). This includes visual effects, camera shaking, particles, affecting the movement of the player character,

animations, and sounds. All these effects help to highlight the characteristics of the objects making the interactions with them look and feel more realistic.

The effects may seem unnecessary but they make the games feel the way they do. Without animations, sound effects, particle systems, and many other modifications the games would look very static and less convincing. They would lose their appeal to the player.

This last part provides an almost complete definition of game feel: “Real-time control of virtual objects in simulated space, with interactions emphasized by polish” (Swink, 2009).

4.3.2.4 Game Feel in Different Game Genres

After defining the game feel it is essential to note that not all the games have the same game feel. Swink developed a scheme that helps classify games by the three building blocks of game feel.

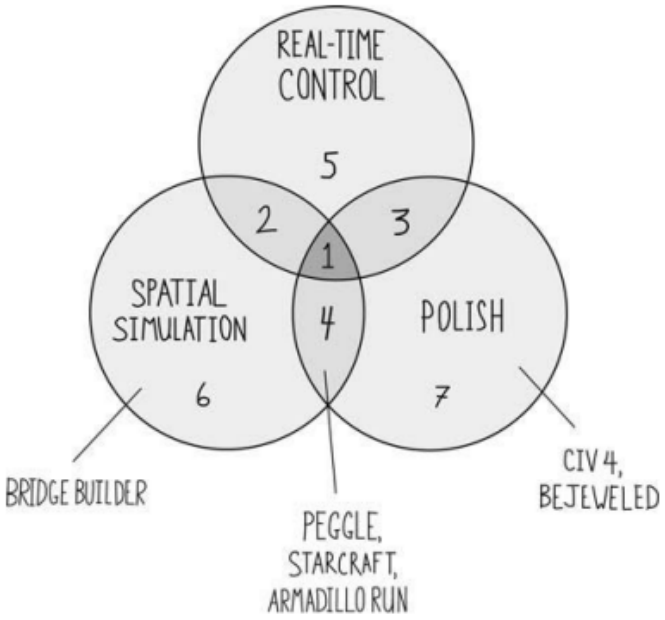


Figure 17: The intersection of the building blocks (Source: Swink, 2009)

Figure 17 represents the intersection of the three building blocks that create different combinations of the game feel. Some games may have just one or two of the building blocks, but this does not mean that they do not have game feel. These games also have a game feel, but it is different from the game feel that is created by

all three building blocks. This project focuses on games that belong to the number 1 on the scheme, meaning that they have all the fundamental characteristics of game feel.

4.3.3 Further Definition of Game Feel

As mentioned above, this project is focused on the game feel that has three building blocks: real-time control, simulated space, and polish. But these aspects are not the only characteristics that form a good game feel, it also consists of experiences.

Swink defined the five most common and important experiences of game feel (Swink, 2009), which are the following:

1. The aesthetic sensation of control.
2. The pleasure of learning, practicing and mastering a skill.
3. Extension of the senses.
4. Extension of identity.
5. Interaction with a unique physical reality within the game.

In his book, Swink talks about personal experiences since this is what experience is about. But in order to understand these characteristics it is necessary to describe each experience in a general way.

The first experience, *the aesthetic sensation of control*, is the feeling of joy the player gets from simply controlling the player character and moving around. It can be compared to the joy of riding a bicycle, skating or dancing, or any activity that involves movement. The player does not need to be experienced or good at the game, the pleasure comes from the simple ability to move around the game world. It is essential to create a character controller that provides the joy of simple movements in order to engage the player. The movement of the player character is the first thing the player learns when they start a new game, and if the movement feels too slow, too fast, interrupted, unresponsive, clumsy, or restricted, the player will likely not engage with the game.

The second experience, *the pleasure of learning, practicing and mastering a skill*, is strongly related to the first one. It happens when the player gains confidence in controlling the player character and starts mastering the mechanics of the game. Some mechanics can take less time to master, some more, but the satisfaction of mastering a skill is a feeling that makes the player engage more with the game. Otherwise, if the player spends a lot of time learning a skill and still cannot master it or at least feel any improvements, this will cause frustration and can make the player abandon the game.

The third experience, the *extension of the senses*, is compared by Swink to driving a car. When a person learns to drive a car, at first they do not realize the extensions of the car and may bump into things while driving or parking. But after some practice, they start “feeling” the car around them. It is based on previous experience and turns into intuition. The same happens with games. When a player spends some time playing the game, they get a better understanding of the dimensions of the player character and some movements become intuitive. This may include the player leaning to the sides while trying to cover up the player character from the enemy’s fire, they unconsciously move their body in the same way they move the player character.

The fourth experience, *the extension of identity*, is very similar to the previous one. It happens when the player starts associating the player character with themselves. For example, instead of saying “My character got shot!” they say “I got shot!”. The player can fully submerge into the player character identity for some time and then get taken out by an unexpected event.

The fifth experience, *the interaction with a unique physical reality within the game*, is strongly related to the polish aspect of the game feel. It is very important to make the objects in the game world feel different for the player. Some objects may be heavier than others, some surfaces may be bouncy, and some slippery. In the real world, not everything is the way it looks: sometimes an object looks heavy but is very light, and vice versa. These seemingly small details help build a more complete and satisfactory experience for the player.

4.3.4 Game Feel as a Skill

As described above, game feel is about the experience the player gets from the game. The experience is based on several factors, and the most important of them is the control of the player character. Character controller is an essential part of a game and it is the most important thing in terms of player engagement.

When a person starts playing a new game, the controls may feel weird, clumsy, and unresponsive. It is normal when people start learning a new kind of activity they always feel not confident and make mistakes at first. But with time they gain experience and get better at this activity, the same should happen with games. After a certain amount of practice, the player should feel more confident controlling the player character and notice how some mechanics that seemed difficult in the beginning, now are easier for them. In some cases, the players cannot fully appreciate and enjoy the game without reaching a certain level of skill and mastering some mechanics. “In this sense, skill is the price of admission for game feel” (Swink, 2009).

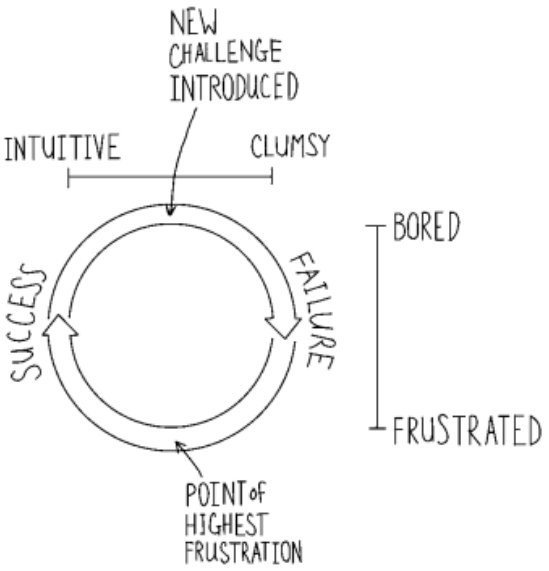


Figure 18: The cycle of skill and game feel (Source: Swink, 2009)

However, mastering a skill is not the only aspect that creates a good game feel. Some players can master a skill in a game but still not enjoy the pure control of the

character and only keep playing the game because of the aesthetic appeal. This means that a good game feel is a combination of aesthetic pleasure and the pleasure of learning and mastering a skill. The connection between these aspects of a game creates a cycle that is represented in Figure 18.

When a player starts a new game, they will be bad at it in the beginning, but they believe that with time and practice, they will get better and be able to fully enjoy the game. That is why it is important to create a character controller that can be learned and mastered. Creating a controller that is too complicated will simply frustrate the player when they realize they cannot get better no matter how hard they try. There will be cases when a player cannot get better for individual reasons, but this should not be the case for the majority of players.

Creating a good game feel and helping the player learn new mechanics without making them frustrated is the key part of getting them into the flow state. The flow state is achieved by balancing the player's skill and the challenge that the game offers.

If the skill level is too low or the challenge is too difficult, the player will feel frustrated and may abandon the game. The same can happen if the challenges are not engaging enough, the player will get bored.

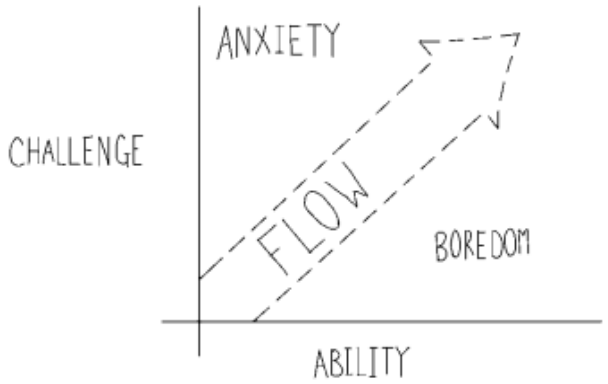


Figure 19: The flow state (Source: Swink, 2009)

The flow state is achieved by correct design decisions and balancing the challenges, this project is focused on part of the flow that is achieved with a good character controller.

4.3.5 Intuitive Controls

The importance of intuitive controls cannot be overestimated. Correctly assigned controls make the games feel more fluid and help players learn and master skills faster. Uncommon or incorrect controls may frustrate the players and make them feel like they cannot control the player character well enough, this may happen also when a mechanic is not assigned to the “usual” control. For example, not assigning the “Jump” to the “Space” key on the keyboard.

Intuitive control means that the intention of the player is translated almost perfectly into the game (Swink, 2009). When the controls are not intuitive, the feeling becomes the opposite. The players may feel that their intentions are reversed or changed by the game and this could lead to frustration. Making the controls intuitive does not mean making them easy, but making them feel familiar to the player.

There are standards for the controls that were established throughout the video game history. This leads to the next two parts dedicated specifically to the two main types of input in video games.

4.3.6 Mouse and Keyboard

Mouse and keyboard setup can be used to play any genre of video game as long as this specific game has this option. It can be more or less comfortable, and the level of comfort depends on several factors. Keyboards provide a wide range of input, since the whole keyboard can be used, making it more useful for games that have a lot of mechanics and actions. These are usually such game genres as Real Time Strategies or Massive Multiplayer Online Role-Playing Games. These game genres tend to be made for the PC platform and therefore most of the players use a mouse and keyboard.

The mouse provides more precision, which can be important for some game genres like First-Person Shooter or Third-Person Shooter. Using a mouse to play these games makes aiming easier and provides a competitive advantage for the players. It

can also feel more intuitive to control the camera by simply moving the mouse in the necessary direction.

4.3.7 Gamepad

Gamepads, or controllers, are mostly used for the games that are played on consoles, however nowadays they can be used to play on the PC too. Gamepads are more portable and comfortable than mice and keyboards since they are designed to fit in humans' hands. They do not have a wide range of input options, but some of these inputs can be more precise than the ones from mouse and keyboard. For instance, changing the player character movement from walking to running can be achieved by pushing the joystick further: the analog joysticks detect the amount of pressure used and can provide a more accurate representation of the player’s actions than a binary input. Meanwhile, the keyboard only provides a binary input, the key is either pressed or not.

Gamepads can be used for games that require more movement precision and make the player feel more connected to the player character since the actions are represented more precisely. They are also more ergonomic and can be more comfortable for players who do not like sitting at a desk while playing.

4.4 Animation Controller

The Animation Controller component in Unity allows the users to organize and link the animation clips for every character in the game. It is a visual graph that represents all the animations used for specific characters and the variables that affect these animations. Animation controller allows to change the look and feel of animations with the speed of animation, transitions between animations, etc.

The animation controller itself can be seen as a state machine where the character has an animation (or several animations) for each state. The changes between the states are usually controlled by the code of each character.

Animation controllers, also known as *Animation Trees*, can have different parameters that the user defines and affect the animations in different ways, for example, the speed of an animation, a trigger that starts a certain animation, and a boolean variable to check the state of an animation.

Each *Animation Tree* has an entry and an exit, the entry transitions to the animation that will be first played when the character is activated. In most cases, it is a default animation or *Idle* animation, the animation that is played constantly when the character is not doing any specific actions. The idle animation is the “heart” of an animation controller since it is the animation that most of the animations transition to. The exit state is used to stop all the animations at a certain animation layer.

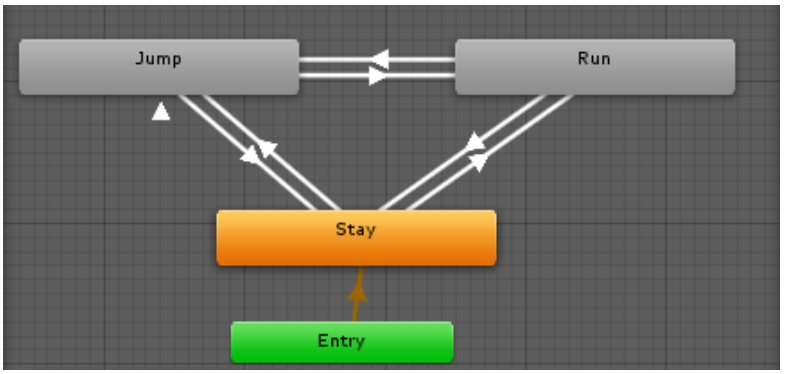


Figure 20: An example of a simple animation controller (Source: Precise Animation Control - Questions & Answers, 2019)

4.4.1 Animation Layers

Animation layers are used for complex animations that involve using different state machines for different body parts of the character. Usually, there would be a base layer for the basic actions, and a separate layer for more specific actions, like shooting or fighting. The layer can affect only some parts of the character, for example, the torso, the arms, the legs, or the head. There are two modes of blending the layers: override, meaning that the base animations will be replaced by the animations from a different layer, and additive, meaning that the animations from this layer will be added on top of the base layer. Override is used when complete control of a certain animation layer is needed. The weight of each layer can also be changed, making it affect the animations more or less, depending on the value.

