INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA

HoopZ - A 3D First Person Multiplayer Basketball Game

MIGUEL ÂNGELO MIRANDA PINTO
Outubro de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

HoopZ

A 3D First Person Multiplayer Basketball Game

Miguel Ângelo Miranda Pinto

A dissertation for obtaining the degree of Master of Science in Computer Engineering, Specialization Area of Graphics Systems and Multimedia

Supervisor: Paula Maria Escudeiro

Porto, October 2022

Para a pequenina Sofia. Que todas as surpresas sejam tão boas como tu

Resumo

Os videojogos apresentam-se no século XXI como a principal indústria de entretenimento a nível mundial, demonstrando ano após ano a sua resiliência e capacidade de se auto-reinventar, ao mesmo tempo que se tornam cada vez mais familiares e acessíveis, e o seu desenvolvimento mais universalmente aberto e disponível ao público.

Esta acessibilidade e facilidade de desenvolvimento leva no entanto a que restem cada vez menos ideias verdadeiramente originais e cada vez menos terra inexplorada no que toca a mecânicas de jogo .

O objetivo desta dissertação passa então por tentar cumprir essa mesma ambição, a de desenvolver um jogo, segundo as práticas de boa programação e arquitetura de sistemas vigentes na indústria, que possua intrinsecamente características, quer mecânicas quer temáticas, que permitam ao mesmo distinguir-se dos demais de forma clara, assim como descrever como as mesmas foram atingidas em cada fase do seu desenvolvimento.

Recorrendo ao uso do Unreal Engine 5, será desenvolvido um videojogo multijogador de basquetebol na primeira pessoa, o primeiro do seu tipo a esta escala, com foco na simulação realística de física da bola, jogabilidade rápida e simples de aprender, num ambiente competitivo 3D com ênfase na criatividade do jogador e na imersão do mesmo no mundo de jogo.

Palavras-chave: Desenvolvimento de jogos, Basquetebol, multi-jogador, Primeira pessoa, Unreal Engine 5

Abstract

Videogames present themselves in the XXI century as the single biggest player in the entertainment industry worldwide, showcasing year after year their resilience and ability to reinvent themselves to go along with the development of new technology, as well as becoming increasingly familiar and accessible, and its development more open and available to the public.

However, this same ease of development leads to there being a continuous shortage of truly original ideas and unexplored territory when it comes to game mechanics.

The objective of this dissertation is to then attempt to achieve that very ambition: to develop a videogame, using the best programming and system architecture practices routinely employed in the video game industry, that possesses intrinsic characteristics and features, both mechanic and thematic, that allow it to clearly distinguish itself from its peers, as well as document and describe each step of its development.

Using the tools provided by Unreal Engine 5, a multiplayer basketball first person video game will be developed, the first of its type at this scale, with a focus on the realistic simulation of ball physics, quick and easy to learn gameplay, inside of a competitive 3D environment with emphasis on player creativity and immersion.

Keywords: Game development, multiplayer , Basketball, First Person, Unreal Engine 5

Agradecimentos

Gostaria de usar esta secção para agradecer a todos os muitos e muitíssimo bons amigos que fiz ao longo de todo o meu percurso académico. As memórias não se vão, ficam-se. As coisas boas não se falam, sentem-se.

De seguida, os meus agradecimentos ao ISEP enquanto instituição e ao Departamento de Engenharia Informatíca e todos os seus docentes, especialmente aqueles que fomentaram em mim o gosto pela aprendizagem e a vontade de me superar a mim mesmo em tudo o que faço. Ensinar é por vezes um trabalho ingrato. É preciso dar valor a quem sabe ensinar, por vezes a quem não sabe ser ensinado.

Um agradecimento em específico à professora Paula Maria Escudeiro, pela disponibilidade que sempre me mostrou quando necessitei, e pela orientação ao longo deste projeto.

A titulo mais pessoal, um carinho especial aos meus colegas de banda, que são como meus irmãos de sangue e sem os quais já não conseguia viver. A música persegue-nos e nós devolvemos o favor, até quando custa sair de casa porque chove lá fora. Se calhar, especialmente quando chove.

Finalmente, e como não poderia deixar de ser, um agradecimento enorme à minha família, de onde tudo vem e onde tudo acaba. À minha Mãe Cristina, ao meu Pai Sérgio, à minha Irmã Cristiana e à Pepper, a melhor cadela do mundo, um muito obrigado do fundo do meu coração por tudo.

Especialmente o que nem precisa de ser dito.

Table of Contents

List of Images

Glossary

Term	Meaning
AAA	A way to refer to game development companies that possess the biggest funding and can afford to produce the highest quality titles.
Actor	Entity that exists inside a game level.
AI	Artificial Intelligence.
Clipping	Visual bug that happens when two meshes intersect when they shouldn't.
Codebase	Collection of source code that is responsible for the functioning of a given piece of software.
CPP/cpp	Name given to the source file of a C++ class, where the implementations of the code being run are written
Desync	Desynchronization. Visible correction of data when its representation in a client strays too far from its representation in the server.
Dolly	Camera movement where a camera is mounted on a dolly, and moves towards, away from, or alongside an actor.
E-Sports	Electronic sports. Branch of sports that involves the usage of the digital medium.
FPS	First Person Shooter- Type of game that consists of some sort of shooting mechanic, portrayed through a First Person perspective. Frames per Second – A way to evaluate how efficient a game's performance is.
Frame	Static image, supplied to be visualized by the user through a graphical hardware component.
Framerate	Rate at which frames are supplied to the user. Evaluates how well a game is transmitting its activity to the player.
GAAS	Games as a Service – Type of revenue model that encourages the use of microtransactions.

Gamification	Application of game-design elements and game principles in non-game contexts.
Grind	Repetition of an activity, normally for some sort of in-game reward.
IP	Intellectual Property.
Indie	Independent. A way to refer to video game companies that are smaller than AAA companies, and normally produce and develop their own games.
Latency	Measure of the time that a data packet takes to travel the network from a server to a client, or vice-versa.
Levels	Game levels. Using a theatre analogy, they represent a new scene of a play, with new actors and environments.
LMB	Abbreviation for Left Mouse Button.
Mesh	Polygonal wireframe that constitutes a 3D model.
Metascore	Score, employed by the Metacritic website, obtained through the aggregation of the review scores for any given game,
Microtransaction	Type of in-game transaction where the player can use fiat money to purchase game content, directly from an in-game store or marketplace
Multiplayer	Game mode that is characterized by the presence of several players inside of the same game session.
Pay-to-win / P2W	Term used to describe video games that allow the spending of fiat money to obtain some sort of gameplay benefit over other players.
Play testers	People that partake in the testing of a game, providing feedback of their experience to the developer
Proof of Concept	A realization of a certain method or idea in order to demonstrate its feasibility, or a demonstration in principle with the aim of verifying that a concept or theory has practical potential
RMB	Abbreviation for Right Mouse Button

Singleplayer	Game mode that is characterized by the presence of only one player in a game session.
SDK	Software Development Kit. Set of tools that aid in the development of a given piece of software.
Thread safe	Programming concept that is used to describe pieces of code that guarantee the safety of their manipulation of data structures that may be shared with other threads.
Trucking	Trucking is a type of tracking shot in which the entire camera moves left or right along a track.
Unreal Engine 5/ UE5 / Unreal	Game engine, developed by Epic Games. Main tool used in developing the project
UML	Unified Modeling Language. Standard language for structuring and portraying software projects.
Widget	Type of object that is used to display UI elements and data in Unreal Engine 5

1 Introduction

Video games and video game development studios are a cultural phenomenon of almost indescribable proportions and unbelievable reach.

They allow people to play an active part in the process of appreciating art, more so than any other art form that exists today. They place the audience behind the wheel of the action and ask, not tell, where to go next. They appeal to a very primal part of the human brain and are capable of retaining our attention and perception in virtually infinite ways.

From the humble beginnings of the industry in the early 50's, where some very basic graphical interfaces allowed humans to play against AIs in digital versions of traditional tabletop games, to their explosion in popularity in the 80s and 90s with the coming of affordable home consoles such as the Sega Saturn, the NES and the PS1 and PS2, followed by the proliferation of network games in the 2000s brought forth by the introduction of the world wide web, before finally arriving at the current day landscape, where video games are the single most lucrative entertainment industry in the world, profiting more than the music and movie industry combined , have entered into the realm of sports with the advent and popularization of ***e-sports*** and have even influenced traditional business ventures and the way client-product interaction is modelled through the use of ***gamification*** mechanics.

This amazing journey has only been possible through the continuous iteration of existing game concepts and mechanics, to find out what works and what doesn't, and a very deliberate focus on creativity and lateral-thinking in order to be able to discover new genres and different approaches to existing ones.

This document is an in-depth look into how such research is conducted in an academic setting, and how a possible viable outcome of such research might be developed.

The current chapter presents the problem, context, motivation, method, contribution, and this document's structure.

1.1 Problem

Sports and their practice worldwide are of the major cultural characteristics of the human species. It has been present, in some way or another, throughout the development and history of mankind, and records of its existence can be found dating back millennia.

It is, also, one of the most immediately recognizable use cases for digital media. It is passively consumed, in its many, many forms, by billions of people around the world, and this number is only ever going to grow.

This incredible demand for sports has led to the creation of a plethora of video game franchises, whose main objective is emulating the real thing with none of the physical effort and years of painful dedication and abstinence that are required to be a successful professional athlete.

These franchises normally release a new title each year, in order to keep up with the everchanging rosters of the professional teams that are represented within them, whose licensing costs are astronomical and often exclusive, meaning no other games can represent them, even if their developers were willing to license them. Because of this, the development cycle of these titles is prohibitively brief and improvements are mostly incremental, which can lead to a creative staleness in the genre, which is further exacerbated by the lack of competition that each franchise has within its respective sport, which is due in no small part to the aforementioned licensing deals.

As such, there is a dire and very palpable thirst for games that are able to capture the inherent excitement of sports, while still feeling fresh and innovative in their own right. This causes a very sizeable chunk of the player base, some of which only play sports videogames, to feel that the diminishing returns they get on the franchises they buy annually may not be worth it anymore.

This creates a vacuum in the video games ecosystem, one that can either be filled by a title that fulfills the expectations of the fans in regard to innovation, or that can cause them to stop investing in sports games altogether, which in turn can lead the companies developing them to also deinvest in the market, leaving everyone in it all the worse for it.

This scenario is both a problem and an opportunity, seeing as it ultimately comes down to a need that isn’t being satisfied properly, mostly due to a lack of competition and the existence of what is effectively a monopoly in the development and distribution of sports games.

This document aims to describe a possible approach showing how to supply the aforementioned vacuum, insofar as it pertains to the basketball sports videogame genre.

1.2 Context

In the particular case of basketball, only one major game franchise exists, in the form of NBA 2K games, developed by Visual Concepts and published by 2K Sports, with the most recent game in the franchise at the time of writing being NBA 2K22, released for consoles and PC on the 10th of September of 2021.

Players have long voiced their dissatisfaction when it comes to the franchise and its inability to properly renew itself with each passing year. While visually the game can, at times, be virtually indistinguishable from its real-life counterpart, with extensive work being devoted to detailed in-game models of the players likenesses and the arenas in which the game takes place, and some new or reworked gameplay features can be introduced, the novelty effect soon wears off, and players often feel like they can't distinguish, in any meaningful way, the newer iteration of the franchise from the one that came the year before.

Worse still, the game is designed around the ***GAAS*** model, providing the user with the possibility of spending even more money through microtransactions to purchase an in-game currency, aptly named ***Virtual Currency*** or ***VC***, that allows players to effectively bootstrap their experience and artificially progress through the game without actually playing the game. This way, the developers employ predatory practices to pad out the early game experience so that players can either choose to ***grind VC*** through in-game activities in order to access the content that they want, or simply pay for it and not have to go through that laborious experience.

While this approach has been an enormous success from a commercial perspective for 2K Sports, growing criticism has been placed on its predatory implementation and how it devalues every single copy being sold to the players. Gone are the days where buying a full-priced NBA 2K videogame would entitle the customer to all of the content within it. Instead, the enjoyment of playing it is now irreparably connected to the game's progression system, effectively making it so significant parts of the game are walled off to the player, and only accessible via the devotion of a significant amount of in-game time, or by purchasing it through the game's ***microtransactions***. This further fragments the playerbase, and inevitably leads to a more unfair experience for everyone. The game is no longer fully skill-based, as players from a richer economic background can now feasibly pay to have a better performance in-game. This is one of the indistinguishable hallmarks of ***pay-to-win***, or ***p2w***, games.

1.3 Motivation

The main objective of this dissertation is the development of a first-person multiplayer basketball video game that delivers fast paced innovative gameplay, with a focus on modularity and customization, allowing the players to cater their experience to their particular needs and wants. A focus on physics-based game mechanics must ensure that the core tenets of a basketball game are easily and immediately identifiable to new players, but a full-fledged simulation of all aspects of the game is both outside the scope of this project and also antithetical to the more carefree, arcade gameplay loop that is intended and idealized.

In addition to this, the research and development of a functional, quick and flexible matchmaking system, with an integrated server browser and voice-chat, is also contemplated and described in detail.

Finally, the analysis and implementation of a brief training mode, with simple and functional AI bots designed to guide the player into performing various facets of the practice of basketball is present as well.

There is, at the time of writing, no game with these particular characteristics released in the market, or in active public development. There are also few first-person games based on sports that ever moved beyond the prototype stage, so there isn't a valid template for what such a game should be, or what the correct way of approaching the particular problem that the first-person perspective places in the realm of game design.

This particular set of challenges and the promise of the discovery of new, unbroken ground in the realm of video game history is, thus, the main motivation behind the undertaking of this project.

1.4 Method

In this chapter, the method by which the project aims to deliver the necessities of the players is further explained and detailed.

The core necessities that have been identified, and their main solutions are as follows:

- Players want an innovative gameplay experience, unlike any other basketball game they have ever played.
    - The first-person perspective effectively guarantees, by its very nature, a new take on basketball games.

- Players want their time to be valued, and don't want to have their progress gated behind microtransactions.
    - All unlockable data tied to the progression system will be strictly cosmetic and won't in any way impact the gameplay experience.
- Players want a responsive and latency-free multiplayer experience.
    - A focus on optimizing the data being sent and received through the internet, allied to the game's arcade look and feel, should guarantee an adequate experience under normal circumstances.

In order to successfully and properly design and implement a multiplayer basketball game, a healthy amount of research and iteration is necessary. As such, each main gameplay system needs to be developed in a modular, airtight way and must be able to be tested in its own game map.

However, all the systems need to work harmoniously in tandem, to ensure the final experience is cohesive and pleasant, so regular on-going and all-encompassing gameplay sessions must be performed and reviewed, both through the use of human play testers, whose input and feedback will be recorded and analyzed accordingly, as well as through the use of AI bots with which deterministic unit and integration tests are possible.

A focus on accurate and realistic rigid body physics is also a cornerstone of the development of the project, so existing academic research on physical interactions between the basketball and the various other actors that are present inside the basketball pitch, or any objects that can be their analogous equivalents, must also be reviewed and serve as the inspiration for the physics in the project. As basketball games are televised worldwide, a vast video archive of basketball matches exists and is readily available for perusing, which allows the constant comparison between real life and the digital behavior of the basketball.

Beneath all of this, the network systems must allow for ***latency***-free gameplay and accurate distributed communication between servers and the clients connected to them, with minimal packet loss. Since dedicated servers are expensive, and the maximum number of players is designed to be small, a peer-to-peer network architecture is going to be the chosen approach, with one player acting as an authoritative server and the other players acting as clients.

The development of this solution is going to heavily borrow the battle-tested framework freely provided by Epic Games and their game engine ***Unreal Engine 5***, and the code for it is mostly going to be written in ***C++***, although some features may use ***UE5***'s inbuilt visual scripting language, Blueprints.

1.5 Contribution

The development of this project contributes to further developing the first-person perspective and its use cases when employed in 3D video games. The research executed during this project is focused specifically on sports games, and the first-person sub-genre within them which is still fairly underexplored. As such, existing techniques that are employed in resolving any given game design obstacles that present themselves can be once again reasserted in their efficacy, while new techniques are discovered and validated with an adequate use case, one that they can be relied upon to be the solution to in the future.

The project hopes to be a meaningful step in the right direction when it comes to breaking apart the monopoly existing today in the basketball videogames sub-genre, and provides a framework by which other projects in the area might guide themselves in the future.

Additionally, this project is also a testament to the amount of work that can be feasibly developed by a one-person team, by taking advantage of the available open-source and freely distributed tools and content in this day and age. The indie game industry in Portugal is unfortunately still stuck in the 80s, and retro 16-bit side scrolling games are a dime a dozen, mostly because indie developers wrongly assume that 3D videogames are too difficult to be developed solo.

Finally, this project hopes to showcase to other students, either prospective or otherwise, that videogame projects, when properly enunciated and conveyed, can be as valid as any other when undertaken as a master's thesis subject. Realtime 3D video games are genuinely amongst the hardest endeavors one can attempt, and ones that implement multiplayer support doubly so, even though mainstream academic opinion can often be oblivious to this fact.

1.6 Document Structure

The rest of this dissertation is structured as follows. Chapter 2 consists of the state of the art, presenting a broad view of how the themes presented in this chapter are being addressed in the state of the art. Chapter 3 contains the value analysis, which explains the value created by this work. Chapter 4 contains the Analysis of the project, which details how the features of the game were conceptualized and designed. Chapter 5 consists of the implementation of the features of the project, and how the final proof of concept was achieved. Chapter 6 shows the Evaluation model, through which the efficacy of the implementation at achieving the project's goals is measured. Chapter 7 contains the conclusions that were achieved by developing the project. At the end of the document, a list of the references is provided.

2 State of the Art

This chapter provides an in-depth view, as of the time of writing, of both the academic world and the video games industry, regarding the main themes and interests of this dissertation. We'll explore existing solutions and implementations to designing problems that need answers if a satisfactory experience is to be delivered to the end user while remaining true to the original vision behind the project.

The main themes that need approaching are the following:

- Digital Basketball video games
- Full body immersion in video games with a First Person perspective
- Real-time aim assist systems.
- Real-time projectile path prediction systems.
- Client-side prediction and network-shared synchronicity systems.
- Scalable and filter-based matchmaking systems for multiplayer video games.

In order to accurately compare the systems that are the focus of this project, other titles and their implementations of these systems will be talked about in depth. The choice of these titles, in themes of which there may be an abundance of material available, will take into account reviews both by professional media and the final player audience alike.

However, reviews evaluate the global status of a game, and are not the perfect measure of specific gameplay features, so examples may be given of titles that employ very good implementations of specific systems, while not being themselves what can be called objectively good games.

2.1 Digital Basketball video games

2.1.1 NBA 2K22

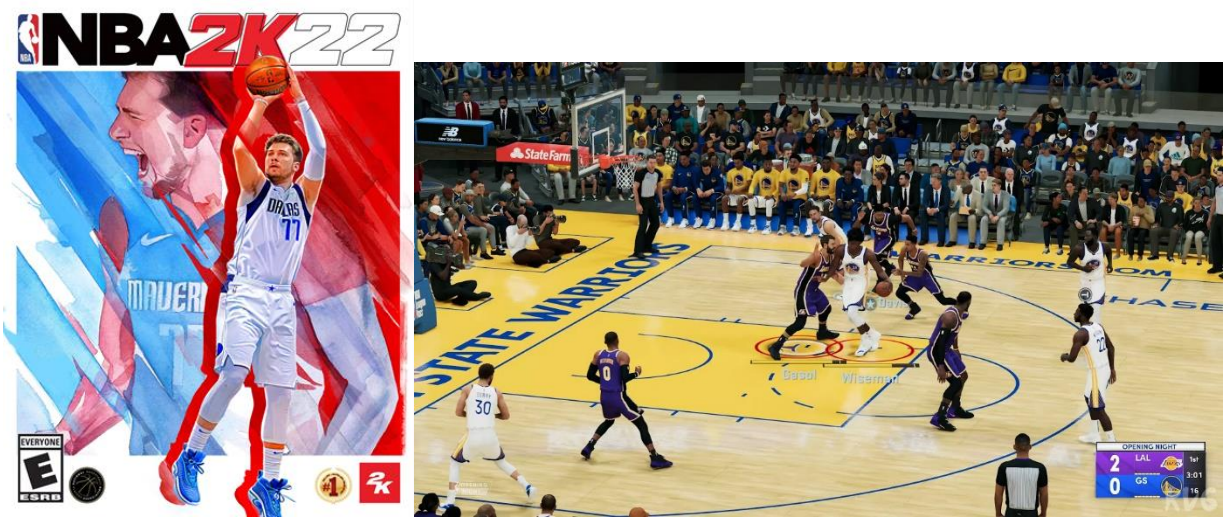


Figure 1 – NBA 2K22 cover art (left). NBA 2K22 gameplay (right)

The NBA 2K series consists of twenty-three primary installments and several spinoff-style titles. All of the games in the series have been developed by California-based video game developer Visual Concepts.

The latest iteration of the series, NBA 2K22, has been released on the 10th of September of 2021. Despite having a lukewarm reception by the video game press, with a 76% rating (Metacritic, 2022a), and a bad reception by the target audience, with a 3 out of 10 rating, the game has been deemed a success financially.

Overall, the game was praised by critics for its photorealistic graphics, especially in regards to player likenesses and arena venues resemblance , immersive soundscape and eclectic soundtrack, reactive in-game commentary and refined animations, as well as an unbelievably accurate simulation of the physics of basketball.

However, the player base showed a fairly vocal discontent in regard to the gameplay aspect, citing concerns that the game played too much like previous iterations and that not enough innovation was present to justify buying a new title while the previous one was still available.

2.1.2 NBA Playgrounds 2

Figure 2 – NBA 2K Playgrounds 2 cover art (left). NBA 2K Playgrounds 2 gameplay (right)

Released on the 16th of October of 2018, developed by Saber Interactive and published by 2K Sports, NBA Playgrounds 2 is an arcadey take on the traditional NBA 2K formula, designed to appeal to players that don't want a rigorous simulation of the sport but instead a carefree, fun way to experience basketball in a setting that allows for some creative license not bound by the laws of physics.

Real-life players and their teams are still present, but their likenesses and features have been exaggerated to reinforce the cartoony depiction of the sport. In addition to this, the players can perform wildly amusing maneuvers, such as front flips and back flips to score points in creative and unexpected ways.

Overall, the game was praised by critics for its over-the-top, simple and fun gameplay and cartoonish, cell-shaded graphics, as well as for its success in repackaging the traditional NBA 2K experience with a freshly placed coat of paint, although criticizing its excessive focus on micro-transactions and lack of game modes to offer a longer play time. The game has a ***Metascore*** of 73 out of 100 and a user score of 5.1 (Metacritic, 2022b)

2.1.3 Grapple Hoops

Figure 3 – Grapple Hoops cover art (left). Grapple Hoops gameplay (right)

Grapple Hoops was developed by Andreas Georgiou and released on Steam on the 18th of October, 2021. Of all the titles mentioned in this chapter, it's the closest mechanically to what the project in this dissertation attempts to achieve, although thematically quite different.

It is an FPS game where the player must traverse a slew of levels populated by AI enemies to overcome, while keeping control of a basketball that must be dunked or thrown into a hoop in order to successfully complete the level. The player must use a grappling hook and various free-running techniques in order to complete the level in the shortest time possible.

While clearly made on a lower budget compared to the rest of the basketball games displayed, and without access to a multi-faceted development team with different departments and specializations, it is without a doubt the most original one of the lot, due to its use of the First Person perspective to convey the feeling of throwing a basketball into a hoop in a way that traditional basketball games can't replicate.

No information regarding the sales of the game has been as of yet disclosed, but user reviews on Steam have been mostly positive, with many players lauding the innovative gameplay and general goofiness of the game, while placing some concerns regarding the overall unfinished and unpolished state of the title.

2.2 Full body immersion in video games with a First Person perspective

Supporting full body awareness on a First Person game comes with a particular set of challenges that may not be immediately apparent, but in return it allows for a level of immersion that is impossible to achieve in normal, old-school approaches to designing the viewport of the player.

Traditional player characters in the first person perspective are normally setup as portrayed in the images below.

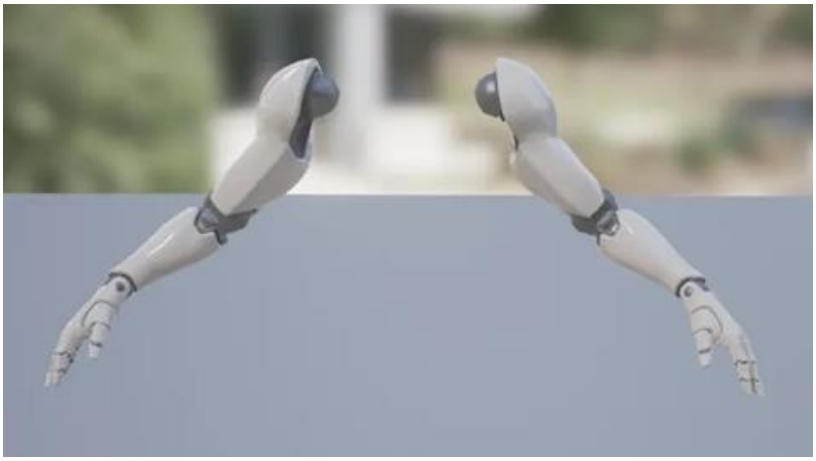


Figure 4 – Floating arms FP mesh in the UE5 Editor

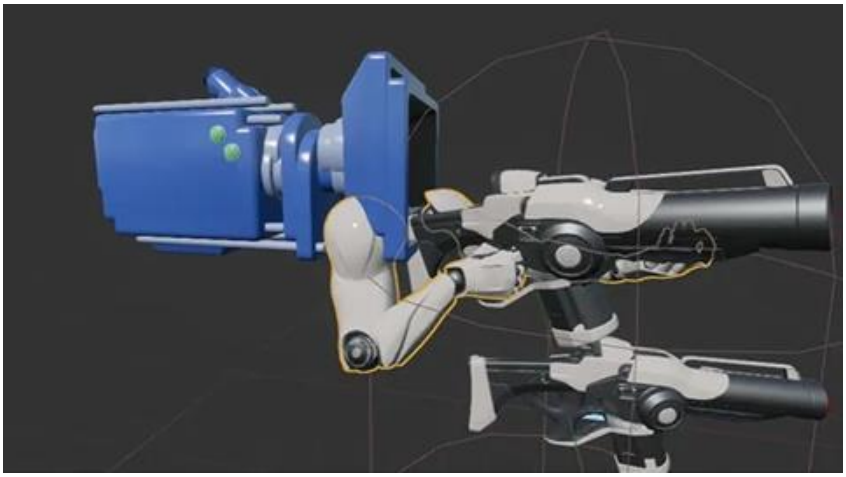


Figure 5 – Template FP character actor in UE5 Editor

This approach consists in modelling a skeletal mesh of a pair of arms, visible only to the player, which is then bound to the player's collision capsule and then positioned accordingly to the position of the camera through trial and error to find an adequate placement in which the player perspective doesn't clip through the arms, allowing the player to see inside of the mesh.

