UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

Development Of
Gameplay Mechanics
– AND –
Ability Interactions
For A Multiplayer Fast-Paced Sports And Shooter Game

Arnau Jiménez Gallego

Director: Lasse Löepfe

Degree: Video Game Design and Development

Course: 2024-25

University: CITM UPC

Index

Summary

This thesis presents the development of a vertical slice for a multiplayer fast-paced sports and shooter game. The final product is the result of work done by three students, each one assuming specific responsibilities and roles in programming, game design, and art. The technical implementation of gameplay mechanics, ability interactions, and networking features seamlessly is the main focus of this thesis.

The game is inspired by titles like Rocket League and Overwatch 1&2, it combines a set of unique movement mechanics, such as wall-running and sliding, with character-based abilities, and a sports-based gameplay.

A great challenge addressed throughout the development is the implementation of physics-driven gameplay with networked multiplayer environment. Unity's Netcode for GameObjects is used together with Unity Multiplay to create a dedicated server environment with an authoritative architecture. The goal was to achieve consistency, and performance while minimizing latency and synchronization issues.

The objectives that I achieved and that the project contains are: a modular and extensible ability system, advanced movement mechanics with physics-based interactions, cloud hosted dedicated server, and a custom game flow and match states for a sports-based game.

Multiple testing phases were conducted with internal and external sessions gathering feedback from diverse player profiles that serve to validate the project.

Overall, this project achieved its main goals and taught me important lessons about networking, gameplay development, balancing, and working in a team. The final result is a strong prototype that shows what I have learned and could be the starting point for a complete game in the future.

Resum

Aquesta tesi presenta el desenvolupament d'un vertical slice per a un videojoc multijugador de ritme ràpid que combina esports i jocs de trets. El producte final és el resultat del treball fet per tres estudiants, cadascun assumint responsabilitats i rols específics en programació, disseny de joc i art. La implementació tècnica de les mecàniques de joc, les interaccions d'habilitats i les funcionalitats de xarxa de manera fluida és el focus principal d'aquesta tesi.

El joc està inspirat en títols com Rocket League i Overwatch 1&2, i combina un conjunt de mecàniques de moviment úniques, com córrer per parets i lliscar, amb habilitats basades en personatges i una jugabilitat basada en esports.

Un gran repte abordat durant el desenvolupament és la implementació de mecàniques de joc basades en física amb un entorn multijugador en xarxa. S'utilitza Netcode for GameObjects de Unity juntament amb Unity Multiplay per crear un entorn de servidor dedicat amb una arquitectura autoritària. L'objectiu era aconseguir consistència i rendiment mentre es minimitzava la latència i els problemes de sincronització.

Els objectius que he aconseguit i que el projecte conté són: un sistema d'habilitats modular i extensible, mecàniques de moviment avançades amb interaccions físiques, servidor dedicat allotjat al núvol, i un flux de joc personalitzat amb estats de partida per a un joc basat en esports.

S'han dut a terme múltiples fases de prova amb sessions internes i externes recollint opinions de perfils de jugadors diversos que serveixen per validar el projecte.

En general, aquest projecte ha assolit els seus objectius principals i m'ha ensenyat lliçons importants sobre xarxes, desenvolupament de mecàniques de joc, equilibrat i treball en equip. El resultat final és un prototip sòlid que mostra el que he après i podria ser el punt de partida per a un joc complet en el futur.

Resumen

Esta tesis presenta el desarrollo de un vertical slice para un videojuego multijugador de ritmo rápido que mezcla deportes y disparos. El producto final es el resultado del trabajo realizado por tres estudiantes, cada uno asumiendo responsabilidades y roles específicos en programación, diseño de juego y arte. La implementación técnica de las mecánicas de juego, las interacciones de habilidades y las funciones de red de forma fluida es el foco principal de esta tesis.

El juego está inspirado en títulos como *Rocket League* y *Overwatch 1&2*, combina un conjunto de mecánicas de movimiento únicas, como correr por las paredes y deslizarse, con habilidades basadas en personajes y una jugabilidad basada en deportes.

Un gran desafío abordado durante el desarrollo es la implementación de una jugabilidad basada en físicas con un entorno multijugador en red. Se usa Netcode for GameObjects de Unity junto con Unity Multiplay para crear un entorno de servidor dedicado con una arquitectura autoritativa. El objetivo era lograr consistencia y rendimiento mientras se minimizaba la latencia y los problemas de sincronización.

Los objetivos que he logrado y que el proyecto contiene son: un sistema de habilidades modular y extensible, mecánicas de movimiento avanzadas con interacciones físicas, servidor dedicado en la nube, y un flujo de juego personalizado con estados de partida para un juego basado en deportes.

Se realizaron múltiples fases de prueba con sesiones internas y externas recogiendo comentarios de perfiles de jugadores diversos que sirven para validar el proyecto.

En general, este proyecto logró sus objetivos principales y me enseñó lecciones importantes sobre redes, desarrollo de mecánicas de juego, equilibrio y trabajo en equipo. El resultado final es un prototipo sólido que muestra lo que he aprendido y podría ser el punto de partida para un juego completo en el futuro.

Key words

Mechanics, Ability, Programming, Physics, Unity, Multiplayer, Server, Netcode for GameObjects, Video game.

Links

Github Project: https://github.com/Historn/HyperStrike

Releases: https://github.com/Historn/HyperStrike/releases

Video presentation: https://youtu.be/oQkHkJfrDX4

Table index

Figure index

Glossary

Mechanics: Rules, elements, and processes that make up a game.

Ability: Action a character can perform, often activated by a button press, that can modify their attributes or have a direct impact on gameplay.

Multiplayer: Video game in which more than one person plays simultaneously in a shared game environment.

Unity: Cross-platform game engine developed by Unity Technologies.

Debug: To detect and remove errors from (a video game).

Network: Several interconnected computers, machines, or operations.

Netcode: frameworks that are specifically designed to help make building certain aspects of networked gameplay easier.

1. Introduction

The implementation of gameplay mechanics and ability-based interactions has become an essential aspect for developing fast-paced multiplayer games, this thesis explores its development for a 3D multiplayer fast-paced sports and shooter video game. The game concept combines aspects from sports-based games and hero shooters, where players control characters with projectile launchers and special abilities to influence the environment and the game's physics-driven ball and player movement.

The game will include explosions, impulse forces, and collision reactions that need to be tweaked in order to achieve a fair mix between realism and arcade-like responsiveness. By introducing interactions like changing the trajectory of the ball, influencing the player's movement, or applying area-of-effect forces, the ability system may also change the gameplay. One of the major problems is to make sure that the mechanics and abilities integrate smoothly into the physics system without unintended behaviors or instabilities, which presents a significant challenge.

1.1 Motivation

I have had an intense love for video games since I was a little child, spending countless hours lost in lots of virtual worlds that tested my abilities, imagination, and inventiveness. After almost six years of studying video games development, has grown my passion into a goal to one day develop my own video games. Other than video games, I have always had a strong interest in science, particularly physics, and computer programming. I enjoy coding to solve intricate issues as well as to comprehend how things operate.

During these last years, I felt attracted to how game mechanics and ability systems work in video games, it is a big world to explore so I wanted to apply my knowledge and research skills to get deeper into this subject, mixing it with game mechanics, ability systems, and network connections. I want to improve my skills and learn new ones that are important in video game development.

This can also be a strong project for my portfolio that can showcase my programming skills and also to improve them.

1.2 Problem formulation

The development of gameplay mechanics and ability interactions for a fast-paced sports-based and shooter game presents a unique set of challenges, particularly when it has to combine elements of multiplayer functionality with physics-driven mechanics. The game merges the competitive dynamics of football with the fast-paced, ability-driven gameplay of hero shooters, presenting characters equipped with projectile launchers, unique abilities that also affect the gameplay, and advanced movement mechanics, such as wall-running.

The principal problem is integrating these elements seamlessly while maintaining balance, responsiveness, and fairness in a multiplayer environment. Physics-driven mechanics, such as the movement of the ball, the collisions of the characters, and the trajectory of the projectiles and forces, must be designed to be predictable, consistent, and enjoyable while accommodating a fast-paced gameplay. At the same time, the character abilities must be carefully crafted to enhance gameplay without overshadowing the core sports mechanics or creating unbalanced multiplayer matches.

Advanced movement systems, such as wall-running and fast traversal, must feel intuitive and responsive, at the same time must avoid exploits or unintended behaviors within the physics-driven gameplay and ability interactions across multiple players in real-time, it presents a significant challenge to achieve a low-latency multiplayer experience

The absence of a definitive framework or established best practices for merging such disparate genres and mechanics further complicates the development process, there are needed innovative solutions and iterative testing to achieve a harmonious and competitive gameplay experience for this kind of game.

1.3 General objectives

The general objectives of this thesis are the two following:

The first objective is to design and implement an optimized system for gameplay mechanics and ability interactions tailored to fast-paced multiplayer environments.

Then, to develop a working prototype that demonstrates smooth synchronization of physics-driven interactions in an online multiplayer context, ensuring a responsive, fair, and performant gameplay experience.

1.4 Specific objectives

- Implement physics-driven mechanics for multiplayer video games
- Implement an ability system for multiplayer video games.
- Set up a dedicated server.
- Achieve a seamless integration between online connection and physics-driven mechanics.
- Create a final prototype of the game.

1.5 Scope of the project

The video game prototype focuses on the programming aspects of gameplay mechanics and ability interactions specifically in a fast-paced multiplayer sports shooter game. The primary goal is to achieve an efficient implementation of coding techniques that ensure responsive gameplay, balanced and fair competition, and smooth synchronization in a networked environment.

The key aspects that this thesis will focus on are:

- Understanding game mechanics and ability interactions:
    - How are gameplay mechanics and ability interactions programmed?
    - Why are they implemented in specific ways, and what are the technical trade-offs?
- Understanding multiplayer games:
    - How do networked multiplayer systems interconnect players?
    - How do multiplayer systems work together with game mechanics?
- Development:
    - Implement core gameplay mechanics.
    - Ability system programming.
    - Multiplayer synchronization techniques for physics-based mechanics.
    - Optimize performance and network efficiency for smooth and balanced gameplay.

While the focus is on the technical implementation, other aspects, such as game design or art design are outside the scope of this thesis. The final deliverable will be a functional vertical slice prototype that demonstrates the effectiveness of the programming techniques in a controlled game environment.

1.5.1. Target Audience of the Project

This project is primarily aimed at players who enjoy fast-paced, competitive multiplayer games that combine elements of sports and shooter mechanics. More specifically, it targets:

- PC gamers who are looking for a fresh experience blending the physics-based gameplay of sports titles like Rocket League with the tactical depth and hero abilities found in games like Overwatch.
- Indie game enthusiasts and those interested in experimental gameplay that pushes traditional genre boundaries.
- Students and developers interested in learning about or testing innovative game mechanics in a real-time multiplayer environment.
- Game design educators and researchers, who may use the game as a case study or prototype for discussing advanced game programming concepts.

2. State of the art / Theoretical framework / Contextualization / Market study

2.1. State Of The Art

For the development of a fast-paced sports and shooter game, it is required a basic knowledge of maths, physics, and multiplayer connections. The main focus of this section is to research the newest and best programming practices needed for the project.

2.1.1. Game Engine

The project final deliverable will be a prototype focusing on performance, ease of development, and multiplayer support. It should be able to run also on lower-end PCs.

- Unity offers good performance on lower-end PCs and offers multiple multiplayer support options. Also it is fast and easy to work with but sometimes can be tricky to work with custom networking solutions.
- Unreal Engine is a good choice for networking but is heavier on hardware, it also has a steeper learning curve if it has never or little been used before.
- Godot is an extremely lightweight ideal for lower-end hardware but has fewer multiplayer and community support compared to Unity or Unreal.

Unity is the best choice for this project, it offers a good balance between performance, ease of development and cloud hosting support. Especifically new Unity's version Unity 6 comes with great improvements for these features.

2.1.2. Physics Engine and Simulation

Physics is a crucial aspect for first-person shooters and sport-based video games. Achieving a good simulation for a video game is not focused only on whether they are realistic or not, it has to fit with your game's objectives and scope, what you, as a developer, want to obtain.

In a 2018 GDC Talk, Psyonix's Jared Cone, Lead Gameplay Engineer for Rocket League, talks about the issues they had with networked physics and how they designed and implemented them. As Cone explains, they weren't looking for the physics to be realistic but to be fast, responsive, and controllable. The physics engine that they used for the game was Bullet instead of the one that was already implemented in Unreal Engine 3, Cones emphasizes that one of the reasons they chose it, was that the engine is open-source allowing the developers to debug it and modify it in the way they wanted to

fix edge case problems they found during the development. Cones says that implementing other physics engines is not bad if they fit with what you want to achieve for the game.

A fixed tick rate allows deterministic physics, meaning that the physics frames work with the same Delta Time independently of how fast the game is rendering. For Rocket League they chose 120Hz instead of 60Hz to achieve a more consistent result on the physics simulation, yet making the game performance more expensive.

So, for this project, to have responsive multiplayer gameplay requires specialized physics mechanics implemented through either Unity 3D, a third-party physics engine, or custom physics formula modifications, and supported by robust network optimization techniques. Unity 3D already comes with a built-in physics engine, NVIDIA PhysX, but it also provides a package called Unity Physics that is addressed for Data-Oriented Technology Stack (DOTS). By looking at Unity's documentation, it is visible the difference between both, it will also depend on the scope and the time of development that the project has.

PhysX is a mature and well-documented physics engine that has been implemented in many known commercial games. It integrates responsive rigid bodies, colliders, and joints system support that are quick and simple to set up with Unity's GameObjects and Monobehaviour without needing to change the structure. Still, it has some performance limitations, a high number of rigid bodies can impact the performance due to the physics engine running on a single thread. It is also non-deterministic so, it introduces inconsistencies due to floating-point precision issues.

The DOTS physics engine is scalable for large worlds meaning that for a high count of objects will ensure a smoother performance, it can handle thousands of physics objects efficiently by implementing multi-threading. In this case, it works with deterministic simulation which means it uses floating-point determinism within the same hardware which makes it better for multiplayer synchronization but DOTS requires a shift in the programming paradigm changing from Monobehaviour to ECS (Entity Component System) adding a steep learning curve if never worked with DOTS, so it makes it harder to implement. Even though DOTS Physics does not sync automatically with netcode solutions, so will still need some custom handling.

Creating a custom physics engine allows complete control over how interactions and collisions behave, enabling to tailor the system to the specific needs of the game. In fast-paced competitive games, such as the one developed in this project, a custom solution can prioritize performance and precision for specific scenarios, eliminating unnecessary calculations. By creating a deterministic custom system, it becomes easier to replicate physics behavior consistently across clients and the server, improving gameplay reliability in networked environments. But building a physics engine from scratch is technically challenging and time-consuming, especially when dealing with accurate collision handling, friction, and bounce responses. Without careful implementation, a custom physics system may lead to desynchronization between clients and the server, particularly in online multiplayer contexts.

For a hero shooter sports game and the short development time there is for the project, PhysX is the best option except if the game is expected to have massive object interactions. The main reasons are that is easier to implement with rigid body physics, probably used for player movement, ball interactions and other mechanics, it needs less development time compared to DOTS and custom physics engine while at the same time works well with Monobehaviour-based gameplay and already Unity existing tools, and by this time there also exist some networking workarounds that will be explained further on in this thesis.

2.1.3. The network infrastructure

The main challenge for developing a game with physics-based mechanics is to be able to synchronize the interactions across all the clients while keeping a smooth and responsive gameplay.

2.1.3.1. Choose a Networking Model

There are two possible multiplayer architecture options, but just one fits with a project where it is needed to have in mind competitiveness, physics simulations, and synchronization issues.

First, there is the Client-Authority Model, where each client does the physics simulation locally and sends the updates to the server, which then validates and broadcasts. This model leads to desynchronization issues and potential cheating due to clients being able to manipulate physics so it is not ideal for a competitive shooter sports game.

The best option is the Authoritative Server Model, the server is the source of truth for all physics calculations, the clients send their inputs (movement, shooting, abilities) to the server that simulates the physics and sends the updates back to all the clients, then the clients do an interpolation or extrapolation to smooth out the movements. This model also prevents cheating, since clients don't control the physics, and ensures consistency making all the clients see the same game state.

2.1.3.2. Synchronization Strategies

As seen before, when choosing the PhysX Engine for the project, it is explained that is not deterministic, so it is essential a way to synchronize the physics across clients.

The best approach for this project is *Snapshot Interpolation*, as explained in the previous section, the server runs the physics simulation and then stores snapshots of key physics objects, such as the players, the ball, or the projectiles. The server sends periodic snapshots (updates) to the clients, for a fast-paced game it would be good every 50 milliseconds. Finally, the client smoothly interpolates between the received snapshots, it minimizes visible lag without adding input delay while working well with rigid body-based movements creating a smooth and natural-looking physics replication.

PhysX is a non-deterministic physics engine, but there are ways to smooth the interactions in a multiplayer setting implementing snapshot interpolation. "*Gaffer on Games: Networked Physics*" (Glenn Fiedler, 2015)

In the previously mentioned GDC's Talk, and in the video "*Let's Talk Netcode*" (Blizzard Entertainment, 2018), explains how Rocket League and Overwatch use snapshot interpolation to sync ball physics, player movement, and interactions for smooth visuals.

For a fast-paced shooter with physics-based interactions, lag compensation is still required to ensure accurate collision detection and ability interactions.

The first technique is Rewind and Replay Physics, the most well-known approach is from *"Source Engine Lag Compensation"* (Valve Corporation, 2005-2024), which explains how past physics states are stored on the server, when a client requests an action, rewind the physics state to when the action should have happened, then simulate forward. For example this prevents possible issues where players hit the ball on their screen but miss on the server.

The second technique is Input Prediction for the player movement. If the player moves using physics, such as wall running or impulses, clients predict their own movement locally without needing to wait for the server. Once the server's snapshot arrives, the client reconciles any differences, if the server disagrees with the client's position or rotation, it smoothly corrects it instead of glitching and teleporting.

The paper, *"Multiplayer Game Programming: Architecting Networked Games"* (Glazer and Madhav, 2015) describes how, despite the Internet's inherent inconsistencies, it is needed to achieve reliable performance and design the game code to reach maximum security and scalability.

2.1.3.3. Networking solution

For the type of video game this project focuses on, selecting an appropriate networking solution is crucial. Having in mind Unity 6, there are multiple networking framework options. This section will explain these options and their pros and cons to take into account for the final decision on which one to use.

Mirror is a high-level networking library made for Unity that evolved from, the now deprecated, UNet System. Mirror's design is especially intuitive for developers familiar with UNet. It has a community that offers extensive support, tutorials and plugins, which can accelerate the development. Even though it is suitable for small to medium projects, it may face issues scaling to large and more complex architectures. As a project developed by the community, it doesn't have official Unity backing affecting long-term sustainability.

FishNet is a networking solution that emphasizes performance, flexibility and feature richness. It is designed for high-performance scenarios that require handling numerous players and complex interactions, and offers some built-in features like client-side prediction for synchronization, lag compensation and seamless scene transitions. The full source code is accessible so it allows customization. Still, requires a steeper learning curve for developers that are new to networking concepts due to the extensive feature set.

Photon is one of the most popular networking frameworks that offers a client-server architecture with cloud services. Is easy to implement and offers managed servers, it is not needed to create a custom server infrastructure. One thing that makes Photon not

viable for this project, as explained in the Networking Model section, is that it depends on client-authoritative design and can allow players to cheat.

Netcode for GameObjects is Unity's official library that is already designed to integrate seamlessly with Unity, so it provides better compatibility with future engine updates. It is adjusted for small co-op games and may not have some advanced features that competitive and fast-paced games require. Comes with Multiplay and Matchmaker multiplayer services.

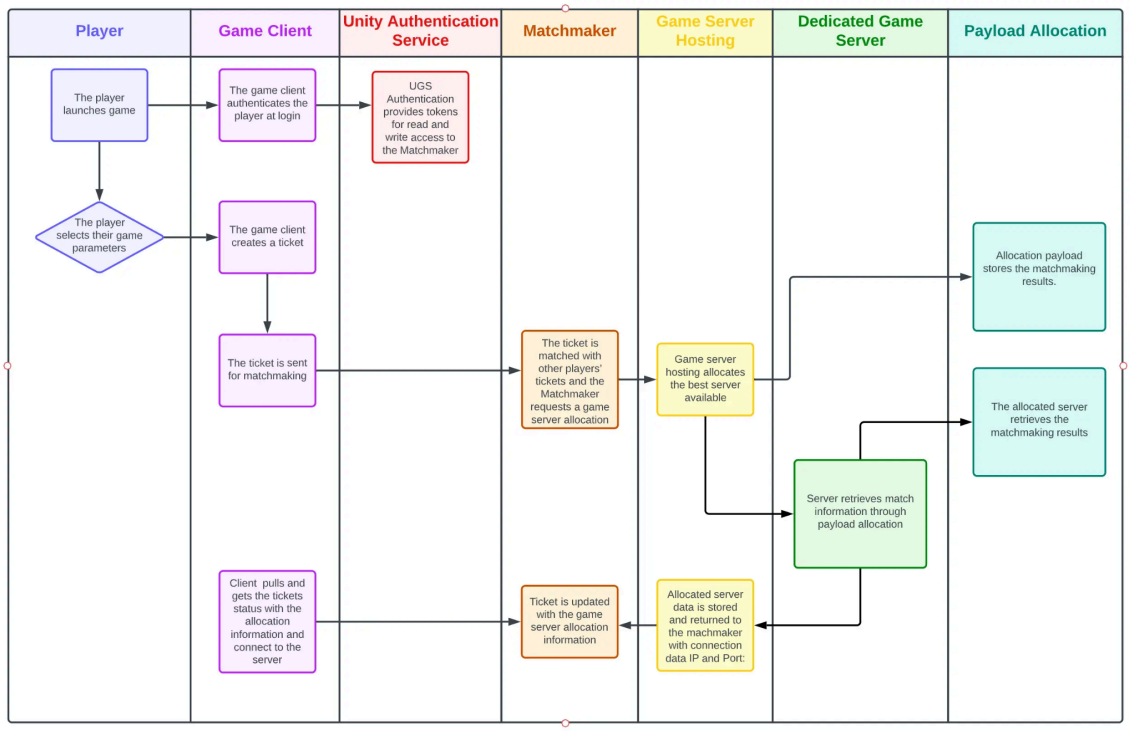
Apart from choosing the networking framework, for an Authoritative Server Model, it is needed to host the game with a custom server. Either can be in a own local machine or, having in mind the scope of the project, use free-tier cloud services. There are three options available for cloud services that would fit the project: *Unity's Multiplay*, *Amazon Web Services* (AWS), or *Google Cloud*.

These options have limited resources for the available RAM or CPU, but for a small-scale project that will have small player counts is enough. Also, they have fast and low-latency networks that may help lag compensation, Multiplay and AWS offer more free-tier time and some users find Google Cloud less intuitive.

Finally, a competitive online multiplayer game requires some sort of matchmaking framework, Unity provides Matchmaker as a reliable choice due to the features it offers.