In Unity, for the animation layers to work, each layer needs an avatar mask, which is a component that allows disabling animations for specific parts of the body. It is based on the rigged 3D model of the character and gives the freedom to enable or disable movement for all the bones in the model. The base animation layer has a base avatar mask that has all the bones activated, secondary layers tend to use only specific bones for certain animations.

4.4.2 Blend Trees

Blend trees serve the purpose of blending animations that are very similar, for instance, the animations of running in different directions. There are different blend types, 1D Blending and 2D Blending, that have one or two parameters that affect them, accordingly. 2D Blending has more variation as it uses two parameters instead of just one. 1D blending is used for more simple animations where the character can run in different directions but does not lean or change the posture depending on the direction.

There is one more type of blending, which is Direct Blending. It is used for animations that require more precise control and do not need to be blended indirectly, such as facial expressions.

4.4.3 Sub-State Machines

Sub-State Machines are used for animations that can form a set to represent a joint action. The complete animation can have several stages and these stages can be grouped into a *Sub-State Machine*. For example, a fall and land animation can have falling, landing, and standing up stages. A shooting animation can consist of crouching, aiming, shooting, and getting up. Having all these animations in *Sub-State Machines* can also help with the visual understanding of an *Animation Tree* since there will not be that many separate animations and transitions.

Sub-State Machines can also be used to diversify the animations by randomizing them. For instance, the damage animation should not always be the same, there should be several damage animations that are chosen randomly every time the character is hit.

5. Methodology

The methodology used for this project is *Agile Incremental*. This methodology allows the creation of a solid structure for a project, based on *milestones* that are distributed along the process of development. Having these milestones allows the developer to better understand the amount of work, distribute it well, and set clear goals for each milestone.

Agile Incremental implies delivering a functional piece of software for each version, that will later form a complete product. The whole development process will be divided into four parts: analysis of the game genres; analysis and definition of the mechanics for each preset; implementation of all the presets; and finally publishing and documentation.

In order to keep track of the tasks that need to be accomplished for each milestone, the application *Trello* will be used. Trello is very helpful in terms of organization because it allows one to keep track of each task and its status (e.g. To Do, In Progress, Done).

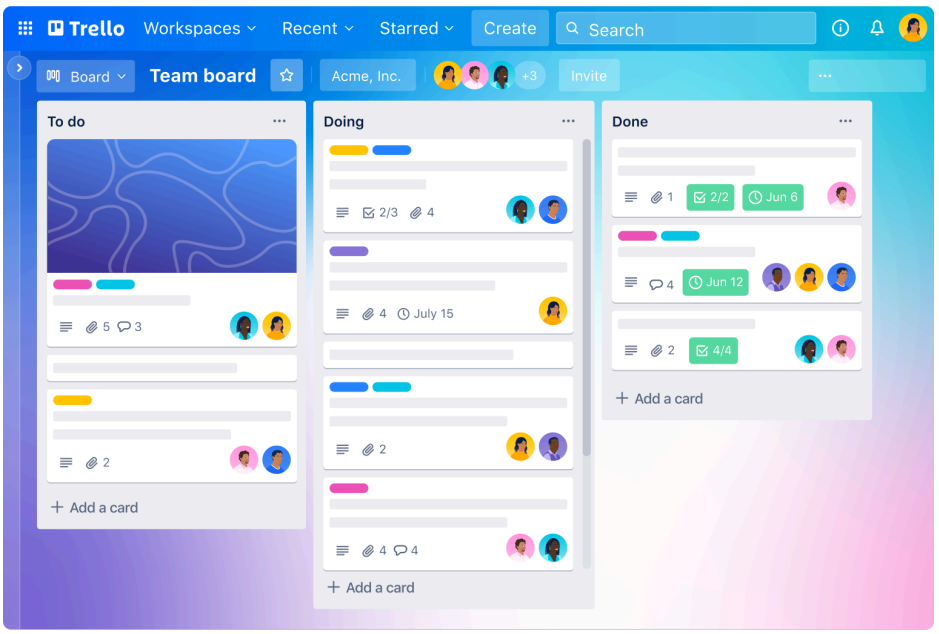


Figure 21: An example of a Trello Board (Source: trello.com)

5.1 Schedule

In order to represent an approximate distribution of work, a Gantt diagram was created. This diagram is based on three big milestones of this project: The Initial Submission, Second Submission, and the Final Submission.

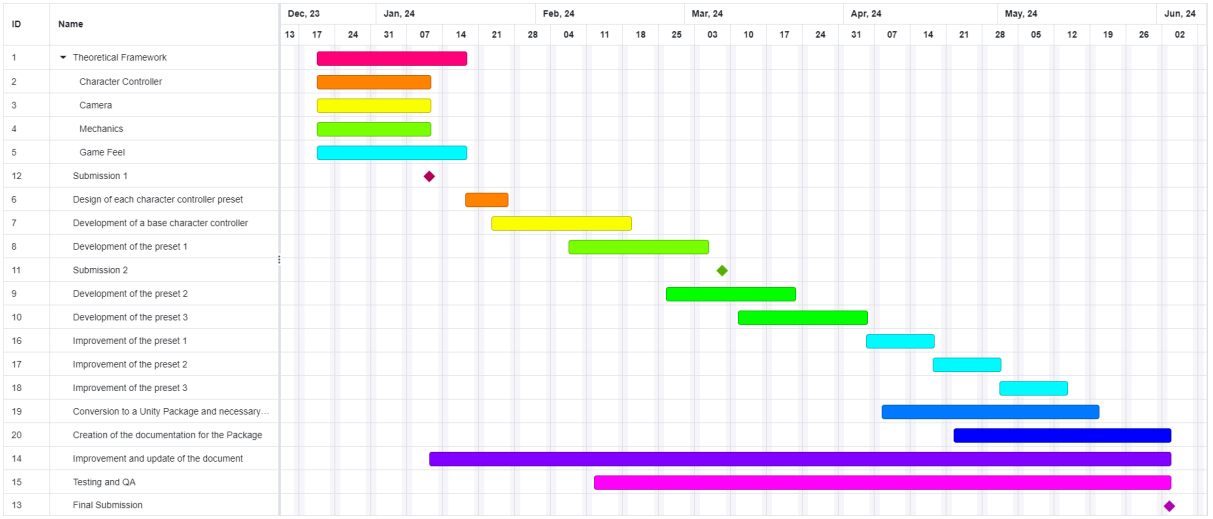


Figure 22: Gantt diagram of the development process (Source: own elaboration).

5.2 Software

The software used for the development of this project is *Unity 3D*, specifically version LTS 2022.3.18f1, *ProBuilder* package for *Unity 3D,* and *Microsoft Visual Studio 2022*. In order to control the versions of the project and have access to it from multiple devices, a GitHub repository was created and every change in the project will be uploaded to the repository.

The 3D model and animations were acquired from the website *Mixamo.com* which allows royalty-free use of all the content available.

5.2.1 Unity and Unity Asset Store

Unity is a video game engine developed by *Unity Technologies* and first released in 2005. It allows the creation of video games for different platforms, such as Windows, Mac OS, Linux, WebGL, PlayStation, Xbox, Nintendo Switch, iOS, Android, and several VR platforms. The games developed with this engine can be 2D or 3D, but it

can also be used to create interactive experiences outside of the video game industry.

Unity Engine is written in C++ programming language, but its primary scripting system uses C#, making it more accessible for different users, since C# is a higher-level language than C++.

The users can develop their own scripts and assets or can use the ones available in the *Unity Asset Store*. It is a website that belongs to Unity Technologies that allows developers to publish their assets and make them accessible to other users, whether for free or by paying a required amount of money. The assets vary from 2D sprites to complex game systems. This project is intended to be a package available in Unity Asset Store.

6. Development Process

In this chapter, the whole process of the design and development of all three presets will be described. In the first two sections, the game genres and mechanics will be researched and analyzed, and the mechanics for each preset will be determined. In the third section, the creation and functioning of the controllers will be described. Finally, in the last section, the creation of documentation and publishing process of the package will be explained.

6.1 Analysis of the Game Genres

The game genres that will be used for this project are analyzed from the perspective of mechanics and user experience in order to determine the characteristics that define them and be able to define the mechanics that are necessary for each genre further on. The analysis will be mostly based on the games that are used as referents for this project (chapter 3. Referents).

6.1.1 Third-Person Shooter

Similar to First-Person Shooters, Third-Person Shooters tend to have a rapid and responsive character controller. The main example used for this analysis is *Mass Effect 3* (BioWare, 2012). There is almost no delay between the action of the player and the response of the character, all the movements feel smooth, the use of objects and interaction with them is fast, and the feedback is immediate. The input sensitivity in these games and the response sensitivity are high, but the games tend to smooth out the movement to make it feel more natural.

In terms of physics, some third-person shooters can be more realistic, while others can use a not realistic gravity or have a higher jump force, but in any case, the control does not feel slow or “floaty”, it is always very responsive.

The camera is usually positioned behind the character over their shoulder, offering this way an unobstructed view of what is in front of the player, but also showing the character’s avatar, unlike the first-person shooters.

The camera is controlled with the mouse and the character is controlled with WASD. In the case of gamepads, the camera is controlled with the right joystick and the character with the left joystick.

The sensitivity of the controls on the mouse and keyboard tends to be different from the sensitivity on gamepads. There is an established speed for walking and running on the keyboard, while on the gamepad it is a range of values, from -1 to 1 on both axes. In terms of the camera, the sensitivity can vary on both, the mouse and the gamepad.

6.1.2 3D Platformer

3D Platformer is one of the most varied game genres. There are many different features and game feels to each game, from the classic *Super Mario 64* (Nintendo, 1996) to *Fall Guys* (Mediatonic, 2020). These games have completely different game feel and different objectives, but they are both 3D platformers. The needs of each 3D platformer game can be very different, but the main characteristics remain the same for most of them: the character is positioned in the center of the screen, the camera follows the character, and the character can move in 3 axes.

The game feel of the 3D Platformer games tends to be light and “floaty”, but it should not feel slow or monotonous. The movement is very responsive and provides a feeling of freedom and satisfaction. The jump is usually not limited to only one fixed jump but has variation to it. It can depend on how long the input button is held and has different types, like double jump and wall jump.

The camera control is also very important for 3D Platformers: the camera has to be dynamic and is usually controlled by the player to give them the possibility to find the best angles and move the player character as they will.

6.1.3 Adventure

Adventure games are based on the exploration of the game worlds and solving puzzles. While some other game genres may not have much narrative to them, adventure games usually have a strong narrative component and the player takes an active role in this story as a player character. Adventure games were inspired by the narrative kind of entertainment that existed way before video games, like books and

movies, but their main difference is the interactivity. They started as text and graphic adventure games like *Colossal Cave Adventure* (the term “adventure games” comes from the name of this game) and made it so far, to the games like *Starfield* (Bethesda Game Studios, 2023) and *Hogwarts Legacy* (Avalanche Software, 2023).

Because of the fact that adventure games are based on different stories, the variety of mechanics can be huge, from simply climbing a hill to using a magic wand. But all the games have some common main mechanics, which are usually the movement and some basic interactions with the world.

The game feel can vary a lot depending on the game, but as in other genres, there are some common characteristics. The camera tends to be controlled by the player in order to give them more freedom and let them experience the in-game world the way they want.

The control of the character should be dynamic and responsive, but not necessarily as realistic as in some Third-Person Shooters. The movement should be free and the change between the movement states should be as seamless as possible for the best experience.

6.2 Analysis of the Mechanics

The common mechanics for the three game genres will be the basic mechanics like movement and interaction with objects. However, the movement will have a different game feel in each genre. Despite having a different game feel, the basics of the movement will remain the same. The player character should be able to move forward, backward, left and right, run, and jump. These forms of movement are considered the most important and the most basic abilities the player character should have. Some games may restrict the movement in one way or another, but for the most part, the player character should be able to move freely.

For this project, only the mechanics that will be implemented in each preset are analyzed.

6.2.1 Genre-specific Mechanics

The genre-specific mechanics are usually more varied and related to the games themselves. For example, in a third-person shooter, the main genre-specific mechanic is shooting, in a 3D platformer it could be a double jump or gliding, in an adventure game it could be something even more specific to the game, like dropping objects on purpose in *Stray*.

These mechanics add variation to the gameplay and make it more engaging for the player since original mechanics created specifically for a game stand out much more than the common mechanics used in each game of the genre.