The player is then, literally, a floating pair of arms, and any shadows cast by the player or reflections where the player may appear make this fact painfully apparent.

The main advantage of this system is that it's quite easy and quick to implement and understand, so much so that it has become the de-facto standard. Virtually all existing, freely distributed game engines that allow for 3D graphics employ this implementation as part of their FPS template, and UE4 is no exception.

However, it is heavily prone to inefficiency, because it forces an animator to have to animate two different sets of animations for the player character, one for the first-person perspective that the player sees and another one for what other players, either friend or foe, actually see when they look at the character. It is also fairly unrealistic since the player can not see the player character's feet when looking straight down, again reinforcing the notion that the player is a floating pair of arms, which obviously is detrimental to immersion.

The solution to this conundrum is then to be found by undertaking the full body awareness approach, the basis of which can be quickly understood by the image below.

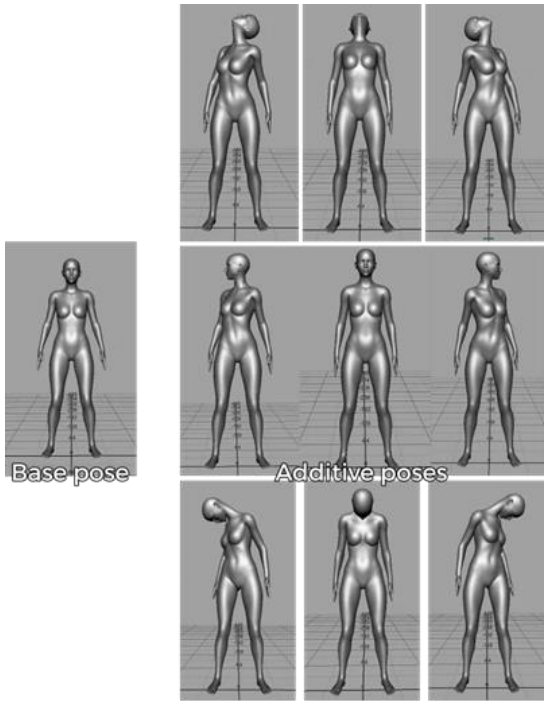


Figure 6 – Base animation pose (left). 8 different orientation poses(right)

Using this approach, the full body of the player character is modelled, the player camera is placed in the location where the eyes of the 3d mesh are, and the movement of the camera, triggered by player

input, is tracked to the resulting skeletal mesh through the use of additive poses that are interpolated between one another according to the camera's orientation and rotation.

These additive poses are so called because they "add" their data on top of any other animation that is previously being played, which means they can be used interchangeably whether the player is running, walking, or jumping in the game.

The advantages of this method are immediately observable. The player character is no longer a floating pair of arms, and now longer breaks immersion when casting shadows or appearing on reflections. In addition to this, only one set of animations needs to be created, possibly using motion capture to obtain the best results. These do still need to be iteratively tweaked to find the perfect balance between immersion and accessibility, because animations that look good in a third-person perspective might not look as good or be as readable when viewed through a first-person perspective.

Additional steps can be taken to perfect this system, such as using the collision data of the camera with the surrounding environment to adapt the animation so that the player is never capable of clipping through the walls, or even creating idle breathing animations for the player character, so that camera bob effects don't have to be faked through code.

2.3 Realtime aim assist systems

The most important part of any basketball game that attempts to achieve even a modicum of success is also the most important part of the real game of basketball, the single most invaluable decision factor determining who actually wins the game. That part is shooting the basketball and, consequentially, the subsequent points scored from doing so. Any basketball videogame that fails to properly address and prioritize this feature is doomed from its inception.

The way to go about this in traditional basketball games, where the camera is either ***trucking*** courtside or performing a ***dolly*** movement up and down the field from an elevated position, was to define as a basic gameplay axiom that it is *always* in the player's best interest to be aiming at the hoop when the shooting gameplay action is triggered.

This way, the player only has to worry about the timing of the release of the shot, and rest assured that the game is smart enough to choose the trajectory of the ball accordingly, and let physics take care of the rest. Aiming is sacrificed in the name of immediacy and straightforwardness.

However, this approach doesn't survive the switch to a First Person Perspective, becoming too simple and heavy-handed, and failing to account for the newly added freedom of motion of the camera, orientation-wise. No gameplay system feels skill-based if the system itself is doing most of the heavy lifting, and not the player. That being said, particular care must be taken to resist the erroneous urge to place all of the responsibility in the player's hands, resorting to no assists of any kind. No peripheral

input device currently existing can be used to do, in a time-sensitive realtime fashion, what the human hand can do instinctively at a whim.

Considering all of this, a new solution needs to be posited. One that balances the need for accessibility and readability that befits a game with a focus on arcade gameplay, while also allowing the player the freedom and the leeway to experiment in a meaningful way.

Borrowing heavily from the work and research of cinema and the well-established library of types of camera movement found in movie-making, a suitable solution has been employed successfully in other titles to resolve similar conundrums, using what is called an arc-shot technique.

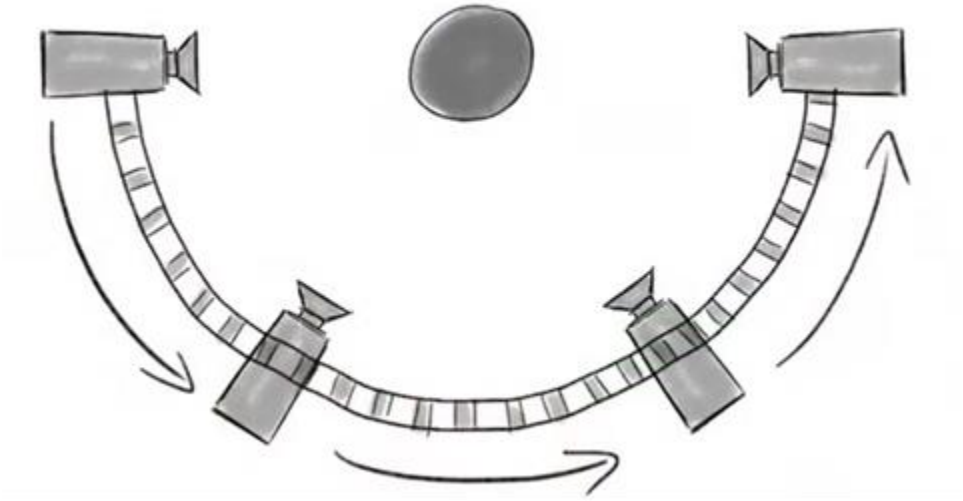


Figure 7 – Arc shot schematic

In this sort of shot, the camera is centered around a given object, and rotates around it describing an arc that guarantees that the orientation of the camera is always directed towards the object at the center, and is thus an adequate solution to guarantee the player is always facing the hoop when aiming the basketball, while still allowing the player the freedom of choosing where exactly in the hoop the ball should go to.

2.3.1. Dark Souls 3

Figure 8 – Dark Souls 3 lock-on mechanic(left). Dark Souls 3 cover art (right)

Developed by Tokyo-based studio From Software and published by Sony Computer Entertainment, Dark Souls 3 was released for the PS4 on the 24th of March, 2016.

The game is a third-person action-adventure experience with a focus on hardcore, punishing combat and exploration. The Dark Souls franchise is most well-known for popularizing the use of a progression system that makes use of certain checkpoints, referred to as bonfires, that allow the player to save progress, replenish health and spend souls, the equivalent of in-game experience, to improve their character, while also reviving all of the enemies in the area and serving as the place of respawning should the player die, bringing a new tactical approach to the simple act of progressing the game.

It was a critical and commercial success, selling over 10 million copies as of May 2020 (Gamestop, 2022a), and amassing a ***Metascore*** of 89 out of 100 and a user score of 8.9 out of 10 on Metacritic, with most of the reviews lauding the game's fluid, hardcore gameplay and beautiful art direction and soundtrack (Metacritic, 2022b).

In regards to its implementation of a real time aim assist system, its main use comes in the form of the game's lock-on mechanic, visible in Figure 8. This mechanic allows the player to focus the camera on a single enemy, and freely move around it without having to worry about recentering the camera again, much like the arc-shot schematic present int Figure 7. In this way, the player only has to worry about attacking, dodging or parrying, and can leave the camera handling aspect to the game. The game uses a small white dot placed on the center of mass of an object, visible on the chest of the enemy in Figure 8, to communicate to the player that the lock-on mechanic is enabled and also to provide a focal point where the player's eyes can latch on to when the action gets too chaotic, which is an effective quality of life feature to improve the system overall.

2.4 Realtime projectile path prediction systems

Working in direct conjunction with the real-time aim assist systems described in the previous chapter, real-time projectile path prediction systems need to be employed so that the player can know, in a readable and time sensible way, where the basketball would land, should a shooting action occur.

Being that the ball movement is intrinsically tied to the physics simulation, that means that the basic physics formula for a free-falling object can be used, with a bit of vector math, to preemptively calculate the trajectory of the basketball, when thrown from a given location and with a given force vector.

A 2D formulation of this particular situation is portrayed below.

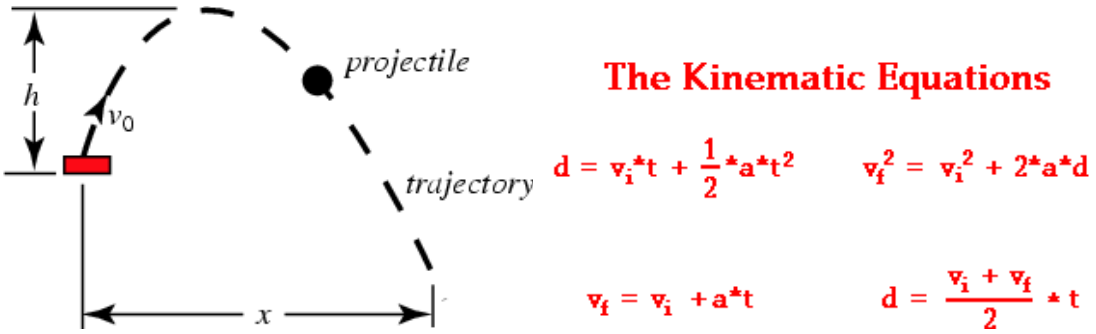


Figure 9 –Projectile trajectory schematic (left). Kinematic equations of movement (right)

Using the kinematic equations of movement as a basis, the location of a projectile can be accurately predicted in vector space. Using these calculations, an UI overlay can display the trajectory to the player.

Several other titles have already found working implementations of this system. One of the best ones is the subject of our brief analysis in the next sub-chapter.

2.4.1 Uncharted: The Lost Legacy

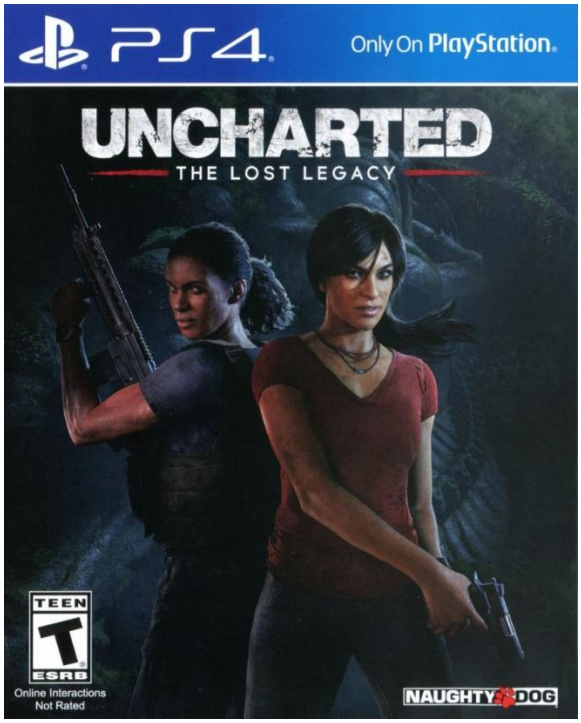


Figure 10 – Uncharted: The Lost Legacy cover

Developed by California-based studio Naughty Dog and published by Sony Computer Entertainment, Uncharted: The Lost Legacy was released for the PS4 on the 22nd of August, 2017, as a stand-alone expansion to Uncharted 4: A Thief's End, also developed by Naughty Dog and published by SCE.

The game is a third-person action adventure experience with a focus on narrative and exploration, and was quite the financial and critical success, boasting a Metascore of 84 out of a 100 and a user score of 8.0 out of 10 in Metacritic (Metacritic, 2022c).

Its main usage of a realtime projectile path prediction system is in the use of grenades and other throwable objects, allowing the player to get an accurate read of the arc trajectory of the object before throwing it, as seen below in Figure 11.

Figure 11 – Grenade arc trajectory mechanic

As we can see, a ***Bezier curve*** with an arrow at the end, rendered in white in order to contrast against every type of surface and help player readability, is employed to display the prospective trajectory the grenade would take if it was to be released, and a circle overlay is placed unto the surrounding environment, both to further boost player readability, but also to display the area of effect of the explosive.

2.5 Client-side prediction and network-shared synchronicity systems

The very nature of multiplayer games implies a rethinking of most of the axioms and presumptions that are pervasive throughout most single-player experiences. First off, a much stricter emphasis needs to be placed on ensuring that all players enjoy the game in a level playing field, hardware components and their myriad of different combinations non-withstanding. Players with inferior rigs needs to be able to compete with players with superior rigs. This issue is especially apparent in PCs, because console hardware is mostly standardized and, as such, cross-play/cross-platform support between consoles is more easily containable and edge cases are fewer, allowing developers to prevent against them more effectively.

Additionally, we also have to account for interruptions or delays in the network stream of data that forms the very basis of communication between players, that can lead to differing levels of desynchronization and ultimately disconnection. These are not only unavoidable but also very prone to being explored by bad actors in the system that may intend to cheat, either for their own benefit or for the hindrance of the experience of everyone else. Cheater detection and prevention is an extremely complex field of study that consistently proves too difficult even for established, AAA game studios to tackle effectively, so it is outside of the scope of the current document.

In order to address these issues various techniques could and should be adopted. Some of them are game specific, such as the particular issue of handling a high number of players across a very big level

that is very characteristic of the ***Battle Royale*** genre, while others exist at a more fundamental level, such as how to properly address latency in a way that doesn't break player immersion but also maintains gameplay coherence. In the present chapter, a brief examination of solutions for these problems is supplied.

2.5.1 Client-side prediction

In every sort of networking application, a central entity, known as the ***server*** or ***master***, is tasked with being the arbiter of truth, effectively acting as both the judge and the executioner, maintaning the accuracy of the data and relaying it as it sees fit to other actors in the network, also known as ***clients*** or ***slaves*** so that they can create their own copy of the data for their personal consumption.

In order to maintain data coherence, the relation between the ***server*** and its ***clients*** is mostly strictly authoritative, meaning that only the server can tell the clients what to do, and not the other way around. Clients must, instead, issue requests to alter data to the server, and the server must validate these requests and update the data accordingly, if the requests are deemed appropriate.

This authoritative stance is very good at maintaining and asserting data accuracy, but fails to account for latency in the updates the server sends to the clients. If no other measures are implemented, gameplay will only be fluid and precise for the player acting as the server in a peer-to-peer network architecture. For everyone else, there will be a noticeable lag between actions and their appropriate and expected reactions, since the server first has to validate and accept any input that is attempted, and then still has to propagate the output across the network. This mismatch is illustrated in Figure 12

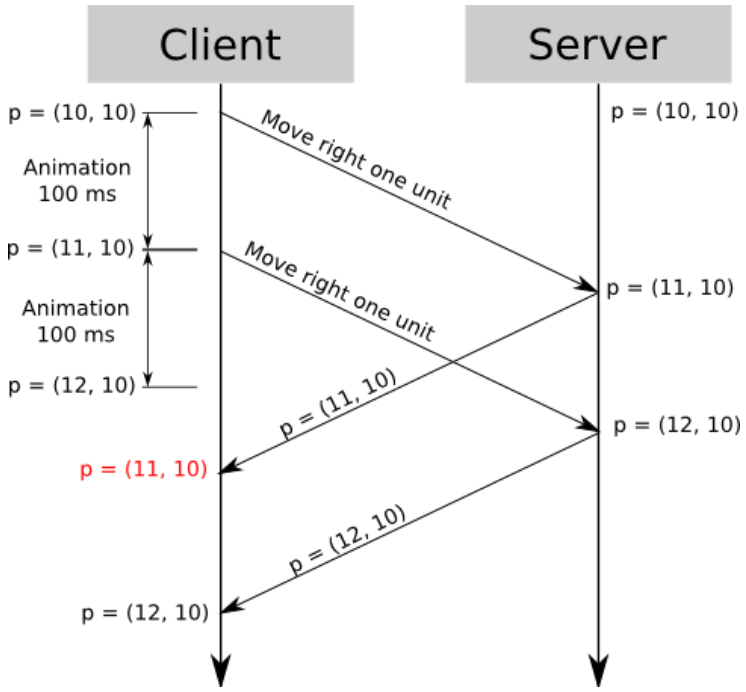


Figure 12 – Simple authoritative server-to-client communication schematic

As we can see, the client is already expecting to be at position p = (12,10) by the time the first message from the server to move to p = (11,10) is received. This leads to visible desync in the position of the player, since clients are effectively running the game in the present, while receiving instructions from the past.

This is where client-side prediction is of extreme importance. Using client-side prediction, the client is allowed to display immediate outputs that result from the player inputs, which are stored alongside a timestamp indicating when they were performed and what output they produced.

Then, upon receiving and applying authoritative server data, also timestamped accordingly, the client retroactively repeats actions undertaken after the timestamp of the latest received server data, interpolating between the output of each stored action to allow for immediate user feedback, while ensuring ***desync*** is kept to a minimum. This process is illustrated in Figure 13.

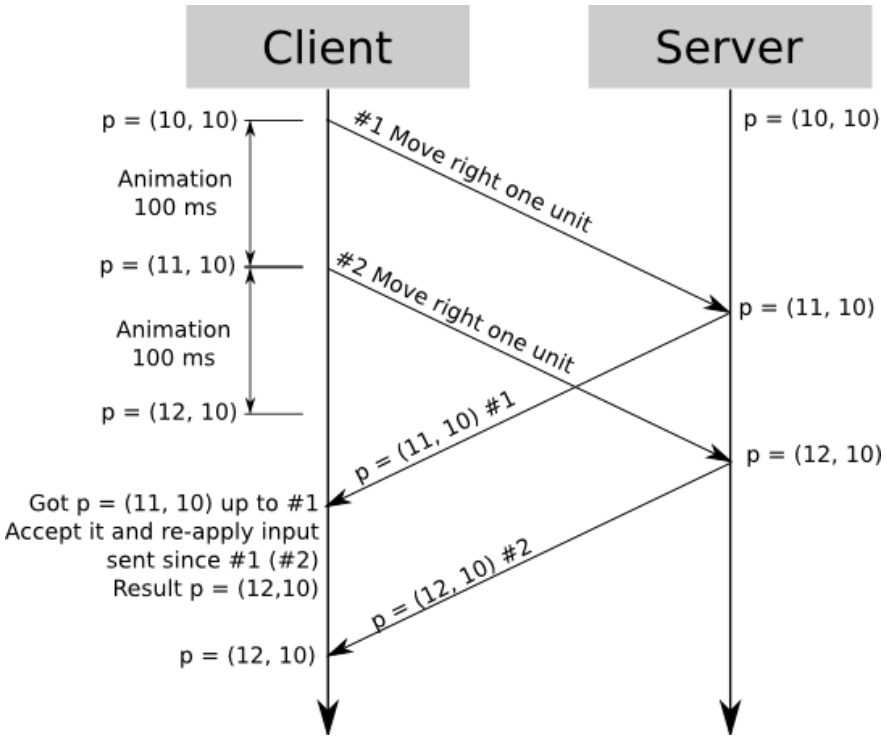


Figure 13 – Server-to-client communication, employing client prediction and server reconciliation

2.5.2 Physics fixed time-step for deterministic physics simulation

We've showcased in the previous chapter how client-side prediction can be used to successfully mitigate network desync issues, but unfortunately, it only solves part of the problem of network synchronicity. In fact, it works quite well when working with deterministic systems, where a given input can reliably be expected to result in a given output, but non-deterministic systems struggle with this notion, by their very nature. One such system is physics simulation.

Most modern videogames employ physics simulation in some sort of way, such as the movement of player characters, projectile simulation or collision handling. It's become such an important part of the functioning of a videogame, that most game engines, custom-built or otherwise, implement physics support by way of a dedicated physics thread, whose only concern is performing real-time physics calculations in a detached manner from logic code, running in the game thread, and graphics code, running in the render thread.

Normally, for single-player games, that's the end of the story as far as physics simulation goes. Game developers know that a dedicated physics thread exists, but they normally don't need to consider how it calculates data, and don't need to interact with it in any way besides using the appropriate engine functions to apply forces to objects in any way they see fit. This is because, no matter the FPS in their system, the physics thread handles calls to it in a proportional way.

However, in multiplayer games, several players must calculate the same data and, in order for there to be no visible ***desync*** issues, arrive at the same outcome in a deterministic manner. That is simply not possible under normal physics systems, where the physics thread is allowed to run as fast as possible, which obviously differs from player to player, depending on their hardware configuration.

Instead, a fixed-step approach must be taken regarding physics calculation, commonly referred to as ***physics sub-stepping.*** Such a system entails that the physics thread is no longer allowed to run rampant and calculate physics as fast as possible, and must instead run at a fixed frame rate. This rate can be tweaked according to the needs of the project. This way, even when the rendering thread is stressed and must compute graphical information in a volume that forces the ***framerate*** to drop, the physics thread can react accordingly, and ensure that the simulation presents realistic and reproducible results that are the same inside every game application in a distributed multiplayer session.

2.6 Scalable and filter-based network matchmaking systems

In order to effectively connect to other players and thus actually take part in a multiplayer experience, some sort of matchmaking system needs to be in place. These are highly dependent on the type of game that's being created.

Games with a higher focus on competitive play are traditionally built on top of a ***skill-based matchmaking system*** or ***SBM***, where players have a gameplay profile that keeps track of their performance across all multiplayer matches they take part in and assigns them a ***rating***, which is then used to group players together based on their ranking when they attempt to find a match. This system is completely automatic, which means players have no way to know who their opponent is beforehand. The main objective of these types of systems is to try to guarantee a fair playing field, so that players do not feel frustrated by playing against other players that are vastly better than them, and to create an adequate challenge that fosters replayability, and does not place them against players who they can defeat easily, without effort.

In addition to this competitive playlist, a casual playlist, that does not use ***skill-based matchmaking***, is often also present, to allow players to simply enjoy the game and not be worried about their ranking or their in-game performance as much.

Other matchmaking systems make use of ***server browsers*** to allow users to choose by themselves directly which server they wish to connect to. These ***server browsers*** are normally deployed alongside a modular network filter system, that allows players to filter the selection of servers that are presented to them. What type of filters to use is up to the developers, but some, such as filters allowing the player to choose what maps should appear in the ***server browser***, or excluding servers that are already full, are quite standard.

Systems of this type normally approach competitive gameplay in a different way, with the creation of servers that are more closely moderated and that possess different sets of rules that are more adequate for competitive players.

Modern videogames and the developer teams responsible for them display a clear preference for ***SBM*** systems, to the point that it has become the de-facto standard in competitive multiplayer games.

In the next chapters, we will analyze some of the most known implementations of both these kinds of systems, providing some additional information about their specific features and use cases.

2.6.1 Counter Strike: Global Offensive

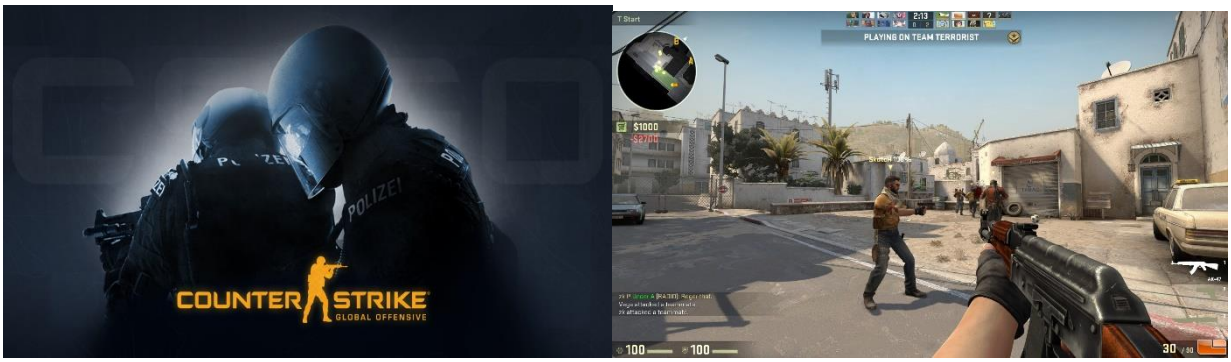


Figure 14 – CS:GO cover art(left). CS:GO gameplay(right)

Developed by Washington-based studios Valve Corporation and Hidden Path Entertainment, and published by Valve Corporation , Counter Strike: Global Offensive was released for all consoles and PC on the 21st of August, 2012, as a sequel to Counter Strike: Source, also by the same developers.

The game was quite the commercial and critical success on release, boasting a ***Metascore*** of 83 out of 100 and a user score of 7.3. The game has since made the transition to fully free to play in November of 2018. (Metacritic, 2022d)

Counter Strike as a franchise is universally considered as being responsible for the popularization of online First Person Shooters, and is one of the main staples of E-Sports competitions to this day, boasting some of the biggest viewer counts on platforms like Twitch and Youtube.