Netcode for GameObjects, Mirror and FishNet are good options. For a fast and easy implementation, the best option is Netcode for GameObjects together with Multiplay and Matchmaker thanks to the support and multiplayer services that are provided and that allow to focus on more important things. Below, there is an image of a flow diagram of how the execution flow works with Netcode for GameObjects together with Multiplay and Matchmaker services.

Figure 2.1.3.3.1: Unity's Multiplayer Services Flow



(Source: Unity's Matchmaker Documentation)

2.1.4. Advanced Movement Mechanics

Developing advanced movement mechanics in first-person shooter (FPS) games requires a combination of programming techniques and an understanding of some mathematical formulas and physics principles, such as kinematics, dynamics and collision detection.

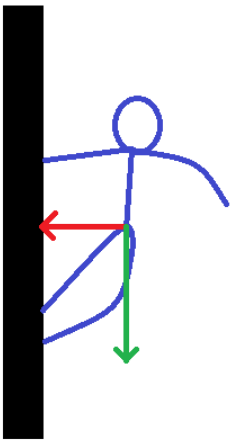
2.1.4.1. Common Approaches

In this section will be discussed the implementation approaches of common advanced movement mechanics and their applications.

Wallrunning adds verticality and fluidity to the players' movement by allowing them to traverse vertical surfaces, it is a feature that enriches the experience of the game. This are the following implementation approaches for this movement mechanic:

- Raycasting-based detection, rays are casted from the player to detect walls that may be nearby ensuring smooth and responsive transitions.
- State machine integration, manage the transitions of the different movement states using finite state machines (FSM).

Figure 2.1.4.1.1: Wall run Raycast



(Own Source)

Sliding mechanics allow the players to navigate obstacles in the game environment by maintaining the momentum[1]. This are the following implementation approaches for this movement mechanic:

- Adjustment of physics materials, modifies the properties of physics materials to create smooth sliding effects, for instance, modifying the friction value.
- Momentum preservation, the momentum of the player is conserved when transitioning between movement states.

Rocket jumping uses the explosion force of the weapons, typically rockets or grenades, to impulse the player to get to higher areas. An impulse force is applied to the player in the opposite direction of the explosion by calculating the vector of the impact between them and applying it to the velocity of the player. To balance this mechanic, it is also applied self-damage penalties or invulnerability frames during the jump.

Strafing involves moving the player laterally while it maintains the aim focused on a target. Then, strafe-jumping improves strafing by combining it with jumping to gain more movement speed and momentum, the implementation of the mechanic must follow:

- Responsive input, for lateral movement controls, to be implemented.
- Apply air acceleration while the player is airborne.
- Limit the maximum speed to avoid players accelerating to infinite values.

[1] Momentum: Measure of the "mass in motion" of an object. It's calculated by multiplying the object's mass by its velocity.

- Avoid resetting the speed when the player lands on the ground.

2.1.5. Ability Systems

It is a challenge to design and implement ability systems for a fast-paced sports and shooter game. In hero shooters, the characters' uniqueness, the strategic depth and the balance of the game are defined by these ability systems that need to ensure competition and at the same time maintain fast responsiveness by handling the cooldowns, activations and in an efficient manner the interactions.

When developing the ability systems, memory management must be efficient, the event-driven architecture has to handle multiple players casting simultaneously without causing drops to the frame rate.

It is needless to say that they are required to be synchronized for all the players in multiplayer games while compensating for the lag.

2.1.5.1. Common Approaches

Abilities systems have to be modular, scalable and performant to ensure a smoother multiplayer environment. In this section will be explained the most common architectures for implementing them and their comparisons with simple code examples.

In the component-based system, the abilities are implemented as separate modules that can be attached, removed, or modified dynamically to an entity, in this case, a character or player.

```
C/C++
public class AbilityComponent : MonoBehaviour
{
    public float cooldown;
    public float lastUsedTime;

    public virtual void Activate() { }
     public bool CanUse() { return Time.time >= lastUsedTime + cooldown;
}
}
```

A data-driven system stores the properties of each ability in external files like JSON, XML or ScriptableObjects, defining them using data and being executed by an ability loader.

```
C/C++
JSON
{
    "abilities": [
        {
            "name": "Dash",
            "cooldown": 3.0,
            "effect": "MovementBoost",
            "parameters": { "force": 10 }
        }
    ]
}
[System.Serializable]
public class Ability
{
    public string name;
    public float cooldown;
    public string effect;
}

public class AbilityLoader
{
    public static Ability LoadAbility(string jsonData)
    {
        return JsonUtility.FromJson<Ability>(jsonData);
    }
}
```

There is the Finite State Machine (FSM)-based system, where each ability represents a state in a state machine.

```
C/C++
public enum AbilityState { Idle, Charging, Active, Cooldown }
```

```
public class AbilityFSM
{
    public AbilityState currentState;

    public void TransitionTo(AbilityState newState)
    {
        currentState = newState;
    }
}
```

And the event-driven ability system trigger the abilities by using events instead of polling them each frame. This is a good choice for an efficient synchronization in multiplayer games.

```
C/C++
public class AbilityEventSystem
{
    public delegate void AbilityActivated();
    public static event AbilityActivated OnAbilityUsed;

    public static void UseAbility()
    {
        OnAbilityUsed?.Invoke();
    }
}
```

2.1.5.2. Choose a Model

For a multiplayer fast-paced sports-based hero shooter game, the best option is a hybrid approach implementing component-based and event-driven systems. This combination grants the modularity, scalability and optimal performance needed for multiple hero abilities and also responsiveness in real-time in an efficient way. The abilities as components allow more flexibility and scalability while utilizing event-driven mechanisms ensures efficient execution and communication between systems.

2.2. Theoretical Framework

In this section will be explained some definitions and historical facts related to the project's objectives and research.

2.2.1. Game and Gameplay Mechanics

Game mechanics are a set of rules and systems that dictate how the player interacts with a game and how the game responds to the input of the player. They are not the same as the game dynamics, which describe emergent behaviors that arise from the mechanics, and game aesthetics, which are related to the experience of the player.

According to Alexander Brazie in *Video Game Mechanics: A Beginner's Guide (with Examples)* *"Gameplay is the flow between challenges, mechanics and outcomes"*. This means that gameplay mechanics are the specific ways that affect directly the flow of the game and the player interactions, how it feels and progresses. Meanwhile, game mechanics refer to the fundamental rules and elements that establish how the layers interact with the game structure.

A comparison to understand better the differences between game and gameplay mechanics would be the following:

Table 2.2.1.1: Game & Gameplay Mechanics Comparison

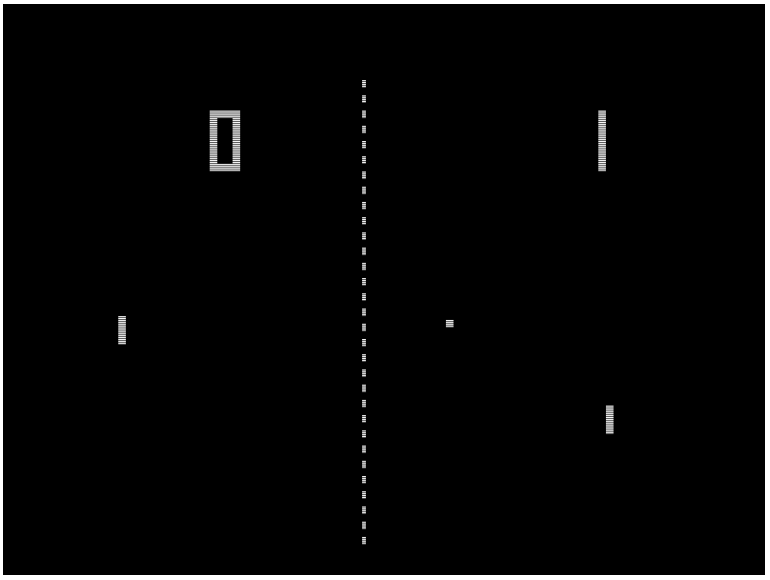
Aspect	Game Mechanics	Gameplay Mechanics
Definition	The core rules and elements governing how a game works.	The mechanics that influence directly the flow and feel of the game.
Example	Health systems, movement physics, enemy AI behavior.	How health regenerates, how movement interacts with the environment, how enemies react dynamically.
Role in Flow	Provides a foundation for interaction.	Affects the pace, the challenge and the engagement.

2.2.1.1. Evolution of Game and Gameplay Mechanics Over Time

The influence of technological advancements and the evolution of the expectations from the players have transformed the development of game and gameplay mechanics since the origin of video games.

In the late 1970s and 1980s, the technological limitations of these years made video games to have straightforward mechanics offering simple interactions. In games like *Pong* (1972) the players control paddles to hit a ball back and forth, while in *Space Invaders* (1978) was introduced the foundation for the shooter genre with basic shooting mechanics.

Figure 2.2.1.1.1: Pong Screenshot



(Source: Wikipedia Pong Page)

In the 1990s, players experienced the implementation of storytelling and exploration elements in titles like *The Legend of Zelda: Ocarina of Time* (1998) which offered a more immersive experience to the players through the combination of action with puzzle-solving and narrative, marking the blending of various genres by going towards gameplay mechanics that were more complex.

In the early 2000s, technology advanced and it was easier to create open-world games that implemented mechanics that supported non-linear gameplay emphasizing the diverse possibilities of the player's interaction like *Grand Theft Auto III* (2001).

The mid-2000s brought realistic physics engines into gameplay mechanics. Valve Corporation developed *Half Life 2* (2004) using the Source engine, they integrated puzzles and interactions based on physics improving the immersion and interactivity with advancements that allowed them to create worlds with more dynamics and credibles.

Then, in the 2010s procedural generation became outstanding by being implemented in games like *Minecraft* (2011) that allowed to generate vast and unique worlds, this offered to the players endless possibilities of exploration and creativity.

Finally, from the late 2010s to the present day, recent technological advancements are focused on adaptive artificial intelligence letting developers to create responsive and personalized gameplay. Thanks to it, the enemies in games like *The Last of Us Part II* (2020) challenge the players and are more immersive by reacting dynamically to the actions of the players.

2.2.1.2. Future Trends in Movement Mechanics

Nowadays, VR and AR technologies are evolving to the point that further on will blur the boundaries between the virtual and physical worlds. This will allow the developers to be able to design and implement movement mechanics that fit with real human movements to create a more immersive experience. For instance, *Half-Life: Alyx* (2020), demonstrates the potential and immersion that virtual reality can have. In the case of AR, there is the app NBA Vision Pro that implements augmented reality to display a small basketball court in 3D during live games.

Figure 2.2.1.2.1: Half-Life: Alyx Screenshot

(Source: Steam Half-Life: Alyx Page)

There is also the integration of haptic feedback and wearable technology that boosts the immersion experience by making the players physically feel sensations related to in-game interactions, such as a shot or an explosion. There are different devices that

provide tactile feedback like gloves, vests, or even full-body suits. A recent and innovative technology is Disney's HoloTile Floor which is described as the "world's first multi-person, omnidirectional, modular, expandable, treadmill floor.". This allows the users to move in any direction while remaining physically in the same space.

Figure 2.2.1.2.2: Lanny Smoot on the HoloTile Floor

(Source: The Walt Disney Company)

2.2.2. Ability Interactions

Abilities are specific powers or actions that a character can perform being differentiated from the own skills of the player, such as spells or special attacks. These abilities make the character have unique traits and capabilities.

An ability system is a defined framework that manages and executes these character abilities and how they are acquired and upgraded enveloped by the rules and the mechanics of the game. For instance, there is Unreal Engine's Gameplay Ability System that is described as a "highly-flexible framework for building abilities and attributes of the type you might find in an RPG or MOBA title."

2.2.1.1. Evolution of Abilities Interactions Over Time

In the 1980s, games like *Final Fantasy* (1987) which were within the role-playing games genre (RPGs), were the first to introduce character classes with different abilities.

Then, the late 1990s and early 2000s saw in games like *Diablo II* (2000) the emergence of different playstyles and the customization of the character abilities, what is called skill trees.

Action-adventure titles started to implement abilities used for unlocking new areas and puzzles mixing it with action, one example is *The Legend of Zelda* series (1986-2024).

Now, modern games focus on extensive customization of the abilities to their preferences, enhancing the engagement and strategic depth of the player.

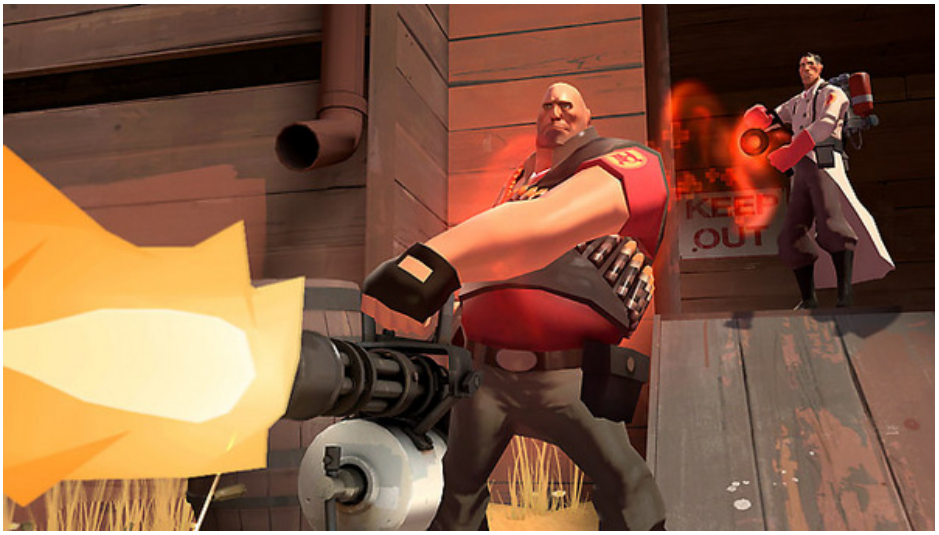
2.2.3. Hero Shooter Games

Hero shooter is a video game subgenre of shooter games that has a character-centric gameplay, focuses on teamwork and strategic hero selection where each hero has unique abilities and weapons.

2.2.3.1. Origins and Evolution of Hero Shooter Abilities

Team Fortress 2 is one of the first hero shooters and with its highest peak of players at the same time of 161.417[2] on July 9th, 2024, developed by Valve Corporation and released in 2007. The game contains several game modes, maps, and equipment. The players can choose between nine different classes and each one offers unique abilities, and even as it says on the Steam store site, personalities. The player can customize the character classes to suit their liking and the way to play. For instance, the Medic is a class that can heal other players from your team, while there is the Spy that can disguise of an adversary and infiltrate enhancing cooperative gameplay due to the players have to think about the other's abilities.

Figure 2.2.3.1.1: Team Fortress 2 Screenshot



(Source: Steam Store TF2 Page)

[2] Data collected from SteamDB.info looking for the highest peak of players from the past year 2024 (https://steamdb.info/app/440/charts/#48h)

The design of hero shooters is a crossover between the abilities and roles of the Multiplayer Online Battle Arenas (MOBAs) within shooter mechanics that creates a blend of gameplay that is oriented to action with strategic depth.

Dynamic skill upgrades are abilities that can be upgraded or modified in real-time during the matches enabling the players to adapt the playstyle in a dynamic way to counter their opponents or adapt to the scenarios, for example, there is *Battleborn* (2016). Overwatch 2 allowed players to adapt the hero's abilities during the matches by offering minor and major upgrades, such as cooldown reductions, by introducing a Perks system. On the other side, a feature that has become also essential, is visual customization, allowing players to change their characters' appearances.

2.2.3.2. Advanced Movement in Hero Shooters

Advanced movement in hero shooters offers players more control and strategic options that influence substantially the gameplay dynamics. This type of mechanics also contributes to the competitive and strategic depth of hero shooters.

There were early innovations in movement mechanics in the 1990s that emphasized speed and verticality, rocket jumping and strafing. For instance, were implemented in *Quake* (1996) and *Unreal Tournament* (1999), which laid a base for more complex movement systems in future games.

Titanfall 2 (2016) is a game where one of the most important movement mechanics is to be able to run along the walls, it also implements double-jump and slide, which the combination of all three mechanics creates a more fluid and dynamic combat.

In *Overwatch* (2016) the hero abilities were combined with different movement mechanics. For instance, the character called Tracer can perform a maneuver called "blink" that teleports the player a short-distance.

Figure 2.2.3.2.2: Titanfall 2 Screenshot

(Source: Steam Store Titanfall 2 Page)

In games like *Deadlock* (2023) are implemented features that add more complexity layers and control allowing the players to outmaneuver other opponents and obtain tactical advantages, these features are dash-jumps and slides.

These advanced movement systems enhance the player's immersive experience and also contribute to the competitive and strategic depth of hero shooter games.

2.2.4. Sports-based Games

In sports-based video games the players immerse in athletic experiences that simulate and recreate traditional sports.

2.2.4.1. Origins and Evolution

In the late 1970s and early 1980s, sports-based games began appearing. The first game based on sports was *Pong* (1972) developed by Atari, it was not a directly simulation of a sport but recreated the concept of table tennis with two simple paddles and a ball rendered in the screen.

In 1978, a video game, developed by a physicist called William Higinbotham, is considered as one of the first true sports games, *Tennis for Two* featured a tennis match for two players with simple physics.

Around the mid-1980s, thanks to the graphics and processing power advancements, the game developers started to create more accurate representations of real-world sports.

For instance, *Intellivision's World Series Baseball* (1983) and *Tecmo Bowl* (1987) even though they were more authentic representations of real sports, the available technology of that time was still limited.

In the late 1980s and early 1990s, Electronic Arts Sports team developed *John Madden Football* (1988) and *FIFA International Soccer* (1993) whose gameplay mechanics were more complex and implemented advanced graphics, becoming popular video game franchises.

By the 2000s, games like *NBA 2K* and *FIFA* started using newer technological advancements, such as motion capture and artificial intelligence, to implement realistic animations and more strategic gameplay. Also, true-to-life physics and detailed player models were improvements that made sports-based games to be closer to a real-life sports experience.

Nintendo Wii's motion-sensitive controllers also allowed the players to participate in the gameplay physically, adding a layer to the immersion experience in games like *Wii Sports* (2006), that included a variety of game modes related to different sports such as tennis and bowling.

Sports-based games play an important role in the globalization of sports, allowing the players from different countries to experience sports that maybe they do not have access to.

2.3. Market study

There are lots of games that follow hero shooter or sport-based genre style, two of the most important or that created a big impact in the industry within these genres are *Overwatch* and *Rocket League.*

2.3.1. Overwatch 1&2

Overwatch (2016) and its sequel *Overwatch 2* (2022) are team-based first-person shooter, launched by Blizzard Entertainment, that popularized the hero shooter genre.

Figure 2.3.1.1: Overwatch 2 Screenshot

(Source: Blizzard News Page)

Since its launch, Overwatch 2 attracted over 50 million active users and generated $225 million in revenue, indicating a strong initial engagement. Overwatch maintains a significant share of the hero shooter market, regardless of the competition that faces from emerging titles.

Blizzard Entertainment transitioned Overwatch 2 to a free-to-play model, this made the game generate the revenue through in-game purchases such as cosmetic items or a battle pass.

Overwatch was promoted as an esport game attracting more audience and sponsorships thanks to the Overwatch League, established in 2018.

2.3.2. Rocket League

Rocket League is a sports-based game, launched by Psyonix in 2015, where users play a football match controlling a rocket-powered car with physics-based mechanics. Its unique concept and engaging mechanics have contributed to its enduring popularity.

Figure 2.3.2.1: Rocket League Screenshot

(Source: Rocket League Page)

The game maintains a robust player base, assisted by its free-to-play model and cross-platform capabilities, fostering a diverse and active community. It occupies a unique niche that contributes to its sustained market presence, thanks to a limited direct competition.

Rocket League has achieved significant financial success, with estimates suggesting total revenue approaching $943 million, including earnings from game purchases and in-game transactions.

Since 2016, Rocket League Championship Series (RLCS) has cultivated a vibrant esports scene, with regular tournaments and a dedicated fan base.

Rocket League employs a free-to-play approach, monetizing through cosmetic items, seasonal passes, and in-game events. And the game features an active item trading community, with platforms like OP.Market providing real-time price tracking for in-game items.

3. Project management

In this section we will look deeply into the management of this project and the tools that will be used to achieve good management, a SWOT analysis of what to take into consideration for the development of the game, the risks that could endanger the project, and its possible solutions, and an initial cost analysis.

Keeping in mind that this project is being developed by a team composed by three people and the roles that each person has, and the software and hardware that are going to be used.

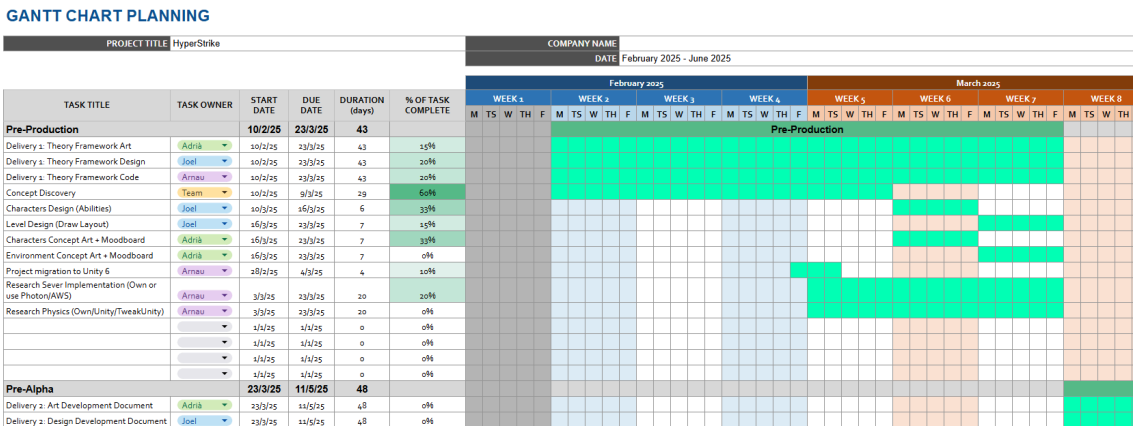
3.1. Methodology and Tools

3.1.1. GANTT

A Gantt chart is a tool for visualizing the planning of the project phases and their length. On the left side there are the main tasks for each phase of the project, together with the team members assigned to them, each one in a different color, the starting and ending dates with the total duration in days between both, and the progress in percentage of the task if it is currently being completed.

Then, on the right side, there are the months distinguished by colors, below the weeks and days. This part is made to see how much time will last each task, so it will be colored the squares from the starting date of the task until the ending date.