For this project, a list of genre-specific mechanics will be defined for each preset of the character controller.

Since the list of genre-specific mechanics can be very long, only the most common mechanics will be implemented for each preset, the users will also be able to program and add their own mechanics if needed.

6.2.1.1 Third-Person Shooter

The mechanics chosen for the Third-Person Shooter are the basic mechanics that are present in most of the games of this genre. They are the following:

- **Movement**: The player moves in four directions depending on the input and the *Forward* vector of the camera. This mechanic includes walking and running. The player character turns to face the direction in which it is moving unless the constant strafe option is activated. The movement is more realistic and the character does not gain the speed instantly.
- **Jump**: The player jumps up with a certain force. The force depends on the time that the jump button has been pressed. If the player jumps while moving, the player character will keep the momentum of the movement. The jump can only be executed if the character is touching the ground and is not crouching. The jump force is low and the fall force is high, which makes the jump more realistic and heavy.

- **Aim**: The camera moves closer to the player character, changes the FOV, and activates the strafing, meaning the player character will rotate following the forward vector of the camera.
- **Shoot**: The player character will shoot once or activate the rapid-fire if the shooting button is being pressed. Different types of weapons are not implemented, however the fire rate and the damage can be changed.
- **Crouch**: The player character will start crouching once the button is pressed, and will stop crouching if the button is pressed again. This slows down the movement but allows the player to avoid possible attacks and have more precision while shooting.

6.2.1.2 3D Platformer

The 3D Platformer games can have a huge variety of mechanics, so the ones chosen for this preset are considered to be the most basic:

- **Movement**: The player moves in four directions depending on the input and the direction of the camera, the mechanic includes both walking and running. The player character turns to face the direction in which it is moving and there is no strafing.
- **Jump**: The player jumps up with a certain force. The force depends on the time that the jump button has been pressed. If the player jumps while moving, the player character will keep the momentum of the movement. The jump force is high and the fall force is low, giving the jump a “floaty” feeling. The jump cannot be executed while the player character is crouching.
- **Double Jump**: The player can only use a double jump while in the air and the player character has not started falling yet.
- **Crouch**: The player character will start crouching once the button is pressed, and will stop crouching if the button is pressed again. The movement speed is lower, but it allows the players to get into the less accessible parts of the level.
- **Dash**: The dashing mechanic gives the player character an impulse of force in the direction of the movement. It is only applied during a short period of time and can be done during any type of movement or even from the idle state.

6.2.1.3 Adventure

Since Adventure games tend to base their mechanics on the story of the game, it is hard to determine what is considered common mechanics for this game genre. However, there are some mechanics that are present in many Adventure games and they are the following:

- **Movement**: The player moves in four directions depending on the input and the direction of the camera, the mechanic includes both walking and running. The player character turns in the direction of its movement, there is no strafing.
- **Jump**: The player jumps up with a force that depends on the input. The longer the jump button is pressed, the higher the jump is. If the player jumps while moving, it keeps the movement inertia.
- **Gliding**: The player can use a glider after they jump or start falling down. It slows down the movement and allows the player to descend slowly.
- **Climbing**: The player character can climb certain surfaces. It moves horizontally and vertically based on the *Up* and *Right* vectors of the player character itself. Once the player character reaches the top, it climbs up automatically.

6.2.1.4 Summary of the Analysed Mechanics

After determining the set of mechanics for each preset, a representative table of all the mechanics was generated and is represented below.

Mechanic	Third-Person Shooter	Adventure	Platformer
Movement			
Jump			
Crouch			
Aim			
Shoot			
Glide			
Double Jump			
Climb			
Dash			

Table 1: All the mechanics and their implementation in each genre (Source: own elaboration).

6.3 Implementation

After the analysis of the genres and the definition of the mechanics for each preset, the whole implementation and development process will be described. It will include the input system, the camera, the basic character controller, and the three presets. All the source code and the presets will be available in the annexes.

This section will be divided into several subsections: throughout the implementation process, there were different stages of the product, leading to the final version. All the stages will include a description of the parts that were implemented during each of them. The software used for the implementation is described in section 5.2.

6.3.1 Version Ⅰ: Basic Character Controller

Given the extension of the project, the implementation started with the creation of a basic character controller that later gave a starting point for the three specific controllers. This subsection also includes the implementation of the input system and the camera.

6.3.1.1 Input System

The input system used for this project is a package called *Input System* created by *Unity Technologies*, which allows the creation of Action Maps for different devices, defining the actions, key binding, and properties of the actions.

The *Unity Events* call user-defined methods to execute certain actions when an input is received. The *Action Maps* bind the events with the input.

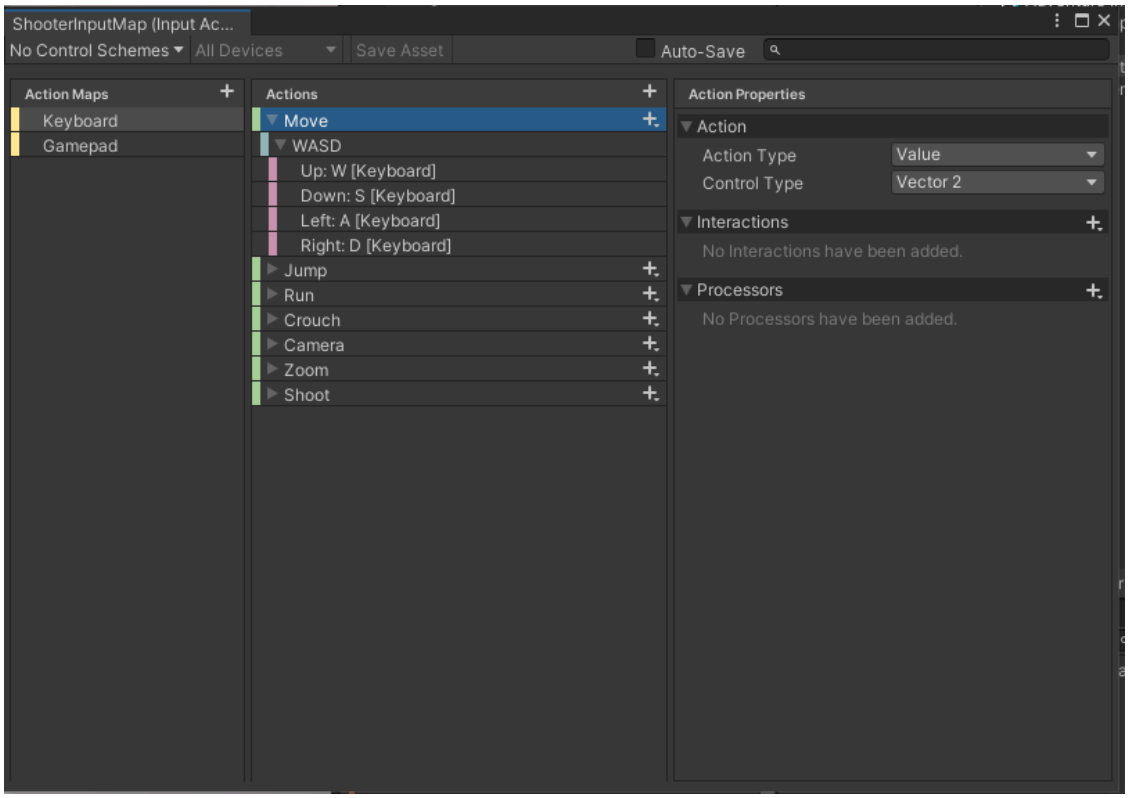


Figure 23: Input Action Map (Source: own elaboration).

The *Input Maps* can have different *Actions Maps* for different devices, in this case, these are Keyboard and Gamepad.

The *Player Input* component that is added to the player character in the Inspector allows to assign a default *Action Map* and connect all the *Events* with the methods.

Figure 24: Player Input component with Events (Source: own elaboration).

The events are separated into the Keyboard and Gamepad ones, but there are also general events that allow to change controls in case a device is connected or lost.

The input is controlled by a script called *PlayerInputManager.cs*. This script detects what device is connected when the scene is executed and sets an according *Input Map*. It also checks the connected devices during runtime to change the *Input Map* if a device is lost or a new device is detected.

6.3.1.2 Camera

The camera's movement was programmed from scratch to make it fully customizable and accessible to the users.

```
//Variables accessible from the inspector
[Header("General")]
[SerializeField] private Transform cameraPivot;
[SerializeField] private Transform cameraPosition;
[SerializeField] private Transform cameraWalls;
[SerializeField] private Transform zoomPosition;
[SerializeField] private Transform cameraPositionStart;
[SerializeField] private GameObject player;
[SerializeField] private float minDistance = 5f;
[SerializeField] private float maxDistance = 10f;
[SerializeField] private float minPitch = -20;
[SerializeField] private float maxPitch = 90;

[Header("Invert Controls")]
[SerializeField] private bool pitchInverted;
[SerializeField] private bool yawInverted;

[Header("Mouse")]
[SerializeField] public float yawRotationalSpeedMouse = 40;
[SerializeField] public float pitchRotationalSpeedMouse = 40;

[Header("Gamepad")]
[SerializeField] public float yawRotationalSpeedGamepad = 60;
[SerializeField] public float pitchRotationalSpeedGamepad = 60;

[Header("Player")]
[SerializeField] private LayerMask avoidObjectsLayerMask;
[SerializeField] private float offset = 0.1f;
[SerializeField] private PlayerInputManager playerInputMng;
[SerializeField] private BaseMovement playerMovement;
```

Figure 25: Variables accessible from the inspector in the *CameraMovement.cs* script (Source: own elaboration).

The camera uses several *Transforms* located in the Player *GameObject* in order to calculate the correct position depending on the location of the transforms and the set distance to them. The *Transforms* are located differently in each preset and can be adjusted to achieve different camera placements (e.g. centered or displaced). The variables *MinDistance* and *MaxDistance* regulate the distance between the camera and its desired position. It also can be adjusted in each preset.

There are different variables that affect the movement of the camera, such as rotational speed in two axes, Yaw and Pitch; minimum and maximum distance between the camera and the player, and minimum and maximum Pitch that are used

to limit the camera's angles. The rotational speed is different for the mouse and gamepad since they have very different game feel. The axes can be inverted and there is an option for it available in the inspector.

The camera also detects objects that are marked with a certain *LayerMask* in order to avoid them and reposition itself if a collision occurs.

For each preset, a separate camera *GameObject* was created. They all function the same way but have different *Transforms* as points of reference and different variables, such as distance and rotational speed. Creating several cameras facilitates the process of switching between controllers and allows the creation of customized cameras.

6.3.1.3 Base Character Controller

The base character controller is a simple third-person controller that does not have any genre-specific mechanics, giving a starting point for the customized presets. It consists of a player character and a camera that follows it. The player character is represented by a capsule, that in customized presets will be replaced by a 3D human model. Since the camera and the player character are separate objects, the player character contains points of reference for the camera. These points are the following:

- *Camera Pivot* is used as the center of rotation for the camera.
- *Zoom Position* is used as a point of reference when the camera is displaced from the center to the side of the character to create a zoom effect.
- *Camera Position* is used as a reference point for the general positioning of the camera.
- *Camera Position Start* is used as the initial position of the camera before it is moved to the *Zoom Position*.
- *Camera Walls* is a position from which a Ray is cast to detect the walls.

Figure 26: Reference points located in the character controller *GameObject* (Source: own elaboration).

The camera's script has several variables that affect its positioning and movement. These variables are the minimum and maximum distance between the camera and the player character, the minimum pitch and maximum pitch that limit the camera's rotation in the X axis, the rotational speed for the Y and X axes, and finally the *LayerMask* for the objects that the camera should detect and avoid. These variables make the camera's movement smoother and more comfortable for the player. They are also adjustable and vary depending on the input device (e.g. mouse or gamepad).

The character controller itself also has variables that affect the movement. There are such variables as Walking Speed, Running Speed, Jump Force, and Gravity Force. The player character also detects different surfaces using *RayCasting* and *LayerMask* to determine whether it can jump on this surface, or not. These are the basic components of the player character movement script that can be adjusted in order to achieve a better game feel.

6.3.2 Version Ⅱ: Implementation Approach

In order to find the best implementation approach for this project, several tests were conducted where different systems were created.

The first system was based on a Finite State Machine (FSM) structure to control the states of the player character. Each state would control the input, physics, and animations that it would include. This structure allowed the creation of independent states for each preset and made the code modular, where one state could be removed without affecting the general functioning of the character too much.

However, during the design of the three controllers and the implementation of the base character controller, some issues were encountered. The main issue was the overlapping states and the repetition of the same pieces of code in different states. Additionally, the created structure was hard to understand and could create more problems as the project grew bigger.

While every state was supposed to be responsible for only one mechanic, the overlapping states were going to implement and execute more than one mechanic. For instance, the normal jump has a different trajectory than the jump in movement, and the possibility to control the character while it is in the air requires reading the movement input while being in the jump state. Another example would be the crouching and shooting states being combined, which would require reading the movement input as well as the shooting input. The overlapping states created the problem of repeated code and made it more complicated to structure and control everything.