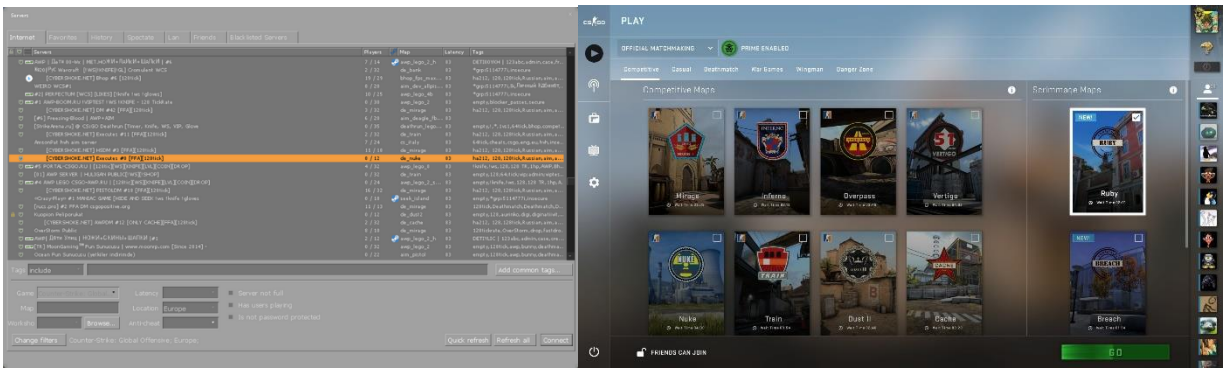


Figure 15 – CS:GO community server browser(left). Normal CS:GO matchmaking menu (right)

Counter Strike: Global Offensive's implementation of a matchmaking system is a mix of the two main systems described in the previous chapter. As is displayed in Figure 16 on the left, a server browser is supplied, through which players can gain access to the community servers in-game. These community servers are hosted by players on their own machine and can as such be the home to player-made maps and mods that wildly transform the gameplay of the game, sometimes so much so that they turn into new game modes that are then converted into full-fledged games of their own.

However, for competitive gameplay, the game uses a ***SBM*** system to place players into multiplayer sessions based on their in-game rank, allowing the player to list a preference for a given map or collection of maps to be taken into account when finding the optimal session. For casual or non-competitive play, the system behaves in the same way, but without taking ***SBM*** into account.

2.6.2 FIFA 22

Figure 16 – FIFA 22 cover art(left). FIFA 22 gameplay(right)

Developed and Published by Electronic Arts, in various development studios working in collaboration around the world, FIFA 22 was released for all consoles and PC on the 1st of October of 2021, as a sequel to FIFA 21, released the prior year.

It was a commercial and critical success, boasting a ***Metascore*** of 78 out of 100, despite having quite a low user score of 2.5 out of 10, an occurrence that is quite prevalent in the sports games presented in this document (Metacritic, 2022e). The fans applauded the game's photorealistic graphics and presentation, as well as the introduction of a new animation system that uses neural networks and a library of thousands of motion-captured animations to improve on the already excellent player-to-player and player-to-ball animation feedback, but heavily criticized the lack of meaningful gameplay improvements and the game's reliance on ***microtransactions*** and ***loot boxes*** to drive the progress in the game's main multiplayer mode, Ultimate Team.

Figure 17 – FIFA 22 Ultimate Team game mode menu.

Within Ultimate Team, players can create a fictional football club and populate its roster with real-life players in the form of player cards, akin to the type trading card games like Magic: The Gathering use to display their creatures and spells. The player can collect these player cards either by completing in-game quests and missions, trading them in the game's inbuilt marketplace, or buying them using a currency that can be earned in-game through gameplay, FIFA Ultimate Team Coins or FUT coins, or a currency that can be bought with real-life money, FIFA Points. The higher the cost of the player cards, the higher their quality, and the higher the team value by extension.

Then, the player can choose to enter competitive online gameplay, where the game will make use of ***SBM*** to match the player with other players of similar rank, but not similar team value.

3 Value Analysis

Value Analysis can be defined as a process of a systematic review that is applied to existing product designs in order to compare the function of the product required by a customer to meet their requirements at the lowest cost consistent with the specified performance and reliability needed (N.Rich, M.Holwegg, 2006). It focuses on the purpose of the product that's being analyzed, as well as the prospective demands of the customer towards it. This way, a higher quality product can be achieved by tailoring the development of the product to the most important needs of the customer, the ones that bring him the most value and prioritizing features accordingly.

In the current chapter, the definition of value and concepts adjacent to it are discussed, especially as they pertain to the video games industry. Afterward, an in-depth view of the value proposition and the opportunities and risks associated with the project. Finally, the proposed business model is detailed, and its main intricacies and characteristics and analyzed

3.1 Value and related concepts

In order to properly understand how to add value to the project being developed, a study into the meaning of value itself and its related concepts, such as perceived value and cost, is necessary beforehand.

Value can be defined as the needs, desires, standards, criteria, beliefs, attitudes, or preferences that an entity possesses (Nicola, et al., 2012). It is directly proportional to the benefit a given customer achieves from a service or product, or the level of customer satisfaction from a given service or product. Some products possess intrinsic value, either derived from the resources they are made of or the resources that have been expended in their creation, while others possess **perceived value.**

Perceived value is presented as a trade-off between benefits and sacrifices perceived by the customer in a supplier's offering (Ulaga & Eggert, 2006). Among other conceptualizations, benefits are conceived as a combination of economic, technical, service, and social benefits (Anderson et al., 1993) or economic, strategic, and behavioral benefits (Wilson and Jantrania, 1995). Sacrifices are sometimes described in monetary terms (Anderson et al., 1993). Other definitions describe sacrifices more broadly as a combination of price and relationship-related costs (Grönroos, 1997). **Perceived value** is, therefore, highly variant from user to user. This inherent variation is even more accentuated in the videogames industry, where emotional attachment and high consumer engagement play an ever-increasing part of the decision-making process.

Cost can be defined as the monetary value of goods and services that producers and consumers purchase, or the measure of the alternative opportunities foregone in the choice of one good or activity over others. It is an important measure in the decision-making process of customers and can easily become the deciding factor in times of economic strife. At the same time, it also plays a great part in the risk assessment procedures of videogame companies when analyzing the undertaking of a new title.

For this project, it is of the utmost importance to keep these concepts in mind when faced with any sort of design decision. Prospective features need to be evaluated according to their **intrinsic** and **perceived value** and adequately balanced out with their **cost**, to avoid the allocation of funds, either temporal or otherwise, on features that cost more than what they achieve.

3.2 Value Proposition

A value proposition can be defined as a group of sentences that accurately describe a product, the value attained from its use by the customer, and which needs of the customer it is intent on satisfying. It is meant to be concise and straightforward, so that any prospective client can evaluate whether to invest in the product or not in a time-efficient way. It must be truthful to what the clear aspirations of the final product are and should avoid overpromising solutions that cannot feasibly be attained using the product.

In order to formulate an accurate value proposition, considerable care should be taken to fully understand what customers consider valuable in a product, what they dislike and appreciate, and what tasks a given product should be expected to fulfill.

3.2.1 Customer Jobs

Customer jobs represent the objectives the customers intend to achieve through the purchase of our product. These can vary from customer to customer and, in the specific case of video games, can even vary across the lifetime of the product, as the user becomes more accustomed to the gameplay loop of the game. In this project, our prospective customer isn't any given corporate entity or company, but the end players themselves, the ones that buy the game through a store, either digital or otherwise. Their main intentions are to:

- Extract entertainment from the product.
- Be exposed to new gameplay elements.
- Share their enjoyment with their friends.
- Share their in-game progress with their friends.
- Be part of a community of players.

3.2.2 Pains

Pains represent the difficulties that the customer faces. They can be understood as the obstacles that are preventing the customer from achieving the objectives they intend to achieve. Through the proper identification of the pains of the customer, valuable information can be derived from what exactly are the problems our solution should tackle.

The main pains of the customer, as pertains to the project, are:

- Staleness of the basketball sports video game genre.
- Over-reliance on the ***GAAS*** model to artificially increase the duration of the experience.
- Over-reliance on ***microtransactions*** to mimic innovation.
- Presence of ***pay-to-win*** progression systems, that reward spending instead of skill or dedication.
- Yearly release cycles diminish the value of the investment in the latest games in the genre.

3.2.3 Gains

Gains represent the opposite of the pains of a customer. They can be perceived as activities or circumstances that increase the satisfaction of the customer. The main gains of the customer, as pertains to the project, are:

- Being presented with innovative gameplay mechanics.
- Being able to enjoy a complete experience from the get-go. No gameplay content should be gated behind a purposely obtrusive progression system.
- All ***microtransactions*** are strictly cosmetic.
- No ***pay-to-win*** mechanics. All progression is directly proportional to the time investment of the players, and players with more skill are able to progress faster, but not unfairly so.
- No need to buy a new title every year. Further content, such as maps or new game modes, can be delivered to the player through the creation of ***expansion packs*** or ***DLC.***

3.3.4 Conclusion

Through the research presented in the previous chapters, the needs of the customers begin to take shape. By avoiding their pains, and aiming to fulfill their gains, their jobs are made easier, and the product is then assigned value accordingly. By using this research to inform the development of the project, a higher chance of success can be attained.

In conclusion, the product that is proposed in this project is a first-person multiplayer game that revolves around the sport of basketball. Players will extract value from it in the form of entertainment, by engaging in both competitive and casual matches with other players across the network. These matches will reward the players with cosmetic trinkets with which to decorate their in-game characters, using a fair and unobtrusive progression system that does not rely on the use of ***microtransactions*** to drive the experience and the enjoyment of the player.

3.3 Business Model Canvas

The Business Model Canvas is a type of design artifact that is used to aid in the process of understanding the structure of a business. It provides a template, through which business owners can map out the different entities of their business, such as their partners, suppliers and customers, as well as the key activities and resources that are involved. Additionally, it also details a business' financial operations, by enunciating its cost structure and revenue streams. Finally, it also allows a business owner to define the customer relationships, the channels through which those relationships are conducted, and the proposition that brings the customers value.

Figure 19 aggregates all this information. Below it, further considerations of each facet of the model are detailed.

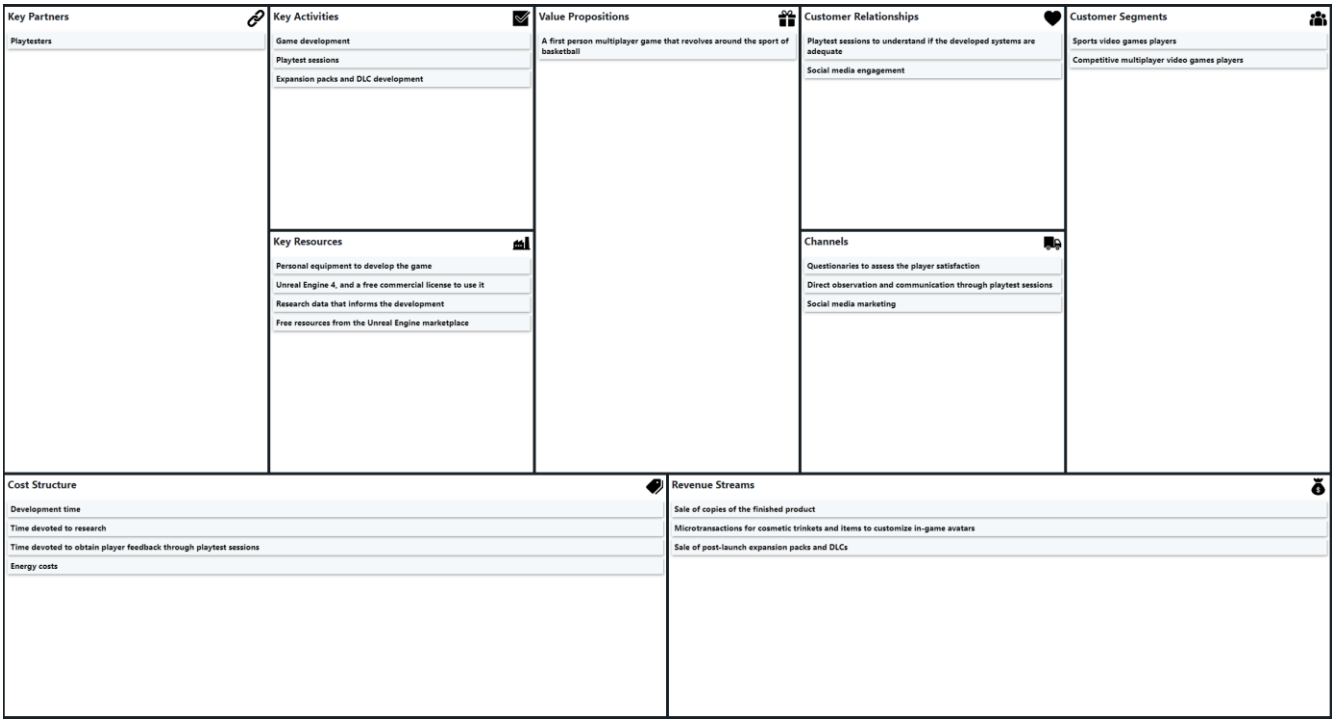


Figure 18 – Business Model Canvas

3.3.1 Key Partners

The key partners section lists any entities that aid the business in a meaningful way, by providing access to resources that couldn't be attained otherwise. The creation of partnerships is a process that inherently improves the development of a product, by lending credibility to it through the involvement of external parties that are outside of the development team. In the particular case of this project, the main key partners are the playtesters, which will inform the development of the game, and provide feedback on its features.

3.3.2 Key Activities

The key activities section lists the activities that are necessary to achieve the business' value proposition, which in turn powers the revenue streams necessary to keep the business afloat and offset the expenses detailed in the cost structure. In the particular case of this project, the main key activities are the development of the game itself and all the activities related to it, the scheduling and execution of playtest sessions in order to assess the feedback from prospective customers, and the development of ***expansion packs*** and ***DLC*** to support the game post-launch.

3.3.3 Key Resources

The key resources section lists the resources that are necessary to achieve the business' value proposition. These resources can be of different types, such as financial, human, intellectual, or logistical, but should all be directly involved in the actions that are necessary to bring the product to fruition. In the particular case of this project, the main key resources are the personal equipment that is necessary to develop the game, as game development can be quite hardware intensive, the usage of ***UE5*** and its various development tools, the research data that informs the development of the game and the free, community generated resources that are available in the Unreal Engine Marketplace.

3.3.4 Cost Structure

The cost structure section lists the most important expenses that result from the business being active, some of which directly related to the consumption of resources previously detailed in the key resources section. In the particular case of this project, the main expenses belonging to the cost structure of the business model are those that incur from the time that is spent on developing the game, the time that is devoted to research into how to improve the systems that are developed, the time that is devoted to obtaining player feedback through playtesting sessions, and the energy costs that power the personal equipment used to develop the game.

3.3.5 Customer Segments

The customer segments section lists the particular pockets of the market for which our product is designed for. These segments are what drives our revenue streams, and assure our project is economically viable. In the particular case of this project, the main customer segments are sports video games players, particularly ones that have a special interest in basketball, and competitive multiplayer video games players.

3.3.6 Customer Relationships

The customer relationships section lists the relationships that exist or are expected to exist with the customers that buy the product. It pertains to how the business aims to communicate with its customers, before, during, and after the sale of the product. In the particular case of this project, the main customer relationships will consist of the communication and contact with the players directly, during the playtest sessions that are designed to understand if the developed systems are adequate, and also the communication and contact through social media platforms, from which the creation of a community can be fostered and dynamically grown.

3.3.7 Channels

The channels section lists the avenues through which the previously mentioned customer segments should be reached and interacted with. This pertains to the marketing of the product, and the ways the business' value proposition can be delivered to prospective customers. In the particular case of this project, the main channels through which customer segments will be reached are through direct observation and communication with the play testers in the playtest sessions, as well as through the usage of social media platforms to share snippets of gameplay and the development progress.

3.3.8 Revenue Streams

The revenue streams section lists the main ways revenue is extracted from the product, to offset the expenses previously detailed in the cost structure section and to maintain the activity of the business. In the particular case of this project, the main revenue streams through which customers are expected to invest their money are the sale of copies of the finished product, ***microtransactions*** for cosmetic trinkets and items to customize the players in-game avatars, and the sale of post-launch ***expansion packs*** and ***DLCs.***

4 Analysis

The current chapter of this document is intended to portray how the analysis of the requirements, motivations, challenges, and all of their related research displayed thus far was translated into artifacts that informed the development of the game and served as proper plans that eased the implementation of the required systems necessary to deliver a working proof of concept that can be deemed adequate when placed upon scrutiny regarding the objectives it set out to achieve in the Introduction chapter.

Within it, information about the project's intricacies is presented to the player, as well as an extended study of the project's ***Domain Model*** and the concepts that make up its ***Game Design Document.*** Snippets of C++ and Blueprint code are present, whenever relevant, as well as schematics and in-game images that help the reader to visualize how the end product of any given system should look like.

4.1 Game Design Document

A **game design document**, or **GDD**, is a "highly descriptive living software design document of the design for a video game" (Oxland, Kevin, 2004). It "is the fun document that details all of the characters, the levels, the game mechanics, the views, the menus, and so on – in short, the game" (Bethke, Eric, 2003)

It is expected to feasibly and adequately inform its reader of the overall direction the development team intends to take with the videogame, so that, for instance, any member of the development team, regardless of its department, can reliably use it to inform the development of any feature that he or she might be tasked with.

A **GDD** should be structured accordingly to the game that it intends to describe. For an example, no story chapter is necessary if the game that is being built has no story to tell. For this project, a three-part approach to the **GDD** was deemed optimal.

The first part pertains to the Gameplay Design of the project. Within it, the overall gameplay of the project is laid out according to the experience that is envisioned for the players and how they might interact with the finished product.

The game's gameplay mechanics are defined, detailing what specific actions the player can take as they control their in-game character, and how those actions should interact with one another and with the game's systems to better serve the gameplay

The game's Input Model is defined, detailing which specific buttons, whether on a keyboard or a controller of some sort, the player must press in order to perform a particular action from the array of possible actions.

The game's gameplay variables are defined, detailing the variables that can exist throughout the gameplay and their importance to the balance of the experience.

Finally, the game's game modes are defined, detailing the rules under which game sessions can be played, both in multiplayer and in single player and how those rules should be enforced.

The second part pertains to the Visual Design of the project. Within it, the overall look and feel of the project are laid out, according to the necessities of the gameplay design and which information must be presented to the player to enable a rewarding and fulfilling playing experience. The game's main visual inspirations are also detailed, to ease the creation of a mental image of the desired result.

The game's menus are portrayed, as well as it's characters and main gameplay elements.

The **UI** and **HUD** for the player are defined, as well as how they should be updated as the gameplay progresses to inform the player of changes in sensitive data that may be of tactical interest for obtaining a better score.

The third part is dedicated to the Technical Analysis of the game's systems, which contains a detailed overview of mechanical systems that are the player's main interaction with the gameplay, answering the questions of how they should be implemented, how they interact with each other, and what are the ways in which the game's systems will answer the necessities posed by the design of the gameplay.

The game's domain model will be defined, detailing the main objects that exist in the project's scope and how they interact with one another

The game's Class Diagram will be defined, detailing how the main objects defined in the domain model have been converted into classes, and what kinds of variables those classes possess.

The game's main game state handling systems will be defined, detailing how the state of a given game should be handled when its variables change

The game's player state handling systems will be defined, detailing how the state of each player present in each game session should be handled in response to that player's actions and the actions of his allies and opponents.

The game's matchmaking and server browser systems will be defined, detailing how players are connected to each other and how they can filter the session they access.

4.1.1 Gameplay Design

The current chapter of this document pertains to the decisions that were made by the development team in regard to the gameplay of the project. Within it, a description of the gameplay variables and their influence is presented, as well as an explanation for their existence, how exactly they interact with each other, and how they can be used modularly to create different game modes, besides the ones that are explicitly defined and described in this chapter. Furthermore, all other facets of game design that may require an explanation can be found in the following chapters.

4.1.1.1 Core Gameplay Tenets

There are 5 core gameplay tenets that are of integral importance to the development of the videogame described in this document. Adherence to these tenets is key, both as a way to simplify and streamline any decision-making process that may derive from its creation, as well as a way to make sure the end goal of the development process and its vision as it pertains to the finished product remains focused and crystal clear. The way these tenets inform the gameplay design of the project will be explicitly alluded to throughout this document.

They are, in no particular order of importance:

- Fun, fast-paced arcade gameplay
- Physics-based approach to gameplay
- Short gameplay sessions that induce a sense of "just one more game"
- Skill-based gameplay, with no ***pay-to-win*** mechanics
- Pleasant, cartoonish visuals that can run effortlessly in a large spectrum of hardware.

The first tenet, ***"Fun, fast-paced arcade gameplay",*** is meant to assure that the game can be quickly picked up and enjoyed by a diverse target audience, that isn't expecting an accurate representation of the game of basketball, but rather a simple and straightforward reimagining of the sport that plays to its strengths while cutting back on the things that may hinder the fun factor for everyone taking part in the activity. The main consequence of this tenet is that there are no fouls in the game, making it closer to street basketball than official, professional, basketball.

The second tenet, "***Physics-based approach to gameplay",*** is meant to assure that the game makes heavy use of real-time physics simulation, and the way it can both provide predictability and unpredictability to a system, to drive its gameplay loop, rather than rely on static, pre-arranged data to present to the player in a deterministic fashion.

The third tenet "***Short gameplay sessions that induce a sense of 'just one more game'",*** is meant to assure that the game maintains an impending sense of urgency related to the duration of its game

sessions, as well as guarantee that the players are always encouraged to keep moving and to attempt to influence the score, keeping the gameplay down-time to a minimum. That way, the decision to start a new game session after one has been finished is favored over quitting the game and playing something else.

The fourth tenet "***Skill-based gameplay, with no pay-to-win mechanics",*** is meant to assure that the game keeps the playing field as level as it could possibly be, mechanics-wise, so that only the skill of any given player can influence the outcome of a match, and higher-skilled players, or players that have more experience, win more matches than lower-skilled players, or players that have less experience. This way, healthy competition, which is the basis of every good sport in the history of sports, is rewarded and encouraged.

The fifth tenet "***Pleasant, cartoonish visuals that can run effortlessly in a large spectrum of hardware"*** is meant to assure that the game values readability, accessibility and ease of use above photorealism and graphical complexity, so that the game can be enjoyed by a rather larger target audience, one that can't or won't care to invest in expensive hardware to be able to enjoy games that employ visual features that are bleeding edge, and thus quite harder to display in a pleasurable frame rate in a typical gaming setup.

4.1.1.2 Game Variables and Mechanics

Identification of the variables and mechanics that make up the gameplay loop of the project is easier if attempts are made to divide them into different groups, all of which interact with each other in fixed, reproducible ways.

These groups are:

- The Gameplay Managers
- The players and their characters
- The level entities that affect gameplay

As far as the Gameplay Managers are concerned, two immediate necessities can be identified:

- A Game Director, responsible for:
    - Starting the game
    - Pausing the game
    - Ending the game
    - Handling a shot clock that forces players to shoot the ball often, in accordance with the **third gameplay tenet**
    - Placing the players inside teams whenever they enter a session
- A Game State Manager, responsible for:
    - Keeping track of existing teams
    - Keeping track of players in existing teams
    - Keeping track of the scores of the existing teams
    - Keeping track of the scores of the players in the existing teams

    - Keeping track of the remaining time in a game session

As far as the players and their characters are concerned, these are the main variables identified:

- A Player Character, responsible for:
    - Handling player inputs
        - Moving the player
            - Influences player speed
            - Influences the player's position
        - Jumping
            - Can be charged to jump higher
            - Influences player's air position
        - Dashing
            - Influences player speed
            - Influences player's position
            - Used to knockdown enemy players, to steal the basketball, in accordance with the ***first gameplay tenet***
            - Used to knockdown enemy players, to cause chaos, in accordance with the ***first gameplay tenet***
            - Used to dodge incoming enemy players
            - Decrements a counter when used, which gets incremented after a given time has passed
        - Shooting a basketball
            - The intended direction of the shot
            - The intended force of the shot
            - The timing of the shot
            - The distance of the player character to the hoop
        - Passing a basketball
            - The intended direction of the pass
            - The intended force of the pass
- A Player Profile, responsible for:
    - Storing player preferences
    - Storing player data across game sessions
- A Player ***HUD***, responsible for:
    - Showing the remaining game time to the player
    - Showing the remaining time in the shot clock to the player
    - Showing the score to the player
    - Showing the remaining dashes to the player
    - Showing the shot trajectory to the player
    - Showing the shot power to the player

As far as the level entities that affect gameplay, these are the main entities identified:

- A Basketball entity, responsible for:
    - Being picked up by players

    - Being thrown by players
    - Physically reacting to the environment in a believable way, in accordance with the **second gameplay tenet**
- A Hoop entity, responsible for
    - Detecting when a valid point was scored
    - Detecting how many points a valid shot is worth
    - Incrementing a team's score in the event of a valid point
    - Incrementing a player's score in the event of a valid point
    - Informing the Game Director that a point was scored, and as such the game should be paused and the player reset to the necessary positions so that the game can be restarted.

Of the previously identified mechanics, three of them deserve further explanation.

The first one is Dashing. Dashing as a mechanic is inspired by the travelling rule in basketball, which dictates that taking more than two steps with control of the ball, without dribbling it on the ground, is considered a foul.

The ***first tenet*** of the gameplay design for this project dictates that no fouls exist that can hinder the progress of gameplay, so the travelling rule is instead reworked so that it can serve the gameplay instead of detracting from it. As such, instead of the three steps that would have indicated a foul, the player can instead perform three dashes that propel the player character towards any given direction, guided by player input.

Once a player performs a dash, a counter is decremented. When this counter reaches zero, the player can no longer dash. After a given time has passed after a dash was performed, the counter keeping track of the remaining dashes is incremented, allowing the player to dash again.

This maneuver can be used both to perform attacking duties, as well as to perform defensive duties.

When a player is attacking, it can be used to protect a player in the same team that's carrying the basketball, by knocking down any defensive players that may intend to steal the ball.

When a player is defending, it can be used to knock down a player on the opposite team that's carrying the basketball. After being knocked down, any player carrying the basketball immediately drops it, allowing the opposite team to assume possession of it, so that they may use it to score points for themselves.

Additionally, Dashing can also be used for evasive maneuvers, to avoid other players from dashing into the player character.

The second mechanic that deserves clarification is the existence of a Shot Clock. This mechanic is again inspired by the rules and mechanics of the real game of basketball, where a clock is used as a countdown timer that defines the amount of time that a team may possess the basketball before attempting to score a field goal. This is done to increase the scoring of field goals and to reduce stalling tactics, that may have otherwise been used to grant teams ahead on the scoreboard an unfair advantage.

It is used to much the same effect in this project, with the main difference being that its duration is reduced to 15 seconds during normal gameplay, down from what is normally employed in the official sport, where its duration ranges from 24 to 35 seconds, depending on the league. This way, both the **first** and the **third** core gameplay tenets are upheld, keeping the pace of the gameplay at a high rate, and encouraging the players to attempt to score field goals often, which contributes to the fun factor and the player's enjoyment when playing the game.

Finally, the third game mechanic that requires further definition is the Shooting mechanic, and its importance can hardly be overstated, since it directly interacts with the main win condition of any score-based game.

The team with the most amount of successful shots wins the game. As such, teams will always attempt to shoot the basketball when given an opportunity, to increase their chance of winning the game. That means that this mechanic is the central piece of the gameplay loop, the one with the most influence on the outcome of a given game session, and the one that can most accurately predict the success of a player and his skill level.