Figure 3.1.1.1: Gantt Chart Planning



GANTT CHART PLANNING

PROJECT TITLE: HyperStrike — COMPANY NAME — DATE: February 2025 - June 2025

TASK TITLE	TASK OWNER	START DATE	DUE DATE	DURATION (days)	% OF TASK COMPLETE
Pre-Production		**10/2/25**	**23/3/25**	**43**	
Delivery 1: Theory Framework Art	Adrià	10/2/25	23/3/25	43	15%
Delivery 1: Theory Framework Design	Joel	10/2/25	23/3/25	43	20%
Delivery 1: Theory Framework Code	Arnau	10/2/25	23/3/25	43	20%
Concept Discovery	Team	10/2/25	9/3/25	29	60%
Characters Design (Abilities)	Joel	10/3/25	16/3/25	6	33%
Level Design (Draw Layout)	Joel	16/3/25	23/3/25	7	15%
Characters Concept Art + Moodboard	Adrià	16/3/25	23/3/25	7	33%
Environment Concept Art + Moodboard	Adrià	16/3/25	23/3/25	7	0%
Project migration to Unity 6	Arnau	28/2/25	4/3/25	4	10%
Research Sever Implementation (Own or use Photon/AWS)	Arnau	3/3/25	23/3/25	20	20%
Research Physics (Own/Unity/TweakUnity)	Arnau	3/3/25	23/3/25	20	0%
		1/1/25	1/1/25	0	0%
		1/1/25	1/1/25	0	0%
		1/1/25	1/1/25	0	0%
		1/1/25	1/1/25	0	0%
Pre-Alpha		**23/3/25**	**11/5/25**	**48**	
Delivery 2: Art Development Document	Adrià	23/3/25	11/5/25	48	0%
Delivery 2: Design Development Document	Joel	23/3/25	11/5/25	48	0%

(Own Source)

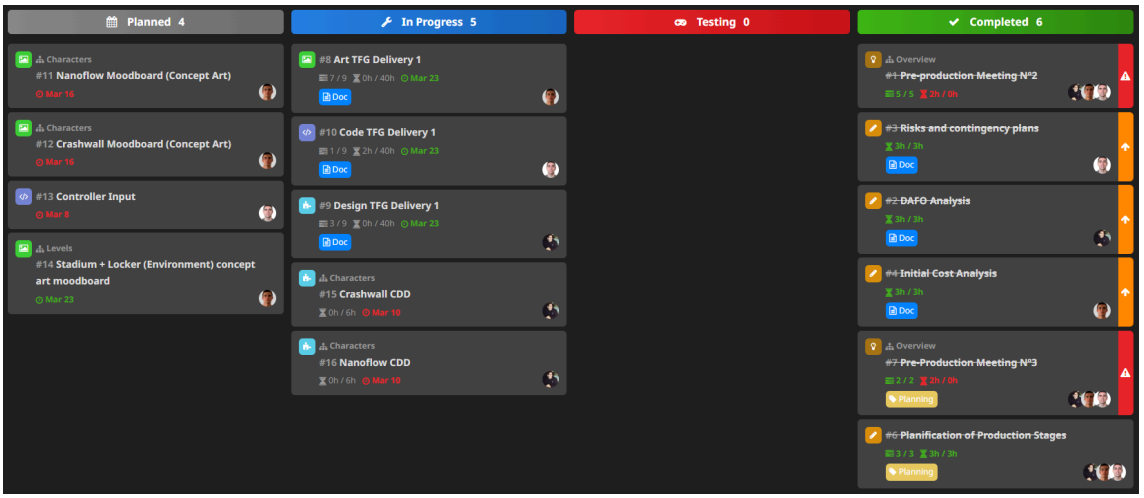
3.1.2. HacknPlan

On the other hand, HacknPlan works with an Agile methodology approach where the tasks presented in the previous Gantt chart are going to be more detailed adding

descriptions, subtasks, hours spent working on each task, and with more continuous feedback that also shows the current state of the task.

For each Gantt chart phase presented, such as Pre-production or another one, there will be sprints inside it. Sprints are shorter phases that focus on the implementation of specific tasks.

Figure 3.1.2.1: HacknPlan Table

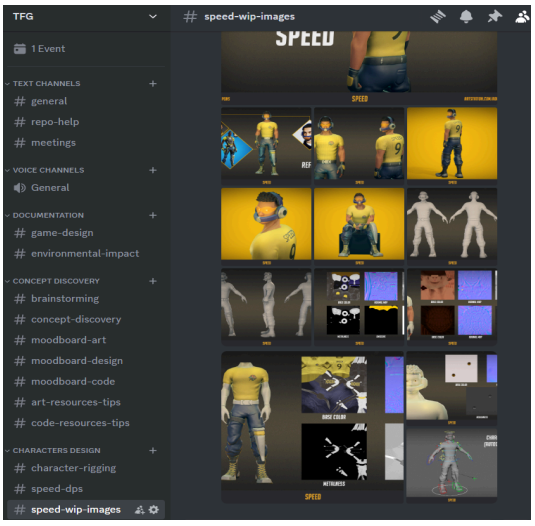


(Own Source)

3.1.3. Discord

Discord is the tool where the team will have weekly meetings in voice calls, stated as events and defined during the Pre-production phase to be each Tuesday at 10:30 am. Also is the place where progression, ideas and some other resources that need to be notified as fast as possible to the team will be shared.

Figure 3.1.3.1: Discord Server



(Own Source)

3.2. SWOT

A SWOT analysis is a way to analyze the strengths, weaknesses, opportunities and threats in its specific target market and strengthen one's business strategies.

Table 3.2.1: SWOT

	Positives	Negatives
Internal Origin	**Strengths** Stylized Art style Unique in its genre Strong visual identity Unity's multiplayer solutions	**Weaknesses** Many FPS games in the market Can become repetitive with only one game mode Small development team Non-balanced experience
External Origin	**Opportunities** Unexplored genre, big potential market Mix of the two most popular game genres Streaming and content creation	**Threats** Existing games could create similar game modes Technical difficulties with servers Multiplayer game market saturation Low player engagement

3.3. Risks and contingency plans

It is very important to detect the risks that could endanger the work and look for solutions so that, if necessary, the project can be resumed.

The possible risks identified for this project, and their corresponding solutions, are as follows, ordered from least to most important:

Table 3.3.1: Risk and contingency plans

Risk	Solution
Running out of time to develop the project	Implement a Gantt development approach using Hack'n Plan and Google Sheets with milestone deadlines.
Team communication and coordination issues	Set up a structured communication channel via a *Discord* server. Do regular meetings to check and align the team on progress.

Server hosting and networking instability	Use a reliable server architecture and test networking with simulated high-latency conditions. Implement algorithms to ensure stability and smooth gameplay.
Performance issues on lower-end PCs	Optimize physics simulations. Profile the game regularly for bottlenecks.
Bugs and exploits	Do QA testing and track bugs. Try to add an anti-cheat system to avoid exploits.
Hero abilities complexity	Use modular code design to make analysis and balancing easier.

3.4. Initial cost analysis

Like any other commercial project, there is a large list of costs regarding its whole production. In this project, the costs have been calculated taking into account all the members of the team, so all costs will be grouped.

The level of developers has been considered Junior, and there are 3 people: 1 Game Designer, 1 3D Artist, and 1 Gameplay Programmer. The number of hours of work per week was 20, and as a consequence salary has been computed based on part-time dedication.

Table 3.4.1: Salaries

SALARIES						
Job Role	**Annual Revenue (avg.)**	**Price/Hour**	**Weekly Work Hours**	**Weeks of Work**	**Total Work Hours**	**Total Cost**
Junior Game Designer	22.000€	11,22€	20	18	360	4.040,82€
Junior 3D Artist	20.000€	10,20€	20	18	360	3.673,47€
Junior Gameplay Programmer	24.000€	12,24€	20	18	360	4.408,16€
TOTAL						**12.122,45€**

The total cost of salaries for 3 developers during 5 months of development would be 12.122,45€. This estimation has been computed considering the average annual

revenue[3] of each job role in Spain. Thanks to these numbers, it has been possible to calculate the price per hour, the total hours of work, and as a consequence, the total cost.

Then there are the costs related to the Software used and its monthly fees. In the following table, it has been computed all the necessary costs regarding the different software used during the overall production. Most of the products have an educational or student license, and as the developers were eligible for these plans, the majority of the monthly fees were free of charge.

Table 3.4.2: Software costs

SOFTWARE		
Name	**Monthly Fee**	**Total Cost (5 months)**
Unity Game Engine (Personal)	0,00€	0,00€
Autodesk Maya (Educational)	0,00€	0,00€
Adobe Photoshop (Students)	35,57€	177,85€
Adobe Substance 3D Suite (Education)	0,00€	0,00€
Maxon ZBrush (Monthly)	55,35€	276,75€
DaVinci Resolve	0,00€	0,00€
TOTAL		**454,60€**

It has also been taken into consideration all the hardware necessary to realize the project. In this case, it has been computed the cost per month based on the lifespan estimation[4] of each product for the total time of development (5 months).

Table 3.4.3: Hardware costs

HARDWARE					
Product	**Price**	**Units**	**Lifespan Estimation (Months)**	**Monthly Amortization (per unit)**	**Cost (5 months)**
Computer	1.199,00€	3	72	16,65€	249,79€
Mouse	29,99€	3	24	1,25€	18,74€
Keyboard	49,99€	3	48	1,04€	15,62€
Monitor	149,00€	6	60	2,48€	74,50€
Digital Graphics Tablet	79,99€	1	36	2,22€	11,11€

[3]Average annual revenues obtained from Glassdoor.com on February 20, 2025. See the salaries table for each salary consultancy link.
[4] Lifespan estimation of office equipment (https://www.pdq.com/blog/equipment-lifecycle-management-guide)

Headphones	69,99€	3	36	1,94€	29,16€
TOTAL					**398,93€**

Last but not least, it has also considered some indirect costs like electricity consumption and Internet fees. In the electricity price[5], it has been taken the average price of January 2025 (0,1598 €/kWh). For the Internet monthly fee[6], it has been considered a 1Gb plan, with a price of 31€/month.

Table 3.4.4: Electricity costs

ELECTRICITY					
Material	**Watts/h**	**Daily Work Hours**	**Price kWh (Avg.)**	**Daily Cost**	**Cost (5 months, 3 developers)**
Gaming Computer	300	4	0,1598	0,192 €	51,775 €
2 Monitors	44	4	0,1598	0,028 €	7,594 €
TOTAL					**59,369 €**

Table 3.4.5: Internet costs

INTERNET		
Product	**Price (Monthly)**	**Cost (5 months)**
Internet Fee	31,00 €	155,00 €
TOTAL		**155,00 €**

Finally, after calculating all costs (direct and indirect), it can be said that the complete production of the project would have a price of 13.190,35€.

Table 3.4.6: Total project costs

TOTAL PROJECT COSTS	
Item	**Cost**
Salaries	12.122,45 €
Software	454,60 €
Hardware	398,93 €
Electricity	59,37 €
Internet	155,00 €
TOTAL	**13.190,35 €**

[5]Average monthly electricity cost Spain on February 20, 2025 (https://www.costeenergia.es/historico/pvpc-anual)
[6] Price from O2 company on February 20, 2025 (https://o2online.es/)

3.5. Environmental impact and social responsibility

In this section will be calculated the carbon footprint that will produce the entire project having in mind and putting together all the phases of the development and even the end of life of it.

In the first sheet it is calculated the usage emissions per hardware with the emission factor[7]. Here we have in mind that hardware components that are directly connected to the computer for its usage and power are using the energy that the computer consumes.

Table 3.5.1: Hardware Usage Emissions

HARDWARE USAGE EMISSIONS							
Device	**Total Hours of Use (h)**	**Energy Consumption (W)**	**Energy Consumption (kWh)**	**Units**	**Total Energy Consumption (kWh)**	**Emission Factor (kg CO2/kWh) - Avg. Spain 2024**	**CO2 Emissions (kg)**
Computer	360	300	108	4	432	0,108	46,656
Monitor	360	40	14,4	6	86,4	0,108	9,3312
TOTAL							**55,9872**

Then, calculate the amortization emissions per production, lifespan, and working time of each hardware component that is going to be used to develop the project. We take into consideration that it will be used a third-party server to host the multiplayer game services.

Table 3.5.2: Hardware Amortization Emissions

HARDWARE AMORTIZATION EMISSIONS							
Device	**Production Carbon Footprint (kg CO2)**	**Lifespan (Years)**	**Working Days**	**Working Hours /Day**	**CO2 Emissions x Hour of Use (kg CO2)**	**Total Working Hours**	**CO2 Emissions (kg)**
Computer	778	6	90	4	0,360	360	129,67
Monitor	63	5	90	4	0,035	360	12,6
Keyboard	19,58	4	90	4	0,014	360	4,895
Mouse	5,86	2	90	4	0,008138888889	360	2,93
Digital Graphics	35	3	90	4	0,032	360	11,67

[7] Average emission factor in Spain in 2024 (https://www.nowtricity.com/country/spain/)

Tablet							
Headphones	12,89	3	90	4	0,01193518519	360	4,296666667
Server	6000	5	60	24	0,83	1440	1200
TOTAL							**1366,06**

In this sheet, it is calculated the usage of services such as Github, a game server, or even the queries answered by an AI ChatBot, in this case ChatGPT.

Table 3.5.3: Network and Cloud Services Usage Emissions

NETWORK & CLOUD SERVICES USAGE EMISSIONS				
Service	**Quantity**	**CO2 Factor (kg CO2/unit)**	**Estimation Details**	**CO2 Emissions (kg)**
Github Commits	285 commits	0,0000024 kgCO2/commit	**Assumption:** A single commit transfers ~100 KB of data. **Energy Consumption:** ~0.06 kWh per GB translates to 0.06 kWh/GB × 0.0001 GB ≈ 0.000006 kWh per commit. **Emission Factor:** 0.4 kg CO2/kWh. **CO2/Commit:** 0.000006 kWh × 0.4 kg/kWh = 0.0000024 kg.	0,000684
Google Drive Storage	5 GB (annual storage)	0,08 kgCO2/GB/year	**Emission Factor:** Approximately 0.08 kg CO2/GB/year.	0,4
AI Queries	100 queries	0,00502 kgCO2/query	**Assumption:** A single query is estimated to produce around 0.00502 kg CO_2 (using online query energy estimates and model training).	0,502
Game Server (16GB)	288 kWh	0,4kgCO2/kWh	**Power:** Assumed constant power of 200 W (0.2 kW). **Total Hours:** 60 days × 24 h = 1440 h. **Energy Consumption:** 0.2 kW × 1440 h = 288 kWh. **Emission Factor:** 0.4 kg CO2/kWh.	115,2
TOTAL				**116,103**

The following table shows the usage emissions per user that is going to play the game. Once more, just having in mind the computer and the monitor as in the first table of this section.

Table 3.5.4: Users Usage Emissions

USERS USAGE EMISSIONS							
Activity	**Power (W)**	**Power (kW)**	**Time per User (h)**	**Units**	**Quantity**	**Emission Factor (kg CO2/kWh) - Avg. Spain 2024**	**CO2 Emissions (kg)**
Device Power Usage (PC)	250	0,25	1,5	100	37,5	0,108	4,05
Device Power Usage (Monitor)	40	0,04	1,5	100	6	0,108	0,648
TOTAL							**4,698**

Table 3.5.5: Usage Impact

Usage Impact	**Positive Impacts**	**Community Building**	Games can be useful as a common way for players from diverse backgrounds to create connections and a sense of belonging between them. Cooperative/team gameplay and challenges can create friendships and collaborative relationships.
		Communication & Teamwork	Multiplayer games often require a good communication and strategic plans. These interactions can boost interpersonal skills like communication.
		Stress Relief	For many people, games offer an engaging escape from everyday pressures. Positive social interactions within the game can provide emotional wellbeing and stress relief.
	Negative Impacts	**Social Isolation**	There is a risk that excessive play might lead to reduced face-to-face social interactions. This could potentially result in isolation if the online world begins to substitute real life.
		Toxicity	Online competitive games sometimes generate aggressive behavior or toxic interactions between players. It can lead to a bigger and worse problem.

Table 3.5.6: Project's End of Life Negative Impacts

Project's End of Life Negative Impacts	**Continuous Energy Consumption**	The centers where servers from Github are hosted need continuous power to be operative. Even though one single project might not use barely any resources, the sum of millions of repositories increases the energy consumption in the long term, and as a consequence, the carbon footprint, especially if energy isn't renewable.
	Utilization & Upgrades of Server's Hardware	The centers where the data is stored need to stay updated by replacing hardware to improve its performance. Keeping old projects that aren't being used anymore can contribute to the frequent update of the hardware, which generates electronic waste and potential recycling issues.
	Increase in Network Traffic & Maintenance	The fact of having an inactive project uploaded to the cloud services contributes to the network traffic infrastructure, which needs constant maintenance, energy, and all kinds of resources that contribute to a negative environmental impact.

Finally, it is calculated the total CO_2 emissions produced by the entire production and usage of the project. The team, during this process, thought about how could the carbon footprint be countered, so we researched how many trees would be needed to be planted.

Per year, a tree absorbs 21,77 kg of CO_2[8], so we did some calculus. The project will last four months, which is equivalent to 7,26 kg of CO_2 a tree, then we multiply it by the total CO_2 emissions in kilograms and we get the total number of trees needed to absorb the total emissions that this project will produce.

Table 3.5.7: Total CO_2 Emissions

TOTAL CO2 EMISSIONS	
Activity	**Total CO2 (kg)**
Hardware Usage	44,32
Hardware Amortization	1366,06
Home Office Environment Usage	14,20
Network & Cloud Services Usage	116,10
User Usage	4,70
TOTAL	**1545,37**

[8] Calculation of CO_2 offsetting (https://www.encon.eu/en/calculation-co2)

Table 3.5.8: Total Trees Needed

Total Trees Needed
213

4. Methodology

4.1. Programs and Tools

It is important to know which programs and tools are required for a project of this scale, choosing the right tools and programs will ensure good team communication, task management, version control, and development.

4.1.1. Google Documents and Sheets

Google Documents and Sheets are tools from Google Workspace that allow documentation and organize data while enabling a seamless collaboration. Documents will be used for writing the Game Design Document, technical documentation, and the thesis reports of each team member. Sheets will be used to track the milestones and for budgeting.

4.1.2. HacknPlan

HacknPlan is a tool used for project management made specifically for game development that unifies agile methodologies and a Kanban board system to organize the tasks, responsibilities assignments and track the progress of the development.

4.1.3. Discord

Discord nowadays is one of the best (written chat and voice chat) programs that are for communicating, its main purpose is to make a reliable way of communication between players but the features it offers to the users make it also a good platform to communicate for small projects.

4.1.4. Github

GitHub is a version control platform made for code management and project tracking allowing an efficient and simultaneous collaboration where multiple developers can work at the same time without conflicts. It will be used to track, store, and update code changes with issue tracking and pull request features that makes easier code reviews and bug fixing.

4.1.5. Visual Studio 2022

Visual Studio 2022 is an integrated development environment (IDE) that is equipped with some important features like code debugging, syntax highlighting, performance profiling and gives code suggestions using IntelliSense. It supports the main programming language that is used for Unity development, C#.

4.1.6. Unity

Unity is a real-time development engine that implements tools that allows to work with game physics, such as PhysX, 2D and 3D rendering, animation, scripting and multiplayer support. The C# scripting language, together with Monobehaviour, allows developers to create complex mechanics and multiplayer interactions.

4.1.7. PhysX

PhysX is a real-time physics engine developed by NVIDIA that integrates physics simulations with rigid body dynamics and collision detection. It is Unity's built-in physics system highly important for handling realistic physical interactions in a game.

4.1.8. Unity Multiplay

Multiplay is a scalable game server hosting platform equipped with features that removes complexities for developers that are new to multiplayer games development.

4.1.9. Unity Matchmaker

Matchmaker is a multiplayer feature included in Unity's multiplayer services that provides matchmaking features allowing players to connect based on some information such as connection quality and skill level.

4.1.10. Affinity Photo 2

Affinity Photo 2 is high-end image editing software developed by Serif, used for photo retouching, digital painting, compositing, and graphic design.

4.2. Development Phases

The main idea is to develop what is called a "Vertical Slice" for the final deliverable of this thesis.

4.2.1. Pre-production

In this phase of the project, the team researches and defines almost all the necessary to start with the production of the game. The roles of each person in the team are assigned together with the responsibilities it will have.

It is established the concept of the game and the core mechanics that will define the gameplay, as well as the game engine, networking solutions, libraries, and other necessary tools.

The team decides the timeline for each phase of the project and their deadlines. This phase lasts from February 10th, 2025 to March 23, 2025.

4.2.2. Pre-alpha

This phase is focused on building the first functional prototype of the game, focusing on gameplay mechanics, abilities, basic visuals, and test features. Implement a basic multiplayer structure for basic online multiplayer and interactions.

There will be implemented basic and temporary models as placeholders for the assets, stadiums, characters and UI, such as any basic gameplay HUD element.

The pre-alpha phase will last from March 23 to May 11th, 2025.

4.2.3. Alpha

During this phase, the game should have most of the essential mechanics in place. Is focused on improving the experience of the game and refining features to prepare it for external playtesting.

The character controls, ball mechanics, abilities and physics interactions should be almost finished. At this point, the multiplayer architecture should already support online matchmaking together with game lobbies and latency handling. Some basic versions of the stadiums, environments and interactions should be already implemented.

The alpha phase will last from May 11th to June 8th, 2025.

4.2.4. Beta

This is the final phase, the Vertical Slice should be completed and ready for a demonstration of the research done and the development. All the features have to be finished and polished, the game should run smoothly with minimal crashes.

It also must be implemented working multiplayer with reliable online support and minimum physics desynchronization issues while the essential mechanics work according to the project's objectives. Some minor bugs and issues can be noted for future iterations but they should not impact the core functionality.

Finally, the beta will last from June 8th to June 30th, 2025 when the final prototype of this project should be delivered.

4.3. Testing Phases

4.3.1. Pre-Alpha Test

Before reaching the Alpha phase of this project development, the implementations done during the Pre-Alpha phase will be tested internally by all three team members.

Testing the mechanics internally is a crucial part of this phase, it ensures that the game is playable and that the core features work as expected. This will help identify if some critical bugs or issues affect the game and address them, the team members will test the game and expose their opinion and experience.

The sessions will be conducted frequently and documented using tools like Google Sheets and HacknPlan to track issues, assign priorities, and propose fixes. During the Pre-Alpha, multiple builds will be generated, and iterative test sessions will be conducted after each major implementation. Bugs and imbalances discovered will be discussed in weekly meetings held on Discord.

The blockout maps done by the team designer will serve as testing scenarios for the implemented mechanics. These simplified environments allow for focused testing on core mechanics such as player movement, ability execution, ball physics, and networking synchronization under controlled conditions.

4.3.2. Alpha Test

In this phase, testing is more focused on both internal and external people to get feedback on the gameplay, the mechanics and the user experience. If there is an unbalanced feature, such as a character ability, critical bugs, performance issues, or crashes, they should be addressed.

A small group of external testers, between 10 to 20 players with a variety of gaming backgrounds, will be selected to play matches in the game. These sessions will be managed remotely with a hosted server using Unity Multiplay services. A feedback process will be done through pre- and post- play surveys using Google Forms to gather qualitative insights, and gameplay sessions may be recorded, with permission, to analyze player behavior and identify unintended mechanics or design flaws.

External playtests will help evaluate the gameplay balance, ability interactions, difficulty curve, performance across different hardware configurations, and the intuitiveness of the UI/UX.