The second system was based on creating a parent script that implemented all the basic methods, such as moving, jumping, and zooming in. This script was responsible for the movement of the player character and implemented virtual methods that could be overwritten by the scripts that would inherit from it, in case it was needed.

This system was less structured but allowed more flexibility, provided more control over the functioning of the controller, and did not have the issue of overlapping states. It was also more compact, easier to understand and organize.

After evaluating the advantages and downsides of both systems, it was decided to implement the second one. Its structure will be described in the following section.

6.3.2.1 Base Code Structure

The second iteration started with the creation of the base script which would be the parent script for each genre implemented. The specific scripts created for each genre inherit from it and implement the same methods differently. The main variables are located in the parent script that is called *BaseMovement.cs*, the methods in this script are virtual so the specific scripts can override them to implement a different approach.

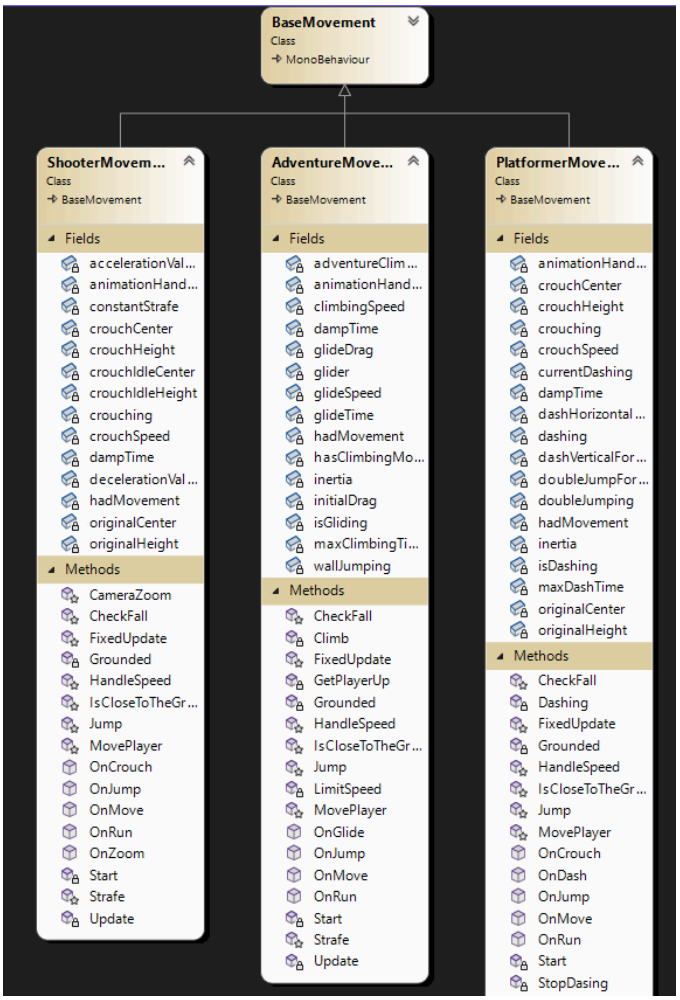


Figure 27: Simplified UML diagram with class hierarchy (Source: own elaboration).

The script *BaseMovement.cs* has public and protected variables and methods. Public variables and methods are exposed in the Unity Inspector tab and are accessible to the user. Protected variables are accessible for the classes that inherit from *BaseMovement.cs*, are not exposed in the *Unity Inspector*, and are used for the internal functioning of the scripts. The variables are separated by their functionality making them easier to read and understand.

```
[Header("Camera")]
[SerializeField] protected Camera camera;
[SerializeField] protected float lerpRotationPct = 0.1f;
[SerializeField] protected Transform feetTransform;
[SerializeField] protected Transform zoomPosition;
[SerializeField] protected Transform cameraPositionStart;
[SerializeField] protected Transform cameraPosition;
[SerializeField] protected Transform cameraIdle;
[SerializeField] protected float cameraDefaultFOV = 60f;
[SerializeField] protected float cameraZoomFOV = 40f;

[Header("Animator")]
[SerializeField] protected Animator playerAnimator;

[Header("Movement")]
[SerializeField] protected LayerMask floorMask;
[SerializeField] protected LayerMask notFloorMask;
[SerializeField] protected float walkSpeed = 3f;
[SerializeField] protected float runSpeed = 6f;
[SerializeField] protected float zoomSpeed = 2f;
[SerializeField] protected Transform capsule;

[Header("Jump")]
[SerializeField] protected float jumpForce = 0.4f;
[SerializeField] protected float currentJumpForce;
[SerializeField] protected float jumpDecrement = 0.1f;
[Range(-2f, -0.0f)]
[SerializeField] protected float fallDetection = -1.0f;
[SerializeField] protected float fallForce = 10f;
[SerializeField] protected float coyoteTime = 0.2f;
[SerializeField] protected float jumpBuffer = 0.2f;
```

Figure 28: Public variables in the *BaseMovement.cs* script (Source: own elaboration).

The methods in the parent script are mostly virtual, giving the children script the possibility to override them. The internal methods are protected, so they are only accessible by the scripts that inherit from the parent. The input methods, however, are made public in order to be accessible for the *Unity Events*.

The *BaseMovement.cs* script implements the following methods:

- *OnMove(InputAction.CallbackContext context)*: A method used for the *Unity Event* that reads the movement input.
- *OnJump(InputAction.CallbackContext context)*: A method used for the *Unity Event* that reads the jump input.
- *FixedUpdate()*: The methods that affect the player's movement and the physics are called here.
- *MovePlayer(Vector3 movement)*: A method that receives the movement vector and sets the *Rigidbody* velocity.
- *CheckFall()*: A method that checks if the player character is falling.
- *Jump()*: A method that calculates the jump force and applies it to the *Rigdbody*.
- *Strafe(Vector3 movement)*: A method that receives the movement vector passed by value and rotates the character depending on whether strafing is active or not.
- *HandleSpeed()*: A method that is responsible for handling the character's movement speed.
- *CollidedWithEnemy()*: A method that is called when the player character collides with an enemy, applies a knockback force to the *Rigdbody* and disables movement for a certain amount of time.
- *IsTouchingTheGround()*: A method that uses *Physics.CheckSphere()* to verify whether the player character is colliding with objects that are marked as ground or not.
- *IsCloseToTheGround()*: A method that uses *Physics.CheckSphere()* to verify whether the player character is close enough to the ground to perform a certain action or change a state.

- *IsTouchingDifferentGround()*: A method that uses *Physics.CheckSphere()* to verify whether the player character is colliding with objects that are marked as not ground.

- *IEnumerator EnableMovement(float time)*: A coroutine that enables the movement after the player collides with an enemy.

All these methods are made virtual, meaning that the scripts that inherit from *BaseMovement.cs* can override them in order to change the implementation. In case they are not overridden, the base method is called.

The methods that affect the player's movement and physics are called in *FixedUpdate()* since the physics simulations are executed with the same frequency as the *FixedUpdate()* method.

6.3.3 Version Ⅲ: Development of the Specific Presets

The basic mechanics were determined during the creation of the Base Character Controller. These are horizontal movement, jumping, and camera control. However, each genre requires different mechanics not present in other genres. The addition of new mechanics requires the addition of new scripts that inherit from the original script, *BaseMovement.cs*. New scripts are created in order to separate the functions of each genre and make the code easy to understand and modify. The process of creation and functioning of all the presets will be described in the following sections of the paper.

6.3.3.1 Third-Person Shooter

The Third-Person Shooter controller is the most complex since it has more mechanics and interactions than other controllers present in this paper. The game feel chosen for this controller is realistic in terms of physics and responsiveness.

The mechanics chosen for this preset are based on the most common mechanics used in the games of this genre (6.2.1.1 Third-Person Shooter). These include running, crouching, aiming, and shooting. To implement these mechanics two scripts were created: one is *ShooterMovement.cs* which inherits from the *BaseMovement.cs* script, and the other one is a specific script for the shooting mechanic.

The *ShooterMovement.cs* script implements all the actions that affect the player's movement. Some of these methods are overrides for the original actions in *BaseMovement.cs*, such as *OnMove()* which reads the movement input, *OnJump()* which reads the jump input, and *OnZoom()* which is responsible for the aiming. Other methods are unique to this script: *OnRun()*, and *OnCrouch()*. All these methods are later bound to the corresponding actions in the inspector.

The movement works the same way as the movement in the *BaseMovement.cs* script. The *OnMove()* method reads the input, which is multiplied by the camera vectors depending on the axis: for the Y axis it is the camera's *Forward* vector, and for the X axis it is the camera's *Right* vector. The result is added to a *Vector3 movement* in the *FixedUpdate()* method and this vector is set as the player's *Rigidbody* velocity in the X and Z axes.

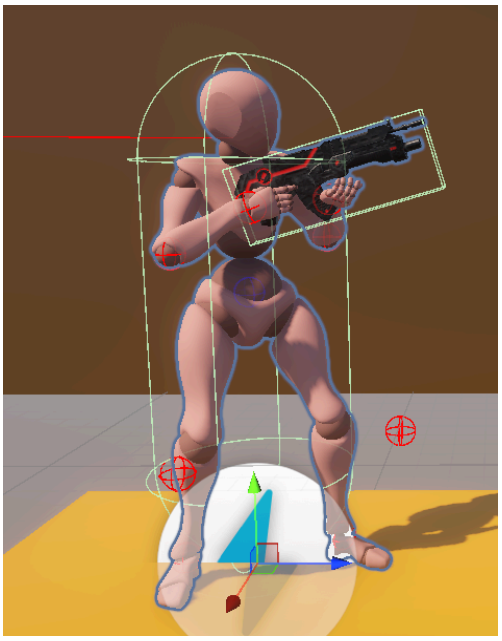
The jump force is also calculated in the *Jump()* method which is later called in *FixedUpdate()*. An initial force is assigned to the *currentJumpForce* variable when the *OnJump()* action is called. The force and the height of the jump depend on the input: the longer the jump button is being pressed, the higher the jump. The *currentJumpForce* decreases with time, preventing this way the character from jumping higher than the established limit. The variable called *jumpDecrement* determines how fast the jump force decreases over time. Apart from the gravity, there is a Fall Force applied to the player character when it starts falling to create a more realistic game feel.

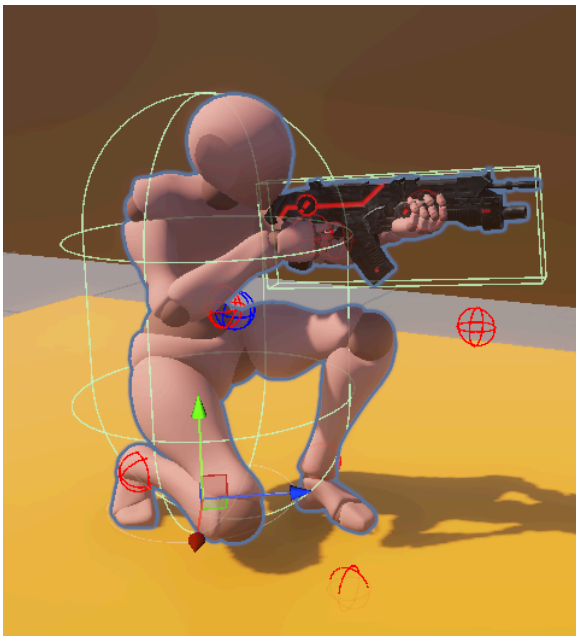
The *OnZoom()* method is responsible for detecting when the zoom input is received and changing the necessary variables. When zoom is pressed, aiming and strafing are set to true, making the player character always look in the same direction as the camera. When the aiming is inactive the player character stays in place and only rotates in the same direction as the camera's forward when it is moving. This method also allows moving the camera to the specified position for aiming when the key is pressed and to its default position when the key is released.

Other methods that affect the player character movement are *OnRun()* and *OnCrouch()*. The *OnRun()* method increases the movement speed while the key is pressed. However, it is not implemented for the gamepad. This is due to the fact that

the movement speed on the gamepad is changed depending on the joystick tilt amount, there is no dedicated run button.

The *OnCrouch()* method sets the crouching mode once the key is pressed and stops the crouching mode when the key is pressed again. It does not need to be pressed continuously, unlike the running key. It also changes the size of the capsule collider to make the player character smaller when it is crouching. When the crouching is stopped, the collider is set back to its original size.





Figures 29 and 30: Original size of the capsule collider and crouched size of the capsule collider (Source: own elaboration).

The other script that was created for this preset is *Shoot.cs*, which is only responsible for the shooting mechanic. It has one action *OnShoot()* that is called when the right input is received. This action calls a method that casts a *Ray* from the camera to the center of the screen, where the crosshair is located. If the *Ray* hits a surface that is a valid surface layer for shooting, a decal is created on the spot where the bullet is supposed to hit. There is also a *GameObject* instantiated to represent a bullet. The bullet gets destroyed if it enters a collision with a wall or floor, and delivers damage if it collides with an enemy. This *GameObject* has its own script, *Bullet.cs*, that is in charge of managing its functioning.