Every shot in HoopZ possesses a success rate that is calculated immediately before the player character shoots a basketball. As the name implies, the higher a shot's success rate, the higher the probability that the trajectory it takes is one that results in a successful field goal. There are four variables that can influence this success rate:

- Shot direction
- Shot power
- Jump timing
- Distance to the hoop

The first variable, shot direction, pertains to where the player chooses to aim the trajectory of the basketball at. Trajectories that are closer to the center of the hoop, similar to real-life basketball, have a higher success rate than trajectories that are farther from the center of the hoop.

The second variable, shot power, pertains to how much power the player chooses to throw the basketball. If the basketball is thrown with too much power, then it will overshoot the hoop and miss. Similarly, if it's not thrown with enough power, then it will undershoot the hoop and miss. The power of the shot must then be just so that it will neither overshoot nor undershoot the hoop, and instead find a sweet spot somewhere in the middle. Additionally, the farther the player is from the hoop, the higher the shot power must be, and the closer the player is to the hoop, the lower the shot power must be.

The third variable, jump timing, evaluates how close to the apex, or the highest point, of the jump the player was when the shot was triggered. By verifying the player's vertical velocity and whether the player is on the ground, this variable is used for increasing the success rate of a shot when the player's vertical velocity is close to zero. Additionally, negative velocities, indicating the player is no longer rising but instead falling, should give a harsher penalty on the success rate than positive velocities, indicating the player is still rising to the apex, to influence the players to avoid late shots and in that way contribute to the fast-paced gameplay described in the **first gameplay tenet.**

The fourth and final variable, distance to the hoop, was already alluded to in the description of the second variable, shot power, as they both work in tandem with one another. This variable tracks how

distant the player character is from the hoop, and applies a penalty to the success rate of shots that are too far away from the hoop the player is aiming towards. Conversely, shots that are close to the hoop the player is aiming towards will receive a bonus to their success rate. This variable contributes to the **second gameplay tenet**, in mimicking the physical difficulty, observable in real life, of throwing a basketball in a long-distance trajectory in a manner that results in a successful point.

4.1.1.3 Input Model

The input model defines how the player can interact with the in-game player character in order to play the game. The following chapters illustrate the main input actions available for the player, both in a Mouse and Keyboard setup, as well as using a Console Controller.

4.1.1.3.1 Mouse and keyboard

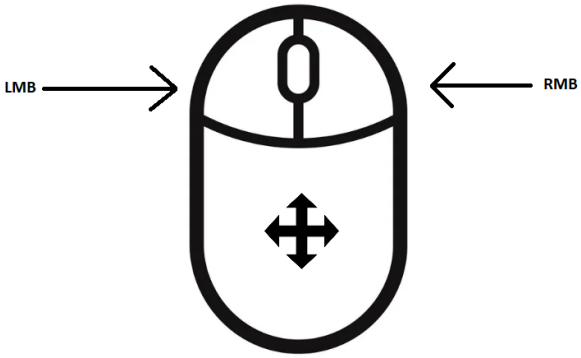


Figure 19 – Mouse Input Schematic

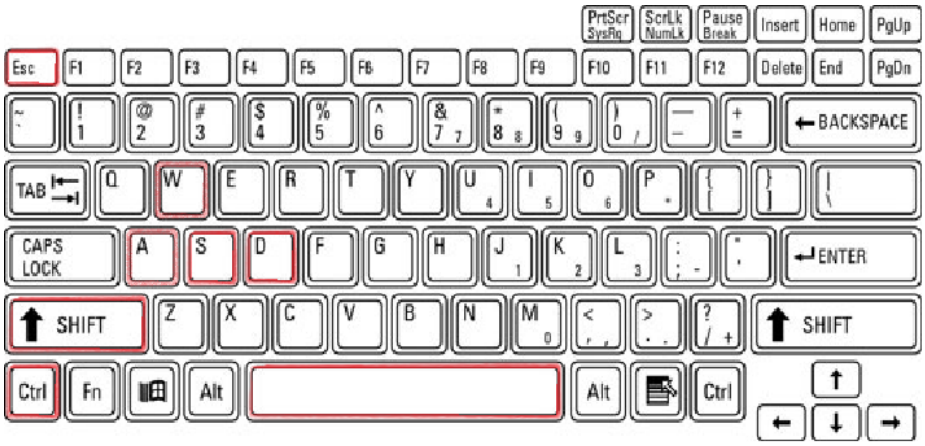


Figure 20 – Keyboard Input Schematic

In the Mouse and Keyboard setup 11 input actions were defined for the mouse, and 10 were defined for the keyboard.

For the Mouse, the input actions are:

- Rotating the camera upwards and downwards, by moving the mouse in a vertical translation.
- Rotating the camera from side to side, by moving the mouse in a horizontal translation.
- Aiming the shot, by pressing the ***RMB*** when the player is in possession of the basketball.
- Stop aiming the shot, by releasing the ***RMB*** after it was initially pressed.
- Charging up the shot, by pressing the ***LMB*** when aiming the shot.
- Charging up the pass, by pressing the ***LMB*** when the pass modifier key is pressed.
- Shooting the shot, by releasing the ***LMB*** after it was initially pressed, while still aiming the shot.
- Shooting the pass, by releasing the ***LMB*** after it was initially pressed, while the pass modifier key is pressed.
- Looking at a point of interest, either a basketball or a player in possession of the basketball, by pressing the ***LMB*** when not in possession of the basketball.
- Stop looking at a point of interest, by releasing the ***LMB*** after it was initially pressed.
- Interacting with the UI, when necessary, by pressing the ***LMB*** and pointing the mouse cursor.

For the Keyboard, the input actions are:

- Moving the player character forward, by pressing the W key.
- Moving the player character backwards, by pressing the S key.
- Moving the player character to the left, by pressing the A key.
- Moving the player character to the right, by pressing the D key.
- Dashing in a given direction, by pressing the SHIFT key while already pressing any of the movement keys.
- Begin charging the jump move, by pressing the SPACEBAR key while on the ground.
- Trigger the jump move, by releasing the SPACEBAR key after it was initially pressed.
- Modifying a shot into a pass, by pressing the LEFT CONTROL key.
- Modifying a pass into a shot, by releasing the LEFT CONTROL key after it was initially pressed.
- Accessing the pause menu, by pressing the ESCAPE key.

4.1.1.3.2 Console Controller

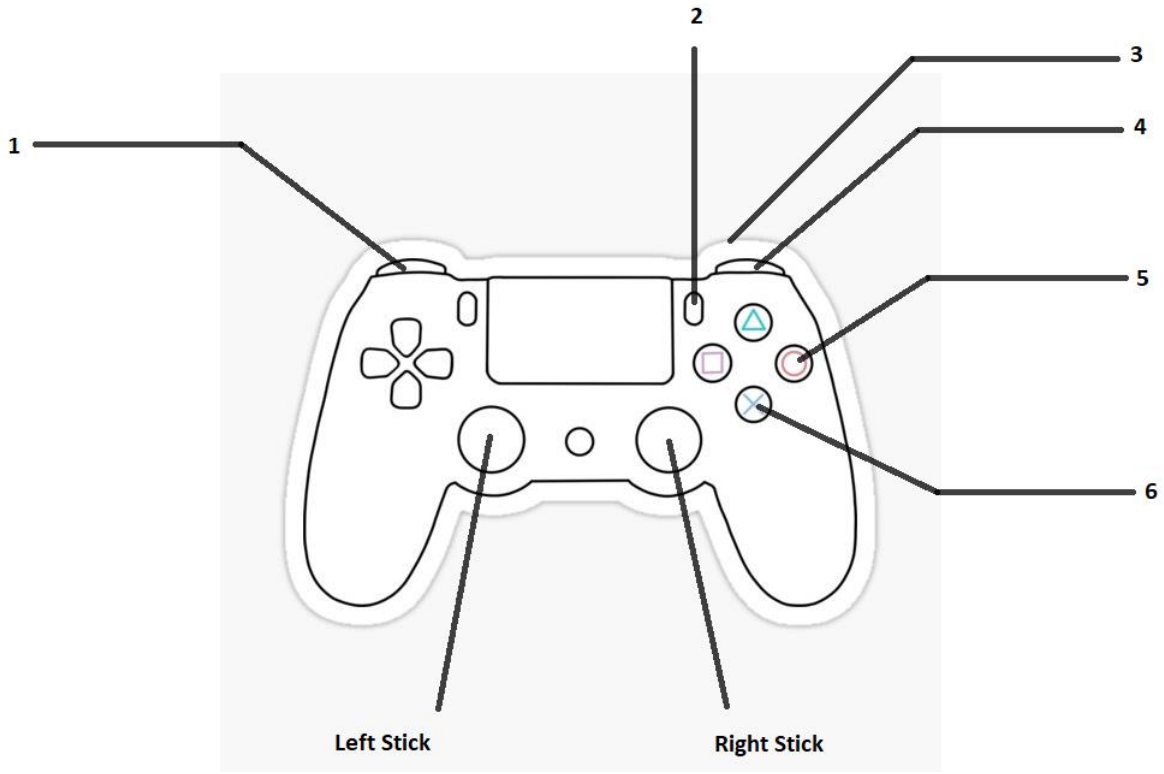


Figure 21 – Console Controller Input Schematic

The above schematic displays the buttons in the controller schematic as numbers to convene that different controller designs, like the ones from Sony's Dualshock controllers or Microsoft's Xbox Controllers, work exactly the same as it relates to the gameplay design of this project.

In the Console Controller setup 21 input actions were defined:

- Moving the player character forward, by tilting the Left Stick upwards.
- Moving the player character backward, by tilting the Left Stick downwards.
- Moving the player character to the left, by tilting the Left Stick to the left
- Moving the player character to the right, by tilting the Left Stick to the left.
- Rotating the camera upwards and downwards, by tilting the Right Stick up or down.
- Rotating the camera from side to side, by tilting the Right Stick to the left or to the right.
- Dashing in a given direction, by pressing button 5 while already moving the character with the Left Stick.
- Begin charging the jump move, by pressing button 6 while on the ground.
- Trigger the jump move, by releasing button 6 after it was initially pressed.
- Modifying a shot into a pass, by pressing button 3.
- Modifying a pass into a shot, by releasing button 3 after it was initially pressed.

- Accessing the pause menu, by pressing button 2.
- Aiming the shot, by pressing button 1 when the player is in possession of the basketball.
- Stop aiming the shot, by releasing button 1 after it was initially pressed.
- Charging up the shot, by pressing button 4 when aiming the shot.
- Charging up the pass, by pressing button 4 when the pass modifier key is pressed.
- Shooting the shot, by releasing button 4 after it was initially pressed, while still aiming the shot.
- Shooting the pass, by releasing button 4 after it was initially pressed, while the pass modifier key is pressed.
- Looking at a point of interest, either a basketball or a player in possession of the basketball, by pressing button 4 when not in possession of the basketball.
- Stop looking at a point of interest, by releasing button 4 after it was initially pressed.
- Interacting with the UI, when necessary, by pressing choosing a button with the Left Stick and pressing button 6

The input model described in this chapter, and the input mappings that were attributed to the buttons, represent the official way to interact with the game and were constructed with the intention of providing players used to First Person Shooters with a sense of familiarity that could potentiate the “pick up and play” potential of the game, in keeping with the **first gameplay tenet**. A key bind remapping feature, allowing the user to choose which buttons do what actions, should also be coupled with this input model to allow the player agency in choosing the combination of input mappings that fits his/hers preferences during gameplay.

4.1.1.4 Game Modes

In regard to the game modes that are planned for the game, three different experiences have been idealized for the initial build that’s being developed as a proof of concept.

The first, and most important, is the competitive multiplayer experience, available through the Multiplayer Game Mode. It allows for players to engage in Player vs Player (PVP) matches, with teams of either 1,2, or 3 players on each side attempting to outscore the enemy team and win the game once the match clock runs out. This Game Mode can be played on any game level that is defined as supporting competitive multiplayer, and the rules of the match are as follows:

- 5-minute match clock.
- Basketball is thrown into the air in the middle of the field at the start of the match, after a 3-second countdown.
- Match clock stops for 3 seconds after any point is scored, and restarts once the basketball reenters the field

- 15-second shot clock, that restarts whenever the team in possession of the basketball attempts a shot at the hoop
    - If the 15 seconds shot clock is expired, then the team in possession of the basketball must cede possession to the other team.
- Players can score either 2 points or 3 points with each shot, depending on the distance to the hoop.
- Players can score points against their own team, although this is not encouraged.

The second experience is the single-player experience, where players can play as part of a team of bots, playing against other teams of bots, in a simulated league where the player can either create his own team with a custom jersey and a custom team logo and name, or choose an existing team. The rules of the matches in this simulated league are much the same as the rules for multiplayer matches.

The third experience is also designed to be enjoyed in a single-player, offline way, but doesn't require that the player chooses a team, instead participating in a set of skill-challenges, designed to allow the player to practice a particular part of their game, such as the shooting mechanics, or the passing mechanics, or the defending mechanics. The player's score in each of these challenges should then be stored in an online leaderboard, so the player can compare his performance with that of other people around the world

4.1.2 Art Design

This chapter intends to present how the approach to the art design of the game was developed, and what were the main visual references and influences that instructed the visuals of the game. When needed, further information is added in how particular examples of some parts of the game's aesthetic were derived from a given influence, what exactly was the reasoning behind those decisions, and how they work for the benefit of the gameplay and the game as a whole

4.1.2.1 Overall Visual Art Style

Figure 22– Fortnite characters style (left). Fortnite overall art style (right)

The main goal of the overall art style of the game is accessibility and performance, while still looking pleasant and enticing for the players, in keeping with the **fifth core gameplay tenet**. Additionally, the art style should be universal enough that it can feasibly depict a large variety of locations, themes and tones, so as to not limit the creative possibilities provided by the project's characteristics, and its disregard for the photorealism that permeates objective reality.

This means that it should work whether the player is playing in an underwater basketball court dressed as a pirate, or in a zero-gravity orbital space station floating above the earth in an astronaut costume. The art style should be flexible enough to accommodate for the limitless choices that the setting provides.

When placed upon this particular conundrum, only one real modern game appeared to have a working solution that fit the project's necessities, namely, the title illustrated in Figure 22, pictured above, Fortnite.

Fortnite is a Battle Royale Third Person Shooter published and developed by Epic Games ever since its original release on the 21st of July of 2017. It's very hard to overstate just how much of an incredible success it has been for the company, having a very significant part to play in the e-sports and video-

game streaming communities, and generating a revenue of approximately 5.8 billion dollars in 2021 for Epic Games(BusinessOfApps, 2022a), mostly through the use of cosmetic microtransactions.

The game has a friendly, bright and stylized art style that allows for 100 players to battle amongst themselves in a fairly open island that's fully explorable, and with much of its scenery fully destroyable through player action. As if this wasn't already enough proof of its incredible technical feat, the game also supports cross play across virtually all modern gaming platforms, including mobile. It's also flexible enough that the fictional island that serves as the main battlefield for the players has been changing, sometimes drastically, for most of the 21 seasons of content updates thus far in the game's development cycle. In addition to this, Fortnite has secured partnerships with some of the biggest franchises in the entertainment industry, such as Marvel's Avengers or the Naruto and Dragon Ball anime series, allowing the players to buy or earn skins that make them look like their favorite characters either solo, or in a team with their friends. While the contrast between a Marvel superhero and some of the other purchasable skins can sometimes be a bit jarring and require some serious suspension of disbelief from the player if any immersion is to be retained, the overall art style of the game successfully accommodates for this wild variety and creativity, and its existence is thus a most invaluable asset and a very clear influence when it comes to the development of the art style of this project.

4.1.2.2 Player HUD

Figure 23 – Rocket League HUD

The player ***heads up display,*** or ***HUD,*** is a tremendously important facet of the art design of any game, especially one that focuses on competitive multiplayer. It is tasked with presenting the player with the

most important data for the gameplay task at hand at any given moment, and it should do so in the clearest, most time-effective and unobtrusive way possible.

The game pictured above in Figure 23, Rocket League, possesses one of the best examples of a successful implementation of these objectives that exist in competitive multiplayer games today. Next, an analysis of the elements of its HUD, identified by numbers and highlighted in red in Figure 24, and how they can be repurposed for HoopZ is presented:

1. Scoreboard element – This element plainly conveys the most important variables in the game session, the score of both teams, in order to determine who's winning, and the remaining gameplay time. These two variables have a direct equivalent in HoopZ, so no further adaptation is necessary.
2. Timeline element – This element conveys to the player the events that led to this specific moment in-game. It presents the main players that were involved in the play, and what their actions were. In HoopZ, these actions be directly converted into assists and field goals.
3. Player Name Tag element – This element allows the player to know the names of the other players inside the game session, as well as the team that they belong to, using two contrasting colors and a white font that eases readability. Again, these variables have a direct equivalent in HoopZ, so they can be approached in a similar manner.
4. The boost counter element – This element is used to inform the player of the amount of boost (used to make the car go faster) remaining in the fuel tank. This is the first element that doesn't have a direct equivalent in HoopZ, but it's position in the bottom right corner of the screen should still be populated with game sensitive data. A possible solution is to display a counter of the player's remaining dashes.
5. The camera type text element – This element informs the player of the camera type of his or her choosing at any given moment. This is another element that doesn't find a direct equivalent in our project, but could instead be used, possibly in a more central position, to present to the player the current shot or pass power when the relevant input was pressed.
6. The celebration message element – This element is used to communicate to the player that a goal was scored, and who scored that goal. This element could be used to much the same purpose in HoopZ, informing the player that a field goal was scored and providing information as to who the scorer is.

These elements strike a delicate balance between usability and readability. They endow the player with the information that is most necessary for the gameplay at any given moment, but have a sleek and straightforward approach, heavily resorting to color-coding to simplify information retention and acquisition.

4.1.2.3 In game Menus

Figure 24 – Revolt Main Menu(left) and Vehicle Chooser Menu(right)

In-game menus, and in particular the Main Menu, are often the first impression the player makes when booting up a new game. For this reason, special care should be put to their presentation, and their navigation and the display of their content should strive to be as self-explanatory as possible, allowing the player to understand, quickly and effectively, what content is available, and what features he or she can interact with.

The basic, most generic and widespread version of in-game menus works by using a 2D canvas where different UI objects, like Buttons and Text boxes, are placed into and then the player camera is pointed directly into this canvas, and never again moves. Developers then create different canvases with different UI elements, and each canvas has a very specific function, such as the canvas containing the main menu, or the canvas containing the settings menu. Then, normally at the press of a button by the player, these canvases are alternated between, the ones that become relevant are enabled, and the ones that are no longer needed are disabled, until there comes a time where they're needed again. This all takes place on top of a static background, most often displaying some visual art referring to the game being played.

Revolt, released in 1999 by Acclaim Entertainment, is a racing game that takes a different approach to the in-game menu navigation. Instead of showing 2D canvases on top of a static background, they are instead shown on top of a dynamic, moving background, because the camera is no longer held in place and made to show an array of alternating canvases. Instead, the camera itself moves in response to player input, showing a different camera angle of the main menu map, and only after the transition into that camera angle is finished is the 2D canvas containing the UI elements constructed and made visible to the player. Then, when the player chooses a new option that requires a new transition into a new camera angle, the existing 2D canvas, no longer necessary, disables itself and informs the camera that the transition it intends to perform is now possible. This is used to great effect in Revolt to display interesting camera angles that heavily improve the immersion of the game, such as the vehicle choice menu being displayed on top of a stack of boxes that represent the vehicles the player can choose, shown in Figure 24.

4.1.2.4 Sound Design

As far as the sound design is concerned, a mix of realism and surrealism is intended. The sounds of the basketball hitting the court should deliver a believable and realistic response, whether on a surface that is wooden, metallic or any other type according to the map design. The sounds of the basketball hitting the backboard of the hoop and the net of the hoop are also to be given major attention, since they act as a dopamine dispenser mechanism, not only in the game but also in real life, as any common basketball player can attest to. Other sorts of generic background sounds, depending on which map is being played, should also contribute to the soundscape, in order to immerse the player in his or her surroundings and improve the user experience.

The shouts of the other player characters, whether when requesting a ball from the player or gasping in pain when knocked down, should deliver a lighthearted and comedic vibe that nonetheless remains clear in its purpose. Sounds uttered by player characters should be distinct enough that they can be picked apart even during the intended chaos caused by the gameplay, but they shouldn't be too brash to the point that they sound unpleasing.

The game's music should strive to compliment the gameplay, boasting a high tempo that can induce the players to remain in motion and avoid inactivity. It should also take into account the theme of any given map the players are currently playing in, and should be easily loopable, so that the action is never unnecessarily stopped by a music track coming to an end.

4.1.2.5 Map Design

As far as the game's Map Design is concerned, very few limitations exist, thematically or visually. The rules of the game and its inherent characteristics make it so creativity is the only guiding principle in idealizing and creating a new game map. Maps can be inspired by real life locations or can depict fantastic sceneries that could only be realized in virtual settings, or any sort of scenario in between. Without being bogged down by the need to faithfully recreate licensed real life player likenesses and arenas, originality is given space to flourish. Additionally, the maps could even influence the gameplay, by presenting interactable environment objects that the players could use to their benefit, such as trapdoors or trampolines, or simply by depicting places that have vastly different physical properties, such as the gravity of the moon being lower than that of the earth, leading to a different basketball behavior.

Currently, the only limitation is one of size. Maps need to be big enough to hold within them 2 hoops, and ideally enough space for players to run around without constantly bumping into each other, although even this could also be used for comedic purposes given a compatible setting.

As such, the two maps that are planned for this project's proof of concept delivery intend to show this wealth of possibilities, properly tempered by the logical constraints placed on its one-man development team. They are named, simply, "Park" and "Rooftop".

Figure 25 – EA Sports NBA Street V3 Park Map

“Park” takes place in a fictionalized depiction of what a basketball court could look like, in the game’s universe, and mainly takes inspiration from the Park Map that is present in EA Sports NBA Street V3, released in 2005 and depicted in Figure 25. It intends to provide players with a familiar sight, not unlike the one they could feasibly have were they to play basketball with their friends in their hometown’s basketball court.

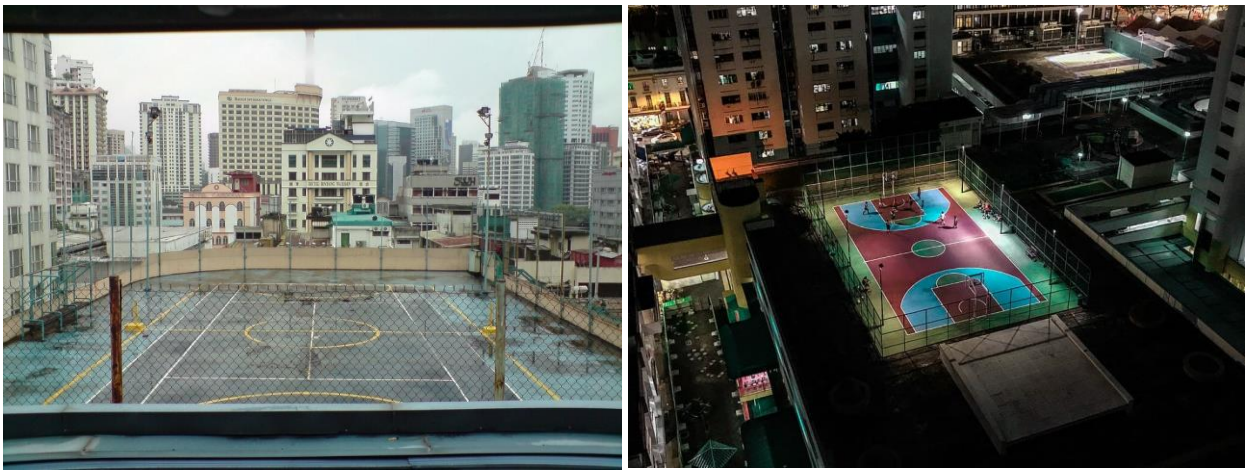


Figure 26 – Rooftop Basketball courts in a daytime(left) and nighttime(right) setting

“Rooftop”, however, intends to provide players with a more surrealist experience atop a skyscraper, surrounded by other skyscrapers in a nondescript city at nighttime, with shining beams of light

illuminating the court and the gusting wind acting as the ever-present reminder of the altitude. The two different real life basketball courts depicted in Figure 26 are the main inspiration behind this map

4.1.3. Technical Design

In the current chapter, information about the technical design of the project is provided. This includes diagrams representing the main systems of the game and how they're going to be implemented, as well as an overall view into how the project's classes are connected to each other.

Additionally, information about Unreal Engine 5 and the way this game engine structures its main classes and entities is supplied, in order for the reader to get a sense of how development is normally approached using this tool.

4.1.3.1 Unreal Engine 5

Unreal Engine 5 code is written in C++ and Blueprint code. The Blueprint Visual Scripting system in Unreal Engine is a complete gameplay scripting system based on the concept of using a node-based interface to create gameplay elements from within Unreal Editor. As with many common scripting languages, it is used to define object-oriented (OO) classes or objects in the engine. These classes or objects are called Blueprints.

In Unreal Engine 5, objects that can be placed or spawned within a game world are called ***actors.*** Actors act as containers unto which different types of content can be added, using ***actor components***. ***Actor components*** can have wildly different uses. An Input Component, for instance, is used to allow an ***actor*** to detect and react to player input, while a Static Mesh Component is used to give an ***actor*** a 3D model to display. ***Actor components*** can be custom-built too, to allow developers to create new features for which the game engine doesn't yet have ***actor components*** for.

The C++ class responsible for the behavior of a given ***actor*** has constructor and destructor functions that are called whenever that ***actor*** is created or destroyed. In the constructor function, the different components that make up an ***actor*** should also be created and setup accordingly. However, to avoid hardcoding references to data that is used by any given component, such as a file path to a 3D model to be used by a Static Mesh Component, Unreal makes use of its Reflection System.

Reflection is the ability of a program to examine itself at runtime. This is hugely useful and is a foundational technology of the Unreal Engine, powering many systems such as detail panels in the editor, serialization, garbage collection, network replication, and Blueprint/C++ communication. However, C++ doesn't natively support any form of reflection, so Unreal has its own system to harvest, query, and manipulate information about C++ classes, structs, functions, member variables, and enumerations.

The Reflection System is opt-in. Any types or properties that should be visible to the reflection system need to be annotated accordingly, and the Unreal Header Tool (UHT) will harvest that information when the project is compiled. This annotation is done through the use of the UPROPERTY macro. When a

variable, or a component of any sort, is labelled as a UPROPERTY in a C++ class's header file, the UHT takes care of exposing that information whenever any Blueprint derived from that class is interacted with in the Editor. This way, no hardcoding of any references is necessary. A programmer or a game designer can simply open the Blueprint class of any given actor in the project, and instead change the value of its data from there. This ensures that, for example, when a given 3D model is moved from one file path to another, any actors that reference it don't need to change that file path in their C++ class. Instead, Unreal already takes care of that seamlessly.