The feedback gathered will be crucial to iterate on game balance, refine abilities, improve network stability, and enhance the overall game experience before transitioning to Beta.

4.3.3. Beta Test

The Beta Test is the final testing phase before submitting the project, focused on validating the overall experience of the game. Testing is also focused on both internal and external people to get feedback to verify the polish, network stability, physics integration with the network, and the completeness of the game in general.

This phase will consist of multiple tests with a wider range of external testers with different backgrounds and skill levels. In an early stage of the phase, the tests will be organized locally and further on online using a cloud hosted dedicated server.

The goal is to observe how the game performs in real use conditions, checking whether the experience is stable, fun, and engaging. It needs to ensure that the game runs smoothly without crashing, while checking if there are frame rate drops, latency, lag or jitter. Also, serves to test the balance of the character mechanics and abilities, and gather feedback through surveys, interviews or gameplay recordings.

During this phase, all the core content should already be implemented, and focus on polishing and fixing bugs found during the tests.

5. Project development

In this section, there is a follow-up of the project's development process, where it is shown how to set it up, the mechanics and abilities implementations, together with the concepts and explanations about network connectivity and performance improvements that make them optimal for a first-person fast-paced sports and shooter game.

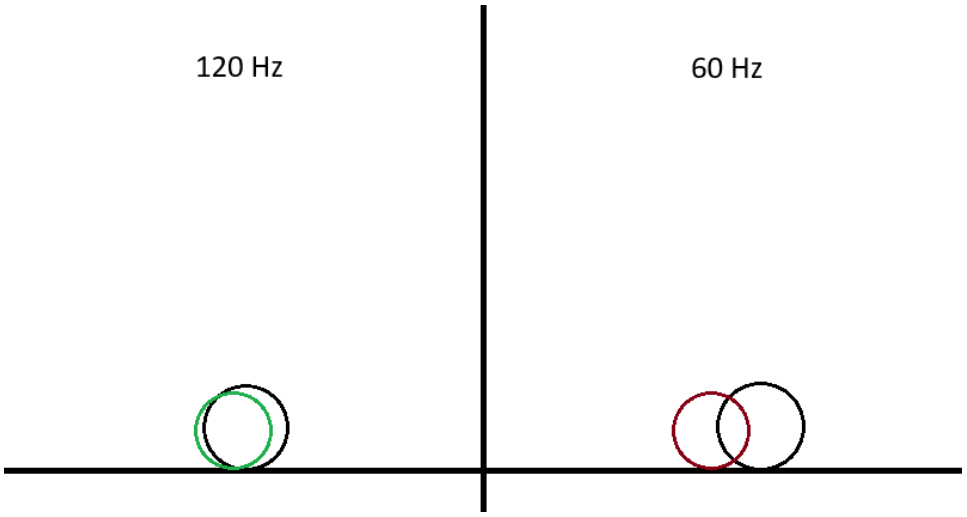
The code snippets of the following sections are simple examples about the techniques used during the development process of the project and are not the final result. The Annexes section contains direct links to the most important scripts of the project, the project is public in Github for more information.

5.1. Project Setting

When creating the project, the latest Unity 6 version released at the moment was 6000.0.43f1. For now, we will start with the *Universal 3D* core template that contains the minimum necessary for starting to develop the project, following the needs of all the team members, on the programming side, any 3D template chosen.

For the physics simulation, a higher frequency is needed due to the fast-paced format of the game, if the frequency is lower, it can create a higher position variation between the player clients for any body in movement. For example, some players will see the ball close to them but others will see the ball a little further. This grants a higher precision for a more reliable synchronization in the network.

Figure 5.1.1: Example - Position variation depending on the simulation frequency

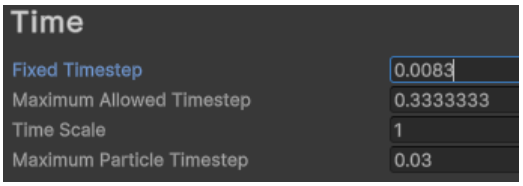


(Own Source)

In the project setting window from Unity it is allowed to change the Fixed Timestep, for this game we want physics to be the most reliable possible so we are going to change it to 120Hz having in mind the study done about the physics simulation.

$$120Hz\colon\ 1/120 \approx 0.0083\ seconds$$

Figure 5.1.2: Project Settings - Time Panel



(Own Source)

5.2. MVC (Model-View-Controller)

MVC is a design pattern that divides, in this case, the player into three primary components. For this project, the main design pattern idea differs a little with the multiplayer implementation:

The Model handles all the player and character data such as health, speed and cooldowns. In this case it also manages some game logic, it handles some character data related behaviour such as applying damage, heal, protection, or death, and subscribes to player analytics events related to the data.

```C/C++
public class Player : NetworkBehaviour
{
	public NetworkVariable<int> Health = new NetworkVariable<int>();
	public NetworkVariable<byte> Team = new NetworkVariable<byte>(0);
	public NetworkVariable<int> Score = new NetworkVariable<int>(0);
	public NetworkVariable<int> Goals = new NetworkVariable<int>(0);
	[SerializeField] private Character Character;

	private PlayerEventSubscriber playerEventSubscriber; // Analytics

	public void ApplyDamage(int damage){}
}
```

The View is responsible for displaying player data on the UI such as score, character name, etc. It listens for updates, like abilities or player stats changes, and reflects them visually through methods subscribed to events. Usually this component is connected to the Controller, but Networking allows to make it simpler just by accessing to the Player (Model) variables.

```C/C++
public class PlayerView : NetworkBehaviour
{
    #region "UI Components"
    #endregion

    Player player;

    private void OnNetworkSpawn()
    {
        player.Health.OnValueChanged += UpdateHealthBar;
    }

    public void UpdateHealthBar()
    {
        float maxHealth = player.MaxHealth.Value;
        float normalizedHealth = Mathf.Clamp01(newValue / maxHealth);
        healthBar.fillAmount = normalizedHealth;
    }
}
```

The Controller is the “brain” within this design pattern. Interprets the player input, controls the movement, attacks, physics, and updates the model. Still, for the project, it has been necessary to separate the behaviour into other components like an Ability Controller (explained further on) to avoid overload a single script.

```CSharp
public class PlayerController : NetworkBehaviour
{
    #region "Model-View Player"
```

```
    private Player player;
    private PlayerAbilityController abilityController;
    #endregion

    PlayerInput input;

    #region "Movement Variables"
    #endregion

    #region "Attack Variables"
    #endregion

    public override void OnNetworkSpawn()
    {
        // Init Player MVC
        player = new Player();
        abilityController = GetComponent<PlayerAbilityController>();

        // Init Physics variables
        // Init Inputs
        // Init Movement variables
        // Init Attack Variables
    }

    private void FixedUpdate()
    {
        // Physics-based + Rigidbody Actions
    }

    #region "Movement Mechanics Methods"
    #endregion

    #region "Attacks and Abilities Methods"
    #endregion
}
```

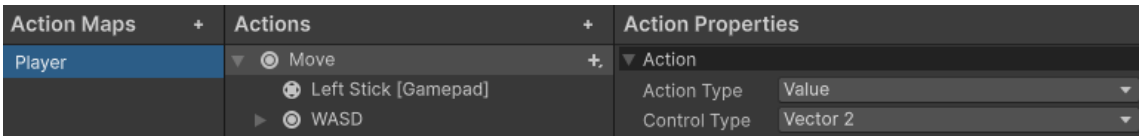
5.2.1. Input System

The project works with Unity's new Input System, which works with Actions. Each Action represents the things a user or player can do in the game. The new Input System allows us to map actions for different input types. For the project, we are going to map Keyboard & Mouse and Gamepad buttons.

It is needed to check if the package is correctly installed or if it needs an update. Then first, each input that is going to be used has to be assigned in the InputSystem_Actions window where an Action Map has to be created, it will contain every Action a player will be able to perform, by setting the Action and Control Type that will define how will be accessed later on in the code and within it, a binding to the controller that will be utilized.

The game will work with keyboard and mouse, and gamepads, so the Actions will contain both binding types as seen in the image below.

Figure 5.2.1.1: InputSystem_Actions - Player Action Map



(Own Source)

Depending on the Action Type, the InputActions will be initialized in a different way, for example, the "attackAction" is a button type instead of a value. This way, it is not needed to put in the "Update()" method the call to the "Shoot()" method, every time the attack button is pressed will automatically invoke the action.

```CSharp
PlayerInput input;

void OnEnable()
{
	input = new PlayerInput();
	input?.Player.Enable();
	InitInputs();
}
```

```
void InitInputs()
{
	input.Player.MeleeAttack.started += ctx => MeleeAttack();
	input.Player.Attack.started += ctx => Shoot();
}
```

5.3. Movement Mechanics Implementation

5.3.1. Player and Camera Rotations

As first choice, there was the normal Unity camera, but it caused jitter when attaching the camera to the player and applying a script to rotate it, while trying to obtain a good first-person behavior.

This happens because, when attaching the camera to the player in any way, it follows the player's Rigidbody position that is updated in the "FixedUpdate()" method, updating its position and rotation in a fixed timestep independently of the framerate, while the camera renders every frame, causing the conflict. There were a few ways to reduce the jitter by code, but none of them fixed it worked perfectly, and doing some research, I found out that there is the Cinemachine Camera package that is recommended because of the improvements it adds to the normal Unity camera. It allows you to do it in an easier way, with smoother transitions, rotation clamping, and applying effects such as camera shake.

During the development process, some tweaks throughout the code were needed for the camera to follow the player's rotation correctly. Even though the Cinemachine Camera handles movement interpolation by itself and fixes jitter issues, it is still needed to update the camera rotation on the horizontal axis while also rotating the player using the Rigidbody to create a correct and smooth first-person camera movement.

```
C/C++
void RotatePlayerWithCamera(Vector2 lookValue)
{
    Vector2 lookValue = lookAction.ReadValue<Vector2>();

    // Get mouse input
```

```
        float mouseX = lookValue.x * sensitivity * Time.fixedDeltaTime;
        float mouseY = lookValue.y * sensitivity * Time.fixedDeltaTime;

        // Adjust xRotation for vertical rotation and clamp it
        xRotation -= mouseY;
        xRotation = Mathf.Clamp(xRotation, -90f, 90f);

        // Update the Cinemachine camera's rotation
        cameraWeaponTransform.localRotation =
                              Quaternion.Euler(xRotation, 0, 0);
        mainCameraTransform.localRotation =
                              Quaternion.Euler(xRotation, 0, 0);

       // Apply horizontal rotation to the player
       rb.rotation = Quaternion.Euler(0, yRotation, 0);
    }
```

In the Cinemachine Brain properties, the update method should be set in “SmartUpdate” and the blend method in “LateUpdate” for a correct interpretation of the camera rotation and to allow us to change other settings correctly during runtime, such as the camera tilt or target. One problem faced is that the blend method at first was set to “FixedUpdate” and was not updating the modified settings in runtime.

5.3.2. Walk and Run

Changing the player position directly using the transform translation gives precise control over the player movement and it is not needed a Rigidbody, but this means that it needs a custom own collision detection and it does not work with other Rigidbodies and physics forces. For the movement mechanics that we wanted to implement, changing the transform position is not the best option in this case scenario. For a physics-based and network handling it has to use Rigidbodies.

There are three ways to modify the player’s position using the Rigidbody component:

- **MovePosition**: It gives precise control over movement and acceleration, and the collisions work correctly if there are not forces involved, forces can cause some issues in the movement simulation with the collisions.
- **Velocity**: It works also correctly with collisions and it overwrites the physics forces adding difficulties when debugging movement problems. Is the easiest one to set up.
- **AddForce**: Works right with the physics forces and collisions, is not as precise as the other methods and requires little more work to get reliable movement.

These options require, in the Rigidbody inspector, set to "Interpolate" the movement and the collision detection to "Continuous" to reduce jittering, "Continuous Dynamic" has a high performance impact if we are working with many objects and it also increases the data sent through the Network, so we try to avoid this last option.

The best option for our game is to work with the last method, reduces some synchronization issues and jittering, it works with forces and velocity changes applying acceleration to the Rigidbody of the players, creating a smooth movement interaction. But it can increase the latency, causing the player to feel some delay when pressing the inputs, explained and fixed forward in the Project Validation section.

The approach works with a speed variable that changes depending on if the player presses a key to activate the sprint. Then for the movement direction, a linear combination of vectors is calculated, multiplying each direction by the corresponding input axis and then adding them together. Finally a force is applied to the player's Rigidbody by multiplying the direction normalized by the desired speed. This approach allows the player rigidbody to react to external forces easier.

```C/C++
void WalkAndRun(Vector2 moveValue, bool isSprinting)
{
    if (isWallRunning && !isGrounded) return;

    float speed = isSprinting && isGrounded ? player.SprintSpeed :
                                              player.Speed;
    Vector3 dir = transform.forward * moveValue.y +
                  transform.right * moveValue.x;
```

```
    rb.AddForce(dir.normalized * speed, ForceMode.Force);
}
```

The force applied to the Rigidbody "ForceMode.Force" is continuous and uses the character mass, this gives the feeling of some characters to be heavier than others by just changing the mass value from the Rigidbody of the character prefab. At the same time, it is combined with the Physics Materials that will be explained later on, to apply friction to the movement avoiding some slippery feeling.

5.3.3. Slide

Sliding is a mechanic that makes it more difficult for the player to be hit by projectiles and abilities, due to the velocity increase, the body's change of animation, and scale.

This approach increases the maximum linear velocity of the player's Rigidbody during this movement, this value is set in each characters' data parameters, the linear damping of the body is lowered to make a faster increase of the velocity and it also reduces the scale and centers to the correct position the player's collider to simulate correctly the collision with other bodies. Then, applies a force that boost the velocity in the direction that the player was applying movement before triggering the mechanic.

```
C/C++
void Slide(bool startSlide)
{
    if (startSlide && !isSliding &&
        isGrounded && rb.linearVelocity.magnitude > 1f)
    {
        isSliding = true;
        slideTimer = slideDuration;

        capsuleCollider.height = characterHeight / 2f;
        capsuleCollider.center = new Vector3(0,
                                 capsuleCollider.height / 2f, 0);

        rb.maxLinearVelocity = player.MaxSlidingSpeed;
        rb.linearDamping = slideDrag;
```

```
        Vector3 slideDirection = new Vector3(rb.linearVelocity.x, 0,
                                 rb.linearVelocity.z).normalized;
        rb.AddForce(slideDirection * slideForce * rb.mass,
                    ForceMode.Impulse);
    }
    else if (!startSlide && isSliding)
    {
        StopSliding();
    }
}

void StopSliding()
{
    isSliding = false;
    capsuleCollider.height = characterHeight;
    capsuleCollider.center = new Vector3(0, characterHeight / 2f, 0);

    rb.maxLinearVelocity = player.MaxSpeed;
    rb.linearDamping = dragCoefficient;
}
```

Finally, we need to reset the player's collider, velocity and drag coefficients back to default values.

5.3.4. Jump

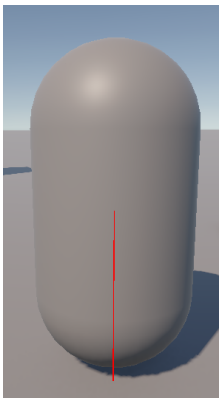
For the jump, first the player casts a ray to the ground to check if it is on the floor. Once the jumping action starts it resets the momentum in Y axis velocity (set the value to 0).

An instant Force is applied in the desired direction setting Force.Impulse, the direction is introduced as a parameter due to the method that executes the wall-run logic, to also use the "Jump()" function.

Finally, a cooldown is applied and when finished it invokes the "ResetJump()" function to avoid jumping exploits that may modify players' maximum velocity such as Bunny Hop.

```
C/C++
void Jump(bool isJumping, Vector3 jumpDir, float forceAdd = 0f)
{
    if (jumpAction.IsPressed() && readyToJump)
    {
        readyToJump = false;
        //Reset Y Velocity
        rb.linearVelocity = new Vector3(rb.linearVelocity.x, 0,
                                        rb.linearVelocity.z);

        rb.AddForce((transform.up + jumpDir.normalized) *
                    (jumpForce + forceAdd),
                     ForceMode.Impulse);

        //Delay for jump to reset
        Invoke(nameof(ResetJump), jumpCooldown);
    }
}
void ResetJump()
{
    readyToJump = true;
}
```

Figure 5.3.4.1: Raycast Ground Checker



(Own Source)

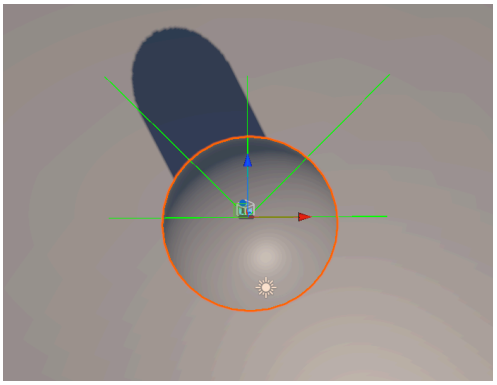
5.3.5. Wall-run

The player is allowed to run through walls for a little period of time. The technique used stores the ray directions that are casted to detect the walls into an array of Vector3.

At first, directions followed the forward and sideways directions, but not backwards, casting five rays to detect the wall the player wants to be attached to. The five rays form an arch to detect just the walls that the players are looking at, but after some testing during the development we were having troubles with this approach.

The rays pointing forward and forward-sideways were causing issues with the wall detection and making the "WallRun" method to apply the logic when it was not wanted, sometimes leading to bugs like sticking the player to the wall and not be able to move the whole game.

Figure 5.3.5.1: Raycast Wall-Run Checker First Approach



(Own Source)

In the end, it only throws two rays, one to the right and another to the left. Checks if the collision is with an object containing the desired LayerMask and wich side is the wall. Then, it applies a force to the direction of where is the wall to make the player stick to it. Next, another force is applied in forward direction and upwards to make the intended behaviour and also avoid the player to fall quickly.

Finally, it checks if the jump input has been pressed to activate the jump function following the normal direction of the wall, and sets the "Dutch" value of the Cinemachine camera that tilts it in Z-axis direction depending on which side is the wall.

The "CheckWalls" method is explained later on in the Utilities section.

```
CSharp
isWallRunning = hyperStrikeUtils.CheckWalls(transform, ref wallHit,
                                        ref refCameraTilt, wallMask);
void WallRun(bool isJumping)
```

```
{
    if (!isGrounded && isWallRunning)
    {
        cameraTilt.Value = refCameraTilt;

        // Stick to wall
        rb.AddForce(-wallHit.normal * stickWallForce, ForceMode.Force);

        // Reduce gravity to stay more time in the wall but not infinite
        rb.AddForce((transform.forward * player.Character.wallRunSpeed)
                    + (transform.up * stickWallForce), ForceMode.Force);

        Jump(isJumping, wallHit.normal, 5f);
    }
    else
    {
        cameraTilt.Value = CameraTilt.NONE;
    }
}
```

5.3.6. Friction and Bounciness

For the movement mechanics execution, frictional forces for the player movement when it is on the floor, the walls, or in the air are handled. Also some bouncing forces that some objects will use when colliding with others.

The first approach attempt for the frictional forces was to apply a drag force[9] to the player's rigidbody for each surface detected by the rays casted and the checkers. This force should be in the opposite direction to the player's velocity.

The second technique consisted of to just modify, depending also on the surface, the linear damping[10] variable from the rigidbody properties. The problem with this approach lies on the acceleration of the body, it is needed to set properly the properties for each medium or surface and the speed of the player, it can create the feel where it takes too

[9] Drag Force: Resistance force exerted by a fluid on a body that moves through it.
[10] Linear Damping: *"The reduction in energy and amplitude of oscillations due to resistive forces on the oscillating system"*

long to reach the maximum speed and when trying to stop the player's movement cause a slippery feeling.

The two first approaches are a little time consuming to find the best values for each character and surface that will be in the game while modifying part of the code.

```C/C++
// Approach 1: Drag Force
Rigidbody rb;
float dragCoefficient;
void HandleDrag()
{
    if (isGrounded)
        dragCoefficient = 0.5f;
    else if (isWallRunning)
        dragCoefficient = 0.25f;
    else
        dragCoefficient = 0.0f;

    Vector3 dragForce = -rb.linearVelocity.normalized *
                        rb.linearVelocity.magnitude *
                        dragCoefficient;

    rb.AddForce(dragForce);
}

// Approach 2: Linear Damping
float groundDrag = 0.25f;
float wallDrag = 0.15f;
void HandleDrag()
{
    if (isGrounded)
        rb.linearDamping = groundDrag;
    else if (isWallRunning)
       rb.linearDamping = wallDrag;
    else
        rb.linearDamping = 0;
```

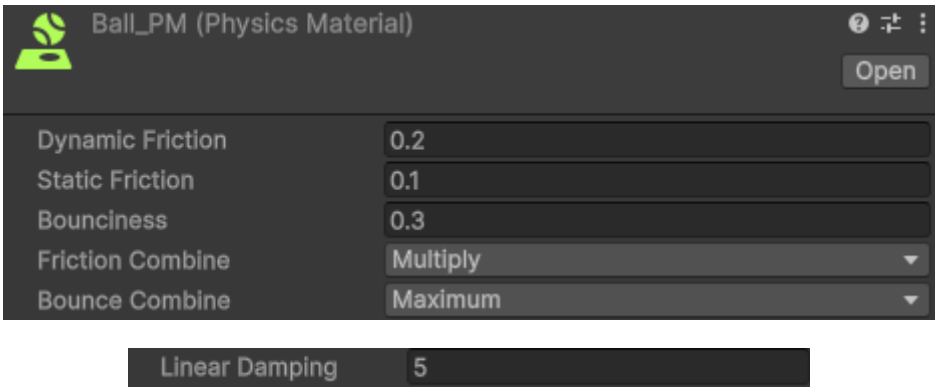
```
}
```

Due to the need of making different surfaces handling for not only the friction but also for bouncing with the pitch scenarios, and Unity's code compilation lasts a lot of time with the minor change in the code, it will be used what is called in Unity, Physics Materials together with the linear damping value set to the Rigidbody in the inspector.

It is easier to implement friction and bounciness in surfaces, and allows a better physics simulation through the network. The server is the one that calculates the collision and then sends the position changes to the players affected instead of each client calculating their own simulation, which would be more prone to network and physics issues.

Each Rigidbody will calculate the friction with the surface within the parameters set in the Physics Material from the inspector.

Figure 5.3.6.1: Ball Physic Material Properties



(Own Source)

Dynamic Friction is the friction encountered by a body while moving across the surface, Static Friction when it starts the movement of the body. Friction and Bounce Combine, tells Unity how it should combine the values of both Colliders in a collision pair to calculate the total between them.