The camera in this preset is displaced to the right, leaving the player character in the left part of the screen. When the player aims, the camera is moved to the Zoom Position which is located more forward than the default position. This changes the player's field of view and creates an effect of zooming in, without changing the camera FOV.

6.3.3.2 3D Platformer

The 3D Platformer preset is focused on the game feel and the jump. The controller should feel snappy and responsive, the jump should feel light and cartoony.

The mechanics used for this controller are simple: the character can move, run, jump, and make a double jump.

The player character movement is implemented in the script called *PlatformerMovement.cs*. The *OnMove()* method reads the input that is later multiplied by the camera's *Forward* and *Right* vectors, and the result is set as the *Rigidbody* velocity in the *MovePlayer()* method. The main differences in the movement are the speed and the way the *OnRun()* method works. In the Third-Person Shooter preset the key has to be maintained in order to run, while in the 3D Platformer preset the running is activated and deactivated by pressing the key. It is made for the player's comfort since it is more common to be moving quickly all the time in the platformers rather than the shooters.

The *OnJump()* and *Jump()* methods are implemented in a different way too. There is a simple jump that is executed when the jump button is pressed for the first time, and there is a double jump that is executed when the jump button is pressed and the player character is in the air. It has to be done quickly because the double jump will not be executed if the player is already falling down. The double jump force is different than the initial jump force and can be changed in the inspector. The jump force is higher than in the Third-Person Shooter and the fall force is lower, which gives the controller a lighter and more bouncy game feel.

The *OnDash()* method is unique to this preset, it gives the player character a force impulse when a certain input is received. It can be executed from any other state of

movement, even jumping and falling. The force of the dash can be also adjusted in the inspector.

The *OnCrouch()* method starts the crouching mode when the key is pressed and stops the crouching mode when the key is pressed again, it also changes the size of the collider so the character can fit into smaller spaces.

The camera in this preset is centered and has a varied horizontal distance, meaning the player has a zone where they can move without the camera following them directly. However, the vertical distance is fixed, meaning that the camera will move immediately when the player jumps.

6.3.3.3 Adventure

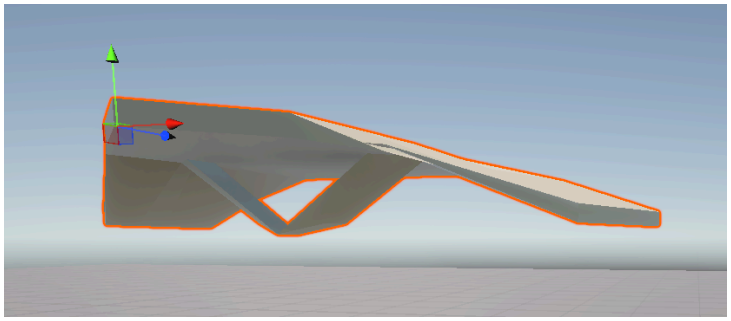
The Adventure controller is more varied since the adventure games usually have very specific mechanics depending on the story and the setting of the game. However, there are some mechanics that are used more than others, such as climbing and gliding.

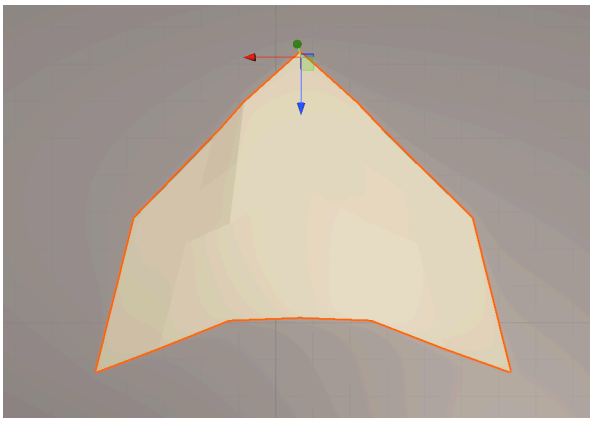
The movement is implemented in the script *AdventureMovement.cs*. The *OnMove()* method provides the input that is used to calculate the direction vector and apply it to the *Rigidbody* velocity. The *OnRun()* method allows the activation and deactivation of the running mode, just as in the 3D Platformer.

To implement the gliding mechanic a method called *OnGlide()* was created. This method is tied to the Input Map and has the same key binding as the *OnJump()* method. It can only be activated when the player character is in the air or is falling, meaning the player can jump off a cliff and then use the glide. The glide is active while the key is being pressed, as soon as it is released, the gliding stops and the player character starts falling. During the gliding, the *Rigidbody* drag is increased to slow down the movement. After the gliding stops the drag is reset to its initial value. The gliding drag value can be changed by the inspector.

To create a more immersive and believable experience, a simple glider was created with the *Unity ProBuilder* package. The glider forms part of the player character *GameObject*, is activated for gliding, and is deactivated when the gliding stops. It is

not accessible from the inspector because the *GameObject* is found through the specific tag “Glider”.





Figures 31 and 32: The glider model (Source: own elaboration).

For the climbing mechanic, a separate script called *AdventureClimbing.cs* was created. This script is responsible for detecting objects that are marked as climbable with a *LayerMask*.

The method *CheckWall()* is constantly checking whether there is a climbable wall in front of the player character. It casts a *Ray* from a certain position in the player’s *Forward* direction. If this *Ray* hits a climbable surface, the boolean variable *wallInFront* is set to true. The other parameter that is checked is the angle between the wall’s *Normal* and the player’s *Forward*: if this angle is smaller than a set maximum angle, the player character can climb the wall. Once these two parameters are true, the climbing process starts.

The climbing itself happens in the *AdventureMovement.cs* script since it forms part of the player character's movement. The movement input is received from the same *OnMove()* method but is used differently. Instead of multiplying it by the camera’s *Forward* and *Right* vectors as it is done in the *MovePlayer()* method, it is multiplied by the player’s *GameObject Up* and *Right* vectors. This is done because when the player character is climbing a wall it is not moving according to the camera’s direction but according to its own axes. Once the climbing movement vector is calculated, it is applied to the *Rigidbody* velocity in the *Climb()* method which is also called in *FixedUpdate()*.

While the player is climbing the gravity is temporarily deactivated to prevent the character from slowly sliding down the wall if the player is not moving. When the player gets back on the ground, the gravity is activated again.

In order to get the player character to the top of the wall that it was climbing, several methods are used. First, the *ClimbToTop()* method in the *AdventureClimbing.cs* script determines whether the player character is close to the top of the wall. This happens if the player is climbing but suddenly stops detecting the wall. Since the detection position is located at the top of the player character, it stops detecting the wall when it is almost at the top.

Then, a method called *TopPosition()* is used to calculate the position where the player character should climb. To do so, a *RayCast* from the detection position in the *Forward* direction of the player is created. To the point of this *RayCast* is added a *Forward* vector multiplied by the necessary distance from the border of the wall and an *Up* vector multiplied by the necessary height from the ground. From this position, a *RayCast* in the *Down* direction is created. The point where it hits the surface is the position where the player character will be moved when climbing to the top of the wall. On the image below it is represented by the cyan sphere.

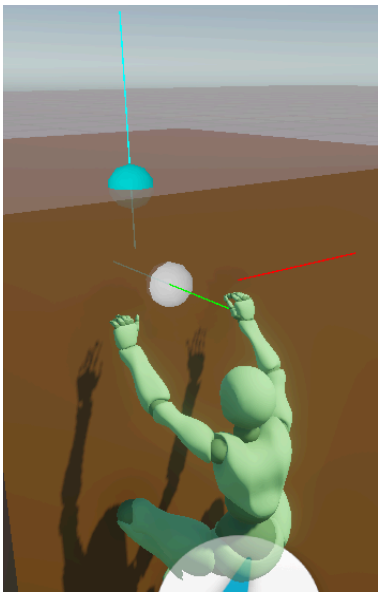


Figure 33: Calculation of the top position (Source: own elaboration).

The method *TopPosition()* returns a *Vector3* that is used in the *AdventureMovement.cs* script as a position to move the player character to.

The normal jump that happens while the player is on the ground works the same way it does in the other controllers. However, there is a variation to both the *OnJump()* and *Jump()* methods that include a wall jump. *OnJump()* reads the jump input and depending on whether the player is climbing or not, a different type of jump is executed.

For the wall jump, in the *AdventureClimbing.cs* script a method called *OffWallJump()* is implemented. This method calculates the *Vector3* for the jump force based on the normal of the wall that the player is climbing and the Up vector of the player character. The obtained vector is applied as a force to the player character's *Rigidbody* in the *Jump()* method.

The camera in one of the references for this genre, the game *A Short Hike* (Adam Robinson-Yu, 2019), is fixed, it was decided to make the camera free for this preset, since it is more common in Adventure games and more convenient for the users. The camera is centered and also has a varied horizontal and fixed vertical distance, like the camera in the 3D Platformer.

6.3.4 Version Ⅳ: Advanced Development of the Presets

During this final stage, an *Animation Tree* for each preset was created and adjusted to achieve the best visual representation possible with the animations acquired from free sources. In addition, the presets were tested with several representatives of the possible target audience, which allowed to adjust the parameters and achieve a better game feel for each preset.

6.3.4.1 Animations

Animations form an important part of the character controllers and add a lot to the game feel. Without the animations the movement of the player character can feel less natural and the special characteristics of each genre would be less appreciated. That is why integrating animations is an important part of this project. All the animations used were acquired from *Mixamo.com*.

A script called *AnimationHandler.cs* was created to control the animations and the parameters that affect them. It is a general script that is used in all the presets to control the animations that function the same way. Some animations depend very much on the movement and cannot be controlled from an external script, so the main *Movement* scripts still affect the animations.

A custom animation controller was created for each preset. The transitions are managed by the parameters created for each controller. Since the animations were acquired from an open source (*Mixamo.com*) and were not custom-made for a specific game, some of them are adjusted to fit the gameplay better. These are such adjustments as the speed of the animations, the rotation of the character, and the transition time between the animations.

The 3D model used for the presets is the same: it is the X Bot model acquired from Mixamo. Based on this model, a *Unity Avatar* was created to later use it for all the animations.

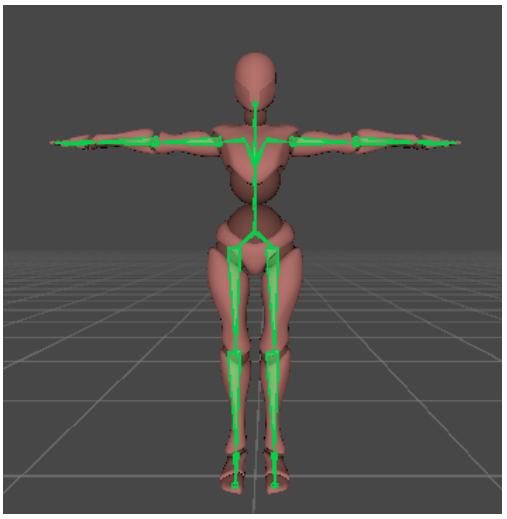


Figure 34: An avatar created from the X Bot model (Source: *Unity Technologies*).

This avatar has all the bones organized and mapped out according to the original model and helps to adapt the animations to this specific model with high precision.

6.3.4.2 Platformer

As can be seen in Figure 35 below, the Platformer animation tree is formed by one *Blend Tree* and six *Sub-State Machines*. The default state is *Movement* Blend Tree which includes three animations: Idle, Walk, and Run. These animations are put in the same state since *Blend Trees* allow smooth transitions between animations based on different parameters, in this case, it is the character's movement speed.

Since the Animation Tree is very extensive, only some states will be explained in detail.

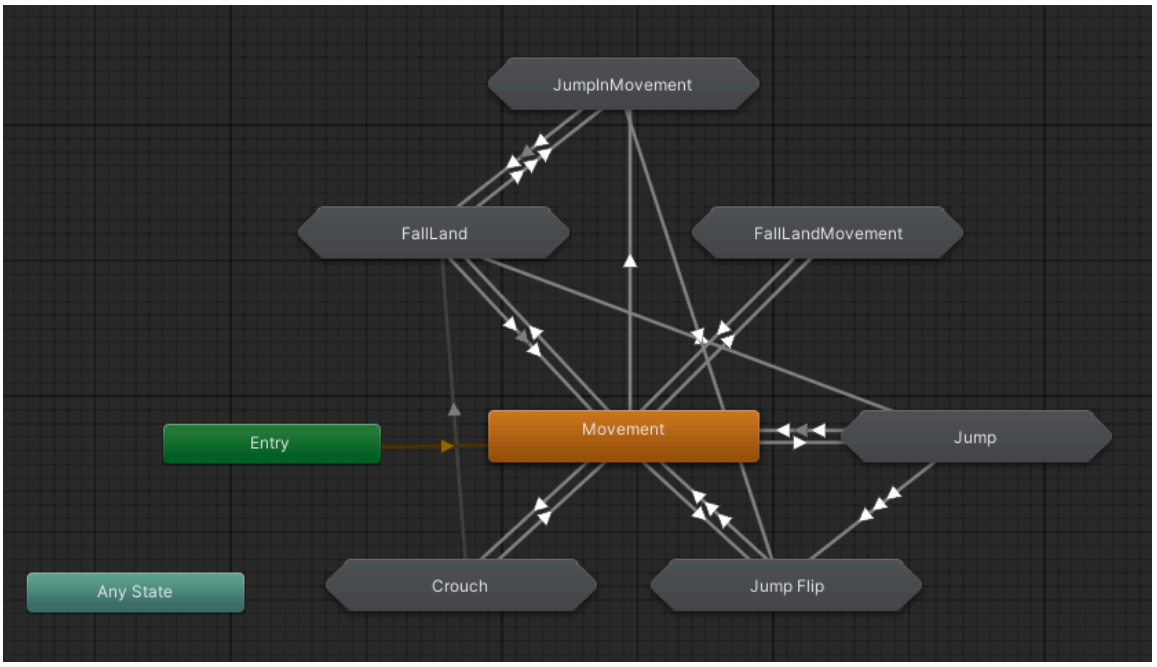


Figure 35: Platformer preset animation tree (Source: own elaboration).