Additionally, this is also the way Unreal takes care of sharing data across the network in a easy and straightforward way, through the use of replicated variables, OnRep Notify functions and Remote Procedure Calls.

Variables can be defined as replicated simply by labelling them as an UPROPERTY with the Replicated modifier when they're defined in a class's C++ header file.

```
UPROPERTY(Replicated)
```

Figure 27 – UPROPERTY macro with Replicated property

A variable being defined as replicated means that the server must always keep track of its server-side values and, in the event that they change, propagate the change in value to all clients that are connected to the server and that qualify to be informed of that change, so they can replicate that variable client-side. Some clients, such as the ones that are too far away to notice the effects of the change in a given variable, can be filtered out of the list of clients to inform, to make network traffic more efficient.

OnRep Notify functions are simply functions that are assigned to run immediately after a variable is replicated, so that they can perform some kind of action based on the newly updated value of that variable. Assigning an OnRep Notify function to a replicated variable is done in much the same way as defining a replicated variable, albeit with a different modifier given to the UPROPERTY macro, like so

```
UPROPERTY(ReplicatedUsing = OnRep_AnInteger)
int AnInteger;

UFUNCTION()
void OnRep_AnInteger();
```

Figure 28– UPROPERTY macro with ReplicatedUsing property and UFUNCTION macro

In Figure 28 above, we can see that the UPROPERTY macro is being instructed to treat the AnInteger variable as a replicated variable, and is also being told to call the OnRep_AnInteger() function whenever a new value for it is replicated from the server. The OnRep_AnInteger function is required to be labelled as an UFUNCTION for the Unreal Reflection System to properly detect it as an OnRep Notify function, which can then call it accordingly.

Finally, Remote Procedure Calls, or RPCs, are simply functions that can be instructed to only run either server-side or client-side. They're the main way clients can communicate to the server their intentions regarding any gameplay sensitive action that must be verified by the server first, to guarantee that no

wrongdoing is being attempted. They're also the main way the server can instruct any given client to do something directly that can't be done server-side, such as giving the order to ignore player inputs in the event of a match ending, for example. RPC functions can be declared in the following way

```
UFUNCTION(Server, Reliable, WithValidation)
void ServerRPCFunction(int IntToSendToTheServer, FString StringToSendToTheServer);

UFUNCTION(Client, Unreliable)
void ClientRPCFunction(int IntToSendToTheClient, FString StringToSendToTheClient);
```

Figure 29– Server and client RPCs using respective UFUNCTION macros

As we can see in Figure 29 above, RPC functions must use the UFUNCTION macro, and given either the Server or Client modifier to distinguish between Server and Client RPCs. Then, they need to be labelled as either Reliable or Unreliable, which controls whether a RPC function should be sent again in the case of a packet loss, as many times as necessary, until an acknowledgment message comes back confirming the RPC reached its destination, or whether the RPC not reaching its destination can be safely ignored, to make network traffic more efficient. Normally functions that concern gameplay logic need to be labelled as Reliable, because the gameplay state must be kept as similar as possible across all connected players, while functions that concern cosmetic changes, such as the triggering of particles from an explosion or the muzzle flash of a weapon being shot, can be safely labelled as Unreliable without much alarm.

Finally, the WithValidation modifier can be used, in Server RPCs, to force a conditional validation of the data inside the actor that has called the RPC, before the code inside the RPC function can run. This is mostly used to handle cheat detection mechanisms, to ensure that players that attempt to corrupt client-side data are properly punished for doing so. For the purposes of this document, this subject of Unreal Engine network programming isn't something that is going to be mentioned any further, but it's inclusion here is necessary to explain the usefulness of the Unreal Reflection System, and how it can streamline otherwise very complicated and convoluted systems that could take up precious development time better used somewhere else.

Another important facet of the development process, and one that is alluded through in the following chapters in this document, has to do with the manipulation of the camera of the player. Most game engines describe camera movement using three different variables, and Unreal Engine 5 is no different. A schematic explaining what these variables affect is available in Figure 30.

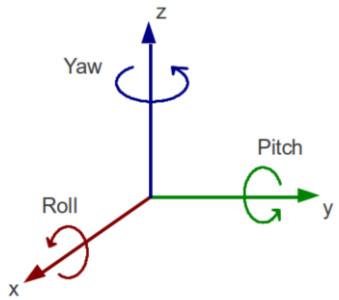


Figure 30 – Camera Rotation Variables

As we can see, all rotation data is communicated using three variables:

- Yaw, for describing rotation along the Z axis.
- Roll, for describing rotation along the X axis.
- Pitch, for describing rotation along the Y axis.

In Unreal, the X axis is the forward vector for the coordinate system. That means that, for FPS projects that follow the traditional setup of having a Camera attached to a Collision Capsule, altering the Yaw variable will make the Camera rotate to the left or to the right, and altering the Pitch variable will make the Camera rotate up and down. The Roll variable is normally never altered, and this is also true in our project.

4.1.3.2 Domain Model

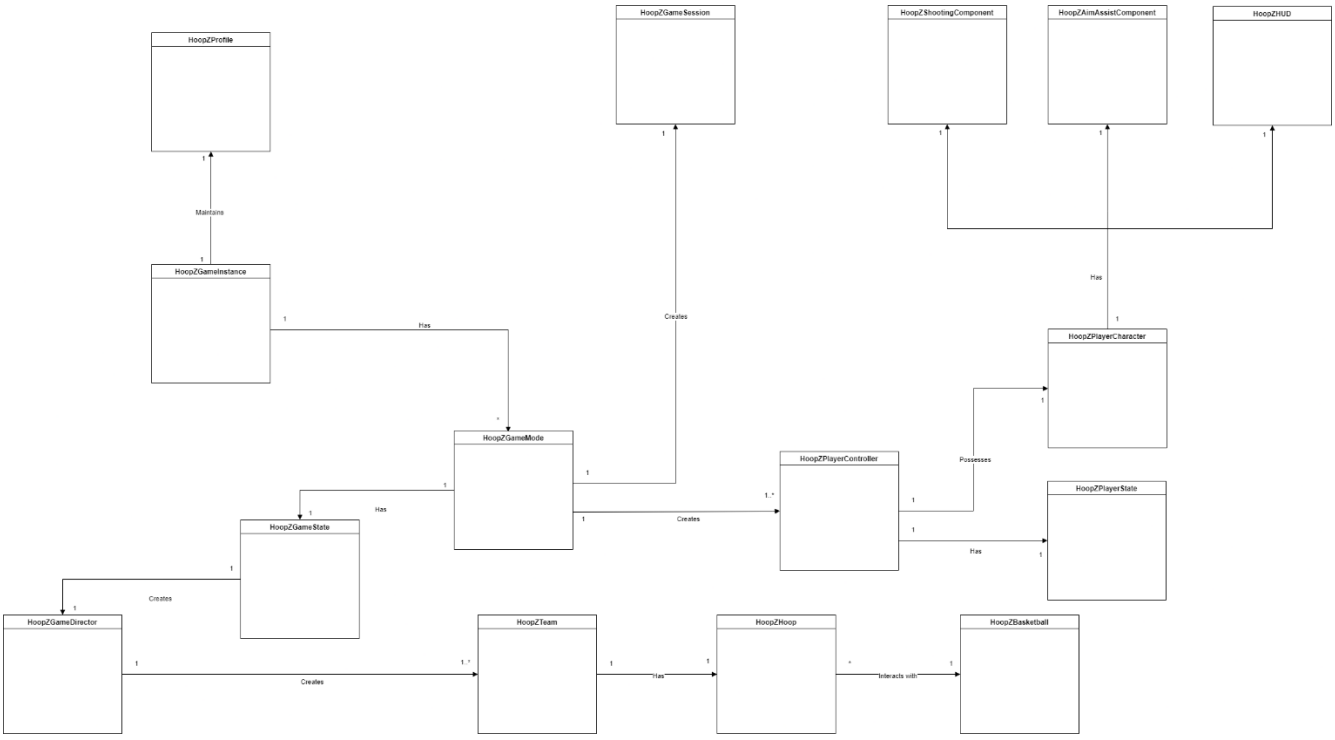


Figure 31– Domain Model Diagram

In Figure 31, depicted above, the Domain Model Diagram for the project is presented. It displays the main connections between each entity in the project and what those connections entail.

- HoopZGameInstance - The root entity in the game, responsible for all transient behavior across the various stages of gameplay. It acts as a ***Singleton***, persisting for as long as the game is running, and can be used as a hub from which information can be transitioned across different HoopZGameModes.

- HoopZProfile – The entity responsible for storing and retrieving all data that should persist between gameplay sessions such as player settings and preferences.
- HoopZGameMode – The entity responsible for defining the rules and the characteristics of any given match during a gameplay session. In the Unreal Engine authoritative server-client network architecture, only servers (dedicated servers or players acting as servers) can access a Game Mode, to ensure the rules can't be cheated by bad actors.
- HoopZGameSession – The entity responsible for creating the game session to which players connect to, responsible for keeping track of the players that have entered the session, left the session or were banned/kicked from the session.
- HoopZGameState – The entity responsible for maintaining data that is related to the state of a given gameplay session. This data includes the score of each team, a list of all the players in a game session, the time remaining in the match and other such data that might be necessary. It should be noted that, unlike the Game Mode, which is only accessible to the server, the Game State is by design accessible to all players.
- HoopZGameDirector – The entity responsible for directing the gameplay session whenever necessary. This includes starting the game, pausing the game and ending the game, but also dealing with cases where the basketball leaves the bounds of the game map, or creating the teams to which players can belong to at the beginning of a game session.
- HoopZTeam – The entity responsible for representing the concept of a team in the gameplay session. Every player in the game needs to have a team. Each team has a hoop that they must defend, and a color that differentiates the player in a given team from the players in the opposing team.
- HoopZHoop – The entity responsible for detecting when a valid field goal has been scored, and communicating that information to the HoopZGameDirector entity so that the team scores can be updated accordingly.
- HoopZBasketball – The entity responsible for interacting with the players in the game session, allowing them to attempt to score field goals in the opposing team's hoop to increase the score of their team.
- HoopZPlayerController – The entity responsible for receiving the players input and turning it into gameplay actions. It should be noted that in the Unreal Engine authoritative server-client network architecture, each client player can only access its own Player Controller, so that players can't meddle with the other players actions. The server, however, can access the Player Controllers of every player in the game world.
- HoopZPlayerCharacter – The entity responsible for responding to the player inputs being transmitted through the HoopZPlayerController. It serves as the visual representation of the player avatar in-game, and is the main gameplay entity
- HoopZPlayerState – The entity responsible for maintaining data that is related to the state of a player in a given gameplay session. This data includes the amount of points that a given player has scored, or the number of assists a player has, as well as the player's nickname. The HoopZPlayerState, same as the HoopZGameState entity, is by design accessible to all players in Unreal Engine 5.
- HoopZShootingComponent – The entity responsible for translating player inputs into shooting actions. Every HoopZPlayerCharacter entity has a shooting component attached to it.

- HoopZAimAssistComponent – The entity responsible for assisting the player in aiming the basketball at the hoop. Every HoopZPlayerCharacter entity has an aim assist component attached to it
- HoopZHUD – The entity responsible for presenting information in the UI layer of gameplay to the player. Every HoopZPlayerCharacter entity has a HUD attached to it.

5 Implementation

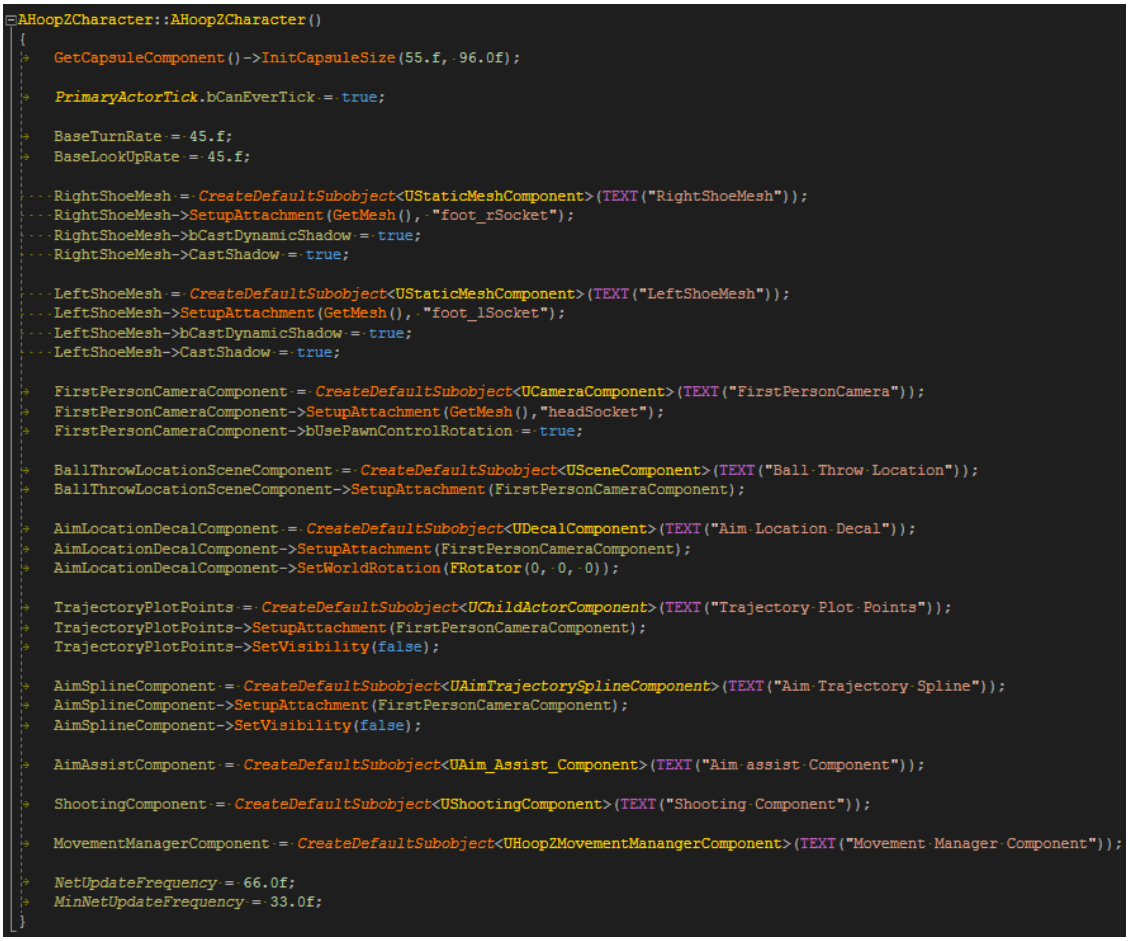
The current chapter describes in detail how the implementation of the main systems of the project was approached and how the final result was achieved. Each sub-chapter within it will focus on a different system. Images of the features described and snippets of C++ or Blueprints code are also supplied when necessary.

5.1 Full Body Immersion In First Person

In order to explain the implementation of the Full Body Immersion system, the HoopZCharacter class needs to be described, particularly its variables, components, and main functions regarding player movement and camera input.

```
/** First person camera */
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category = Camera, meta = (AllowPrivateAccess = "true"))
class UCameraComponent* FirstPersonCameraComponent;

/** Component used to determine where the basketball should be spawned when performing a shot */
UPROPERTY(EditDefaultsOnly, Category = Aiming)
class USceneComponent* BallThrowLocationSceneComponent;

/** Decal component used to display where the basketball is expected to hit */
UPROPERTY(VisibleAnywhere, Category = Aiming)
class UDecalComponent* AimLocationDecalComponent;

/** Component that is responsible for performing aim assist when the player is aiming the ball */
UPROPERTY(EditAnywhere, Category = Templates)
class UHoopZAimAssistComponent* AimAssistComponent;

/** Component that is responsible for spawning and shooting the ball*/
UPROPERTY(EditAnywhere, Category = Templates)
class UShootingComponent* ShootingComponent;

/** Component that is responsible for dealing with movement variables and abilities (dash, jump,etc) */
UPROPERTY(EditAnywhere, Category = Templates)
class UHoopZMovementManagerComponent* MovementManagerComponent;

/** Component that is responsible for creating the trajectory plot points that show the player the trajectory of the basketball if thrown*/
UPROPERTY(VisibleAnywhere, Category = Templates)
class UChildActorComponent* TrajectoryPlotPoints;

/** Component that is responsible for interacting with the trajectory plot points to display the ball trajectory*/
UPROPERTY(EditAnywhere, Category = Templates)
class UAimTrajectorySplineComponent* AimSplineComponent;

/** Right Shoe mesh component*/
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = Mesh)
class UStaticMeshComponent* RightShoeMesh;

/** Left Shoe mesh component */
UPROPERTY(VisibleDefaultsOnly, BlueprintReadWrite, Category = Mesh)
class UStaticMeshComponent* LeftShoeMesh;

/** Enum for the current gameplay action state of the player (whether player is shooting, jumping, aiming, etc)*/
UPROPERTY(EditAnywhere, Category = Gameplay)
EPlayerActionState PlayerActionState;
```

Figure 32– HoopZCharacter class header file



```
AHoopZCharacter::AHoopZCharacter()
{
    GetCapsuleComponent()->InitCapsuleSize(55.f, 96.0f);

    PrimaryActorTick.bCanEverTick = true;

    BaseTurnRate = 45.f;
    BaseLookUpRate = 45.f;

    RightShoeMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("RightShoeMesh"));
    RightShoeMesh->SetupAttachment(GetMesh(), "foot_rSocket");
    RightShoeMesh->bCastDynamicShadow = true;
    RightShoeMesh->CastShadow = true;

    LeftShoeMesh = CreateDefaultSubobject<UStaticMeshComponent>(TEXT("LeftShoeMesh"));
    LeftShoeMesh->SetupAttachment(GetMesh(), "foot_lSocket");
    LeftShoeMesh->bCastDynamicShadow = true;
    LeftShoeMesh->CastShadow = true;

    FirstPersonCameraComponent = CreateDefaultSubobject<UCameraComponent>(TEXT("FirstPersonCamera"));
    FirstPersonCameraComponent->SetupAttachment(GetMesh(),"headSocket");
    FirstPersonCameraComponent->bUsePawnControlRotation = true;

    BallThrowLocationSceneComponent = CreateDefaultSubobject<USceneComponent>(TEXT("Ball Throw Location"));
    BallThrowLocationSceneComponent->SetupAttachment(FirstPersonCameraComponent);

    AimLocationDecalComponent = CreateDefaultSubobject<UDecalComponent>(TEXT("Aim Location Decal"));
    AimLocationDecalComponent->SetupAttachment(FirstPersonCameraComponent);
    AimLocationDecalComponent->SetWorldRotation(FRotator(0, 0, 0));

    TrajectoryPlotPoints = CreateDefaultSubobject<UChildActorComponent>(TEXT("Trajectory Plot Points"));
    TrajectoryPlotPoints->SetupAttachment(FirstPersonCameraComponent);
    TrajectoryPlotPoints->SetVisibility(false);

    AimSplineComponent = CreateDefaultSubobject<UAimTrajectorySplineComponent>(TEXT("Aim Trajectory Spline"));
    AimSplineComponent->SetupAttachment(FirstPersonCameraComponent);
    AimSplineComponent->SetVisibility(false);

    AimAssistComponent = CreateDefaultSubobject<UAim_Assist_Component>(TEXT("Aim assist Component"));

    ShootingComponent = CreateDefaultSubobject<UShootingComponent>(TEXT("Shooting Component"));

    MovementManagerComponent = CreateDefaultSubobject<UHoopZMovementManangerComponent>(TEXT("Movement Manager Component"));

    NetUpdateFrequency = 66.0f;
    MinNetUpdateFrequency = 33.0f;
}
```

Figure 33– HoopZCharacter cpp file

Figures 32 and 33, pictured above, show the components that the HoopZCharacter utilizes. Their definition in the classes header file, in Figure 35 shows how they are exposed to the editor, using the UPROPERTY macro, in order to be manipulated in their blueprint class. Their purpose is plainly stated in the comments above their respective definition.

In Figure 33 we can see the contents of the class's constructor function, where the creation of all the necessary components takes place. It should be noted that the FirstPersonCameraComponent, that creates the Camera through which the player can look around and see the world, is attached to the player's Mesh, which renders the player characters in-game body. The SetupAttachment function is used to hierarchically attach components to each other, so that they can move in tandem if need be, and it has 2 arguments. The first one is a pointer to the component to attach to, and the second one is an optional parameter that allows us to send a socket name to control the location on which the component should be attached, using the socket system in Unreal.

The socket system is used to attach actors, or actor components, to each other using sockets. A socket is an object that can be created when interacting with the skeleton of a skeletal mesh. A skeleton is a hierarchical grouping of objects, called bones, that is used to create animations. A skeletal mesh is a

mesh that is bound to a skeleton, and can thus use that skeleton, and the animations that were created for it, to display motion. It's also important to note that, in Unreal, skeletons are the base of the animation system. When importing a skeletal mesh into the project, one must select a proper skeleton for that skeletal mesh to use, if one exists, or the engine can import the one that was made in whichever 3D modelling tool that was used to create that skeletal mesh. Unreal Engine 5 has a default template skeleton that teams are encouraged to use, because it's the standard skeleton that's used in all the engine's template projects, and an enormous amount of documentation and tutorials exist for it, as well as a lot of Unreal Marketplace animation asset packs that are specifically made for the Unreal Default Skeleton.

In the interest of time, ease of development and a general lack of necessity to reinvent the wheel, that is the skeleton that is being used to power the animations of all the player characters in HoopZ. Its contents are displayed as follows.

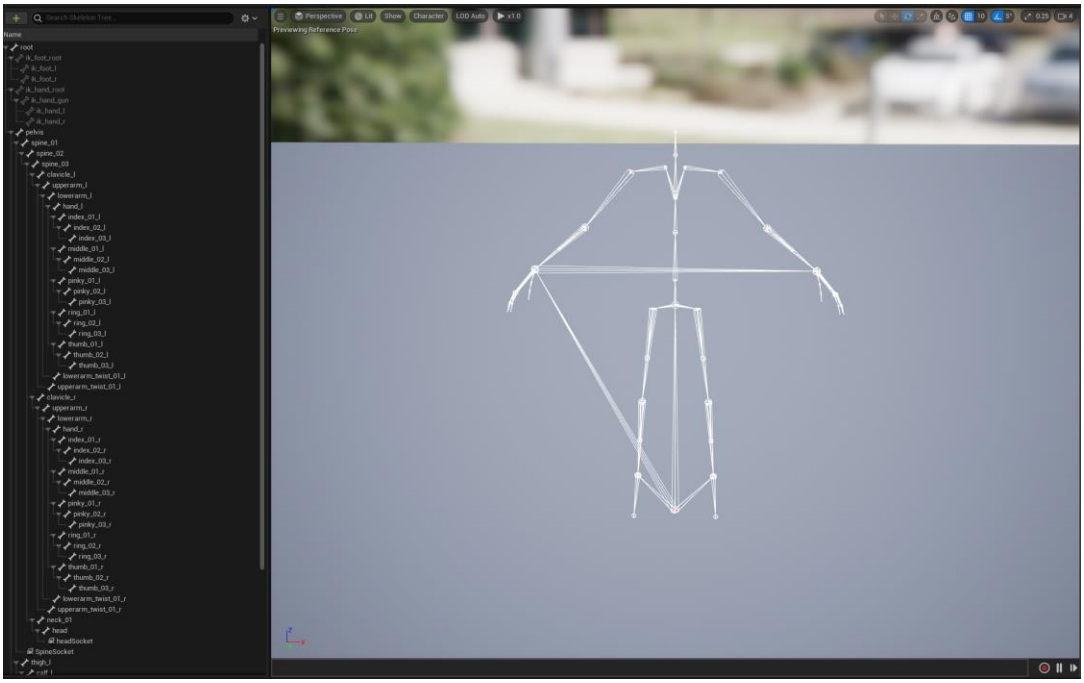


Figure 34– Character Skeleton in Unreal Engine

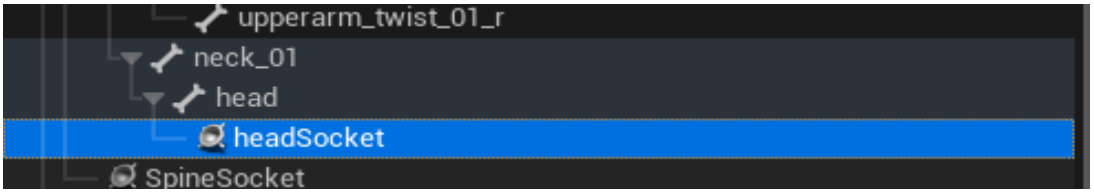
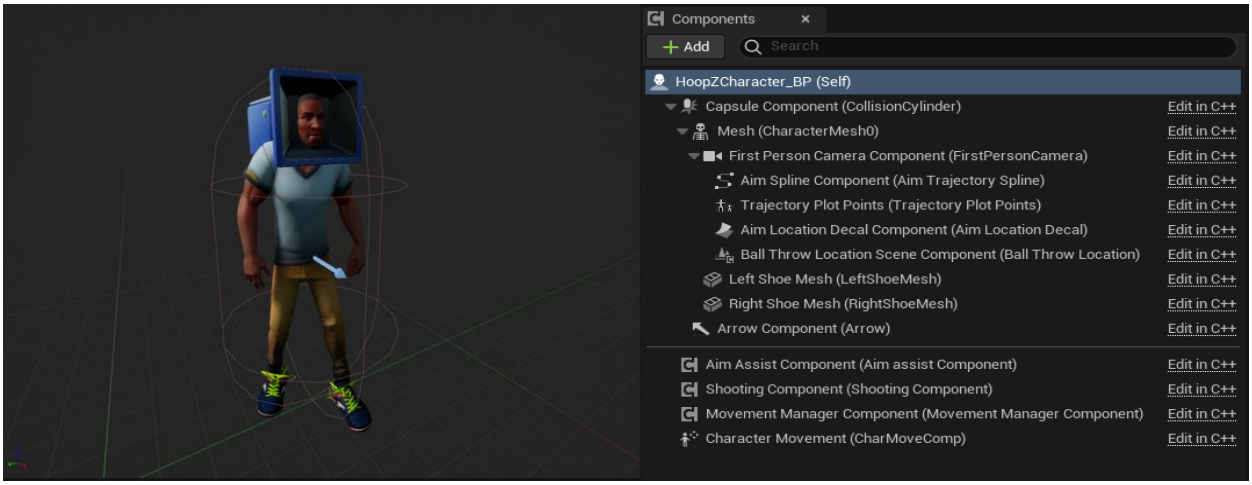


Figure 35– Head Socket definition in head bone

In Figure 34 portrayed above, we can see, in the center of the screen and rendered as a white wireframe, the actual bones that make up the skeleton we're using for the characters. On the left, a nested list displaying the hierarchy of the bones and how they're connected to each other can be consulted. On the next figure, Figure 35, a zoomed—in screenshot of the head bone is shown. Attached to that bone, a socket was created an given the name headSocket. It is to this socket that the SetupAttachment function in the HoopZCharacter constructor will try to attach the FirstPersonCameraComponent. By doing this, the Camera will move whenever the headSocket attached to the head bone moves, achieving full body immersion. After tweaking the values of the headSocket location inside the skeleton, here's how the HoopZCharacter Blueprint class works, inside the Unreal Engine editor



Figures 36 – HoopZCharacter Blueprint class viewport view(left) and component hierarchy viewer (right)

As we can see, the camera was successfully attached to the socket, and appears in the correct position, as can be seen in Figure 36, to the left. To the right, the overall hierarchy of the actor components in the actor can be seen, in precisely the same order as it was defined in the class's constructor function. We can see that the First Person Camera Component is correctly attached to the Mesh of the Character, which in turn is attached to the Capsule Component which serves as the Root Component of the Actor. We can also see that the components that don't require a physical location in the world, instead just performing logical functions, like the Aim Assist Component or the Shooting Component, are kept on a different list, and don't need to be attached to any other component.