In the Player Controller a small method is implemented before applying any movement, to handle grounded and mid air movement, by setting the linear damping to 5 if its grounded and 0 if falling. It is also applied a downward force when the player is not grounded to avoid a floating sensation.

```CSharp
private void SetFriction()
{
    bool isFalling = rb.linearVelocity.y < 50f;
    if (!isGrounded)
    {
        rb.linearDamping = 0f;
        if (!isWallRunning)
        {
            if (isFalling)
                rb.AddForce(Physics.gravity * 10f,
                            ForceMode.Acceleration);
            else if (wasJumpPressed)
                rb.AddForce(Physics.gravity * 4f,
                            ForceMode.Acceleration);
        }
    }
    else rb.linearDamping = dragCoefficient;
}
```

5.4. Ability System

Each character has three abilities, the main system follows a Component-based design, the abilities are built by attaching components to the GameObject. Those components are Ability types scripts that each one contains a set of variables and the core logic of the ability type. Then the three abilities are referenced to the *PlayerAbilityController.cs* script that manages the ability casting logic.

The principal class is the Ability abstract class that is derived from Network Behaviour. It works as base class for each ability type, contains the base variables and virtual functions that all the derived ability types will need in order to work properly and avoid modifying a lot of code in this project, making it an easy to implement and modular system.

This approach allows abilities to contain charges, reloading them in a short period of time they are not spent at all.

CSharp

```
public abstract class Ability : NetworkBehaviour
{
    [Header("Basic Info")]
    public string abilityName;
    public Sprite icon;
    public float chargeReloadTime;
    public float fullCooldown;
    [SerializeField] protected float castTime;
    public float maxCastTime;
    public byte maxCharges;

    public bool isReloading;
    public bool requiresTarget;
    public bool isOnCooldown;

    [Header("Networking")]
    public NetworkVariable<byte> currentCharges =
                                        new NetworkVariable<byte>(1);
    public NetworkVariable<float> currentCooldownTime =
                                        new NetworkVariable<float>(0f);
    public NetworkVariable<float> currentReloadTime =
                                        new NetworkVariable<float>(0f);

    // Reference to the player who owns this ability instance
    protected Player owner;

    // Called when ability is assigned to a player
    public virtual void Initialize(PlayerAbilityController player)
    {
        owner = player;
        // Initialize Variables
    }

    // Server-side execution
    public virtual void ServerCast(ulong clientId, Vector3 initPos,
                                   Vector3 dir)
```

```
    {
        // Charges
        currentCharges.Value--;
        if (currentCharges.Value < 1)
        {
            isOnCooldown = true;
        }
        else if (!isOnCooldown && currentCharges.Value < maxCharges)
        {
            isReloading = true;
        }
    }
    // Visual/audio effects that run on all clients
    public virtual void PlayEffects(Vector3 position) { }
}
```

Different ability class types are created depending on the intended behaviour, such as movement, projectile, area and targeted, even a combination of these previous types to execute multiple logic for a single ability.

The abilities are attached to the player's GameObject and referenced in the Player Ability Controller, when it is spawned, initializes each ability.

Then the Player Controller component manages the input, checking if any of the abilities is being casted and calls the "CastAbility" method which takes the index of the selected ability and the "OwnerClientId". The "CastAbility" method checks if the ability is set properly, is on cooldown or reloading charges, and then if all is correctly set calls the "ServerCast" method from the ability.

CSharp
```
public void CastAbility(int abilityIndex, ulong clientId)
{
    if (abilityIndex < 0 || abilityIndex >= abilityInstances.Count)
        return;
```

```
        var ability = abilityInstances[abilityIndex];

        if (ability == null) return;
        if (ability.isOnCooldown || ability.currentCharges <= 0) return;

        ability.ServerCast(clientId, castTransform.position,
                           castTransform.forward);

        // Start cooldown/reload timer
        if (ability.isOnCooldown)
        {
            StopCoroutine(StartReloading(abilityIndex));
            StartCoroutine(StartCooldown(abilityIndex));
        }
        else
        {
            StartCoroutine(StartReloading(abilityIndex));
        }
    }
```

5.4.1. Movement Ability

The Movement Ability is a type of active skill that modifies the player's position by dashing a certain distance in a specified direction.

Before the ability is executed, the system calculates the final destination that the player will have once the dash is performed in order to prevent undesired behaviour such as clipping through geometry, or exiting the arena boundaries by modifying the final position to one inside the arena boundaries.

This validation is done by the “CalculateDashEndPosition” method, which performs a forward raycast from the player's current position in the dash direction. If the ray intersects with a collider within the specified dashDistance, the dash endpoint is clamped just before the collision to avoid overlap. If it does not detect a collision, checks whether the end position would place the player outside the boundary of the

arena. If so, the direction is reversed and calculates a fallback position to make the player remain inside.

```CSharp
private Vector3 CalculateDashEndPosition(Vector3 startPos, Vector3 direction, float distance)
{
    RaycastHit hit;
        if (Physics.Raycast(startPos, direction, out hit, distance, collisionLayers))
    {
        return startPos + direction * (hit.distance - 0.5f);
    }
    Vector3 endPos = startPos + direction * distance;

    if (Physics.OverlapSphere(endPos, 0.5f,
       LayerMask.GetMask("Boundary")).Length > 0 || endPos.y < 0)
    {
        Vector3 returnPos = (startPos - endPos).normalized;
        endPos = returnPos;
    }
    return endPos;
}
```

Once the final destination is determined, the ability accesses the owner player reference to start a coroutine to handle the movement logic. During the execution, it retrieves the owner's Rigidbody to disable the gravity and set the velocity to zero to avoid undesired behaviour.

It supports instant transition or interpolation between the player's starting position and the final destination, by easing out[11] the end part of the dash producing an acceleration-deceleration feel.

[11] Ease out: timing function where an animation starts quickly and then slows down as it approaches its end point.

At the end, the linear velocity that the player had before dashing is partially restored, it is scaled down to 50% of its initial value, to give a satisfying dashing feel while the player regain the control.

5.4.2. Projectile Ability

The Projectile Ability type is one of the most simple ability types in the project. It is responsible for spawning one or more projectile objects that travel in the direction the player is facing at the moment of casting.

When the ability is activated, it spawns projectiles server-side using an object pooling system, explained deeply forward in the Net Object Pooling section, to minimize the performance impact of instantiating new objects during runtime. The server is the responsible for the projectile's logic and physics simulation, once spawned and initialized, the projectile objects are synchronized with all clients.

```CSharp
public class ProjectileAbility : Ability
{
    [Header("Projectiles Settings")]
    public GameObject projectilePrefab;
    public uint projectilesQuantity = 1;

    public override void ServerCast(ulong clientId, Vector3 initPos,
                                    Vector3 dir)
    {
        base.ServerCast(clientId, initPos, dir);

        owner.StartCoroutine(EnableProjectiles());

        PlayEffects(initPos);
    }

    private IEnumerator EnableProjectiles()
    {
        for (uint i = 0; i < projectilesQuantity; i++)
        {
            var projectileNO =
```

```
            NetworkObjectPool.Singleton.GetNetworkObject(projectilePrefab,
                                                    position, rotation);

                if (projectileNO != null)
                {
                    if (!projectileNO.IsSpawned)
                    {
                        projectileNO.Spawn();
                    }

                    var projectile =
                          projectileNO.GetComponent<Projectile>();

                    projectile.projectilePrefabUsed = projectilePrefab;
                    projectile.Activate(position + direction, rotation,
                                        owner.OwnerClientId, owner);
                }
                yield return new WaitForSeconds(maxCastTime);
            }
        }
    }
```

5.4.3. Area Ability

The ability casts the effect logic within a defined area for the duration of the cast time set. It can apply different effects inside the radius, such as pushing away other objects with a force or applying damage, heal or protection to players, checking also if they are from the player owner's team or adversaries to apply the desired effect. The logic includes checks to determine which players are affected, the caster (self), allies, or enemies.

A sphere collider is used to detect the objects that are inside of the area radius using "Physics.OverlapSphere". In order to check if a player left the area of effect, it stores the players that were inside of it in the previous frame and compares it with the current frame's list. If a player left the area, their protection effect, if it was active, is revoked.

In some cases, such as in Craswall's "Mobile Shield" ability, a visual area indicator can be instantiated and also optionally parented to the player that casts the ability. The prefab used provides feedback and also sets the radius of the affected area.

One of Speed abilities, "Ground Quake", is a type of an Area ability that inherits from the Movement Ability class, allowing it to execute and contain the same values and logic as a movement ability while also apply area effects. It first casts the player's movement from the inherited class logic, and then when it is finished it casts the desired area effect. It allows the player to execute with a single ability cast an hybrid behaviour using both ability types.

```CSharp
public class AreaAbility : Ability
{
    // Affected objects checker variables
    // Values variables
    // Prefab Area

    Collider[] colliders;
    private HashSet<Player> previouslyAffectedPlayers =
                                                new HashSet<Player>();

    void ApplyAreaEffect()
    {
        // Instantiate Area prefab if needed and parent to player

        colliders =
             Physics.OverlapSphere(owner.transform.position, radius);

        HashSet<Player> currentlyAffectedPlayers =
                                                new HashSet<Player>();

        foreach (Collider collider in colliders)
        {
            if (collider.TryGetComponent<Player>(out Player player) &&
                owner.TryGetComponent<Player>(out Player ownerPlayer))
            {
```

```
            if (!selfAffect &&
                player.OwnerClientId == OwnerClientId) return;
            if (!affectAllies &&
                player.Team.Value == owner.Team.Value) return;
            if (!affectEnemies &&
                player.Team.Value != owner.Team.Value) return;

            if (useDamage)
            {
                player.ApplyEffect(EffectType.DAMAGE, damage);
                currentlyAffectedPlayers.Add(player);
            }
            // Apply Heal
            // Apply Protection
        }

        Rigidbody rb = collider.GetComponent<Rigidbody>();

        // Apply forces

        // Revoke protection for players who left the area

        previouslyAffectedPlayers = currentlyAffectedPlayers;
    }
}

private IEnumerator AreaCast()
{
    castTime = 0;
    while (castTime < maxCastTime)
    {
        castTime += Time.deltaTime;
        ApplyAreaEffect();
        yield return null;
    }
```

```
        if (areaObject != null)
        {
            areaObject.GetComponent<NetworkObject>().Despawn();
            useAreaPrefab = true;
        }
        foreach (Collider collider in colliders)
        {
            if (useProtection)
            {
               Player p = collider.GetComponent<Player>();
               if (p) p.ApplyEffect(EffectType.UNPROTECT);
            }
        }
    }
}
```

5.4.4. Targeted Ability

The Targeted Ability locks the casting player's camera to the closest player within a specified range for a period of time.

When the ability is cast, it executes a scan within an spherical area around the caster, defined by a range. The scan filters all the colliders inside the area using a layer mask and checks, such as if the detected player is ally or enemy. Then, for each player found, it calculates the distance to the caster, and the closest one is selected as the target.

CSharp
```
private Transform FindClosestTarget()
{
    Transform closestTarget = null;

    Collider[] colliders =
             Physics.OverlapSphere(owner.transform.position, lockRange,
                                   targetLayers);
    float closestDistance = Mathf.Infinity;
```

```
    foreach (Collider target in colliders)
    {
        Player player = target.GetComponent<Player>();

        if (player == null ||
            player.OwnerClientId == owner.OwnerClientId) continue;

        if (!targetOwnerTeam && player.Team.Value == owner.Team.Value)
            continue;

        float distance = Vector3.Distance(owner.transform.position,
                                          target.transform.position);
        if (distance < closestDistance)
        {
            closestDistance = distance;
            closestTarget = target.transform;
        }
    }
    return closestTarget;
}
```

Once the target is selected, the server sends a Remote Procedure Call (RPC)[12] method to the client who casted the ability using a "NetworkObjectReference" to ensure the correct target is tracked. When the client receives the call, the ability sets the Cinemachine camera's "TrackinTarget" to the obtained target's Transform, causing tha camera to automatically follow the target player during the ability's cast time.

After the cast time finishes, the server sends another RPC method to reset the camera, allowing the caster to control again the camera by setting the "TrackinTarget" to null.

```CSharp
private IEnumerator LockCameraToTarget()
{
    castTime = 0;
```

[12] Remote Procedure Call (RPC): Function call that executes other processes on remote systems.

```
        currentTarget = FindClosestTarget();
        if (fixCameraToTarget && currentTarget != null)
        {
            var targetNetObj = currentTarget.GetComponent<NetworkObject>();
            if (targetNetObj != null)
            {
                NetworkObjectReference targetRef =
                                new NetworkObjectReference(targetNetObj);
                LookAtTargetClientRPC(targetRef, new ClientRpcParams
                {
                    Send = new ClientRpcSendParams
                    {
                       TargetClientIds =
                                   new List<ulong> { owner.OwnerClientId }
                    }
                });
            }
        }

        while (castTime < maxCastTime)
        {
            castTime++;
            yield return new WaitForSeconds(1f);
        }

        ResetLookAtTargetClientRPC(new ClientRpcParams
        {
            Send = new ClientRpcSendParams
            {
               TargetClientIds = new List<ulong> { owner.OwnerClientId }
            }
        });
    }
```

In the *PlayerController.cs* was required a function that handles the camera and player rotations when the camera has a target set. If the camera does not have a target it still executes the "RotatePlayerWithCamera".

```CSharp
void FollowTarget(Vector2 lookValue)
{
    Vector3 targetDirection =
         (cinemachineCamera.Target.TrackingTarget.position -
          transform.position).normalized;

    float mouseXx = lookValue.x * Time.fixedDeltaTime;
    float mouseYy = lookValue.y * Time.fixedDeltaTime;

    xRotation -= mouseYy;
    xRotation = Mathf.Clamp(xRotation, -10f, 10f);
    yRotation += mouseXx;
    yRotation = Mathf.Clamp(xRotation, -10f, 10f);

    cameraWeaponTransform.localRotation =
                                Quaternion.Euler(xRotation, 0, 0);
    mainCameraTransform.localRotation =
                                Quaternion.Euler(xRotation, 0, 0);

    Quaternion targetRotation = Quaternion.LookRotation
             (new Vector3(targetDirection.x, 0, targetDirection.z));
    rb.rotation = Quaternion.Slerp(rb.rotation, targetRotation,
                                 Time.fixedDeltaTime * 100f);
}
```

5.4.5. Melee and Shoot Attacks

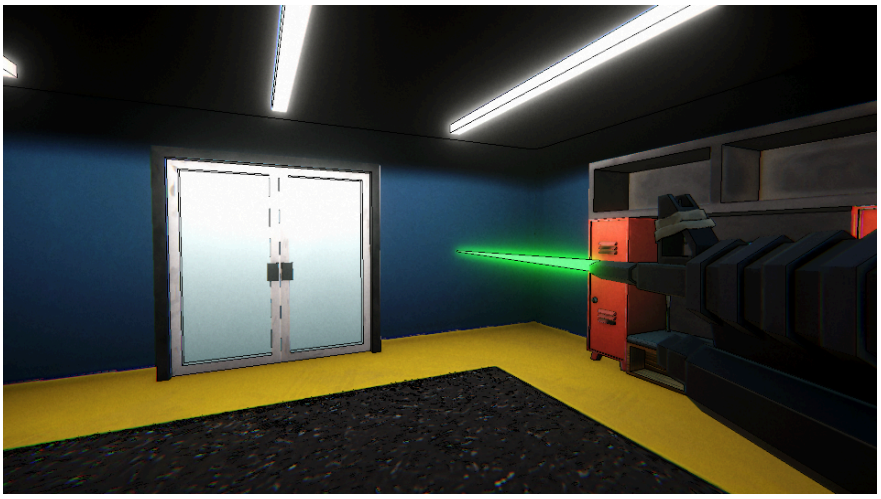
Even though the melee and shoot attacks are not abilities at all, they work similarly. When the input callback is invoked client-side, it sends an RPC method to execute the logic server-side.

The melee attack throws a Raycast in the forward direction that the player camera is looking at, checking if it collides with an object that contains a Rigidbody to apply an impulse force to it, and then if it is a player, apply damage.

```CSharp
void MeleeAttackServerRPC()
{
    if (meleeReady)
    {
        meleeReady = false;

        if (Physics.Raycast(cameraWeaponTransform.position,
            cameraWeaponTransform.forward,
            out RaycastHit hit, player.MeleeOffset))
        {
            Rigidbody rb = hit.rigidbody;

            if (rb != null)
            {
                rb.AddForce(transform.forward * player.MeleeForce,
              ForceMode.Impulse);
            }

            if (hit.transform.TryGetComponent<Player>(out Player p))
            {
                p.ApplyEffect(EffectType.DAMAGE,
                              player.MeleeDamage.Value);
            }
        }
        Invoke(nameof(ResetMeleeAttack), 1f);
    }
}

void ResetMeleeAttack()
{
    meleeReady = true;
}
```

In the case of the shoot attack it works almost equally, but in this case instead of a Raycast it throws a projectile. Depending on the character there are different projectiles, Speed shoots baseball balls but Craswall and Nanoflow work also with Raycast showing a visible line.

Figure 5.4.5.1: Shoot Attack Raycast Projectile



(Own Source)

5.5. Networking

Netcode for GameObjects includes a feature called "Multiplayer Play Mode" that allows to test multiplayer functionalities during the development process by simulating up to three virtual players together with a real player in the same device, eliminating the need to build and run the game and without leaving the Unity Editor. During the network development features of this project, this tool is essential for testing reducing iteration time and Multiplay's cloud service resources.

Unity's Multiplay comes with a free trial period of up to $800 of credit that lasts up to 6 months. The free trial credit will not be totally spent during the development process, due to the limited development time and the low use of multiplayer resources the game spends, just host and play a single match at a time. However, during the development process, it is essential to optimize those resources for a better performance, reliability, and especially to avoid high impact consumptions of the multiplayer fee.

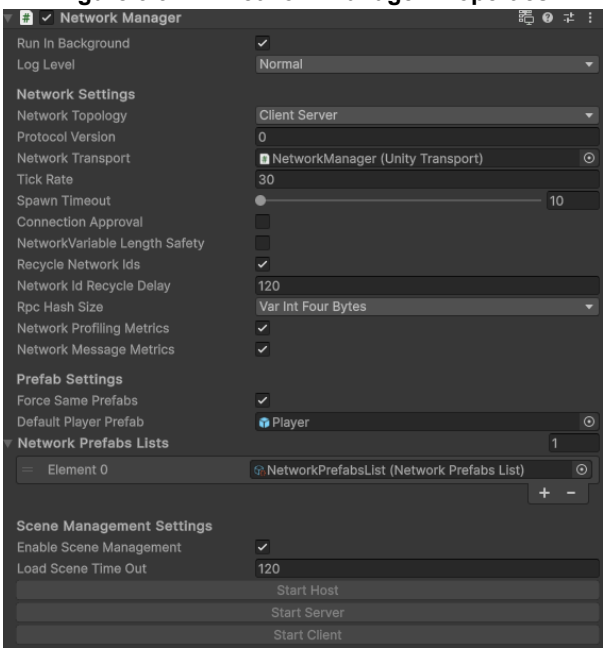
5.5.1. Set Up

To set up the networking environment, it was necessary to add a "Network Manager" GameObject that contains two component scripts: *NetworkManager.cs* and

UnityTransport.cs. The *NetworkManager.cs* script converts the GameObject into a non-destructible object when loading scenes to be persistent the whole game flow, it is placed in the MainMenu scene, where clients and server will need it to initialize the network connections. From that point onward, it will manage all the networking operations through all the scenes.

The *NetworkManager.cs* provides a range of configuration options that are essential for establishing the multiplayer system. One important setting, is the "Network Topology", which must be set to "Client-Server". Allow to implement and work further with a Server Authoritative model.

Figure 5.5.1.1: Network Manager Properties



(Own Source)

Ensuring a Server Authoritative model is essential to prevent potential cheating and synchronization issues between the clients. Cheating would be easier by allowing clients to fully control their movement, instead of the server simulating it and having the control of all clients movement. This could lead to uncontrolled changes in the position of not just the client's characters, but also in the abilities and attacks creating unbalances in the game.

The Network tick rate[13] must fit with the server's physics timestep. Achieve correct behaviour through each client and send the correct quantity of data packets is essential

[13] Tick rate: frequency that the server runs user code updating the state and sends out data.

to prevent affecting the smoothness and consistency of the gameplay. The following table shows how different tick rate and fixed timestep combinations affect performance and reliability.

Table 5.5.1.1: Tick rate & Physics timestep relation

Setting	Result
Physics: 120 Hz Network: 30 Hz	Smooth simulation, but only every fourth physics update is sent. Can feel laggy or jittery in a fast-paced game.
Physics: 60 Hz Network: 60 Hz	Perfectly aligned, but slightly less responsive simulation compared to 120 Hz physics. Acceptable for many games.
Physics: 120 Hz Network: 60 Hz	Smooth simulation, and network sends updates every second physics frame. Good balance for responsiveness.
Physics: 120 Hz Network: 120 Hz	Best possible fidelity and responsiveness, but high performance cost. Only recommended for high-end, low-latency games.

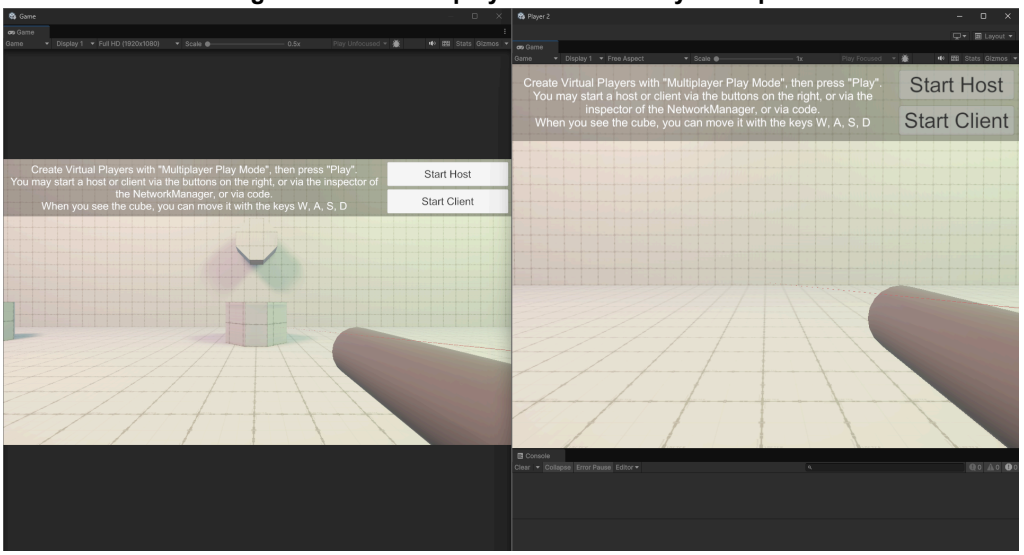
Given the fixed timestep set at the beginning of the project (120 Hz) for a correct simulation, is evident that the higher tick rate the netcode has, the smoother the clients responsiveness and physics simulation is, particularly for fast-paced mechanics, such as hit detection or rapid movement. For this reason, a minimum tick rate of 60 Hz is recommended in this game to reduce visible lag and jitter.

Unity Netcode allows assigning a “Multiplayer Role” to GameObjects, depending on the roles selected (client, server or both), changes the rendering and the execution of the components behaviour. It can also be set for each attached component individually. Some scripts parts can be executed on the server and others on the client-side, this is managed through application of a boolean check by adding “IsServer” or “IsClient” or directly from the Inspector, if the script is just needed from one side.