The type of blending used in this case is *1D* since it only uses one parameter. The thresholds in the *Blend Tree* are the values that determine when one animation will transition into the other, they can be automated or configured manually. In this case, they were automated because there are only three animations and the default values worked well for the transitions.

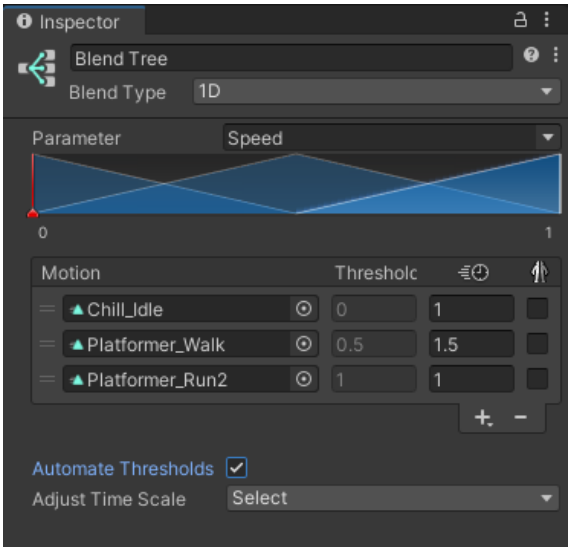


Figure 36: Blending parameters (Source: own elaboration).

The *Sub-State machines* represented in Figure 35 include several animations each and represent different states of the movement of the player character. These states have transitions between them based on the movement and the possibility of the player character transitioning from one state to another.

For instance, the *Crouch Sub-State Machine* has four animations: Stand to Crouch, Crouch Idle, Crouch Walk, and Crouch to Stand. The animation Stand to Crouch is the first executed when the Crouch state is entered. After it can transition to Coruch Idle if there is no movement, and Crouch Walk if the player character is moving. The Crouch Idle animation is executed while the player character is crouching but not moving. It can transition to Crouch Walk and Crouch to Stand. The Crouch Walk has the same transitions but is only executed while the player character is moving. The last animation is Crouch to Stand and it is executed when the player character exits the Crouching state. These animations and transitions are represented in Figure 37 below.

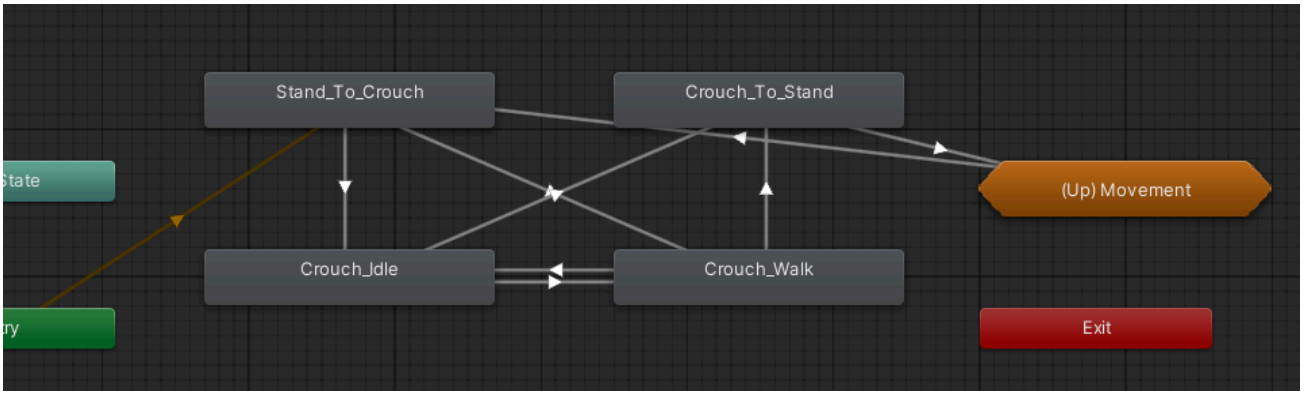


Figure 37: Crouch *Sub-State Machine* (Source: own elaboration).

6.3.4.3 Third-Person Shooter

The Third-Person Shooter *Animation Tree* is more complex since the controller has more states and transitions between them. It has one main *Blend Tree*, five *Sub-State Machines*, and two independent animations. The *Movement Blend Tree* is implemented the same way as in the Platfromer controller. The *Sub-State Machines* represent different states of the player character, and the two animations are the shooting animations that do not specifically belong to any *Sub-State Machine*. The structure is represented in Figure 38 below.

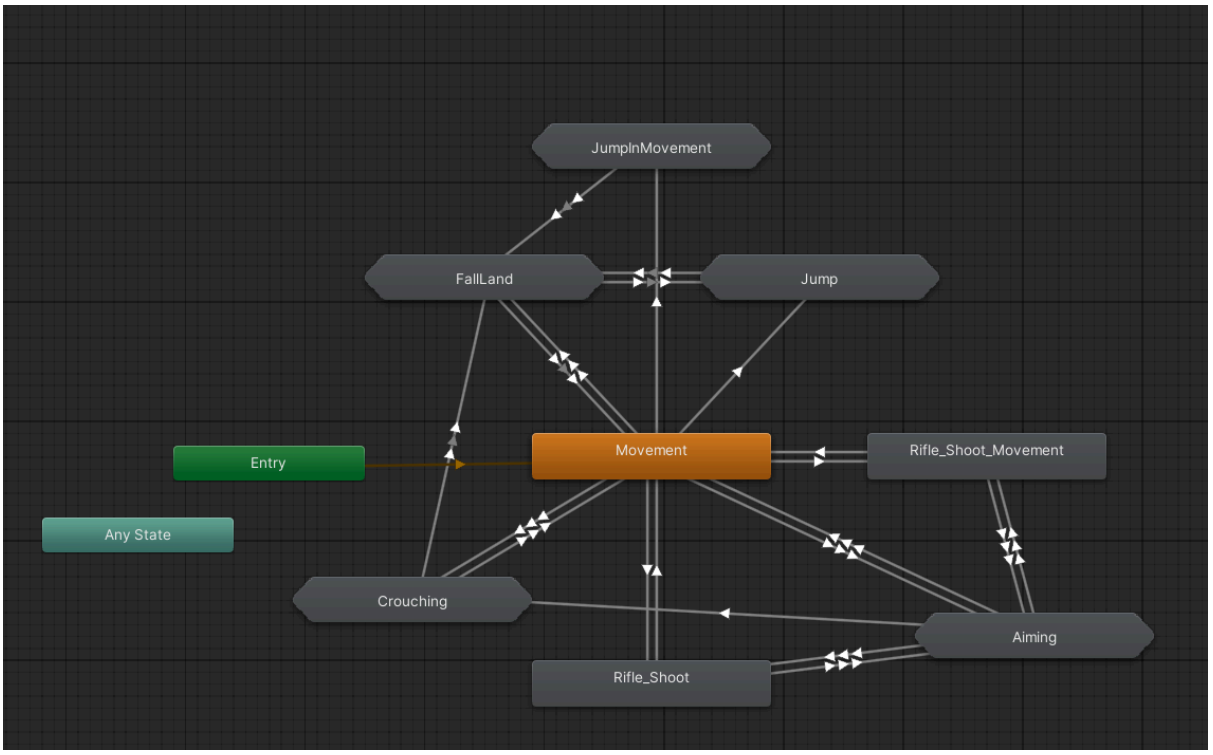


Figure 38: Third-Person Shooter preset *Animation Tree* (Source: own elaboration).

The most curious *Sub-State Machine* in this *Animation Tree* is *Aiming*. It has two animations and one *Blend Tree*. The animations are Aiming Idle and Aim Shooting. These animations are executed when the player is simply aiming, but not moving, and when the player is moving, aiming, and shooting at the same time. The *Aiming Blend Tree* is used for smoother transitions between movement animations while aiming. There are eight animations for eight directions of movement. The type of blending used in this *Blend Tree* is *2D Freeform Directional* blending which uses two parameters: movement input in the X and movement input in the Y axis.

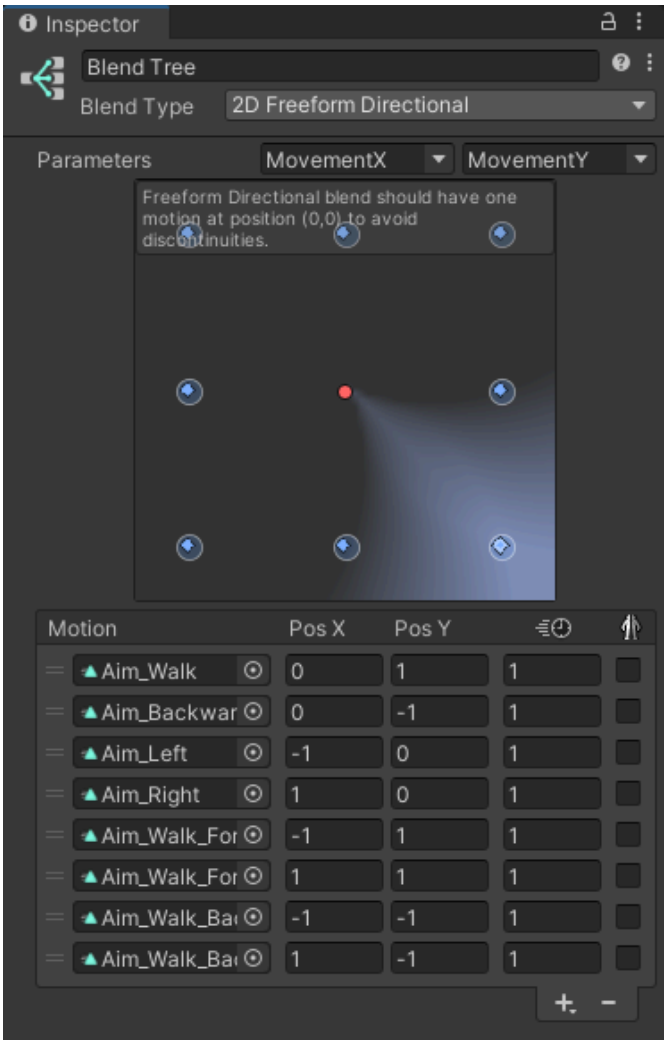


Figure 39: Aiming Blend Tree (Source: own elaboration).

This *Blend Tree* helps achieve smooth movement and transitions in all directions, since while the player character is aiming, strafing is applied and it does not rotate in the direction of the camera's *Forward* vector.

6.3.4.4 Adventure

The Adventure *Animation Tree* has a structure similar to the other *Animation Trees* explained earlier. It also has the main *Blend Tree* and six *Sub-State Machines*. The *Movement Blend Tree* has three animations that smoothly transition between each other depending on the speed of the movement. The *Sub-State Machines* are executed when the character enters a certain state. The whole *Animation Tree* is represented in Figure 40 below.