Finally, all of these steps are combined during gameplay to deliver an overall cohesive approach to the problem of Full Body First Person Immersion, the final product of which can be observed in Figure 37 below.

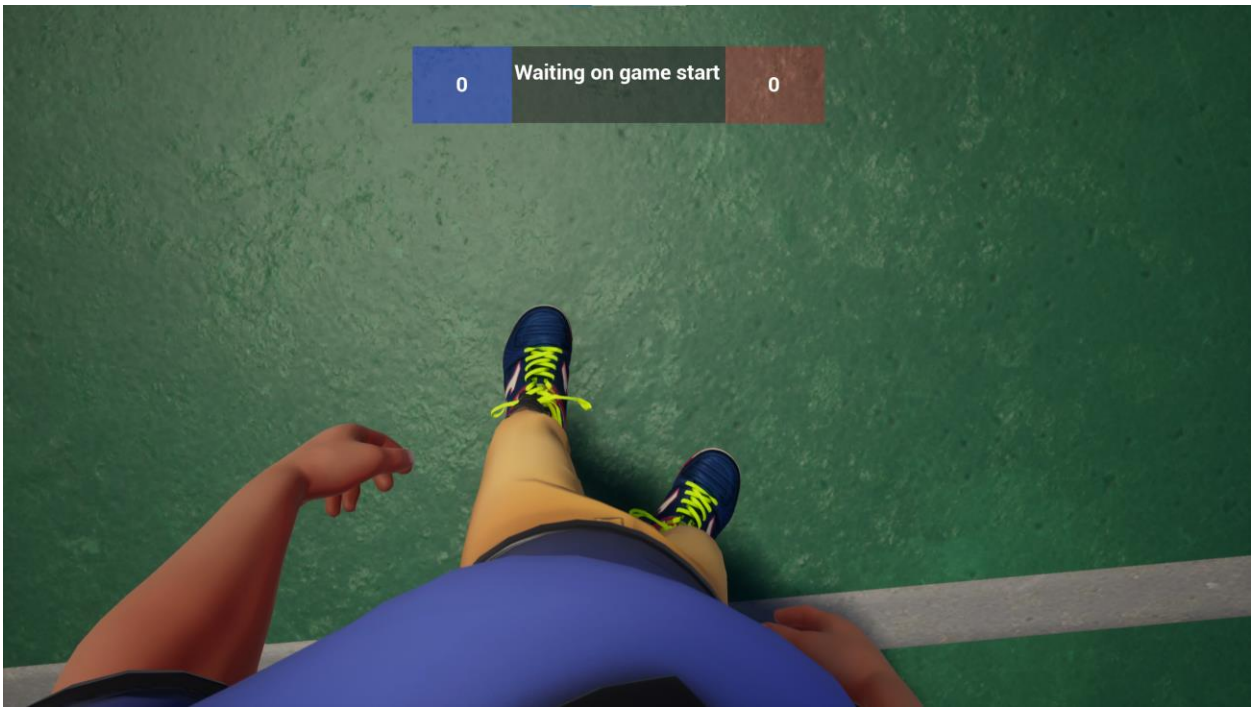


Figure 37– In-game example of the Full Body First Person Immersion System

As we can see, the player can simply look down and notice that he or she is no longer a floating pair of arms, and instead, has a fully rendered body that looks and feels immersive. At the same time, this system also streamlines the development of the game as far as new animations are concerned, because the animator in charge of creating them only needs to worry about one set of animations, instead of two, even if tweaking the animations to make sure they look good both in a First Person and a Third Person perspectives might take a little bit longer.

However, this system, and others like it, are infamous for being broken quite easily in games that allow players to customize the Field Of View (FOV) of the Player Character Camera. This happens because, for higher FOV values, the camera will start to render the inside of the player's head, so visual artifacts like the players mouth or eyes will begin to be viewed by the players, which obviously hurts immersion. The solution to this is often to simply instruct the player's own Camera, not the Cameras of other players, to not render the player's head, so the FOV can be safely changed without any risk of visual artifacts. For this current proof of concept, since no FOV changing functionality is planned, this is not necessary, but a solution such as the one just mentioned will be developed for the game's later versions.

5.2 First Person Aim Assist System

The First Person Aim Assist System intends to help players to better aim at the hoop and, consequently, be able to improve their shot accuracy and efficiency so that they can fully engage in the game's main loop, namely, the cycle of scoring points and improving your team's score so your team can win.

In order to ensure that this system is modular and can be reused in other ways in the future, possibly by the AI characters, an Actor Component was created, and all Aim Assist functionality was developed within it. This actor component and its functions is then called by the HoopZCharacter class whenever the related inputs are pressed. The header file for it, where the functions that are used are defined, can be viewed in Figure 38 below.

```
	/** Sets the speed at which the camera should be interpolated when in aim assist */
	UPROPERTY(EditAnywhere)
	float CameraInterpolationSpeed;

public:
	/** Called when the game starts */
	virtual void BeginPlay() override;

	/** Called every frame */
	virtual void TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction) override;

	/** Sets up necessary references for the activity of this component */
	void SetupReferences(class UCameraComponent* InPlayerCameraComponent, class USceneComponent* InPlayerRootComponent, class AHoopZCharacter* InOwningCharacter);

	/** Stores a reference to the actor being currently targeted */
	void SetTargetActor(AActor* InActorBeingTargeted, uint8 InActorType);

	/** Used to store the target point to look at when first aiming the basketball at the hoop */
	void SetTargetPoint(const FVector& InCurrentTargetPoint);

	/** Used to offset the originally stored target point in response to player input */
	void AddToTargetPoint(const FVector& InTargetPointOffset);

	/** Returns whether the aim assist component is currently targeting an actor */
	bool HasTarget();

	/** Used to look at the currently targeted actor */
	void LookAtTarget(float DeltaTime);

	/** Used to stop looking at the currently targeted actor, and to clear up all data related to the previous aim assist */
	void Stop();

	/** Used to adjust the ball trajectory that is dependant on a given FVector */
	void AdjustBallTrajectory(FVector& InAimVectorForce);

private:

	/** Pointer to the actor currently being targeted by the Aim Assist System */
	AActor* ActorBeingTargeted;

	/** Current Target Point of the Aim Assist System when performing Aim Assist on a Hoop */
	FVector CurrentTargetPoint;

	/** Pointer to the Camera Component of the Character that owns this component */
	class UCameraComponent* PlayerCameraComponent;

	/** Pointer to the Player Controller of the Character that owns this component */
	class APlayerController* OwnerPlayerController;

	/** Pointer to the RootComponent of the Character that owns this component */
	USceneComponent* PlayerRootComponent;

	/** Pointer to the Character that owns this component */
	class AHoopZCharacter* OwningCharacter;

	/** Actor type of the actor currently being targeted */
	uint8 ActorType;
```

Figure 38– HoopZAimAssistComponent header file

The comments above the functions in Figure 38 explain what their purpose is. Additionally, all the variables that are used are also present and properly commented.

The main entry point into the Aim Assist System is the SetTargetActor function. It gets sent a pointer to an actor that should be targeted, and an unsigned integer with 8 bits, that functionally acts as an enum, that identifies the type of the actor whose pointer has been sent.

The definition of the SetTargetActor function can be seen below, in Figure 39

```
void UHoopZAimAssistComponent::SetTargetActor(AActor* InActorBeingTargeted, uint8 InActorType)
{
    ActorBeingTargeted = InActorBeingTargeted;
    ActorType = InActorType;

    if (OwnerPlayerController == nullptr)
    {
        OwnerPlayerController = OwningCharacter->GetController<APlayerController>();
    }

    // From the moment we have an actor target, then the TickComponent function should be active, so we
    // begin aim assist functionalities
    PrimaryComponentTick.SetTickFunctionEnable(true);
}
```

Figure 39– SetTargetActor function definition

As we can see, the function simply stores the data that's being sent in, then checks if the OwnerPlayerController variable has a valid pointer stored already, and if it doesn't then it gets a valid pointer from the OwningCharacter variable. After that, it calls the SetTickFunctionEnable function in order to activate the component's TickComponent Function, pictured below in Figure 40

```
// Called every frame
void UHoopZAimAssistComponent::TickComponent(float DeltaTime, ELevelTick TickType, FActorComponentTickFunction* ThisTickFunction)
{
    Super::TickComponent(DeltaTime, TickType, ThisTickFunction);

    // While the TickComponent function is active, we should always be looking at the target, whichever it is
    LookAtTarget(DeltaTime);
}
```

Figure 40– TickComponent function definition

The TickComponent function, in turn, simply calls the LookAtTarget function and sends the DeltaTime as a parameter. The DeltaTime variable is simply used to store the amount of time that has gone by since the last time the TickComponent function was called. Its execution and calling are handled by the engine in the background, and nothing else needs to be done. Tick functions get called every frame that is calculated by the Game Loop.

```cpp
void UHoopZAimAssistComponent::LookAtTarget(float DeltaTime)
{
    if (ActorBeingTargeted != nullptr)
    {
        FVector TargetLocation;

        switch (ActorType)
        {
        //HOOP
        case 1:
        {
            //If we're targeting the hoop, then we need to obtain our target location from our CurrentTargetPoint variable
            TargetLocation = CurrentTargetPoint - FVector(0,0,100);
        }
        break;

        //BASKETBALL
        case 2:

        //CHARACTER
        case 3:
        {
            //If we're targeting a basketball or a character, then we need to obtain our target location from our ActorBeingTargeted variable
            TargetLocation = ActorBeingTargeted->GetActorLocation() - FVector(0, 0, 100);
        }
        break;

        default:
        {
        }
        break;
        }

        FRotator OldRotation = OwnerPlayerController->GetControlRotation();
        FRotator NewRotation = UKismetMathLibrary::FindLookAtRotation(PlayerRootComponent->GetComponentLocation(), TargetLocation);
        FRotator InterpolatedRotation = FMath::RInterpConstantTo(OldRotation, NewRotation, DeltaTime, CameraInterpolationSpeed);

        FRotator NewPlayerControlRotation = FRotator(0, 0, 0);
        NewPlayerControlRotation = InterpolatedRotation;

        //Make sure to zero out the Roll of the Rotation, since the character's feet should always be perpendicular to the floor
        NewPlayerControlRotation.Roll = 0;

        if (OwnerPlayerController)
        {
            OwnerPlayerController->SetControlRotation(NewPlayerControlRotation);
        }
    }
}
```

Figure 41– LookAtTarget function definition

The LookAtTargetFunction, displayed in Figure 41, is where the actual camera manipulation is performed. Since the camera is set to use the Pawn's Control Rotation as we can see in the HoopZCharacter constructor in Figure 33, changing the Control Rotation, which is stored in a Character's Player Controller, will also change the rotation of the camera.

In order to figure out what is our desired rotation of the camera, we first need to figure out what is the location that we're trying to look at. When the actor we're targeting is a hoop, this is done through the CurrentTargetPoint variable, which will be explained in the next chapter. When the actor we're targeting is a basketball or another character, then the location we're trying to look at is simply the location of these actors.

Then, after storing that location in the TargetLocation variable we can now use it to calculate the desired rotation of the camera. This is done through the FindLookAtRotation function from the UKismetMathLibrary, which is a library provided by the engine that supplies a wide variety of math related functionality. This function receives our current location, which we'll call A, and the location we want to look at, which we'll call B, and returns a rotation that, if applied to A, would look directly at B, which is exactly the behavior we're looking for. Now that we have the desired rotation, we simply need to replace our Control Rotation with it. However, this makes the camera movement appear too robotic and abrupt, so before doing this, we need to interpolate our desired rotation with our current rotation,

obtained from our current Control Rotation. This interpolation is calculated through the FMath::RInterpConstantTo function, which is yet another function that the Unreal Engine provides, and the results of its calculations can be influenced by the DeltaTime variable that was previously described and the CameraInterpolationSpeed variable. While the DeltaTime variable is supplied to us by the engine, through the TickComponent function, the CameraInterpolationSpeed can be set in the HoopZCharacter Blueprint class to control how fast the interpolation should work. Higher CameraInterpolationSpeed variable values mean faster interpolation, and lower values mean slower interpolation.

Once we no longer need the aim assist because the player is no longer pressing that input, then the Stop function gets called. Its definition can be viewed in Figure 42.

```
void UHoopZAimAssistComponent::Stop()
{
    // Clear the current target actor reference pointer.
    ActorBeingTargeted = nullptr;
    // Clear the actor_type byte
    ActorType = 0;
    // Disable the Tick function, so that the aim assist component no longer attempts to look at the target
    PrimaryComponentTick.SetTickFunctionEnable(false);
}
```

Figure 42– Stop function definition

This function simply clears all the data that was being used to perform the camera rotation, and also disables the TickComponent function, so that the LookForTarget functions is no longer called every frame.

5.3 Shooting System

The Shooting System intends to allow players to shoot the basketball, whether at the hoop to score points, or in any trajectory that they might desire. It's the main way the players interact with the game's central loop.

In order to ensure that this system is modular and can be reused in other ways in the future, possibly by the AI characters, an Actor Component was created, and all shooting functionality was developed within it. This actor component and its functions is then called by the HoopZCharacter class whenever the related inputs are pressed. This component's header and source code files are quite larger than the ones from the Aim Assist Component, so only the most important parts of them are going to be inspected in this document. Additionally, the Shooting system has a secondary responsibility, namely the creation and the rendering of the shot trajectory when the player is aiming the basketball, so this chapter will be divided into two sub-chapters, one devoted to the procedures related to the rendering of the projectile trajectory, and one devoted to the spawning and application of the previously rendered trajectory into a valid shot.

5.3.1 Projectile Trajectory Creation

The creation of the projectile trajectory, with which the player can control and predict the destination of the basketball when thrown, is initially triggered by player input. When the player chooses to aim the basketball, an initial force vector, with a default length, is instantiated. This vector, more specifically it's direction and length, is what the player input will influence, as it is the source data from which the trajectory for the basketball will be calculated. In Figures 43 and 44, the initial instantiation of this vector is shown. In Figures 47 and 48, the posterior manipulation of the force vector is displayed.

```
void AHoopZCharacter::PointBasketball()
{
    if (bHasBasketball)
    {
        PlayerActionState = EPlayerActionState::Player_Aiming;
        TrajectoryPlotPoints->SetVisibility(true);
        ShootingComponent->PointBasketball();
    }
}
```

Figure 43– HoopZCharacter's PointBasketball function definition

```
void UShootingComponent::PointBasketball()
{
    //Make sure we turn on the tick function
    PrimaryComponentTick.SetTickFunctionEnable(true);

    //Multiplier constant that interacts with the length of the AimForceVector. The higher this is, the larger the length of the vector.
    const int Multiplier = 750;

    // Assign the AimForceVector with the value of the camera's forward vector to guarantee it is always pointing ahead of the player.
    // Then, multiply it by the Multiplier constant. We do this because the forward vector is normalized, and thus of length 1.
    // If we didn't multiply it, the player would have to needlessly offset the ball trajectory forward whenever it was initially created.
    AimForceVector = CharacterRef->GetFirstPersonCameraComponent()->GetForwardVector() * Multiplier;

    // Then, we need to give the AimForceVector some vertical length, otherwise the initial ball trajectory just points straight to the ground,
    // again, needlessly forcing the player to offset it.
    AimForceVector += FVector(0, 0, 500);

    //Preemptively create the first iteration to give some time for the tick function to buffer in.
    CreateBallTrajectory();

    // Now we set bIsAiming to true, to control the execution flow of the Tick function.
    bIsAiming = true;
}
```

Figure 44– ShootingComponent's PointBasketball function definition

Figure 43 depicts the PointBasketball function's definition, which handles the input of the player first attempting to aim the basketball. It initially checks whether the player is in possession of the basketball, and if it is, it then alters the PlayerActionState variable, which is an Enumerator that details the current gameplay action that the player is engaged with, so that it properly shows that the player is engaged in aiming procedures. After, it turns the TrajectoryPlotPoints actor component, that communicates the visual representation of the shot trajectory to the player, visible. A pointer to this component was previously stored inside the Shooting Component when the HoopZCharacter actor was created, so the Shooting Component can interact with it from within its own functions. Finally, it calls the

PointBasketball function declared in the ShootingComponent class, the contents of which are displayed in Figure 44.

As we can see, this function starts by enabling the TickFunction of the component, much like the Aim Assist System does when the initial target actor is defined. It then properly instantiates the AimForceVector with a default value. The reasoning for this default value can be extracted from the comments above the associated code. This AimForceVector variable is the variable from which the aim trajectory of the basketball is then calculated inside the CreateBallTrajectory function. Figure 45 shows the contents of said function.

```
void UShootingComponent::CreateBallTrajectory()
{
    //We need to create the PathParams struct and then fill it with the details of the projectile path prediction we intend to perform
    FPredictProjectilePathParams PathParams = FPredictProjectilePathParams();
    PathParams.TraceChannel = ECollisionChannel::ECC_WorldDynamic;
    PathParams.bTraceWithChannel = true;

    //The BallThrowingLocationComponent should always be the origin of the prediction
    FVector ThrowingPositionVector = BallThrowingLocationComponent->GetComponentLocation();

    PathParams.StartLocation = ThrowingPositionVector;

    //The initial Launch Velocity should always be the AimForceVector, so the player can interact with the prediction in real time through input
    PathParams.LaunchVelocity = AimForceVector;

    //We want the path prediction to detect collisions, so we can detect the end position of the trajectory
    PathParams.bTraceWithCollision = true;

    FPredictProjectilePathResult PathResult = FPredictProjectilePathResult();

    //We now predict the projectile path according to the PathParams we have filled, and send a reference to the PathResult struct to
    //receive the results of the calculations
    UGameplayStatics::PredictProjectilePath(GetWorld(), *&PathParams, *&PathResult);

    //The resulting prediction includes an Array of path points that show the location of the projectile across time.
    TArray<FPredictProjectilePathPointData> PathData = PathResult.PathData;

    //We need to check if the PlotPoints variable is Hidden in game, and if it is, we set it to not be hidden any more
    if (PlotPoints->IsHidden())
    {
        PlotPoints->SetActorHiddenInGame(false);
    }

    //Activate plot points that show the player the primary trajectory the ball is going to take
    PlotPoints->SetPlotPointsPosition(PathData);

    //The resulting prediction also includes a FHitResult struct that stores the impact point of the prediction
    FHitResult HitResult = PathResult.HitResult;
    FVector ImpactPoint = HitResult.ImpactPoint;

    //Properly update the decal that shows the player the place the ball is going to hit
    AimLocationDecalComponent->SetWorldLocation(ImpactPoint);
    AimLocationDecalComponent->SetActive(true);
    AimLocationDecalComponent->SetVisibility(true);

    if (AActor* HitActor = HitResult.GetActor())
    {
        //Now we should check if the prediction detected a collision with a Hoop actor.
        if (AHoop* Hoop = Cast<AHoop>(HitActor))
        {
            FVector HoopHoleLocation = Hoop->GetHoopHoleLocation();

            //If it collided with a Hoop, then we should call the CheckIfNearGoodHoopPosition to change the material of the
            //plot points according to the distance between the HoopHoleLocation and the ImpactPoint.
            PlotPoints->CheckIfNearGoodHoopPosition(HoopHoleLocation, ImpactPoint);
        }
    }
}
```

Figure 45– CreateBallTrajectory function definition

The CreateBallTrajectory is split into two main parts. The first one takes care of supplying the PredictProjectilePath function, yet again supplied by Unreal Engine 5 through the UGameplayStatics library, with the parameters through which the projectile path prediction should be performed. This function then internally calculates the projectile path and returns an Array of the points in 3D vectorial space that the projectile will move through. This Array is then used as the basis for the second part of the function, which takes care of interacting with the PlotPoints pointer to draw the trajectory of the

basketball on the player's screen, color grading it to communicate to the player whether the trajectory is conducive to a successful shot, based on the distance between the end point of the predicted path and the hoop hole location. The class to which the PlotPoints pointer points to is called HoopZTrajectoryPlotPoints, and it merely defines 5 points that can be interacted with, as well as 3 materials that can be used to paint the points with different colors:

- A default color, for whenever the player isn't aiming at the hoop.
- A success color, for whenever the player is aiming the basketball at the hoop in what is deemed to be a good trajectory.
- A failure color, for whenever the player is aiming the basketball at the hoop in what is deemed to be a bad trajectory.

The Blueprint class for this actor can be viewed in Figure 50.

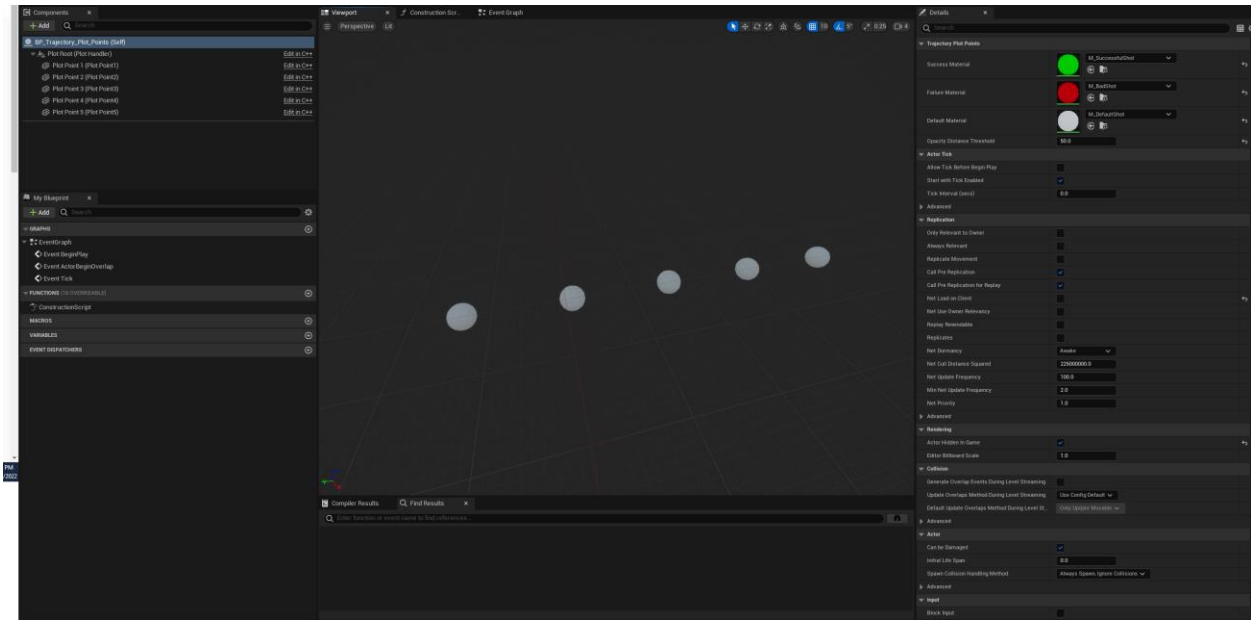


Figure 46– HoopZTrajectoryPlotPoints Blueprint Class

As we can see the 5 points that can be interacted with were made to display sphere meshes. This, however, is something that can be altered in the future, possibly by drawing a line rather than 5 separate spheres. For the purposes of this proof of concept, this approach deems acceptable results, as we'll see at the final part of this sub-chapter, when the final result of the implementation of the current system is shared.

First though, further information must be supplied related to how the player input, specifically the one that interacts with the AimVectorForce variable and thus the rendered trajectory, is collected and communicated to the trajectory system. In order to do so, the contents of Figures 51 and 52, shown below, need to be analyzed.

```
void AHoopZCharacter::LookUp(float Rate)
{
    if (!GetController()->IsLookInputIgnored())
    {
        if (PlayerActionState == EPlayerActionState::Player_Idle)
        {
            if (!AimAssistComponent->HasTarget())
            {
                AddControllerPitchInput(Rate);
            }
        }
        else if (PlayerActionState == EPlayerActionState::Player_Aiming)
        {
            FVector AmmountToAdd = (FirstPersonCameraComponent->GetForwardVector() + FirstPersonCameraComponent->GetUpVector()) * -Rate * AimAssistVerticalMouseSensibility;

            ShootingComponent->AddOffsetToAimForceVector(AmmountToAdd);
            AimAssistComponent->AddToTargetPoint(AmmountToAdd);
        }
    }
}
```

Figure 47– LookUp function definition

```
void AHoopZCharacter::LookRight(float Rate)
{
    if (!GetController()->IsLookInputIgnored())
    {
        if (PlayerActionState == EPlayerActionState::Player_Idle)
        {
            if (!AimAssistComponent->HasTarget())
            {
                AddControllerYawInput(Rate);
            }
        }
        else if (PlayerActionState == EPlayerActionState::Player_Aiming)
        {
            FVector AmmountToAdd = FirstPersonCameraComponent->GetRightVector() * Rate * AimAssistLateralMouseSensibility;

            ShootingComponent->AddOffsetToAimForceVector(AmmountToAdd);
            AimAssistComponent->AddToTargetPoint(AmmountToAdd);
        }
    }
}
```

Figure 48 – LookRight function definition

Figures 47 and 48 show the definition of functions LookUp() and LookRight(), which deal with player camera input in the vertical and horizontal axis, respectively. Depending on the PlayerActionState variable they either move the player's camera or interact with the AimVectorForce variable, inside the Shooting Component.The Rate variable is collected directly from the mouse, or the controller, of the player, and represents how quickly the player motion is, in either axis. Then, the AimAssistVerticalMouseSensibility and AimAssistLateralMouseSensibility constants, which are defined in the HoopZCharacter Blueprint class, can be used to tweak how quickly or slowly the changes to the AimVectorForce variable are. As we can see the AmmountToAdd variable is calculated and then sent to the ShootingComponent class through the AddOffsetToAimForceVector function, the contents of which are self-explanatory, as well to the AimAssistComponent class through the AddToTargetPoint function. This function communicates to the Aim Assist Component that its target point, which is the variable that controls the location that the player camera must be rotated towards when the player is aiming at a hoop, must be updated to reflect the new shot trajectory

The final result of this system can be viewed in Figure 49.