This way, physics behaviour controlled by a script can be set just for the server to execute it, while for example the activation of particle systems or animations will be executed on both sides or just for the clients, making the server authoritative.

As a clear example, in the following image the bouncer's GameObjects from the Pinball Blockout scene are set to clients only, these objects are only instanced client-side, and are automatically destroyed server-side. This is a useful tool that can improve performance on both sides preventing the execution of unnecessary scripts or rendering if not needed. This feature will play an important role throughout the development process.

Figure 5.5.1.2: Multiplayer Role Visibility Example



(Own Source)

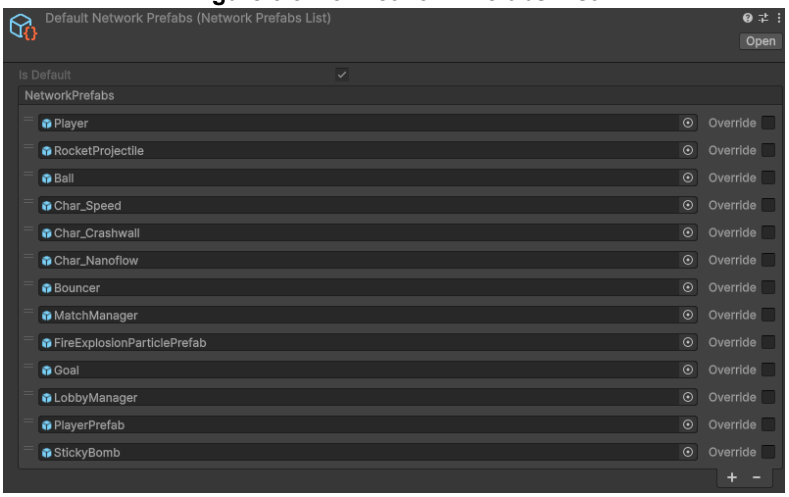
Each GameObject containing a NetworkBehaviour script component, must have at least the NetworkObject component attached to it to work correctly with Netcode for GameObjects.

For the correct implementation of the network behaviour it is essential to define what data and functionality need to be synchronized. There are variables and functions that need to be shared between clients, sent from client to server, or from the server to other clients.

As mentioned in previous sections, the project follows a Server Authoritative model. It does not only means that it simulates the Physics, but also that manages the variables and behaviour that we want to synchronize through all the clients at the same time, a Server Authoritative model makes the server to be the only one modifying those variables while the clients can only read them.

All the prefabs created that contains the NetworkObject component, need to be added in the Network Prefabs List, a default list is created when compiling the scene for the first time.

Figure 5.5.1.3: Network Prefabs List



(Own Source)

5.5.2. Connection

The connection data must be set properly in order to establish the connection between the server and the clients. It can be done by accessing the UntiyTransport component attached previously to the NetworkManager GameObject, calling the function "SetConnectionData" and in the inspector set "Allow Remote Connections?" to true.

On the server's side, the IP must be always "0.0.0.0" to accept connections from any IP address. The port can be any one unused but in Unity's Multiplay Hosting service, the first server established usually has the port 9000. The serverl listen address must be set to "0.0.0.0" again, allows the server to accept connections regardless of which network adapter the traffic arrives on.

On the client's side, the IP address is the one Multiplay provides when allocating the server in the cloud. During the development process, the game needs to be tested locally before uploading it to the hosting service, in this case the IP address can be set to "127.0.0.1" or the computer's IP address where the server is being hosted, for example "192.168.1.20", used when playing in LAN (Local Area Network).

```
CSharp
SetConnectionData(ipAddress, port, serverListenAddress);
SetConnectionData(ipAddress, port);
```

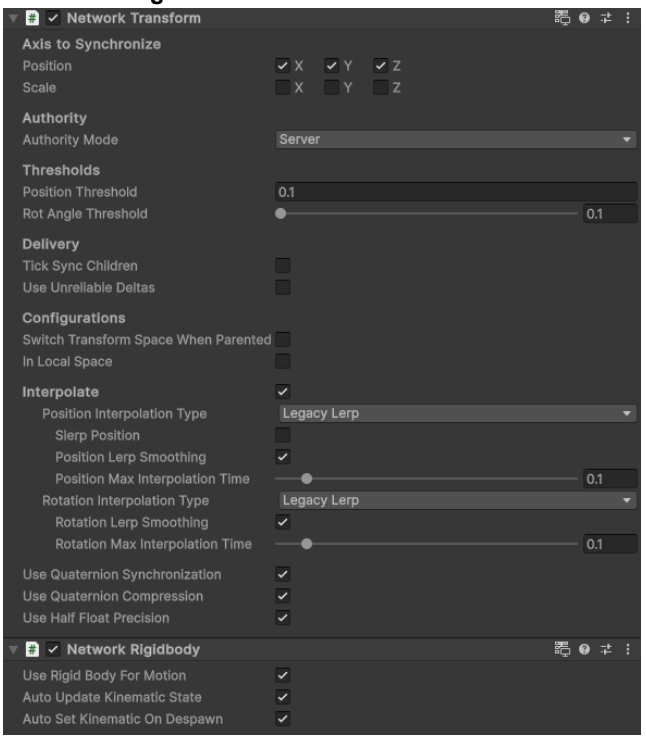
5.5.3. Net Player

The NetworkTransform allows to synchronize the necessary transformations while at the same time, set the authority model about who has the priority to modify the GameObject Transform. Following the Server Authoritative model, it must be set to Server.

If an object uses the Rigidbody component, the NetworkRigidbody component must be attached to the GameObject and set "Use Rigid Body For Motion" to true for a correct physics simulation, and to avoid discordances with the Transform.

The NetworkTransform component implements an interpolation feature which takes the previous and current values and does a transition in a period of time. Depending on the value set in the "Max Interpolation Time" setting, the lower it is the greater the precision of the movement but with a cost, it is less smooth and increases stutter due to jitter, latency or a high threshold value.

Figure 5.5.3.1: Network Transform



(Own Source)

Continuous movement throughout the game requires frequent synchronization of position and rotation data. To ensure the transformations to be smooth and reliable, especially to avoid visual artifacts such as clipping or stuttering on the client side, this data must be transmitted with a sufficiently precise threshold.

During the project development, first was implemented some movement mechanics, so it was necessary to modify the *PlayerController.cs* script explained previously in the MVC and Movement Mechanics sections. For a correct server authoritative implementation, the server needs to execute the whole player behaviour related to Physics or movement, even though is the client who presses the inputs.

The clients send the input data to the server by calling an RPC method, this sends a callback to the server which will be the only one to execute the logic inside that method together with the data that the client sent as function parameters.

The first approach was to use a single RPC method for each movement mechanic and ability implemented. After some testing using the profiling tools that Unity provides, I realized that each client was sending too many RPC calls and bytes of information that were not necessary or reliable at all, and needed to be optimized.

5.5.3.1. Player Optimization

Bearing in mind the tick rate set at the beginning of the project, it sends the RPC calls 60 times per second all the time, it was not checking if it truly needed to send the call to the function, causing a huge amount of bytes sent to the server by each client, and data loss. So, an optimization was required for a correct network performance.

The best technique is to send the major quantity of data in a single packet or RPC. Changing the multiple RPC calls into a single one containing almost all the necessary input parameters, highly reduced the bandwidth overhead and the bytes sent to the server, increasing the performance of the network. It also fixed some of the latency related issues that provoked the server to execute in incorrect order some methods due to delayed RPC calls. Before sending the RPC method is highly recommended to check if the player can move or attack to avoid sending RPC calls, it also reduces the bandwidth overhead.

For the single method to be sent properly to the server we must set above it, [ServerRPC], then in the method name add at the end “MethodName*ServerRPC*(...)”.

The "SendInputServerRPC" function sends the data that would be modified continuously in this case.

```CSharp
[ServerRpc]
void SendInputServerRPC(InputData input)
{
    isGrounded = hyperStrikeUtils.CheckGrounded(transform,
                                                characterHeight);
    isWallRunning = hyperStrikeUtils.CheckWalls(transform,
                                ref wallHit, ref refCameraTilt);
    if (input.look != Vector2.zero)
        RotatePlayerWithCamera(input.look);

    if (input.move != Vector2.zero)
    {
        WalkAndRun(input.move, input.sprint);

        WallRun(input.jump);
    }
    else
    {
        animator?.Animator.ResetTrigger(WalkingHash);
    }

    if (input.slide != wasSliding)
    {
        Slide(input.slide);
        wasSliding = input.slide;
    }

    if (input.jump && isGrounded && !isWallRunning)
    {
        Jump(input.jump, Vector3.zero);
    }
    animator?.Animator.SetFloat(VelocityHash,
                                rb.linearVelocity.magnitude);
```

```
}
```

In this case Input Data is a custom struct created to store the input data into different variable types, for this to be sent through an RPC method without it giving us an error must be Network Serializable. A proper way to create a Network Serializable struct is the following:

```CSharp
[Serializable]
public struct InputData : INetworkSerializable
{
    public Vector2 move;
    public bool moveInProgress;
    public Vector2 look;
    public bool sprint;
    public bool jump;
    public bool slide;

    public void NetworkSerialize<T>(BufferSerializer<T> serializer)
    where T : IReaderWriter
    {
        serializer.SerializeValue(ref move);
        serializer.SerializeValue(ref moveInProgress);
        serializer.SerializeValue(ref look);
        serializer.SerializeValue(ref sprint);
        serializer.SerializeValue(ref jump);
        serializer.SerializeValue(ref slide);
    }
}
```

For example for the attacks and abilities, or the emotes, we can send them separately, calling an RPC method for each one because they will not be triggered each frame or fixed time step, only when needed, and depending on the situation checking if the player can move or interact at that moment to avoid send an unnecessary call.

```CSharp
private void InitInputs()
{
    input.Player.MeleeAttack.started += ctx => MeleeAttackServer();
    input.Player.Attack.started += ctx => ShootServer();
    input.Player.Ability1.started += ctx => ActivateAbility1Server();
    input.Player.Ability2.started += ctx => ActivateAbility2Server();
    input.Player.Ultimate.started += ctx => ActivateUltimateServer();
    input.Player.Emote1.started += ctx => Emote1ServerRPC();
    input.Player.Emote2.started += ctx => Emote2ServerRPC();
}
```

Even though at this point the network performance and the quantity of data sent were highly improved, there still was some notorious latency since the player pressed the input button until it received the server updated information and applied the movements.

The “SendInputServerRPC” was sent in the “FixedUpdate” method, if the Physics time step is at 120Hz and the tick rate set to 60Hz, two input RPC calls were sent batched per each network tick. This caused bandwidth overhead, information overload, and unreliable data that produced latency client-side.

“NetworkManager.NetworkTickSystem.Tick” is an Action provided from Unity Network Manager class that is called once per network tick, aligned with Netcode’s internal loop, then all the physics that are modified by the RPC call are simulated correctly server-side reducing latency.

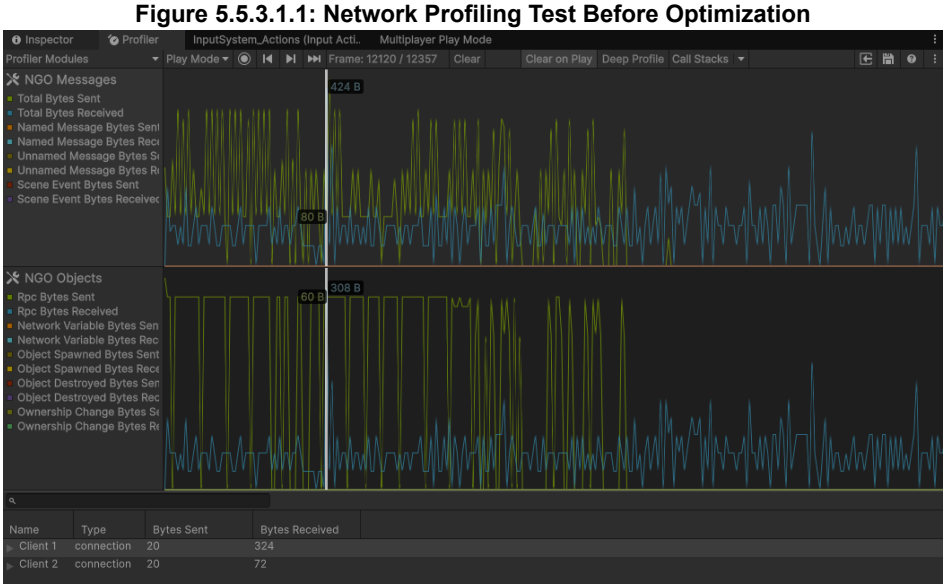
```CSharp
public override void OnNetworkSpawn()
{
    if (IsClient && IsOwner)
    {
        NetworkManager.NetworkTickSystem.Tick += OnNetworkTick;
    }
}
private void OnNetworkTick()
{
```

```
    if (!IsClient || !IsOwner) return;
    InputData data = new InputData
    {
        move = input.Player.Move.ReadValue<Vector2>(),
        moveInProgress = input.Player.Move.IsInProgress(),
        look = new Vector2(input.Player.Look.ReadValue<Vector2>().x,
        invertY != 0 ? input.Player.Look.ReadValue<Vector2>().y :
        -input.Player.Look.ReadValue<Vector2>().y),
        sprint = input.Player.Sprint.IsPressed(),
        jump = input.Player.Jump.IsPressed(),
        slide = input.Player.Slide.IsPressed(),
    };
    SendInputServerRPC(data);
}
```

In the following images, there is a clear difference between them. The first one is before optimizing the transform and RPC calls. The data is inconsistent, unreliable, and with unnecessary high peaks, the game was stopped when the green graph lines ends.

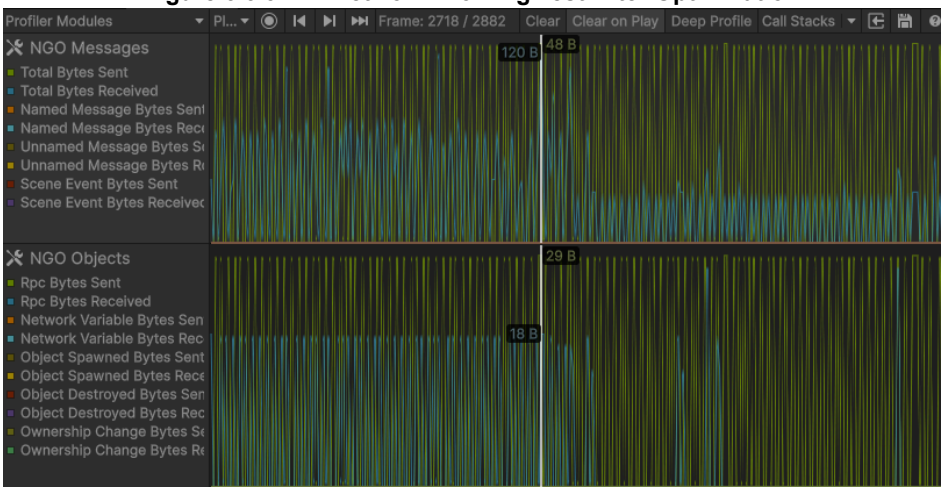
In the second one, after all the optimizations, the information sent is a lot more consistent and uniform than the previous image, causing more reliable information and latency reduction, leading to cleaner and more deterministic behaviour.

Figure 5.5.3.1.1: Network Profiling Test Before Optimization



(Own Source)

Figure 5.5.3.1.2: Network Profiling Test After Optimization



(Own Source)

This table shows an easier to understand comparison between both images:

Table 5.5.3.1.1: Net Player Optimization Comparison

	Unoptimized (Image 1)	Optimized (Image 2)
Avg. Bytes Sent	~424B per frame at peaks	~48B per tick
RPC Timing	Per FixedUpdate (120 Hz)	Per Net Tick (60 Hz)
Transform Sync	Raw & Unfiltered	Interpolated & Filtered
Bandwidth Usage	High and erratic	Low and predictable
Synchronization	Jittery, bandwidth-heavy	Smooth, responsive, and efficient

Finally, with all the optimizations employed, there is approximately an 88.7% of bandwidth usage reduction.

$$Reduction = ((424B - 48B)/424) * 100 \approx 88.7\%$$

5.5.3.2. Cinemachine Camera Owner Issue

An issue found with the players during the playtests was that if every Player prefab includes an active MainCamera and CinemachineBrain, and if nothing disables or deactivates the cameras for remote players, then Cinemachine will respond to the last active CinemachineBrain and input, this also applied to other components. The fix to this issue is to disable the components client-side if the instance of the player is not the owner, controlled by the current client who sends the input.

5.5.3.3. Character Data Network Variables

Scriptable Objects are used for establishing each characters' data in a fast and modular way, these are called Character Data. Some character stats need to be Network Variables in *Player.cs* that retrieves the initial values from the Character Data assigned, allowing to modify them server-side and execute all the logic while the clients can read those variables to display them on UI, giving important visual feedback, and apply stats effects, such as heal, damage, protect or boost.

5.5.4. Net Object Pooling

Projectiles are one of the most used object types in the game, each character of the vertical slice has at least one ability of this type and also are employed for the shoot attack, this means that each projectile is being instantiated in the server, and then in all the clients connected. Even though the projectiles logic and physics are only being executed server-side, to instantiate a new object in runtime each time that is needed can cause overhead, producing a huge performance impact in the game and the Network bandwidth that can also cause lag, desynchronization or latency.

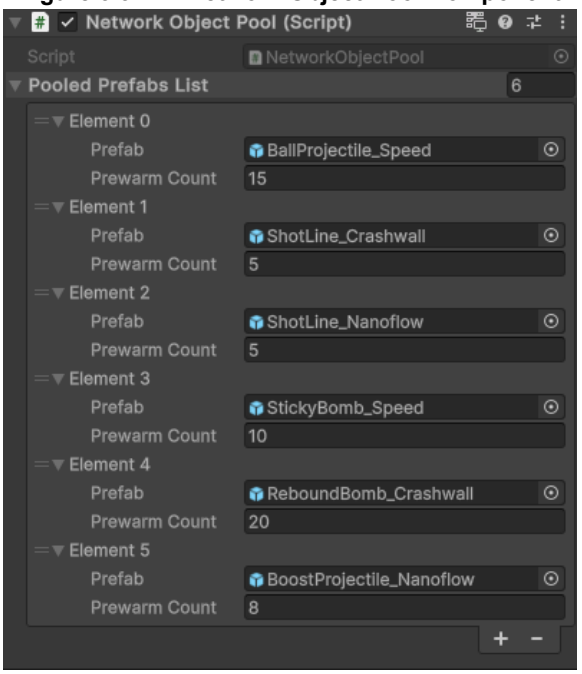
To fix this problem there is a design pattern called "Object Pool". It optimizes performance by reusing pre-instantiated objects stored, disabling and enabling them, instead of constantly creating and destroying new ones. Reusing objects minimizes memory allocation, garbage collection costs, and reduces overhead.

Netcode for GameObjects already provides support for it. "Allows to override the default Netcode destroy and spawn handlers with your own logic", Unity Multiplayer Object Pooling (Last updated on May 21, 2025 by Amy Reeve). This means that the "Spawn" and "Despawn" methods from Netcode, will have a different behaviour for the objects prefabs that are stored in the "pool". When "Spawn" is called on the server, the object will enable for all the clients, when "Despawn" is called, then the object is disabled and stored once more in the pool waiting to be enabled again.

For it to work properly, the script available in Unity's Multiplayer web page mentioned earlier has to be created in the project, and attached to a GameObject on each scene this is required. The objects that are going to be instantiated multiple times during the game are the ones that have to be added to the Object Pool from the script component, in the

following example it is shown the component with a list of multiple projectile objects from the project.

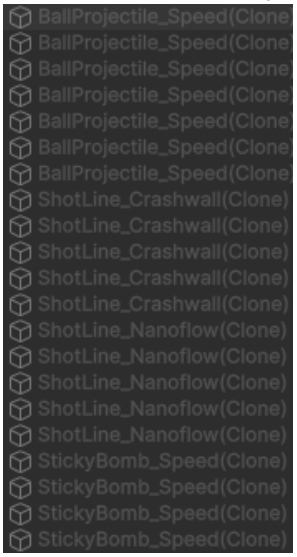
Figure 5.5.4.1: Network Object Pool Component



(Own Source)

Once the game is executed, each client and the server will instantiate the objects set for each scene, and disable them until they are spawned. In the Unity Editor a list of objects disabled must appear if the implementation is successful.

Figure 5.5.4.2: Instantiated Objects List



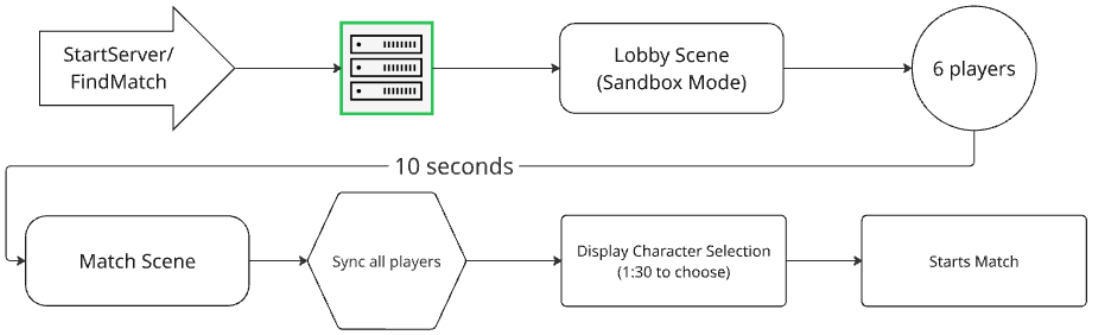
(Own Source)

5.5.5. Game Flow

For the game's flow, we will follow a Finite State Machine design approach, where each scene contains their own manager which controls, through an automatically controlled recursive Finite State Machine, what should be executed on each state. The server will have absolute control over how and when the state changes depending on events happening in the game. When changes are made, the server will send the information that will also invoke events, which the clients are subscribed to.

In the following picture can be seen the flow between scenes of the game, starting from the Main Menu where clients and server start the connection, going to the Lobby where clients will be able to play while waiting for other clients to connect to the server. Once all players needed are connected and synchronized, then it goes to the next scene where players choose their character and play the match.