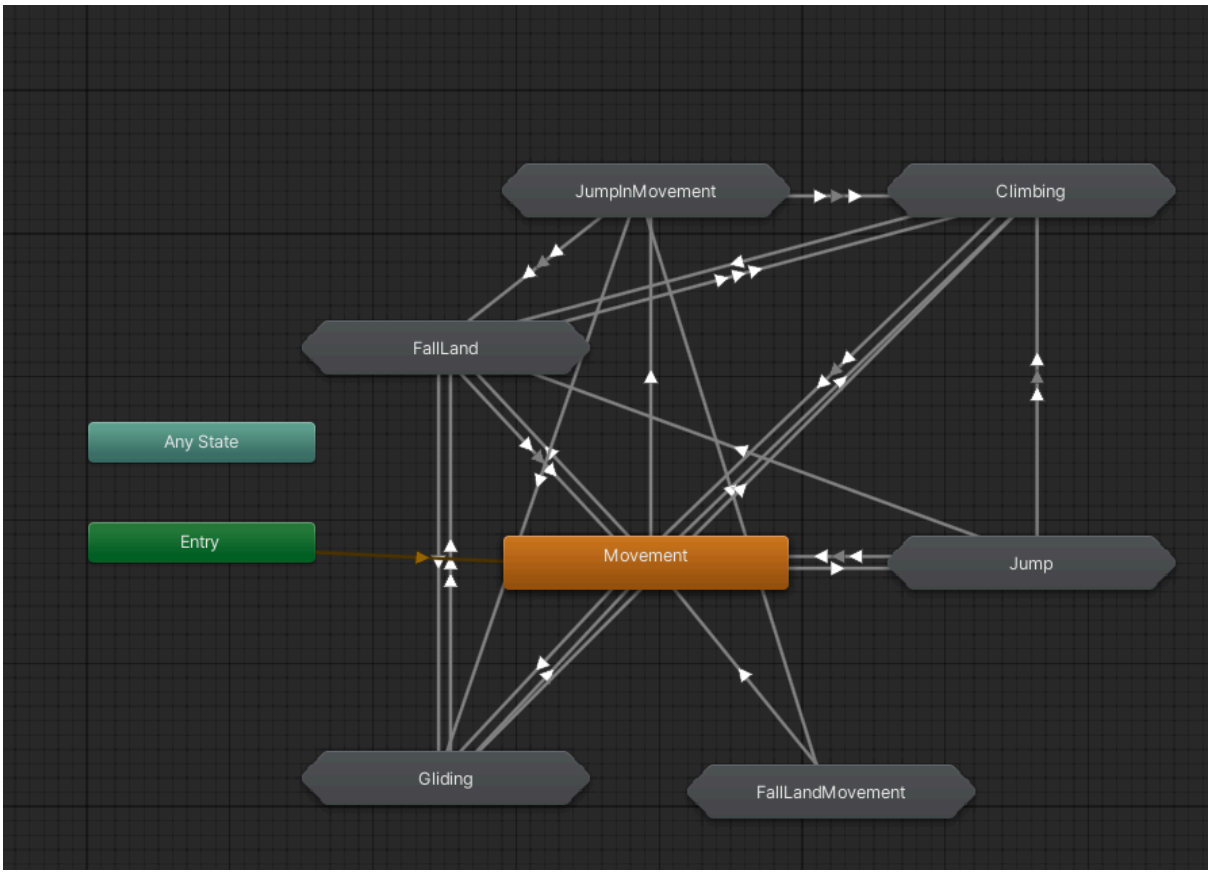


Figure 40: Adventure preset *Animation Tree* (Source: own elaboration).

The most complex *Sub-State Machine* in this *Animation Tree* is the *Climbing* one. The initial animation that is executed when the character enters the climbing state is Idle to Climbing. Then it transitions to Climbing Idle. This animation can transition to two different *Blend Trees*: Climb Vertical and Climb Direction. The type of blending used for these *Blend Trees* is *1D* since each of them only uses one parameter. Climb Vertical only has vertical movement animations and Climb Direction only has animations where the character moves horizontally. It was decided to separate the animations this way to achieve smoother transitions between vertical and horizontal

climbing and avoid a very fast change from one animation to another. Both of these *Blend Trees* can transition to Climbing to Jump animation that represents the Wall Jump. However, only Climb Vertical can transition to Climbing to Top animation that represents the character climbing to the top of the wall. Climbing to Top animation then transitions to the main *Movement Blend Tree.* The whole Climbing Sub-State Machine is represented in Figure 41 below.

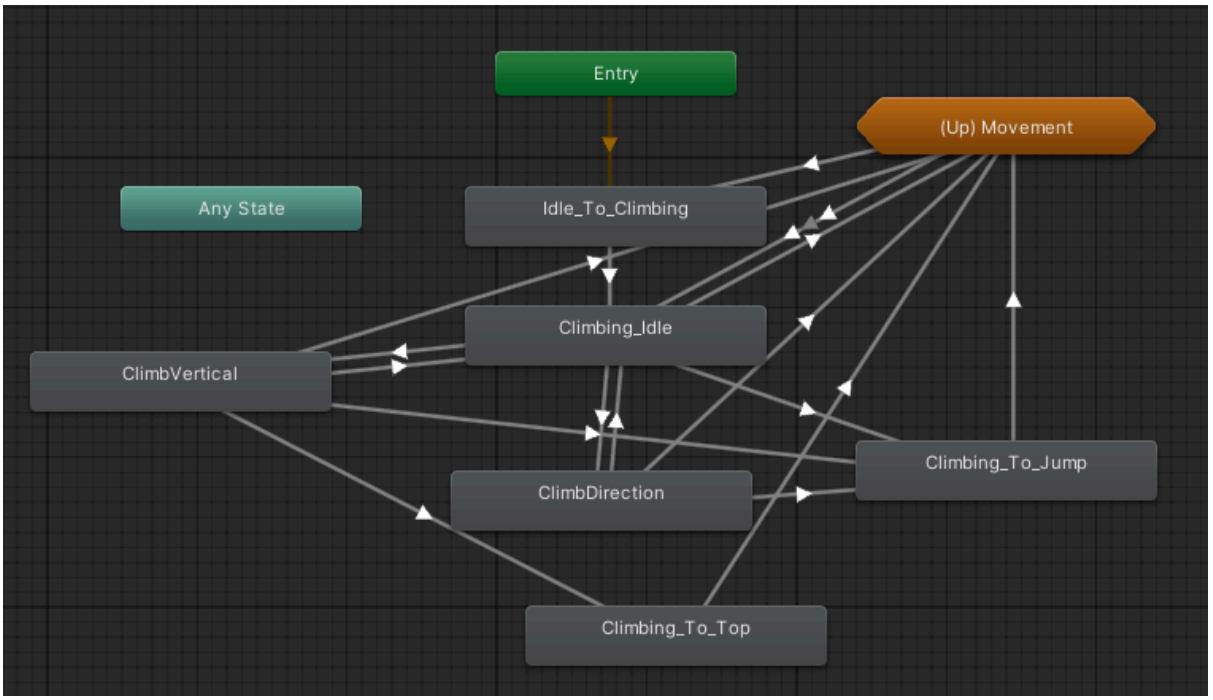


Figure 41: Climbing Sub-State Machine (Source: own elaboration).

6.3.4.5 Parameter Tuning

In order to achieve a better game feel for each character controller, all the presets were tested with potential representatives of the target audience. Several Level Designers and Technical Designers were asked to participate in the testing of the character controllers throughout the development process. The feedback that they provided allowed the improvement of the functioning of the controllers, the organization of the elements, and the positioning of the camera in each preset.

For instance, the camera in the Third-Person Shooter preset was brought closer to the character, especially in the zoom state, to provide better visibility of the scenario for the player. The minimum pitch was also adjusted to allow a wider angle of rotation and give the player the possibility to shoot higher.

The jump and double jump forces were adjusted in the Platformer preset to give the player the feeling of power while jumping, but at the same time not create a feeling of floating in the air and slow down the gameplay.

The Adventure preset was adjusted in terms of the climbing speed and the drag applied to the *Rigidbody*. The drag was increased to slow down the descent and the gliding speed was increased to allow the player to reach farther locations while gliding.

All the values that were modified thanks to the feedback of the participants can still be adjusted by the final user. However, the process of tuning the parameters helped achieve a better game feel with the default variables that each preset has.

6.4 Documentation and Publishing

In order to facilitate the use of the package, a document that explains the process of installation and the functioning of each preset was created. It will be available in the annexes. The documentation contains detailed information on each variable that affects the behavior of each character controller and will be available to every user.

The publishing process started with the creation of a user profile at the *Unity Publisher Portal* and starting the process of creating a new package.

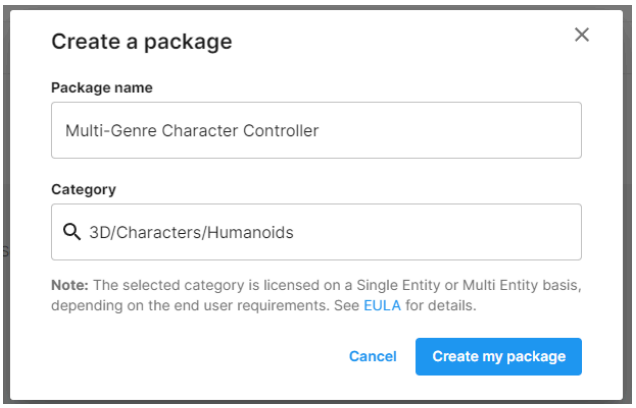


Figure 42: Package creation (Source: *Unity Publisher Portal*).

To upload a package there are several steps to follow:

1. Create a new package.
2. Add release notes.
3. Add a summary, description, and technical details.
4. Set a price.
5. Upload screenshots, videos, and marketing images.
6. Add localization.

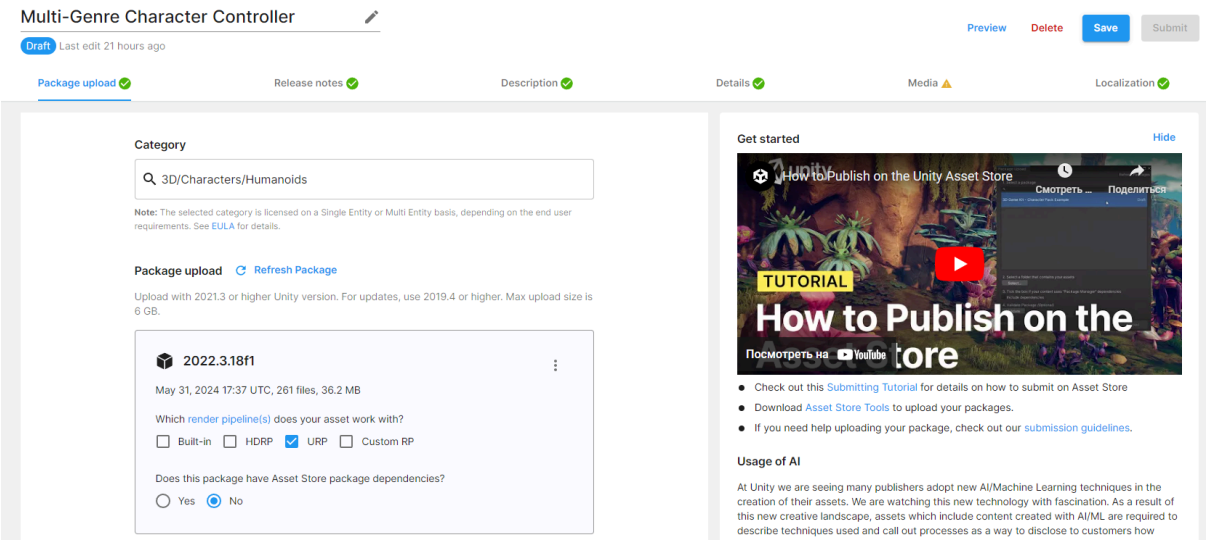


Figure 43: Process of explaining the details of the package (Source: *Unity Publisher Portal*).

After completing these steps, the *Asset Store Tools* package was downloaded. It is an official Unity package that allows one to validate and submit the custom package. The submission stage is shown at the *Unity Publisher Portal*, it usually takes around 30 working days for the package to be reviewed.

7. Conclusions

After completing this project, there are various conclusions to be made based on the development, the obtained results, and the limitations that were encountered during the process.

7.1 Evaluation of the Results

The results are a representation of the objectives set at the beginning of this project.

- An asset pack with several character controllers was created. The genres of the controllers are Third-Person Shooter, 3D Platformer, and Adventure. The package allows users to test their levels and prototypes, and adjust the necessary parameters as they please.
- The character controllers, the cameras, and the mechanics of three game genres were analyzed.
- The common and the genre-specific mechanics were defined for each character controller.
- A base character controller that was used as a template for the presets was implemented. It does not have any specific mechanics but allows the modification of the parameters and the creation of new controllers based on it.
- The three specific presets were implemented for three genres: Third-Person Shooter, 3D Platformer, and Adventure.
- Custom *Animation Trees* were created for each character controller.
- Each preset supports two input systems: mouse and keyboard, and gamepad systems.
- The documentation and the manual for the asset pack were created where the process of installation, use, and functionality of all the parameters are explained.
- An asset pack was created for the *Unity Asset Store*, however, at the time of the submission of this paper, it is still unknown if the pack is approved.

The organization of the project and the development approach allowed the implementation of most of the mechanics and characteristics proposed for each preset.

The initial schedule created for the project was followed through for the delivery of the big milestones, while the smaller objectives and deadlines were modified due to the change in the workflow. However, the presence of a general schedule helped to keep track of the big milestones and redistribute the work depending on the priority.

There were technical issues and limitations that prevented the development and implementation of some mechanics in different presets, or made the final implementation differ from the original idea.

7.2 Limitations

There were different technical limitations encountered throughout the development process, however the biggest limitation was the lack of animations.

Some mechanics that were present in the initial design of each preset could not be implemented due to the fact that there were no animations available in any public source, and implementing a mechanic without proper animations would completely ruin the game feel of the presets.

Table 2 below represents the mechanics that were not implemented. They are not described in section 6.2 Analysis of the Mechanics since the research conducted at the time showed that the animations for these mechanics could not be found, so they were eliminated from the list of the mechanics that were going to be implemented for each genre.