Figure 49– Final result of the Projectile Trajectory System

As is shown in Figure 49, the player can accurately ascertain both the trajectory that the basketball will take at any point of the process, as well as quickly understand whether the trajectory can be translated into a successful shot, if the plot points are colored green, or a failed one, if the plot points are colored red. This distinction between good shots or bad ones is further detailed in the next sub-chapter, as well as the situations where the Shooting System might see it fit to alter the trajectory that the basketball will take, after being spawned.

5.3.2 Basketball Spawning and Trajectory Application

The previous chapter detailed how the player character made use of the Shooting Component to render and displace the predicted basketball's path based on the value of the AimForceVector variable, and how this variable could be affected by player input. The current chapter will make use of the knowledge imparted unto the reader on the previous chapter and analyze how the results of the path creation inform the trajectory of the basketball, once the shooting inputs are supplied by the player.

The starting point of this journey is in the HoopZCharacter's class ShootBasketball function, whose content can be viewed in Figure 50.

```
void AHoopZCharacter::ShootBasketball()
{
    if (bHasBasketball)
    {
        switch (PlayerActionState)
        {
            case EPlayerActionState::Player_Aiming:
            {
                ShootingComponent->ShootBasketball();
                TrajectoryPlotPoints->SetVisibility(false);
                PlayerActionState = EPlayerActionState::Player_Idle;
            }
            break;

            case EPlayerActionState::Player_ModifyingShot:
            {
                ShootingComponent->ShootBasketballStraightLine();
                PlayerActionState = EPlayerActionState::Player_Idle;
            }
            break;

            default:
            break;
        }
    }
}
```

Figure 50– HoopZCharacter's ShootBasketball function definition

This function, as can be extracted from the contents of Figure 50, uses the contents of the PlayerActionState variable, in much the same way as the Projectile Trajectory Creation system detailed in the previous sub-chapter, to control the flow of the code. If the player is aiming, then it calls the ShootBasketball function from the ShootingComponent to spawn and direct a basketball in a previously calculated trajectory. If the player is pressing the ShotModifier input however, the function calls the ShootBasketballStraightLine from the ShootingComponent to spawn and direct a basketball across a straight line starting from the player's viewpoint. Both of them then reset the PlayerActionState variable to its default value, indicating the player, after shooting the basketball previously in his possession, is now in an idle state. The contents of both functions related to shooting the basketball are detailed in Figure 51 below.

```
void UShootingComponent::ShootBasketball()
{
    FVector ShotLocation = BallThrowingLocationComponent->GetComponentLocation();
    FVector DecalComponentLocation = AimLocationDecalComponent->GetComponentLocation();
    ServerShootBasketball(ShotLocation, ShotStrength, CurrentShotDifficulty, CurrentlyTargetedHoop, DecalComponentLocation);
    StopPointingBasketball();
}

void UShootingComponent::ShootBasketballStraightLine()
{
    FVector ShotLocation = BallThrowingLocationComponent->GetComponentLocation();
    ServerShootBasketballStraightLine(ShotLocation, ShotStrength);
}
```

Figure 51– ShootBasketball and ShootBasketballStraightLine function definitions

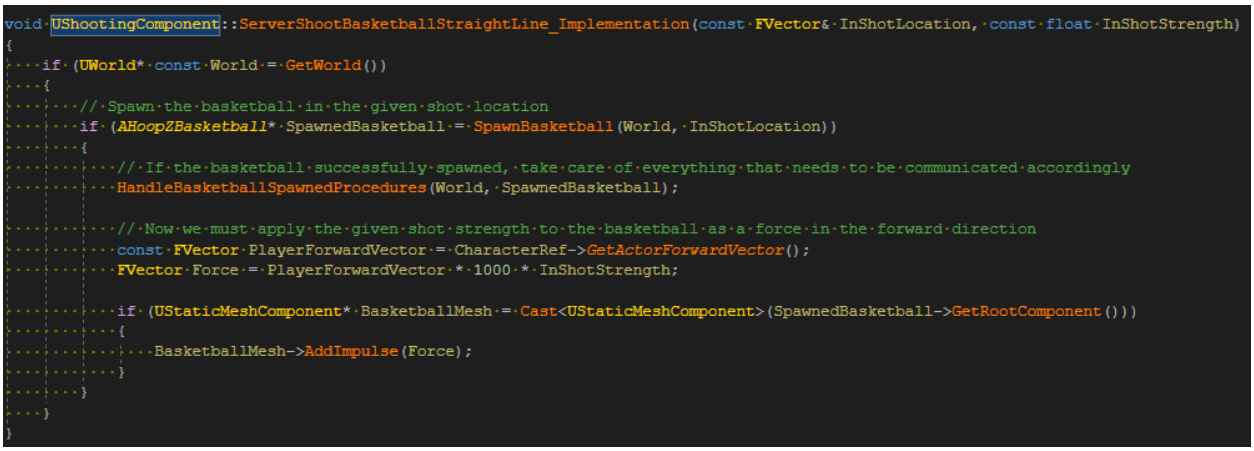
The figure above marks the first usage of Server Remote Procedure Calls (RPCs) in this document this far. The particularities of RPCs were presented to the reader in Chapter 4.1.3.1. The reason they're used in this particular system is because basketballs are one of the most, if not the most, important actors as far as gameplay goes, and as such we can't allow clients to have any authority over their spawning, because that would lead to a very exploitable possibility of mischievous players being able to cheat. Additionally, client players can't replicate variables to other clients, only servers have this capability. That means that even if client players were allowed to spawn the basketball, the other clients would be totally unaware of their creation, and only the client that spawned the basketball could see it being rendered in their in-game world.

As such, by instructing the Server to spawn the basketball and to take care of all shooting procedures, every connected client player will receive the replicated information related to the newly created basketball actor at roughly the same time, assuming all clients possess the same amount of ping.

The two RPC functions, ServerShootBasketball and ServerShootBasketballStraightLine, share a lot of functionality, as can be seen in Figures 52 and 53, but the latter doesn't need to calculate any sort of optimal shot trajectory based on the shot variables sent as arguments, so its inner workings are quite simpler.

```
void UShootingComponent::ServerShootBasketball_Implementation(const FVector& InShotLocation, const float InShotStrength,
                                                              const float InShotDifficulty, AActor* InNewLastActorLookedAt,
                                                              const FVector& InDecalComponentLocation)
{
    if (UWorld* const World = GetWorld())
    {
        // Spawn the basketball in the given shot location
        if (AHoop2Basketball* SpawnedBasketball = SpawnBasketball(World, InShotLocation))
        {
            // If the basketball successfully spawned, take care of everything that needs to be communicated accordingly
            HandleBasketballSpawnedProcedures(World, SpawnedBasketball);

            // Now we must apply a shot trajectory to the spawned basketball
            ApplyShotTrajectory(World, SpawnedBasketball, InNewLastActorLookedAt, InDecalComponentLocation, InShotStrength, InShotDifficulty);
        }
    }
}
```

Figure 52 – ServerShootBasketball RPC function definition



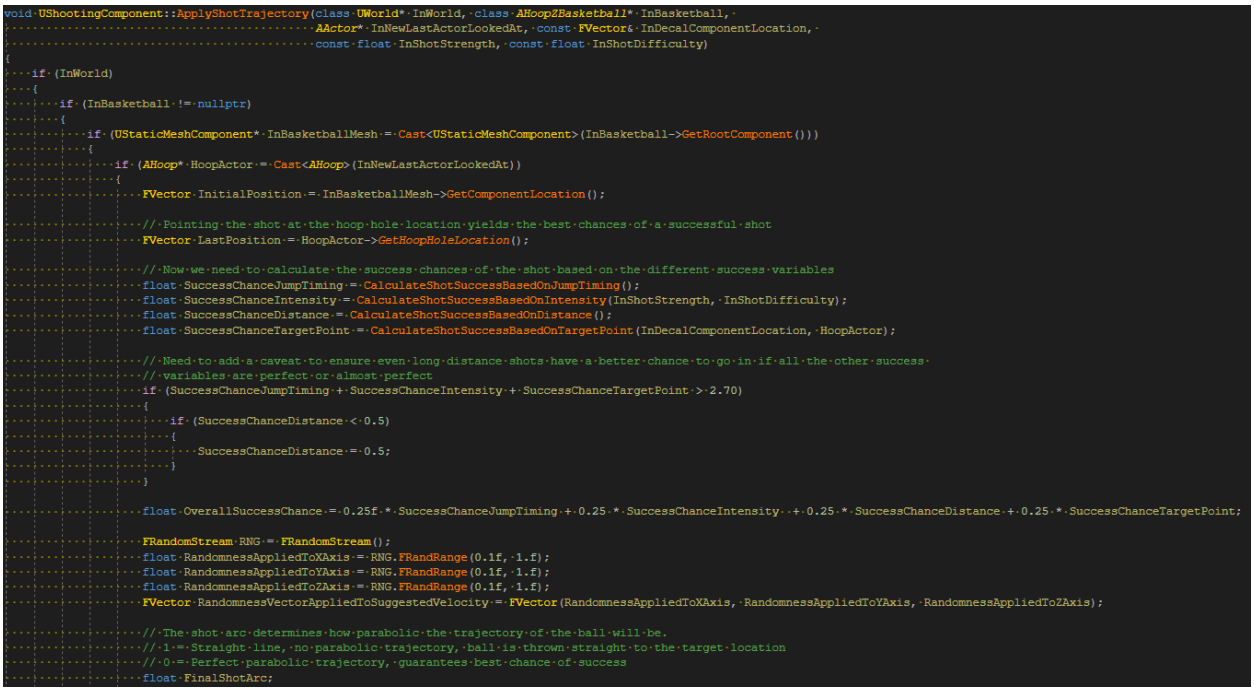
```
void UShootingComponent::ServerShootBasketballStraightLine_Implementation(const FVector& InShotLocation, const float InShotStrength)
{
    if (UWorld* const World = GetWorld())
    {
        // Spawn the basketball in the given shot location
        if (AHoopZBasketball* SpawnedBasketball = SpawnBasketball(World, InShotLocation))
        {
            // If the basketball successfully spawned, take care of everything that needs to be communicated accordingly
            HandleBasketballSpawnedProcedures(World, SpawnedBasketball);

            // Now we must apply the given shot strength to the basketball as a force in the forward direction
            const FVector PlayerForwardVector = CharacterRef->GetActorForwardVector();
            FVector Force = PlayerForwardVector * 1000 * InShotStrength;

            if (UStaticMeshComponent* BasketballMesh = Cast<UStaticMeshComponent>(SpawnedBasketball->GetRootComponent()))
            {
                BasketballMesh->AddImpulse(Force);
            }
        }
    }
}
```

Figure 53 – ServerShootBasketballStraightLine RPC function definition

As we can see, both functions make use of the SpawnBasketball and HandleBasketballSpawnedProcedures functions. The former takes care of spawning the HoopZBasketball actor, setting the necessary data within it – like the player that performed the shot, or the location from where the shot was triggered – and resetting the variables inside the HoopZCharacter actor that control player possession of the basketball, while the latter takes care of handling all of the procedures that inform the HoopZGameDirector and the HoopZGameState classes that a basketball was spawned, so that they can, for example communicate to the players that the game is now in progress or that the Shot Clock was reset. The ServerShootBasketballStraightLine then simply uses the InShotLocation and InShotStrength variables to add an impulse to the Basketball actor that was just spawned. The ServerShootBasketball RPC, however, calls the ApplyShotTrajectory instead, that uses the following data in order to calculate and apply an appropriate shot trajectory:

- World – Pointer to the UWorld object, that represents the game level where the game session is happening
- SpawnedBasketball – Pointer to the newly created Basketball actor
- InNewLastActorLookedAt – Pointer to the last actor that the trajectory creation system collided with, normally a Hoop actor
- InDecalComponentLocation – Location of the Decal Component, indicating where the player thinks the basketball is being aimed at
- InShotStrength – Float variable indicating the shot strength that the player placed into the shot
- InShotDifficulty – Float variable indicating the distance of the player to the InNewLastActorLooketAt, previously calculated when the shot trajectory was created.

The way all this information is used is present in Figures 54,55 and 56, all of them presenting the contents of the ApplyShotTrajectory function, divided into three Figures for an easier viewing.



```
void UShootingComponent::ApplyShotTrajectory(class UWorld* InWorld, class AHoopBasketball* InBasketball,
                                             AActor* InNewLastActorLookedAt, const FVector& InDecalComponentLocation,
                                             const float InShotStrength, const float InShotDifficulty)
{
    if (InWorld)
    {
        if (InBasketball != nullptr)
        {
            if (UStaticMeshComponent* InBasketballMesh = Cast<UStaticMeshComponent>(InBasketball->GetRootComponent()))
            {
                if (AHoop* HoopActor = Cast<AHoop>(InNewLastActorLookedAt))
                {
                    FVector InitialPosition = InBasketballMesh->GetComponentLocation();

                    //Pointing the shot at the hoop hole location yields the best chances of a successful shot
                    FVector LastPosition = HoopActor->GetHoopHoleLocation();

                    //Now we need to calculate the success chances of the shot based on the different success variables
                    float SuccessChanceJumpTiming = CalculateShotSuccessBasedOnJumpTiming();
                    float SuccessChanceIntensity = CalculateShotSuccessBasedOnIntensity(InShotStrength, InShotDifficulty);
                    float SuccessChanceDistance = CalculateShotSuccessBasedOnDistance();
                    float SuccessChanceTargetPoint = CalculateShotSuccessBasedOnTargetPoint(InDecalComponentLocation, HoopActor);

                    //Need to add a caveat to ensure even long distance shots have a better chance to go in if all the other success
                    //variables are perfect or almost perfect
                    if (SuccessChanceJumpTiming + SuccessChanceIntensity + SuccessChanceTargetPoint > 2.70)
                    {
                        if (SuccessChanceDistance < 0.5)
                        {
                            SuccessChanceDistance = 0.5;
                        }
                    }

                    float OverallSuccessChance = 0.25f * SuccessChanceJumpTiming + 0.25 * SuccessChanceIntensity + 0.25 * SuccessChanceDistance + 0.25 * SuccessChanceTargetPoint;

                    FRandomStream RNG = FRandomStream();
                    float RandomnessAppliedToXAxis = RNG.FRandRange(0.1f, 1.f);
                    float RandomnessAppliedToYAxis = RNG.FRandRange(0.1f, 1.f);
                    float RandomnessAppliedToZAxis = RNG.FRandRange(0.1f, 1.f);
                    FVector RandomnessVectorAppliedToSuggestedVelocity = FVector(RandomnessAppliedToXAxis, RandomnessAppliedToYAxis, RandomnessAppliedToZAxis);

                    //The shot arc determines how parabolic the trajectory of the ball will be.
                    //1 = Straight line, no parabolic trajectory, ball is thrown straight to the target location
                    //0 = Perfect parabolic trajectory, guarantees best chance of success
                    float FinalShotArc;
```

Figure 54 – ApplyShotTrajectory function definition (1/3)

```
if (OverallSuccessChance >= 0.875f)
{
    //If the overall success chance is in the [90, 100] range, then we shouldn't apply any randomness to it
    //and instead just use the suggested Velocity
    RandomnessVectorAppliedToSuggestedVelocity = FVector::ZeroVector;
    FinalShotArc = 0.3f;
}
else if (OverallSuccessChance < 0.9f && OverallSuccessChance >= 0.7f)
{
    RandomnessVectorAppliedToSuggestedVelocity *= 25;
    FinalShotArc = 1 - (OverallSuccessChance * 0.8f);
}
else if (OverallSuccessChance < 0.7f && OverallSuccessChance >= 0.55f)
{
    RandomnessVectorAppliedToSuggestedVelocity *= 100;
    FinalShotArc = 1 - (OverallSuccessChance * 0.90f);
}
else if (OverallSuccessChance < 0.55f && OverallSuccessChance >= 0.35f)
{
    RandomnessVectorAppliedToSuggestedVelocity *= 150;
    FinalShotArc = 1 - (OverallSuccessChance * 0.95f);
}
else
{
    RandomnessVectorAppliedToSuggestedVelocity *= 300;
    FinalShotArc = 1 - OverallSuccessChance;
}
```

Figure 55 – ApplyShotTrajectory function definition (2/3)

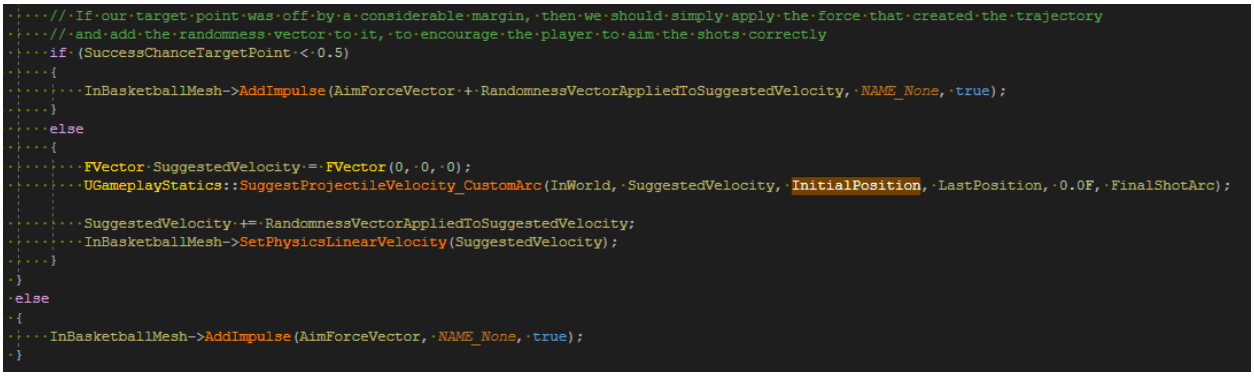


Figure 56 – ApplyShotTrajectory function definition (3/3)

Figure 54 shows the initial collection of the success chances of the shot across the following criteria, previously defined in chapter 4.1.1.2:

- Jump Timing – How close to the apex of the jump the player was when the shot was released.
- Shot Intensity – Whether the correct amount of power was applied by the player into the shot, according to how far away the player was from the hoop
- Distance – Regardless of the shot intensity, shots from farther away are always less probable to be successful than shots from closer to the hoop
- Target Point Location – The location of the Decal Component, representing where the player chose to aim the shot. This is the main difference between the bad shots and good shots dichotomy initially defined in the end of the previous chapter.

The functions calculating these success chances all return a float variable that ranges from 0 to 1.

The contents of these variables are then assigned a weight, currently 0,25 so that all the success chances have the same importance, and are then compounded into the OverallSuccessChance variable. This variable is ultimately the deciding factor on how successful the shot is. In order to granularly interact with the shot, a Vector variable, called RandomnessVectorAppliedToSuggestedVelocity, is instantiated and its value along each axis is provided by a Random Number Generator. Finally, the FinalShotArc float variable is also defined. This variable controls how parabolical the trajectory of the basketball will be in its flight to the hoop. The more parabolical the trajectory, the higher the physical chance of the basketball entering the hoop without colliding with the rim, and so the higher the chance of a point being scored.

On to Figure 55, we see how the OverallSuccessChance variable is used to affect both the RandomnessVectorAppliedToSuggestedVelocity and the FinalShotArc variables. The values affecting both inside each if clause, as well as the conditions that govern the entrance into the if clauses themselves, are completely arbitrary, and were obtained through the playtesting of the system while it was being constructed. Better values for these are not only possible but outright probable and can mainly be obtained through prolongated play sessions with human play testers within the game's target audience. They do, however, provide an acceptable starting point.

In Figure 56, the final part of the function can be viewed. The system evaluates the value of the SuccessChangeTargetPoint variable, to ascertain if the player was aiming at an acceptable location. If the player wasn't, and the success rate as pertains to the location of the Target Point was less than 0.5, then

the shot is deemed a bad shot and the system simply adds impulse to the basketball actor using the AimForceVector, that created the predicted trajectory with which the player aimed the shot, adding the RandomnessVectorAppliedToSuggestedVelocity to it. This is made to incentivize the player to aim the basketball properly, by punishing him or her when their aiming is subpar.

If the players aim is deemed as acceptable, however, then we use the SuggestProjectileVelocity_CustomArc, provided by the UGameplayStatics library, to determine the optimal velocity the basketball should have to reach the hoop, using the FinalShotArc variable. Then we add to this velocity the RandomnessVectorAppliedToSuggestedVelocity vector. If the shot was deemed as a good one, with an overall success chance of over 0.9, then this vector will have length zero.

Finally, this resulting velocity is then applied to the basketball actor.

Although the final result of this system can't be properly evaluated through static images, Figure 57 shows the execution of a successful point, resulting from a perfect shot across all success chance variables.

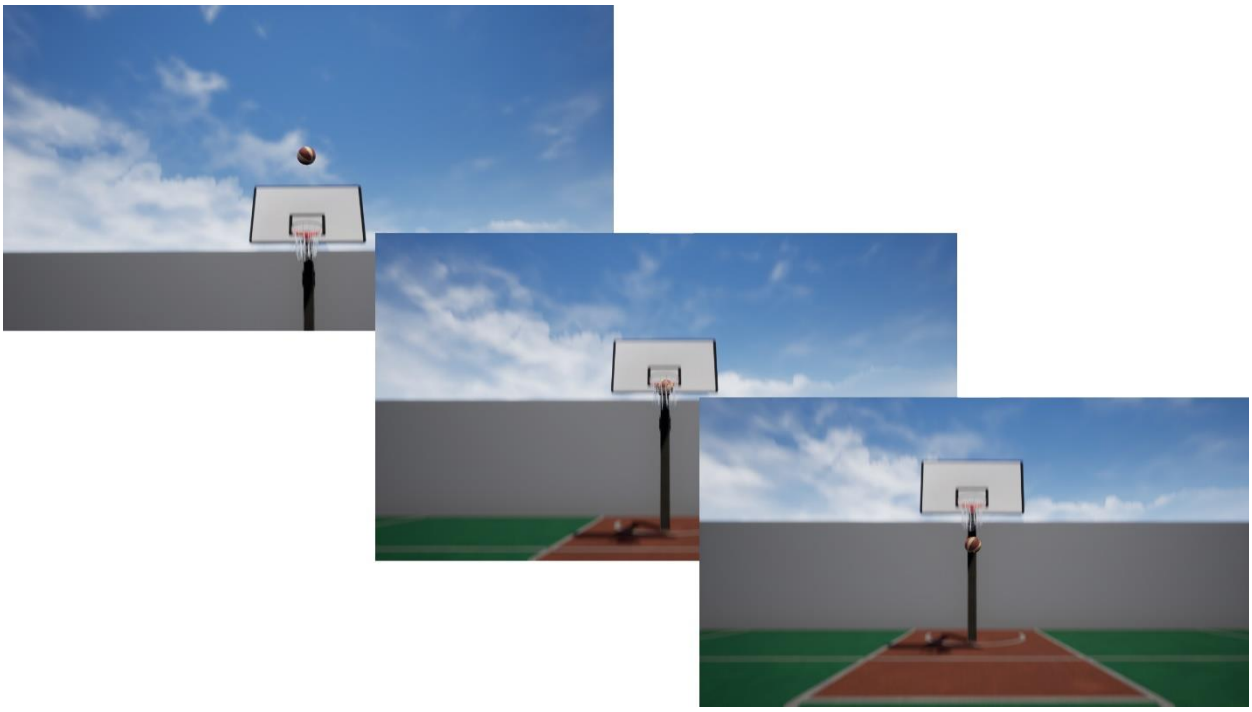


Figure 57 – Successful shot montage

5.4 Dashing and Ball Stealing System

The current chapter intends to present the reader the work that was put into the game's dashing and ball stealing systems. These two systems are part of the game's movement system. In order to ensure that this system is modular and can be reused in other ways in the future, possibly by the AI characters, an Actor Component was created, and all movement functionality was developed within it. This actor component and its functions is then called by the HoopZCharacter class whenever the related inputs are pressed.