Figure 5.5.5.1: Game Flow Diagram



(Own Source)

5.5.5.1. Scene loading synchronization

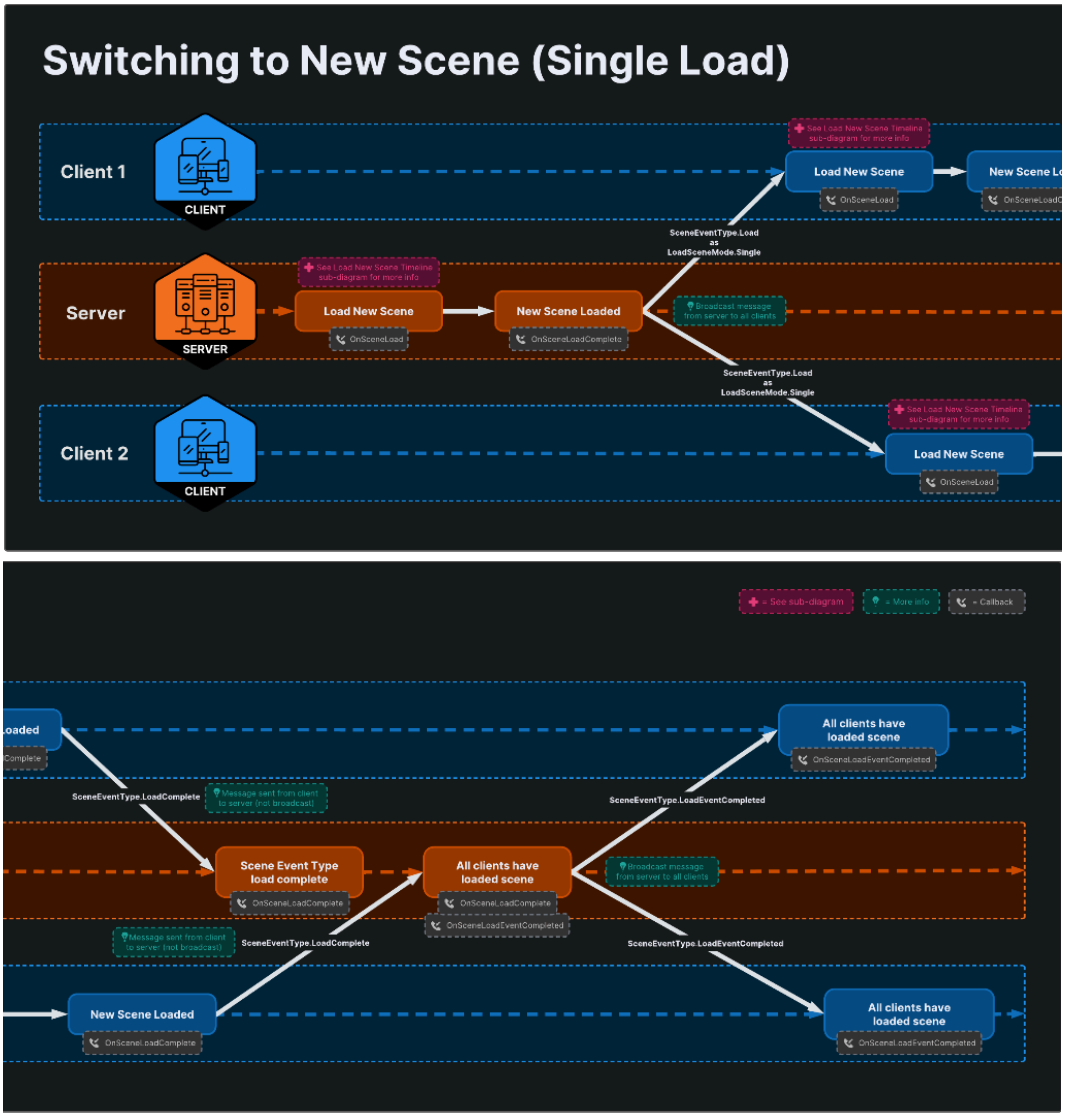
When loading a new scene, it is a must to take into consideration that all the players are synchronized with the server, to compensate latency or lag issues.

If the clients are not synchronized, then the behaviour of the whole online gameplay could suffer important unbalances, such as clients sending requests to do an action, when in the server's side or in other clients, are not needed to be executed because the action has or has not occurred yet.

Netcode for GameObjects brings us with an important feature, the Network SceneManager. When transitioning from one scene to another, the second one is loaded in "Single" mode, meaning that all previous scenes loaded will be unloaded to free memory. This will help to improve performance, especially on the server-side that will

need to handle all the Physics simulations and almost all the scripts behaviours from each scene.

Figure 5.5.5.1.1: Scene Single Load



(Own Source)

As we can see in the previous image, there is still a small variance between both clients even though they completed the synchronization process, there will still be a small latency of 10 to 100 milliseconds.

5.5.5.2. Lobby

The Lobby scene, contains the LobbyManager GameObject, in which the *LobbyManager.cs* script is attached. The LobbyManager handles the clients connection while allowing the players to practice before starting the match, it is completely server authoritative. In this scene there are just three states:

- Connecting
- Wait
- Completed

Once the server has loaded the scene, it spawns a ball for the incoming players to allow them to have fun while waiting other players.

First, before the player connects, the server check if their connection is allowed. An event is invoked server side with the client's request information, the server checks if the scene is full or not and then, it sends to the client a "ConnectionApprovalResponse" to allow the client to join the server.

```CSharp
private void ConnectionApproval(NetworkManager.ConnectionApprovalRequest request, NetworkManager.ConnectionApprovalResponse response)
{
    if (NetworkManager.Singleton.ConnectedClientsIds.Count >= 6)
    {
        response.Approved = false;
    }
    else
    {
        response.Approved = true;
    }
}
```

When a client is connected to the lobby it sets them a random character, making the players to try other characters during the sandbox space, while in the match are allowed to select their favourite.

The server waits for all six players to connect, once the Lobby is full it changes to the "Wait" state starting a timer that, once it gets to zero, will lead to the ArenaRoom scene where the match will be played.

```CSharp
void OnClientConnected(ulong clientId)
{
    if (!IsServer) return;

    Characters[] enumValues =
       (Characters[])System.Enum.GetValues(typeof(Characters));

    var character =
       (byte)UnityEngine.Random.Range(0, (enumValues.Length - 2));

    if(character != (byte)Characters.NONE)
    {
        GameObject player = Instantiate(charactersPrefabs[character],
                            Vector3.zero, Quaternion.identity);

        NetworkObject netObj = player.GetComponent<NetworkObject>();
        netObj.SpawnAsPlayerObject(clientId, true);
    }
    if (NetworkManager.Singleton.ConnectedClientsIds.Count >= 6)
    {
        SetLobbyState(LobbyState.WAIT);
    }
}
```

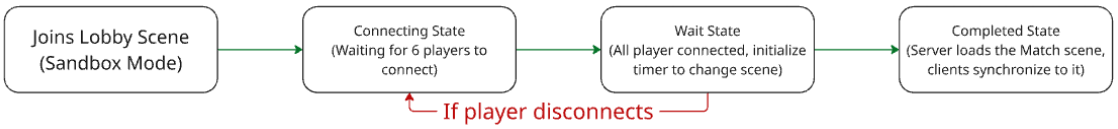
When a client is disconnected, it will receive a callback from the server allocated in the *NetworkManager.cs* script, called "OnClientDisconnect". This event will execute, in case of the Lobby scene, a scene load client-side to go back to the Main Menu scene, while the server will check if the Lobby is full or not to change or not to the "Connecting" state again.

```CSharp
void OnClientDisconnected(ulong clientId)
{
    if (IsClient && !IsServer)
    {
```

```
            StartCoroutine(DisconnectAndLoadMenu());
            return;
        }

        if (NetworkManager.Singleton.ConnectedClientsIds.Count < 6)
        {
            SetLobbyState(LobbyState.CONNECTING);
        }
    }
    private IEnumerator DisconnectAndLoadMenu()
    {
        yield return new WaitForSeconds(0.5f); // Let the server clean up
        SceneManager.LoadScene("MainMenu", LoadSceneMode.Single);
    }
```

The image shows the flow that the Finite State Machine follow:

Figure 5.5.5.2.1: Lobby Flow Diagram



(Own Source)

5.5.5.3. Match

This scene contains the MatchManager GameObject, where the *MatchManager.cs* script is attached. It controls the flow of the whole match, it is completely server authoritative as all the other managers.

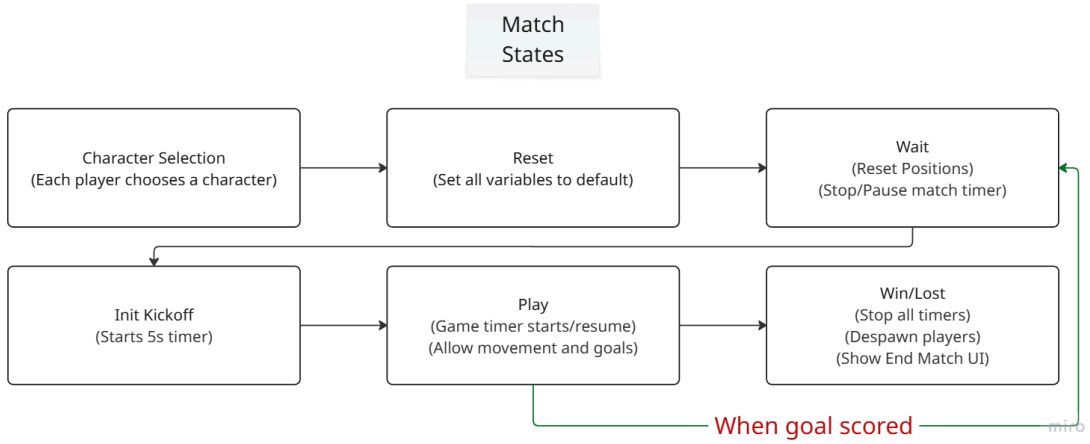
Here, the "ConnectionApproval" event is overridden and instead of checking the amount of players in-game, checks if the current scene, where the match happens, is loaded to decide the approval. In this case, it is handled this way because the lack of time at that point of the development process. The best approach is to handle if the session lacks of players, and a reconnection process in case a client that was previously connected, disconnects and wants to re-join.

When loading the scene from the Lobby to the Match, the server subscribes to an event that is invoked once the Match scene is completely loaded and all the clients synchronized. The subscribed method to that event, sets a new state if all the players are still in the game and synchronized, the "Character Selection" state.

In the "Character Selection" state each client is set to a team, Local or Visitant, and a timer begins. The server invokes an event that will call an RPC method to activate the character selection UI, displaying a button for each character available. The buttons disable or enable their interactability depending on the selection of the team mates, if a player is already chosen, then it is not available, except if they change it.

Once the timer gets to 0, it goes to the next state, the "Reset" state. The match variables are reset, such as the goals and the timer. Then all the players' character prefabs are spawned in their team spawn point positions, and the server sets the state to "Wait".

Figure 5.5.5.3.1: Match Flow Diagram



(Own Source)

The "Wait" state is responsible for resetting the positions and rotations of both the players and the ball. First, setting their respective Rigidbody "isKinematic" property to true, to avoid undesired physics behaviour that can cause the bodies to appear in other unwanted positions and rotations. The position and rotation is set to the team spawn point, the Cinemachine Camera component rotation is also realigned.

After the reset, the state changes to "Init", where the kickoff countdown timer begins. Once the timer reaches zero, the state transitions to "Play", allowing the players and ball to move freely and score goals.

If a goal is scored, the state changes into the "Goal" state. Here, the score is updated depending on the scoring team. Also checks if the game has to end. the game follows what is called the "Golden Goal" rule, which states that if the time ends and the score is tied, the next team to score wins, transitioning to the "Finalized" state.

In case that the match has not yet concluded, the state transitions back to "Reset" re-executing all the behaviour previously explained recursively until the match finally reaches its end condition.

Once the match is completed, the game enters the "Finalized" state, during which a UI panel is displayed to all the players. This panel clearly shows whether their team has won or lost, providing appropriate feedback to conclude the session.

5.5.6. Build the Game

To create a proper build, the Unity version installed must contain the following modules implemented:

- **Server Build:** Linux Dedicated Server Build Support. Other server build types can add additional costs when using a cloud hosting service.
- **Client Build:** Can be any platform the project is focused for, it must not contain the word "Server", for this project is used the Windows Build Support (IL2CPP).

It is highly recommended to do a "Clean Build" when re-building the game in case there is some error or changes done, due to the Multiplayer Role property that can cause building the game with a different role than the expected.

5.5.6.1. Server Build

The build profile platform must be in "Linux Server" and the Multiplayer Role to "Server" in order to produce the build correctly.

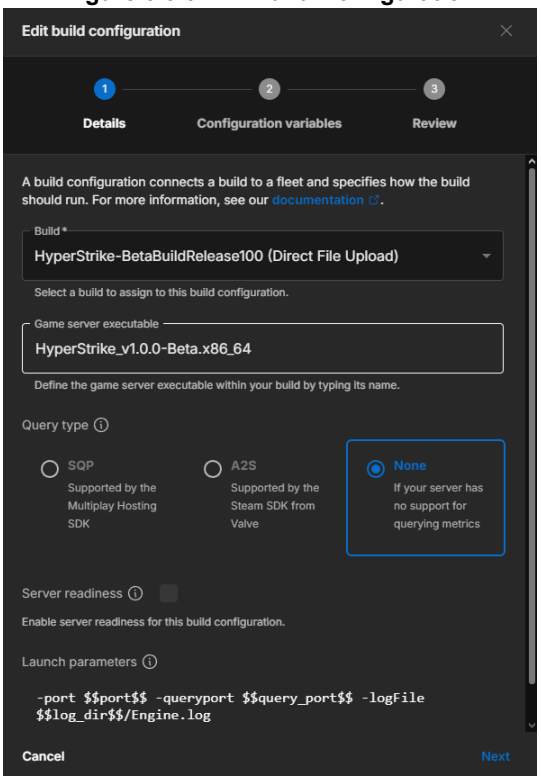
Once it is built, the files generated need to be uploaded to the Untiy Multiplay Hosting service found in the Unity Cloud web page to create a new server instance. If it is first time use it appears a setup guide about the steps to follow to not get lost creating the instance.

First, the project needs to be linked to a Unity Cloud organization, this can be set in Unity Hub. Once done, the next step is to create a build in the Multiplay Hosting service

by uploading the files previously created, with the direct upload method, and selecting the appropriate operating system, in the case of this project, it will be Linux.

For the build to execute appropriately, it needs to be generated a "Build Configuration", it will contain the information about which build to use, the server executable (.x86_64 for Linux), and the launch parameters. The query type and server readiness are not handled in this project, they need to be disabled in order to prevent the server to crash while running.

Figure 5.5.6.1.1: Build Configuration



(Own Source)

A fleet holds a collection of servers in specific regions, the region for our server is Europe but it can be anyone that fits your project. The minimum available servers is recommended to be set to 0, this reduces costs by automatically deallocating the servers after an hour of being online, the inconvenience is that the server must be restarted in order to continue playing.

Finally, before initializing the server a test allocation must be created to ensure the server works correctly.

5.5.6.2. Client Build

For this build type it is needed the build profile platform set to "Windows", the Multiplayer Role to "Client" and the scenes that the game needs to run appropriately.

And to ensure that the client connects to the server, set the IP generated by the Multiplay Hosting service when the server is allocated.

5.6. Environment mechanics

The environment mechanics for the Vertical Slice that the team planned, were easy and simple to implement. For the length of the project we planned just one environment where the main theme was Pinball.

5.6.1. Pinball Arena Pitch

This arena contains bumpers and corner bouncers that creates an impulse force in the direction of the normal where the object collided, in this case we wanted to only affect the players and the ball. The quantity of force that each bumper or bouncer applies, and the objects that are affected, can be set manually in the inspector.

The behaviour checks if the object that collided with the bumper contains the desired tag to apply the force. The bumpers contain a capsule collider and the corner bouncers need a mesh collider.

```CSharp
public class Bouncer : NetworkBehaviour
{
    [SerializeField] float force = 100f;
    public string[] allowedTags;
    private void OnCollisionEnter(Collision other)
    {
        if (other == null) return;

        // Check all the tags the object can impulse
        for (int i = 0; i < allowedTags.Length; i++)
        {
            if (other.gameObject.CompareTag(allowedTags[i]))
            {
              Rigidbody rb = other.gameObject.GetComponent<Rigidbody>();
```

```
                if (rb != null)
                {
                    Vector3 dir = other.GetContact(0).normal;

                    rb.AddForce(dir.normalized * force,
                                ForceMode.Impulse);
                }
            }
        }
    }
}
```

The walls and ramps from the pitch have smaller drag coefficients than the ground, an individual friction value is applied with their respective Physics Material.

The ramps have a high slope, to prevent the players to get stuck on them with the drag coefficient they have, the dynamic and static frictions were set to 0. The walls were also set to 0 to prevent the players to fall fatser than expected and also avoid getting stuck when moving the player while colliding with them.

Finally, the last mechanic implemented in the arena were the movable walls that are in front of the goals on each side of the pitch. Those were implemented with the help of the team designer that created the script behaviour. The *AlternatinBoxesAnimated.cs* script, moves smoothly the object upwards and downwards as we can see in the following code snippet.

It works with a simple timer that controls the speed in which the walls move and doing an interpolation between the initial and the end positions.

CSharp
```
void StartAnimation()
    {
        isAnimating = true;
        animationTimer = 0f;
```

```
        if (isBoxAUp)
        {
            movingUp = boxB;
            movingDown = boxA;
        }
        else
        {
            movingUp = boxA;
            movingDown = boxB;
        }

        isBoxAUp = !isBoxAUp;
    }
```

Once the animation is finished it swaps the initial and end positions to revert the movement.

```CSharp
void Animate()
    {
        animationTimer += Time.deltaTime;
        float t = Mathf.Clamp01(animationTimer / animationDuration);

        if (isBoxAUp)
        {
            boxA.localPosition = Vector3.Lerp(boxADownPos,
                                 boxAUpPos, t);
            boxB.localPosition = Vector3.Lerp(boxBUpPos,
                                 boxBDownPos, t);
        }
        else
        {
            boxA.localPosition = Vector3.Lerp(boxAUpPos,
                                 boxADownPos, t);
```

```
            boxB.localPosition = Vector3.Lerp(boxBDownPos,
                                    boxBUpPos, t);
        }

        if (t >= 1f)
        {
            isAnimating = false;
        }
    }
}
```

5.7. HyperStrike Utilities

The utility class called "HyperStikeUtils", provides various methods to assist with environment awareness and collision detection during gameplay. These methods are especially useful for wall detection, checking whether the player is grounded, or identifying if an object is fully enclosed within colliders.

5.7.1. CheckGrounded

This method checks whether an object is grounded using a raycast. It calculates the origin point at half the object's height. Then, a ray is cast downwards from the origin point to the object's base, if the ray hits a surface the method returns true, indicating that the object is grounded.

It is essential to determine if the player is on the ground before allowing jumping or other actions that require the player to be grounded or in air.

```CSharp
public bool CheckGrounded(Transform transform, float objHeight = 1.0f)
{
    if (transform == null) return false;
    float dist = objHeight * 0.5f + 0.1f;
    Vector3 og = new Vector3(transform.position.x,
                transform.position.y + (objHeight * 0.5f),
                transform.position.z);
```

```
    return Physics.Raycast(og, Vector3.down, dist);
}
```

5.7.2. CheckWalls

This method checks especially objects that have the desired layer mask applied and sets a parameter called "CamerTilt" employed to set the Cinemachine Camera's "Dutch" value when the player is wall-running. It only checks collision to the left and to the right.

CSharp

```
public bool CheckWalls(Transform transform, ref RaycastHit wallHit,
                       ref CameraTilt cameraTilt, LayerMask wallMask)
{
    bool[] wallChecked = new bool[]
    {
        false, false, false, false, false
    };
    Vector3[] directions = new Vector3[]
    {
        transform.right,
        -transform.right
    };

    for (int i = 0; i < directions.Length; i++)
    {
        RaycastHit hit;
        Vector3 og = new Vector3(transform.position.x,
                    transform.position.y + (transform.localScale.y),
                  transform.position.z);

        wallChecked[i] = Physics.Raycast(og, directions[i], out hit,
                         transform.localScale.x * 0.5f + 0.05f,
                         wallMask);

        if (wallChecked[i])
```

```
        {
            wallHit = hit;

            if (directions[i] == transform.right ||
                directions[i] == transform.right + transform.forward)
            {
                cameraTilt = CameraTilt.RIGHT;
            }
            else
            {
                cameraTilt = CameraTilt.LEFT;
            }
        }
    }
    return wallChecked.Contains(true);
}
```

5.7.3. CheckObjectInsideCollision

This methods determines if an object is fully inside a collider. Raycast are thrown in six directions from the object's center following positive/negative X, Y, Z axis. If all raycasts return true, the object is considered completely inside a collider.

Helps detect edge cases where an object may be stuck inside geometry, due to physics bugs or scene setup issues, and to detect if the ball is scored inside a goal.

```CSharp
public bool CheckObjectInsideCollision(Transform transform)
{
    bool[] completelyInside = new bool[]
    {
        false, false, false, false, false, false,
    };

    Vector3[] directions = new Vector3[]
    {
```

```
            transform.right,
            transform.forward,
            transform.up,
            -transform.right,
            -transform.forward,
            -transform.up
        };

                for (int i = 0; i < directions.Length && i < completelyInside.Length; i++)
        {
                    Debug.DrawLine(transform.position, transform.position + directions[i], UnityEngine.Color.green);
                completelyInside[i] = Physics.Raycast(transform.position, transform.position + directions[i], transform.localScale.magnitude / 2);
        }

        return !completelyInside.Contains(false);
    }
```

5.8. UI

Even though my thesis is focused on the project's code, I have created the logo of the game that can be found in the Main Menu scene and the pause menu, and the icon for the application executable as additional implementations.

Figure 5.8.1: HyperStike Logo

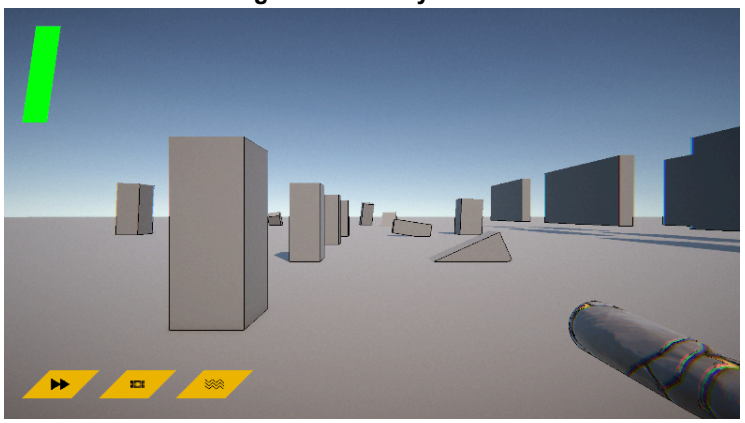
(Own Source)

Figure 5.8.2: HyperStike Icon

(Own Source)

The images were created using *Affinity Photo 2*, which was also used to design the in-game user interface elements, including buttons and player HUD components, except for the ability icons.

Figure 5.8.3: Player HUD



(Own Source)

5.8. Pre-Alpha Tests

During the tests, we found out that the designed pinball-themed map needed some level design improvements to achieve a good feeling with the mechanics implemented.

The ball size was deemed too small for the scale of the environment and the fast player movement during the first tests. It was often difficult for players to track the ball visually, particularly during aerial movement or fast actions. Increasing the ball size was identified as a necessary change to improve readability and better match the arcade-style physics-based gameplay.

The players occasionally got stuck when running or jumping near the walls, the team realized that the issue was in the collision system and the ray detection logic used to manage wall proximity for movement state transitions. In some particular cases, the rays failed to detect a valid wall surface or returned an unintended player state value, causing the player to sometimes "snap" or freeze, especially when the walls were on the left side of the player. The ray casting and player state logic were adjusted to address the bug.