The quality of the animations also compromised some of the mechanics that were implemented. For instance, the aiming animations in the Third-Person Shooter preset do not have the positions of aiming up and aiming down, therefore sometimes the character looks like it is not aiming in the right direction.

The jumping animations for the 3D Platformer preset do not have the style that the 3D platformer characters usually have. They look heavier than they should due to the realism of all the animations present in *Mixamo*.

Mechanic	Third-Person Shooter	Adventure	Platformer
Roll			
Take Cover and Shoot			
Wall Jump			
Swim			

Table 2: The mechanics that could not be implemented in each genre (Source: own elaboration).

Another limitation was the unawareness of the fact that *Unity Technologies* takes 30 working days to analyze and approve the new package. Because of this, the result of the submission of the package could not be presented for the final submission of this project.

7.3 Further work

In terms of future development, there are several parts of the project that will be improved:

- To obtain a better game feel and polish, custom animations would be needed. That is why the main objective of further development is to collaborate with 3D artists and/or animators. This would make it possible to implement all the mechanics needed and not be limited by the presence or quality of the animations available on public sources.
- The structure of the code would be changed in order to make it more scalable for the implementation of other mechanics. It would also make it easier to maintain as the project would become more and more extensive.
- To make the asset pack more powerful and give it more flexibility, new mechanics and new genres would be added. This would make it more useful as it covers more genres and allows the addition of more mechanics.
- To ease the process of modifying the presets, visual scripting would be included in the project. This way the users would not have to deal directly with any scripts.

- Finally, making each asset more complex, and customizable, and adding more mechanics to each genre would make it possible to use the asset pack to create complete games based on it, and not only use it for testing or prototyping.

In conclusion, there were limitations encountered during the development of this project that affected it in important ways, and the unawareness of the *Unity Publisher Portal* policy made it impossible to publish the asset pack on time. However, despite the changes during the process and the limitations, the final results obtained meet most of the objectives set at the beginning of the project and are found to be satisfactory.

8. Bibliography

Adam Robinson-Yu (2019). *A Short Hike* (PC) [Video game].

Aedryn. (n.d.). Pixilart. Retrieved May 15, 2024, from https://www.pixilart.com/art/health-bar-empty-8c3bdb90a0cb308

Aircraft Rotations | Glenn Research Center | NASA. (2022, July 21). NASA Glenn Research Center. Retrieved February 23, 2024, from https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/aircraft-rotations/

Avalanche Software (2023). *Hogwarts Legacy* (PC) [Video game]. Warner Bros. Games.

Barnett, B. (2023, March 1). *Lil Gator Game Developer Interview: Make Your Own Fun With Adorable Sandbox Exploration*. IGN. Retrieved February 23, 2024, from https://www.ign.com/articles/lil-gator-game-developer-interview-make-your-own-fun-with-adorable-sandbox-exploration

Bethesda Game Studios (2023). *Starfield* (PC) [Video game]. Bethesda Softworks.

BioWare (2012). *Mass Effect 3* (PC) [Video game]. Electronic Arts.

BlueTwelve Studio (2022). Stray (PC) [Video game]. Annapurna Interactive.

Buehler, A. (2019, November 11). *Third Person Camera View in Games - a record of the most common problems in modern games, solutions taken from new and retro games - Third Person Camera View in Games - a record of the most common problems in modern games, solutions taken ...* Game Developer. Retrieved January 4, 2024, from https://www.gamedeveloper.com/design/third-person-camera-view-in-games---a-record-of-the-most-common-problems-in-modern-games-solutions-taken-from-new-and-retro-games

Chaotic Cat Studio (2023). *Bubble Heights* (PC) [Video game].

Character Controllers — NVIDIA PhysX SDK 3.4.0 Documentation. (n.d.). NVIDIA Docs. Retrieved January 4, 2024, from https://docs.nvidia.com/gameworks/content/gameworkslibrary/physx/guide/Manual/CharacterControllers.html

Cinemachine. (n.d.). Unity. Retrieved January 4, 2024, from https://unity.com/unity/features/editor/art-and-design/cinemachine

Comparing Orthographic and Perspective Cameras - Questions & Answers. (2016, July 21). Unity Discussions. Retrieved January 4, 2024, from https://discussions.unity.com/t/comparing-orthographic-and-perspective-cameras/169520/2

Crawford, C. (1984). *The Art of Computer Game Design*. Osborne/McGraw-Hill.

Crosshairs Plus | 2D Icons. (n.d.). Unity Asset Store. Retrieved May 15, 2024, from https://assetstore.unity.com/packages/2d/gui/icons/crosshairs-plus-139902

FOV visual representation script. Rotation causes offset - Questions & Answers. (2016, June 12). Unity Discussions. Retrieved February 6, 2024, from https://discussions.unity.com/t/fov-visual-representation-script-rotation-causes-offset/167189

Fundamentals of Real-Time Camera Design. (n.d.). AWS. Retrieved January 5, 2024, from https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc05/slides/GD_Haigh-Hutchinson_FundamentalsReal-TimeCameraDesign2.pdf

Haigh, M. (2009, July 1). *Real-Time Cameras - Navigation and Occlusion*. Game Developer. Retrieved January 16, 2024, from

https://www.gamedeveloper.com/design/real-time-cameras---navigation-and-occlusion

Haigh-Hutchinson, M. (2009). *Real Time Cameras: A Guide for Game Designers and Developers*. Morgan Kaufmann.

If game object is in camera's field of view. (2014, June 3). Unity Discussions. Retrieved January 4, 2024, from https://discussions.unity.com/t/if-game-object-is-in-cameras-field-of-view/106943

I think i just killed this guy... This Peacemaker didn't stand for like 3 minutes now... : r/stray. (2022, December 10). Reddit. Retrieved February 23, 2024, from https://www.reddit.com/r/stray/comments/zhe89c/i_think_i_just_killed_this_guy_this_peacemaker/

Lynch, M. (2023, June 9). *First-Person Games vs. Third-Person Games: What Are the Differences?* The Tech Edvocate. Retrieved February 12, 2024, from https://www.thetechedvocate.org/first-person-games-vs-third-person-games-what-are-the-differences/

Manual: Animation Layers. (n.d.). Unity - Manual. Retrieved February 12, 2024, from https://docs.unity3d.com/Manual/AnimationLayers.html

Manual: Animator component. (n.d.). Unity - Manual. Retrieved January 4, 2024, from https://docs.unity3d.com/Manual/class-Animator.html

Manual: Animator Controller. (n.d.). Unity - Manual. Retrieved February 12, 2024, from https://docs.unity3d.com/Manual/class-AnimatorController.html

Manual: Blend Trees. (n.d.). Unity - Manual. Retrieved February 12, 2024, from https://docs.unity3d.com/Manual/class-BlendTree.html

Manual: Camera component. (n.d.). Unity - Manual. Retrieved January 4, 2024, from https://docs.unity3d.com/Manual/class-Camera.html

Manual: Sub-State Machines. (n.d.). Unity - Manual. Retrieved February 12, 2024, from https://docs.unity3d.com/Manual/NestedStateMachines.html

Mass Effect 3 Gameplay Xbox 360 - Part 3 - Priority: Mars (Chasing Dr. Eva) | *WikiGameGuides*. (2012, March 6). YouTube. Retrieved February 23, 2024, from https://www.youtube.com/watch?app=desktop&v=Tgda1ZbD2tY

Mediatonic (2020). *Fall Guys* (PC) [Video game]. Epic Games.

MegaWobble (2022). *Lil Gator Game* (PC) [Video game]. Playtonic Games.

Mixamo. Retrieved May 10, 2024, from https://www.mixamo.com/#/?page=1&query=xbot&type=Character

Naftis, M., Tsatiris, G., & Karpouzis, K. (2021, September 8). *[2109.03750] How Camera Placement Affects Gameplay in Video Games*. arXiv. Retrieved January 4, 2024, from https://arxiv.org/abs/2109.03750

Nintendo (1996). *Super Mario 64* (Nintendo 64) [Video game]. Nintendo.

Orthographic projection Definition & Meaning. (n.d.). Merriam-Webster. Retrieved January 4, 2024, from https://www.merriam-webster.com/dictionary/orthographic%20projection

Pignole, Y. (2015, September 28). *Third person camera design with free move zone - Third person camera design with free move zone*. Game Developer. Retrieved January 4, 2024, from https://www.gamedeveloper.com/design/third-person-camera-design-with-free-move-zone

The PlayerInput component | Input System | 1.5.1. (n.d.). Unity - Manual. Retrieved January 4, 2024, from

https://docs.unity3d.com/Packages/com.unity.inputsystem@1.5/manual/PlayerInput.html

Precise animation control - Questions & Answers. (2019, October 18). Unity Discussions. Retrieved February 12, 2024, from https://discussions.unity.com/t/precise-animation-control/228125

Sci-Fi Gun Light | 3D Guns. (n.d.). Unity Asset Store. Retrieved May 10, 2024, from https://assetstore.unity.com/packages/3d/props/guns/sci-fi-gun-light-87916

Setting Virtual Camera properties | Package Manager UI website. (n.d.). Unity - Manual. Retrieved January 4, 2024, from https://docs.unity3d.com/Packages/com.unity.cinemachine@2.3/manual/CinemachineVirtualCamera.html

Swink, S. (2007, November 23). *Game Feel: The Secret Ingredient - Design*. Game Developer. Retrieved January 4, 2024, from https://www.gamedeveloper.com/design/game-feel-the-secret-ingredient

Swink, S. (2009). *Game Feel: A Game Designer's Guide to Virtual Sensation*. Taylor & Francis.

Understanding the rendering of the raycasting on flat screen. (2019, March 31). Game Development Stack Exchange. Retrieved February 23, 2024, from https://gamedev.stackexchange.com/questions/169546/understanding-the-rendering-of-the-raycasting-on-flat-screen

Understanding the View Frustum - Unity Manual. (n.d.). Unity - Manual. Retrieved January 4, 2024, from https://docs.unity3d.com/es/2018.3/Manual/UnderstandingFrustum.html

Unity - Scripting API: CharacterController. (n.d.). Unity - Manual. Retrieved January 4, 2024, from https://docs.unity3d.com/ScriptReference/CharacterController.html

Unity - Scripting API: MonoBehaviour.FixedUpdate(). (n.d.). Unity - Manual. Retrieved May 20, 2024, from https://docs.unity3d.com/ScriptReference/MonoBehaviour.FixedUpdate.html

Unity - Scripting API: Rigidbody. (n.d.). Unity - Manual. Retrieved January 4, 2024, from https://docs.unity3d.com/ScriptReference/Rigidbody.html

Westecott, E. (2009, January). *The Player Character as Performing Object*. ResearchGate. Retrieved January 4, 2024, from https://www.researchgate.net/publication/255629301_The_Player_Character_as_Performing_Object

Willumsen, E. C. (2021, September 22). *Avatar Control and Character Complexity: Defining and Typologizing Character Autonomy*. ResearchGate. Retrieved January 4, 2024, from https://www.researchgate.net/profile/Ea-Christina-Willumsen-2/publication/336497340_Avatar_Control_and_Character_Complexity_Defining_and_Typologizing_Character_Autonomy/links/5da35efa92851c6b4bd33bd1/Avatar-Control-and-Character-Complexity-Defining-and-Typ

9. Annexes

General link to all the contents: 🗀 Bolshakova_MultiGenreCharacterController_TFG

9.1 Annex 1: Unity Project and Source Code

This annex contains the whole Unity Project and a link to the GitHub repository.

- Path: *Bolshakova_MultiGenreCharacterController_TFG/Unity Project*
- Name: Unity Project
- 🗀 Unity Project

9.2 Annex 2: Unity Package

This annex contains the final version of the Unity Package of this project.

- Path: *Bolshakova_MultiGenreCharacterController_TFG*
- Name: MultiGenreCharacterController_URP.unitypackage

9.3 Annex 3: Technical Documentation

This annex contains the documentation on how to install the Unity Package.

- Path: *Bolshakova_MultiGenreCharacterController_TFG*
- Name: TechnicalDocumentation_VeronikaBolshakova_TFG

9.4 Annex 4: Manual

This annex contains a manual that explains how to use and adjust the character controllers.

- Path: *Bolshakova_MultiGenreCharacterController_TFG*
- Name: Manual_VeronikaBolshakova_TFG

9.5 Annex 5: Demonstrative Videos

This annex contains demonstrative videos of how the final product works.

- Path: *Bolshakova_MultiGenreCharacterController_TFG/Videos*
- Name: Videos

- 🗀 Videos

9.6 Annex 6: Executables

This annex contains three builds (executables) for each character controller created.

- Path: *Bolshakova_MultiGenreCharacterController_TFG/Executables*
- Name: Executables
- 🗀 Executables

9.7 Annex 7: Third-Party Materials

- Sci-Fi Gun Light (Factory of Models, 2017): https://assetstore.unity.com/packages/3d/props/guns/sci-fi-gun-light-87916
- Crosshair Plus (Asset Bag, 2019): https://assetstore.unity.com/packages/2d/gui/icons/crosshairs-plus-139902
- 3D model and animations: mixamo.com