The sliding mechanic presented multiple bugs. In particular, when a player slid into a corner or complex collision geometry, an unintended impulse force was applied when releasing the slide-binded button, launching the character in upwards direction. Debugging this issue we realized that this feature needed a refinement in the momentum preservation and slope detection to handle some collision cases properly.

The feedback obtained from the team members have been essential in refining the physics-based mechanics and adjusting design elements to match the desired gameplay experience. The team documented each issue and finally created tasks in HacknPlan to resolve them in the following sprints, to address them in future builds before reaching the end of the Alpha phase.

5.9. Alpha Tests

During this phase multiple tests were done, specifically internal tests by the team members, and one external test by the end of the phase to test the gameplay in LAN connection.

The use of profiling tools to debug the quantity of data the objects were sending, the jitter and latency, were essential to determine issues related with net objects data and RPC calls overhead. Those issues were fixed following the optimizations explained in the Player Optimization section.

The bouncers' impulse force was not correctly applying the desired direction, this was fixed by normalizing the force direction, we wanted to apply a force in a direction, not based on how far away the objects are. If it is not normalized, the force becomes stronger the farther apart the objects are when the collision happens, which is not what we wanted for the bounce mechanic.

5.9. Beta Tests

The beta testing phase marked the final stage of the development process and played a crucial role in polishing the final version of this thesis project. All tests were conducted externally with a diverse set of players to ensure broad and unbiased feedback.

Three separate testing groups were organized, each with distinct player profiles:

- **Group 1**: Composed of players familiar with both PC and console gaming platforms.
- **Group 2**: Focused more on PC users and included a mix of experienced hero shooter players as well as individuals with little to no gaming background.
- **Group 3**: Primarily consisted of players who regularly engage with shooter-style games on consoles.

This structured approach to testing provided a wide range of perspectives, allowing for comprehensive feedback on gameplay, controls, UI/UX, and the overall experience. Due to the lack of time, not all the issues from the feedback were addressed.

5.10. External Beta Test (v0.5.0 - 23/06/2025)

This first beta test was done without an online connection and it was aimed to evaluate the core gameplay mechanics, visual feedback, and the player experience. The following feedback list was gathered:

- The ball's bounciness made it feel too light. To fix this, the bounciness value of the Physics Material was reduced to 0.3 to give it a heavier and more realistic feel related to a pinball ball.
- Players noted that the characters movement felt slippery. **The surfaces fricitonal values and rigidbodies' drag coefficient were increased.**
- Player noticed the lack of visual feedback for the abilities, such as VFX, it was difficult for them to understand when and how the abilities were activated.
- UI elements for ability cooldowns and other character stats were missing.
- Some minor bugs related to the use of gamepad when navigating through the UI when pausing. Button selection was not correctly set.

- The projectile explosions were not noticeable enough to relate the knockback effect they applied ("It's hard to tell what pushed me"). **The scale of the VFX was increased.**
- The menu layout was generally well received, although some suggested slightly reducing the size of the buttons for improved readability and aesthetics.
- Characters had difficulty running up ramps, often getting stuck or losing momentum. **Friction values from the Physics Material was reduced to 0.**
- Craswhall and Nanoflow characters' projectiles looked more like laser beams than bullets, also the effect was applied from side to side of the room. **The raycast range was reduced.**
- The bounce effect from bumpers was praised and considered fun and well executed.
- The camera dynamics were well received, with smooth movement.
- Testers appreciated that the movement system introduced a slight challenge.
- The maps and characters were complimented for their visual style and design.
- Scoring a goal lacked music or celebrations. **Music and other sound effects were added.**
- Testers pointed out the lack of characters' role and abilities description in the character selection screen to better inform their choices.
- There was a lack of visual feedback for the shoot attack reload.

5.11. External Beta Test Online (v0.5.5 - 26/06/2025)

This beta test went wrong due to some errors that produced the server to crash. If there were more than four players in the match it thrown an error when resetting the players positions. The list of the visitant players was not set properly so it caused the error.

We could not continue with the test for that day. The next day the errors were fixed and the test was successfully executed with the players using gamepad, due to mouse sensitivity issues. Even though some feedback gathered was similar to the previous beta test session, a distinct profile provided new and important feedback:

- There is a noticeable lack of feedback sounds during critical events, such as taking damage, dying, or explosions. Currently, only the background music and goal scoring sound effects are audible.
- Adjusting the mouse sensitivity in the options menu was not applying a difference. We **implemented the *GameManager.cs*** script which follows the Singleton design pattern, controls the mouse sensitivity and invert Y value, the GameObject is non-destructible, being **present in all the scenes to apply the changes**.
- Implementing a weapon reload mechanic could improve gameplay. Players tend to spam the shoot action to hit the ball, which becomes the dominant strategy to defense and score.
- The gameplay feels more focused on interacting with the ball than on combat or using abilities.
- Moving around the map and rotating the camera feels unintuitive and sluggish. This issue was fixed by **reducing the "Rotation Interpolation Max Time"** from the player and camera Network Transform, **from 0.1 to 0.05**, a higher value was producing the interpolation to delay the rotation movement causing latency.
- Some collisions seem to push players farther than expected, possibly due to input sensitivity or friction values. The problem lied on sending **too much RPC input calls**, producing undesired body acceleration, it was addressed the way explained in the Player Optimization section, by **sending the calls on network tick** instead of fixed timestep.
- Controls beyond basic movement (WASD) are not immediately clear. Displaying keybindings in the UI or making them more accessible from the main menu would improve onboarding.
- The game is enjoyable and fun, but movement feels slightly clunky and lacks fluidity. **Also fixed by sending RPC calls through network ticks**.
- The game concept is solid, though there's room for improvement in the overall user experience (UX).

- Adding control information as decals or hints in the lobby could help players to familiarize with the game before the match.
- The artistic style is visually appealing, though the UI could be refined slightly.
- There is excessive friction on surfaces, causing players to get stuck in certain areas. **The ground Physics Material friction values were slightly reduced**.
- The character feels "floaty," making the player feel that the character was almost flying when jumping from high areas. **A force downwards when the player is falling was implemented in order to give higher gravity feeling**.
- The weapon visual occasionally twitches or glitches during matches. **This issue was related to the rotation interpolation and mouse sensitivity addressed previously.**
- Team visibility and identification needs to improve, it is unclear which team a player belongs to.
- Collisions with other players can be disruptive or frustrating due to how they're handled.

5.12. External Beta Test Online (v0.7.0 - 28/06/2025)

This was the last beta test conducted before the project's deadline arrived. The session was organized to assess the stability of the network, and identify last-minute bugs or balancing issues:

- When entering the options menu using a game controller, there is no clear way to exit.
- Players are unsure which team they belong to.
- Player names could be displayed to improve team identification.
- Only the stunned player is aware they are stunned, a visual indicator is needed for the other players.
- Base movement speed feels too slow.
- It is currently possible to have the same character multiple times if the player waits for the match to start without selecting a character.

- The character selection screen should make the selected character more visually obvious.
- After being stunned, there should be a short invulnerability period to avoid receiving damage immediately.
- Character abilities should also be displayed on the character selection screen, and each ability should have a brief description.
- Sometimes, the game does not allow players to select a character.
- Moving walls movement is not synchronized.
- Character rotation should be reset when a player respawns to avoid incorrect orientation.
- The key bindings for each ability should be clearly displayed in the UI.

6. Project validation

During the last week of the project's development process, tests were conducted before the project's deadline. These sessions played a crucial role in validating the project and refining the final product. The feedback collected not only helped us polish the game before the deadline but also provided valuable insights for our growth as future game developers.

The beta tests revealed recurring themes, allowing us to identify core strengths and areas for improvement in our game.

The most common reported issues were related to:

- The player's movement and rotation fluidity, which was not reported during the last session after the fix implementation.
- The visual and audio feedback, due to the lack of time we could not implement enough VFX, sound effects, or even polished UI for the final product.
- The confusion about which team belongs the player, and knowing what each character ability did. Indicating a lack of clarity in the game.
- The players finding out that shooting the ball was often more effective than using the abilities. The lack of a shoot reloading mechanic made the players spam the attack, reducing strategic depth.

The most praised aspects were:

- The visual style and artistic direction of the environments and characters, which increased the enjoyment of the gameplay experience.
- The combination of fast-paced shooter mechanics with football-like gameplay was well received.
- The ball's bounciness, bumpers, and projectile impacts were fun and satisfying.
- The gamepad implementation was praised in comparison to the keyboard controls, for its responsiveness and layout that felt more intuitive.
- The network connection did not present lag, or unwanted behaviour. It is reliable enough for a fast-paced shooter game.

At the end, through the use of events, we gathered data based positions where the players received more damage, where the ball was hit by a player before scoring, and where were stunned the most. We sent the data to an SQL database and retrieved it to an

Excel. The main idea was to implement a heatmap shader in the scene where the match was executed to visualize correctly the data retrieved, but we could accomplish it before the deadline. It is hard to understand what is happening by just looking at the graphs that Excel generates, so I could not get to any conclusion at all.

7. Conclusions

This was an ambitious project with the idea to create a video game and mix some of the genres I like the most while focusing this thesis on what I want to focus on professionally. From the beginning, our goal was not only to create a fun and working vertical slice of a video game but also to explore difficult technical challenges that are typically addressed in larger productions. The final result that my teammates and I achieved is better than I expected, considering the time we had to develop it.

I have gained invaluable experience developing HyperStrike in areas that are crucial to game development. One of the most challenging and rewarding aspects of the project was to implement physics-driven mechanics in a multiplayer online game. While Unity provides tools to develop multiplayer games, it is complex to achieve reliable physics together with a networked system. Many aspects of a video game must be taken into account, in order to achieve the reliability and responsiveness proper of a fast-paced game, requiring continuous testing and sometimes rewriting core systems.

Another important objective accomplished was creating an ability system, implementing a modular system which makes it fast and easy to implement using the ability types available. I had to ensure that the abilities were fully synchronized for all the players and that worked correctly with the unique logic each ability type executes.

I have also managed to set up a dedicated server using Unity Multiplay Hosting service, handling an automatized game logic and simulating all the physics behaviour. It was my first time developing a video game which needed to execute all the logic by itself while managing other players.

My main goal was to make physics to seamless integrate with online gameplay, and I think I have successfully completed it to a large extent. It still has some problems, but generally the game works well. It feels fast and fun to play, and the visuals help make it enjoyable.

7.1 Future Lines

There is always room for improvement, apart from addressing the beta tests issues gathered from the players' feedback, I would like to implement new features that I could not include on the project due to the lack of time. Also, if my colleagues agree, I would like to continue working on this project idea and polish it for a final market product.

Some of the new features I would like to implement would be:

- Client-side prediction and reconciliation to make the movement and actions feel smoother and more responsive, even though it is not implemented on the project due to the lack of time, is an important feature in networked games, especially for a Server Authoritative model.
- Implement client reconnection handling, if the player disconnects, they can re-join th match without issues.
- Authentication system for players to have an account in the cloud managed by username and password.
- Include cloud synchronized leaderboards.
- Use Scriptable Objects for data that does not need to change in runtime.
- Implement and improve UI elements for character stats and abilities.
- Finish the incomplete abilities, some are still in development or missing final effects and polish.

There are many things that can be added or improved, and I believe this game has a lot of potential.

8. Bibliography

[1] 'Real-Time 3D Development Platform & Editor', Unity. [Online]. Available: https://unity.com/products/unity-engine

[2] 'La herramienta de creación 3D en tiempo real más potente', Unreal Engine. [Online]. Available: https://www.unrealengine.com/es-ES/home

[3] G. Engine, 'Features', Godot Engine. [Online]. Available: https://godotengine.org/features/index.html

[4] 'Unity 6 is here: See what's new'. [Online]. Available: https://unity.com/blog/unity-6-features-announcement

[5] U. Technologies, 'Unity - Manual: Physics'. [Online]. Available: https://docs.unity3d.com/6000.0/Documentation/Manual/PhysicsSection.html

[6] 'Unity Physics overview | Unity Physics | 1.3.10'. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.physics@1.3/manual/index.html

[7] 'GitHub - NVIDIAGameWorks/PhysX-3.4: NVIDIA PhysX SDK 3.4'. [Online]. Available: https://github.com/NVIDIAGameWorks/PhysX-3.4

[8] 'PhysX SDK', NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/physx-sdk

[9] 'Networked Physics', Gaffer On Games. [Online]. Available: https://gafferongames.com/categories/networked-physics/

[10] GDC 2018, It IS Rocket Science! The Physics of Rocket League Detailed, (Apr. 24, 2018). [Online Video]. Available: https://www.youtube.com/watch?v=ueEmiDM94IE

[11] J. Glazer and S. Madhav, Multiplayer Game Programming: Architecting Networked Games, 1st ed. Addison-Wesley Professional, 2015.

[12] 'Insights about Multiplayer Game Design: Strategies and Insights'. [Online]. Available: https://www.searchmyexpert.com/resources/game-development/multiplayer-game-design

[13] 'Source Multiplayer Networking - Valve Developer Community'. [Online]. Available: https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking#Lag_compensation

[14] 'Multiplayer Lag Compensation'. [Online]. Available: https://vercidium.com/blog/lag-compensation/

[15] DevinDTV, How It Works: Lag compensation and Interp in CS:GO, (Nov. 22, 2015). Accessed: Mar. 23, 2025. [Online Video]. Available: https://www.youtube.com/watch?v=6EwaW2iz4iA

[16] PlayOverwatch, Developer Update | Let's Talk Netcode | Overwatch, (Apr. 05, 2016). Accessed: Mar. 23, 2025. [Online Video]. Available: https://www.youtube.com/watch?v=vTH2ZPgYujQ

[17] 'Multiplayer Game Development Made Easy | Photon Engine'. [Online]. Available: https://www.photonengine.com/

[18] 'Introduction | Fish-Net: Networking Evolved'. [Online]. Available: https://fish-networking.gitbook.io/docs

[19] 'GitHub - MirrorNetworking/Mirror: #1 Open Source Unity Networking Library'. [Online]. Available: https://github.com/MirrorNetworking/Mirror#made-with-mirror

[20] 'Multiplay Hosting Service', Unity. [Online]. Available: https://unity.com/products/game-server-hosting

[21] 'Customizable Game Matchmaking Software Service | Unity'. [Online]. Available: https://unity.com/products/matchmaker

[22] 'Networking & Netcode Software Solution', Unity. [Online]. Available: https://unity.com/products/netcode

[23] GDC 2017, Overwatch Gameplay Architecture and Netcode, (Feb. 08, 2019). [Online Video]. Available: https://www.youtube.com/watch?v=W3aieHjyNvw

[24] G. Kroupp, 'Mastering Multiplayer Game Architecture: Choosing the Right Approach - Getgud.io'. [Online]. Available: https://www.getgud.io/blog/mastering-multiplayer-game-architecture-choosing-the-right-approach/

[25] 'What is Amazon GameLift Servers? - Amazon GameLift Servers'. [Online]. https://docs.aws.amazon.com/gamelift/latest/developerguide/gamelift-intro.html

[26] R. Nystrom, Game Programming Patterns, 1st ed. Genever Benning, 2014.

[27] 'The Entity-Component-System - An awesome game-design pattern in C++ (Part 1)'. [Online]. https://www.gamedeveloper.com/design/the-entity-component-system---an-awesome-game-design-pattern-in-c-part-1-

[28] 'Entities overview | Entities | 1.3.10'. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.entities@1.3/manual/index.html

[29] 'Gameplay Ability System for Unreal Engine | Unreal Engine 5.2 Documentation | Epic Developer Community'. [Online]. Available: https://dev.epicgames.com/documentation/en-us/unreal-engine/gameplay-ability-system-for-unreal-engine?application_version=5.2

[30] G. David, 'Unreal Engine's Gameplay Ability System, from programming framework to designer's tool'.

[31] 'Networking Scripted Weapons and Abilities in "Overwatch"'. [Online]. Available: https://www.gdcvault.com/play/1024653/Networking-Scripted-Weapons-and-Abilities

[32] L. Gelinas, 'Simplify multiplayer game creation - Unity Engine', Unity Discussions. [Online]. https://discussions.unity.com/t/simplify-multiplayer-game-creation/1529805

[33] 'The Evolution of Game Mechanics: A Journey Through Time'. [Online]. Available: https://toxigon.com/the-evolution-of-game-mechanics

[34] 'The Evolution of Adventure Game Mechanics - Games Verge'. [Online]. Available: https://www.gamesverge.com/the-evolution-of-adventure-game-mechanics/

[35] M. Alvarado, 'How have video games improved over the years?', Gaming Pedia. [Online]. https://www.ncesc.com/gaming-pedia/how-have-video-games-improved-over-the-years/

[36] 'Gaming Intelligence: How AI is revolutionizing game development'. [Online]. https://interestingengineering.com/innovation/gaming-intelligence-how-ai-is-revolutionizing-game-development

[37] 'The Vision Pro NBA app turns some games into a miniature 3D diorama | The Verge'. https://www.theverge.com/news/613796/nba-tabletop-ar-vision-pro-app-league-pass

[38] 'Half-Life: Alyx is turning 5, so we look back at its impact | Polygon'. [Online]. https://www.polygon.com/gaming/542162/half-life-alyx-5th-anniversary-vr-pc

[39] gbyt55, 'AR and VR: The Future of Gaming Immersive Experiences', Future Vision. [Online]. https://gbyte.co/how-ar-and-vr-are-transforming-the-gaming-industry/

[40] Admin, 'The Future of VR and AR in Gaming: Trends to Watch', Ocean of Games. [Online]. https://oceanofgames.com/the-future-of-vr-and-ar-in-gaming-trends-to-watch/

[41] A. Agrawal, 'Unveiling Haptic Feedback: Complete Guide To This Innovation'. [Online]. Available: https://thefuturisticminds.com/haptic-feedback/

[42] matthew burgos I. designboom, 'Disney's HoloTile floor moves any person or object like telekinesis for immersive VR gaming', designboom | architecture & design magazine. https://www.designboom.com/technology/disney-holotile-floor-treadmill-vr-lanny-smoot-01-23-2024/

[43] 'Skills and Abilities', Game Mechanics Wiki. [Online]. Available: https://gamemechanics.fandom.com/wiki/Skills_and_Abilities

[44] 'The Evolution Of Video Gaming: From Consoles To New Digital Horizons | GamesCreed'. https://www.gamescreed.com/blogs/the-evolution-of-video-gaming-from-consoles-to-new-digital-horizons/

[45] 'The evolution of character abilities - GamesFandom'. [Online]. Available: https://gamesfandom.com/the-evolution-of-character-abilities/

[46] 'History of Video Games - Playing History'. [Online]. Available: https://playinghistory.org/history-of-video-games/

[47] 'Player Skill, Character Skill, and Skill Progression Systems'. [Online]. Available: https://www.gamedeveloper.com/design/player-skill-character-skill-and-skill-progression-systems

[48] 'The Evolution of Hero Shooters: A Journey Through Gaming History', Toxigon. [Online]. Available: https://toxigon.com/the-evolution-of-hero-shooters/

[49] 'What the strange evolution of the hero shooter tells us about the genre's future | PC Gamer'. https://www.pcgamer.com/what-the-strange-evolution-of-the-hero-shooter-tells-us-about-the-genres-future/

[50] 'Overwatch 2 Spotlight: Every big announcement from Blizzard's event | Polygon'. [Online]. https://www.polygon.com/news/522603/overwatch-2-spotlight-recap-perks-stadium-new-heroes

[51] M. McWhertor, 'Overwatch 2's new perks system and Stadium mode, explained', Polygon. https://www.polygon.com/news/522508/overwatch-2-perks-system-stadium-mode-explained

[52] L. Zhouxiang, 'The Birth and Development of Sports Video Games From the 1950s to the Early 1980s', Sport History Review, vol. 54, no. 2, pp. 200–224, Nov. 2023, doi: 10.1123/shr.2022-0037.

[53] 'Former Overwatch 2 marketing head says 50 million active users since launch'. [Online]. Available: https://esports.gg/news/overwatch/overwatch-2-active-users-data/

[54] 'Overwatch's Share of the Hero Shooter Market + Statistics - General Discussion', Overwatch Forums. [Online]. Available: https://us.forums.blizzard.com/en/overwatch/t/overwatchs-share-of-the-hero-shooter-market-statistics/947420

[55] T. Newham, N. Scelles, and M. Valenti, 'The Consequences of a Switch to Free-to-Play for Overwatch and Its Esports League', JRFM, vol. 15, no. 11, p. 490, Oct. 2022, doi: 10.3390/jrfm15110490.

[56] 'Rocket League Population Stats | Population | RLStats'. [Online]. Available: https://rlstats.net/population

[57] D. Andric·Statistics·, 'How much money has Rocket League made? — 2025 statistics | LEVVVEL'. [Online]. Available: https://levvvel.com/rocket-league-statistics/

[58] 'Rocket League: A Phenomenal Game | Success Story'. [Online]. Available: https://gamematrix.io/rocket-league-game-success-story/

[59] 'Gantt vs Agile: differenze e combinazioni - Twproject', Twproject.com. [Online]. Available: https://twproject.com/blog/gantt-vs-agile-differences-and-combinations/

[60] 'Análisis SWOT: Definición, guía y ejemplo | SafetyCulture'. [Online]. Available: https://safetyculture.com/es/temas/analisis-swot/

[61] A. C. Team, 'Sprint in Agile: Sprint Project Management | Adobe Workfront'. [Online]. Available: https://business.adobe.com/blog/basics/sprints

[62] 'Salary: Junior Game Designer in Barcelona, Spain 2025', Glassdoor. [Online]. Available: https://www.glassdoor.com.hk/Salaries/barcelona-spain-junior-game-designer-salary-SRCH_IL.0,15_IM1015_KO16,36.htm

[63] 'Sueldo: Junior 3d Artist en España 2025 | Glassdoor'. [Online]. Available: https://www.glassdoor.es/Sueldos/junior-3d-artist-sueldo-SRCH_KO0,16.htm

[64] 'Sueldo: Junior Software Developer en España 2025', Glassdoor. [Online]. Available: https://www.glassdoor.es/Sueldos/junior-software-developer-sueldo-SRCH_KO0,25.htm

[65] 'CO2 emissions per kWh in Spain - Nowtricity'. [Online]. Available: https://www.nowtricity.com/country/spain/

[66] 'Carbon Footprint of a Laptop vs MacBook vs Desktop Computer vs iPhone'. [Online]. Available: https://8billiontrees.com/carbon-offsets-credits/carbon-footprint-of-a-laptop/

[67] 'Carbon Labeling and Measuring Carbon Impact of Products'. [Online]. Available: https://www.logitech.com/en-eu/sustainability/carbon-labeling-measuring.html

[68] 'U.S. PC gaming session length 2022', Statista. [Online]. Available: https://www.statista.com/statistics/1339296/us-pc-gaming-session-length/

[69] 'Calculation of CO2 | Encon'. [Online]. Available: https://www.encon.eu/en/calculation-co2

9. Annexes

All the final scripts developed and the final build of the game can be found in the Github project web page following the link from the Links section.