As far as the dashing system goes, the main entry point into the system is the HoopZCharacter class's Dash function, that is called whenever the player presses appropriate input. This function can be viewed in Figure 58 below

```
void AHoopZCharacter::Dash()
{
    MovementManagerComponent->Dash();
}
```

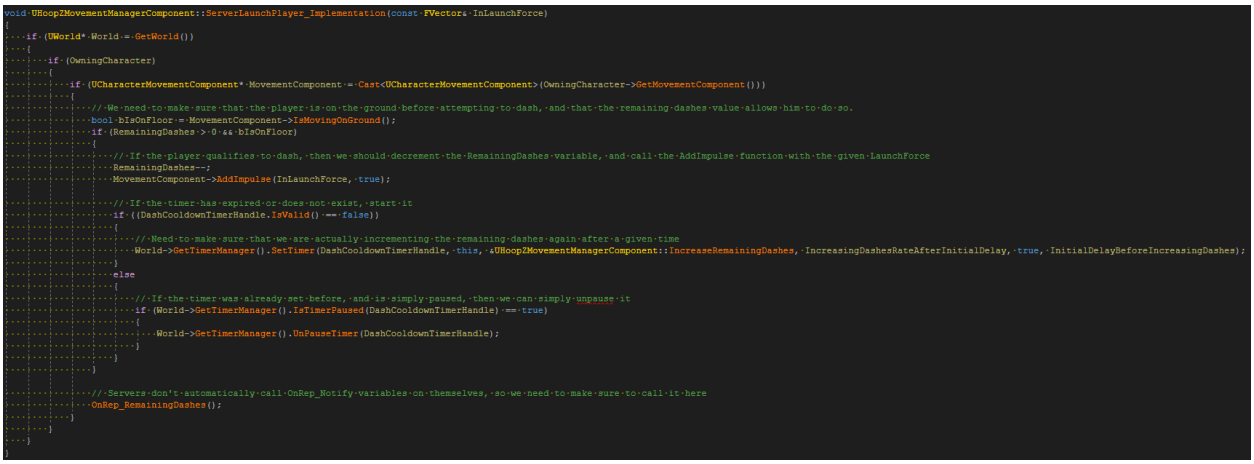
Figure 58 – HoopZCharacter Dash Function definition

As we can see, the function simply calls the Dash function from the component in charge of the movement managing. In Figure 59, we can see the contents of that function.

```
void UHoopZMovementManagerComponent::Dash()
{
    if (OwningCharacter)
    {
        if (OwningCharacter->GetPlayerActionState() == EPlayerActionState::Player_Idle)
        {
            FVector PlayerDirection = OwningCharacter->GetVelocity();
            PlayerDirection.Z = 0;
            FVector DashVector = PlayerDirection * DashForce;
            ServerLaunchPlayer(DashVector);
        }
    }
}
```

Figure 59 – HoopZMovementManagerComponent Dash Function definition

The system firstly ensures that the player is in the correct action state to be able to trigger a dash, and then obtains the vector information about the players velocity at that given time. Then, after setting the Z axis to zero, making sure that the dash movement can't be performed in a vertical direction, the DashVector variable is initialized with the previously obtained player direction vector, multiplied by the DashForce constant variable. This variable is exposed to the HoopZCharacter Blueprint class, to allow quick tweaking of its value in order to achieve the best result. Then, the DashVector variable is sent as a parameter to the ServerLaunchPlayer RPC function, the contents of which can be viewed in Figure 60



```
void UHoopZMovementManagerComponent::ServerLaunchPlayer_Implementation(const FVector& InLaunchForce)
{
    if (UWorld* World = GetWorld())
    {
        if (OwningCharacter)
        {
            if (UCharacterMovementComponent* MovementComponent = Cast<UCharacterMovementComponent>(OwningCharacter->GetMovementComponent()))
            {
                //We need to make sure that the player is on the ground before attempting to dash, and that the remaining dashes value allows him to do so.
                bool bIsOnFloor = MovementComponent->IsMovingOnGround();
                if (RemainingDashes > 0 && bIsOnFloor)
                {
                    //If the player qualifies to dash, then we should decrement the RemainingDashes variable, and call the AddImpulse function with the given LaunchForce
                    RemainingDashes--;
                    MovementComponent->AddImpulse(InLaunchForce, true);

                    //If the timer has expired or does not exist, start it
                    if (!(DashCooldownTimerHandle.IsValid() == false))
                    {
                        //Need to make sure that we are actually incrementing the remaining dashes again after a given time
                        World->GetTimerManager().SetTimer(DashCooldownTimerHandle, this, &UHoopZMovementManagerComponent::IncreaseRemainingDashes, IncreasingDashesRateAfterInitialDelay, true, InitialDelayBeforeIncreasingDashes);
                    }
                    else
                    {
                        //If the timer was already set before, and is simply paused, then we can simply unpause it
                        if (World->GetTimerManager().IsTimerPaused(DashCooldownTimerHandle) == true)
                        {
                            World->GetTimerManager().UnPauseTimer(DashCooldownTimerHandle);
                        }
                    }
                }

                //Servers don't automatically call OnRep_Notify variables on themselves, so we need to make sure to call it here
                OnRep_RemainingDashes();
            }
        }
    }
}
```

Figure 60 – ServerLaunchPlayer RPC function definition

This function makes sure that the player is on the floor and checks the value of the replicated variable RemainingDashes, which tracks how many dashes the player can still do at any given time. If the player is on the ground and has enough dashes, then the DashVector variable previously described will be used to apply an impulse into the player character, performing the dash. A timer is then used to make sure that the RemainingDashes variable is replenished after a set duration controlled by the IncreasingDashesRateAfterInitialDelay and InitialDelayBeforeIncreasingDashes variables, that are, yet again, exposed to the HoopZCharacter Blueprint class to allow for quick iteration on their optimal values.

This concludes the implementation of the Dashing mechanic.

The Ball Stealing System exists as a by-product of the impulse that was applied to the player character.

The entry point into this system is the HoopZMovementManagerComponent class's OnCharacterHit function, that controls what should happen when a collision event is detected on the player character's capsule component. This function and its contents are visible in Figure 61, pictured below.

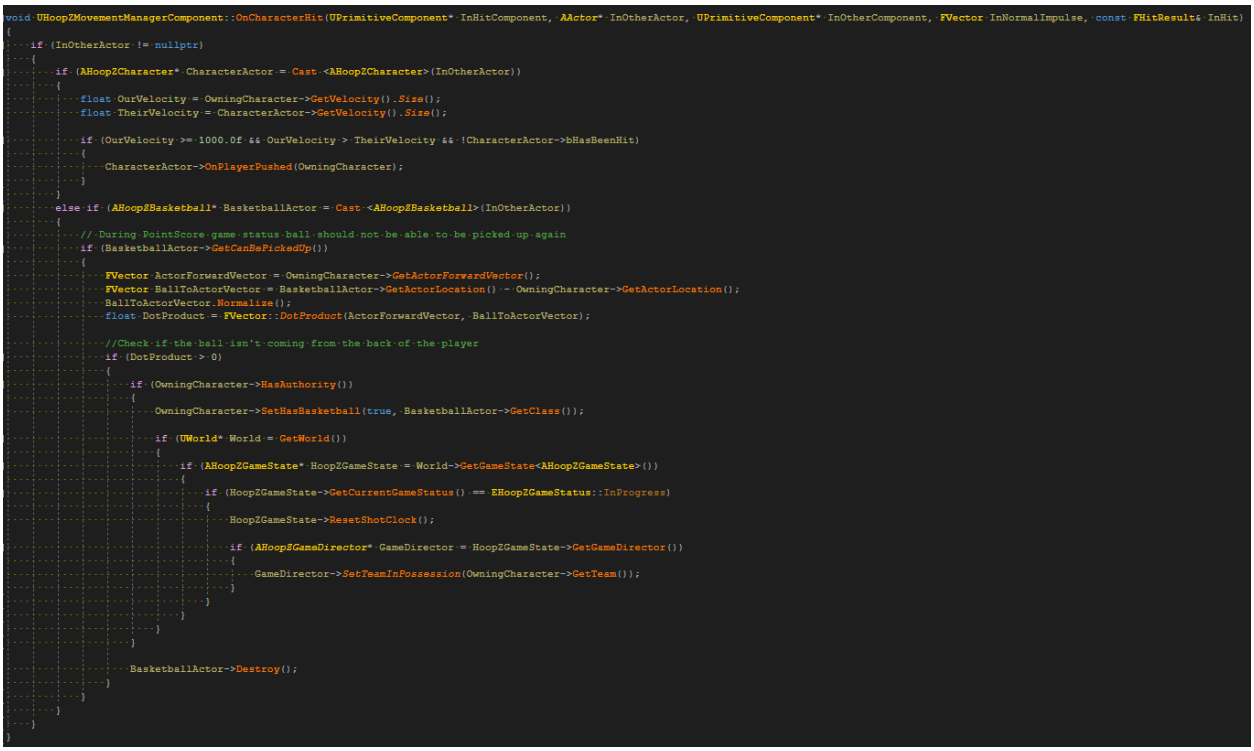


Figure 61 – OnCharacterHit function definition

As we can see, this function's responsibility is two-fold. It is tasked with detecting whether the capsule component collided with a basketball actor, in which case it updates all of the required variables in regards to player possession of the ball and resetting the shot clock accordingly, and it is also tasked with detecting if the collision was with another HoopZCharacter instead. If this is the case, then it checks the velocities of each player involved in the collision, and in the event that the player with the faster velocity is also the one to which this movement manager component belongs to, then the OnPlayerPushed function will be called on the other player. The OnPlayerPushed function will then take care of informing the other player's movement manager component that it is being pushed, as can be seen in Figure 62.

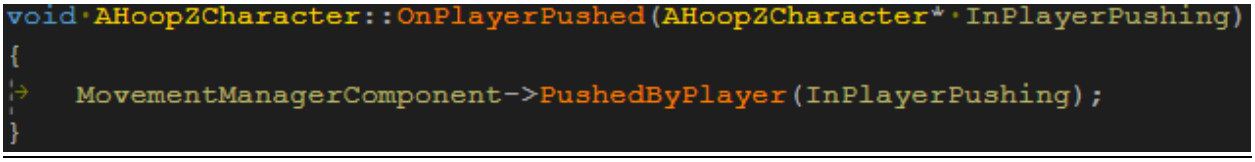


Figure 62 – OnPlayerPushed function definition

Which in turn, calls the ServerPushedByPlayer RPC, in Figure 63, to take care of pushing the player server-side. The contents of this RPC can be viewed in Figure 64

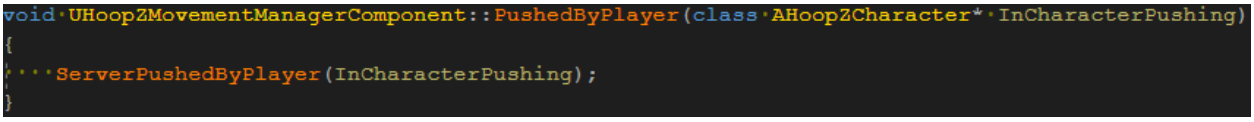


Figure 63 – PushedByPlayer function definition

```
void UHoopZMovementManagerComponent::ServerPushedByPlayer_Implementation(class AHoopZCharacter* InPlayerPushing)
{
    if (InPlayerPushing)
    {
        if (!bHasBeenHit)
        {
            bHasBeenHit = true;
            if (UCharacterMovementComponent* MovementComp = Cast<UCharacterMovementComponent>(OwningCharacter->GetMovementComponent()))
            {
                FVector CurrentLocation = OwningCharacter->GetActorLocation();
                FVector PlayerPushingLocation = InPlayerPushing->GetActorLocation();
                FVector ImpactDirection = CurrentLocation - PlayerPushingLocation;
                FVector ImpulseToAdd = FVector(ImpactDirection.X, ImpactDirection.Y, 10.0f);
                ImpulseToAdd += (ImpactDirection * PushForceMultiplier);
                MovementComp->AddImpulse(ImpulseToAdd, true);

                //Update the PlayerDownedMode, so that the animation blueprint can show the appropriate animation
                PlayerDownedMode = EPlayerDownedMode::PlayerFallingDown;

                //Client will already run this function, but the server actually needs to call it
                OnRep_PlayerDownedMode();

                //Check if the player had the basketball
                if (OwningCharacter->GetHasBasketball())
                {
                    //If it did, then it needs to be dropped, effectively stealing it
                    OwningCharacter->DropBasketball(ImpactDirection);
                }

                //Force the Net Driver to update, to make sure that this information is communicated as soon as possible
                OwningCharacter->ForceNetUpdate();
            }
        }
    }
}
```

Figure 64 – ServerPushedByPlayer RPC function definition

This function is the final stop of the ball stealing system. It calculates the push direction by obtaining the location of both the actors involved, and then creates the ImpulseToAdd vector, multiplying it by the PushForceMultiplier variable, exposed to the HoopZCharacter Blueprint class, to obtain the final force to be applied to the player being pushed. Then it sets the PlayerDownedMode enumerator variable accordingly, to signal to the Blueprint Animation system that it should play the player falling down animation. Then, if the player is currently in possession of the basketball, it calls the DropBasketball function in the HoopZCharacter class so that the basketball can be dropped in a given direction, effectively stealing it from the player.

The visual representation of final result of this system can be gathered from the image montage present in Figure 65.

Figure 65 – Player Push montage

As we can observe, the player is initially in an upright stance, then falls to the ground, landing on his back, and then rolls over to his chest in order to display the getting up animation, that isn't displayed in the montage.

5.5 Main Menu and HUD

The present chapter intends to present to the reader the work that was put into the game's Main Menu system, as well as the Cinematic Camera Manager that powers its dynamic cutscenes, called Sequences by Unreal Engine, from one ***Widget*** to another.

The initial screen with which the player is greeted is the result of the following sequence.

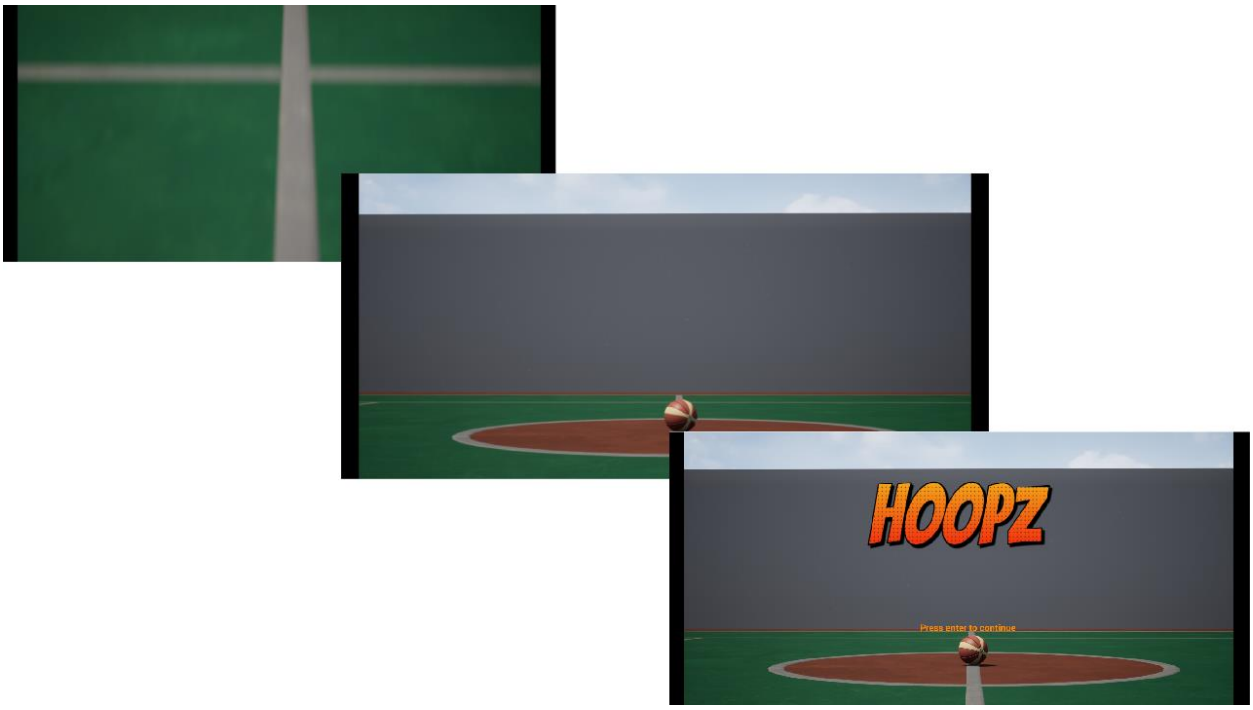


Figure 66 – Splash screen sequence montage

Then, upon pressing the enter button, the following sequence plays

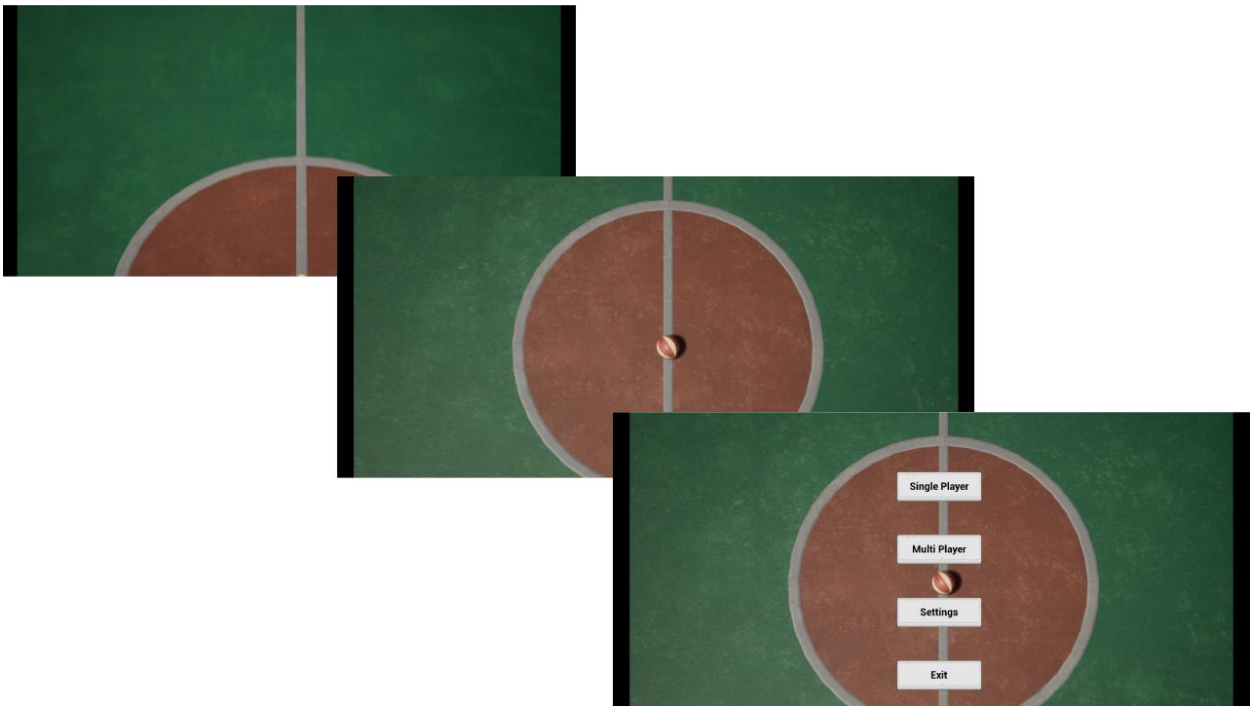


Figure 67 – Main Menu sequence montage

The player can then either choose to enter single player or multi player mode. The sequences in charge of directing the player to their chosen mode's widget are the portrayed next.

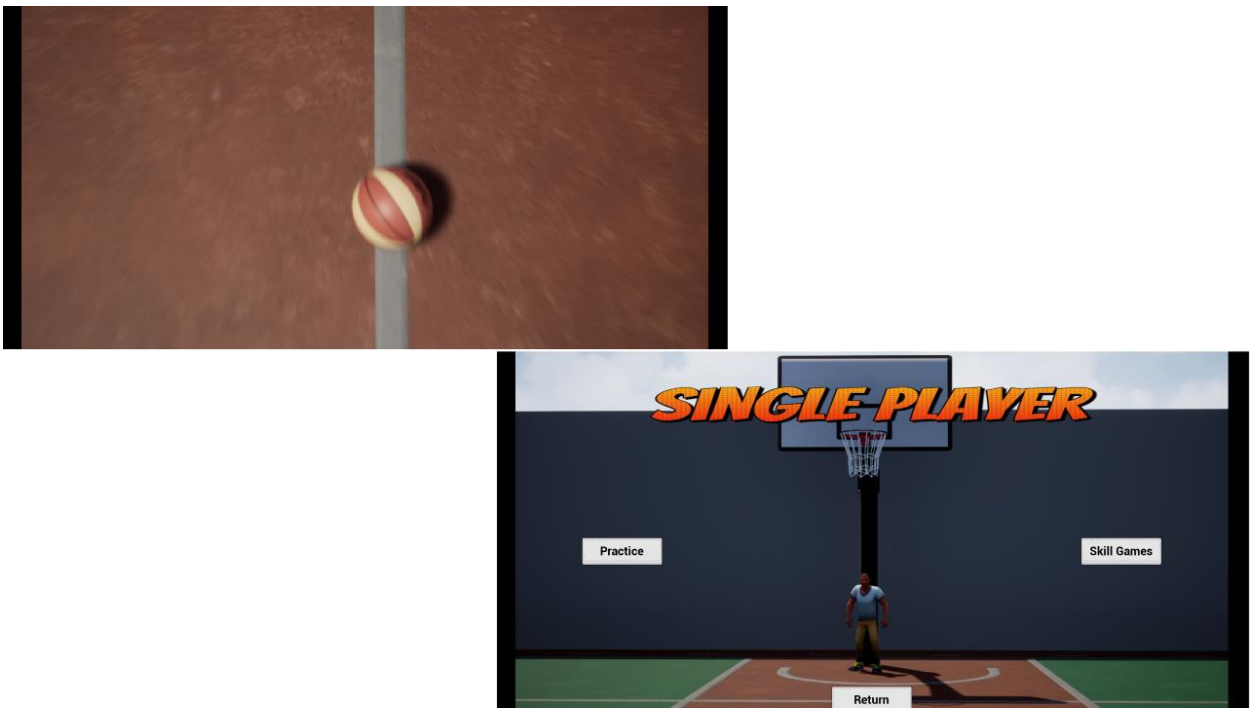


Figure 68 – Single player sequence montage

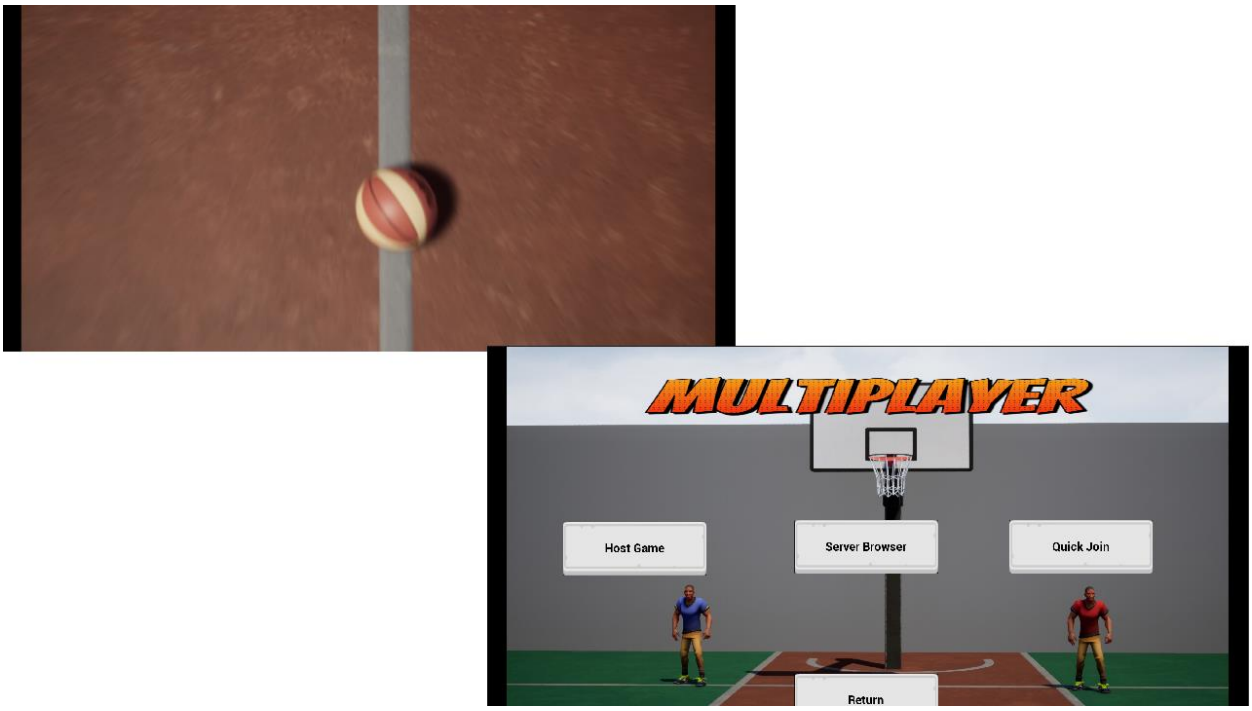


Figure 69 – Multiplayer sequence montage

Pressing the return button, visible on the lower center of the screen, after either mode's sequence ends returns the player, through a reversed version of the sequence, back to the main menu.

The Multiplayer choice screen, additionally, shows the following canvases when pressing the "HostGame" and "Server Browser" buttons, respectively

Figure 70 – Host Game Screen

Figure 71 – Server Browser Screen

The development of this dynamic main menu system mainly resorted to the creation of the individual sequences, which were triggered by either a button press, in the case of interaction with the ***widgets*** in the various canvases, or by the pressing of the enter button when the player was in the Splash screen ***widget.*** Each one of these events communicates with the HoopZCinematicCameraManager actor class, that then takes care of triggering the adequate sequences. This is mainly done through the following functions:

```
//·Called·to·begin·a·transition·to·a·given·state,·at·the·end·of·which·a·user·widget·should·be·added·to·the·viewport
UFUNCTION(BlueprintCallable)
void·BeginTransitionToState(ECineCameraState·InNewCineCameraState,·TSubclassOf<class·UHoopZUserWidget>·InPendingUserWidget);

//·Called·from·the·sequence·to·trigger·the·end·of·the·transition·and·add·the·pending·widget·to·the·viewport
UFUNCTION(BlueprintCallable)
void·EndTransitionToPendingState();
```

Figure 72 – Cinematic camera function declarations

```
void·AHoopZCinematicCameraManager::BeginTransitionToState(ECineCameraState·InNewCineCameraState,·TSubclassOf<class·UHoopZUserWidget>·InPendingUserWidget)
{
	//·We·can't·allow·transitions·into·the·same·cine·camera·state
	if·(InNewCineCameraState·!=·CurCineCameraState)
	{
		CurCineCameraState·=·InNewCineCameraState;

		if·(UWorld*·World·=·GetWorld())
		{
			ULevelSequence*·Cinematic·=·nullptr;

			switch·(InNewCineCameraState)
			{
				case·ECineCameraState::Splashscreen:
				{
					Cinematic·=·MainMenuCinematics[0];
				}
				break;
```

Figure 73 – BeginTransitionToState function definition(1/2)

```
    if (InPendingUserWidget != nullptr)
    {
        PendingUserWidget = InPendingUserWidget;
    }

    if (Cinematic != nullptr)
    {
        CreateLevelSequencePlayer(Cinematic);
    }
```

Figure 74 – BeginTransitionToState function definition(2/2)

```
void AHoopZCinematicCameraManager::EndTransitionToPendingState()
{
    if (OwnerController)
    {
        if (PendingUserWidget)
        {
            if (AHoopZHUD* HUD = OwnerController->GetHUD<AHoopZHUD>())
            {
                HUD->ShowMenuWidget(PendingUserWidget);
            }
        }
    }
}
```

Figure 75 – EndTransitionToPendingState function definition

In Figure 72, we can see the declaration of the main functions of the HoopZCinematicCameraManager class. BeginTransitionToState receives an enumerator variable of type ECineCameraState, that indicates what sequence we're attempting to play, and also receives the widget that should be displayed at the end of the sequence, through the InPendingUserWidget variable. EndTransitionToPendingState has no parameters.

In Figure 73, we can see the definition of the first part of the BeginTransitionToState function. The InNewCineCameraState variable is evaluated to make sure it's not a duplicate, and if not, it is stored. Then, depending on its value, a sequence will be chosen to be played, from the MainMenuCinematics array that is exposed to the Blueprint class of Camera Manager, and will be stored inside the Cinematic variable. As we can see in Figure 74, displaying the second part of the function, both the InPendingUserWidget and the Cinematic variables are null pointer checked, and then the CreateLevelSequencePlayer function is called. This function simply creates a LevelSequencePlayer actor, that actually has the functionality to play sequences through the player's camera.

Those sequences, when they end, call the EndTransitionToPendingState function, displayed in Figure 75, that simply shows the PendingUserWidget that was set when the transition begun.

As pertains to the player HUD, Figure 76 displayed below analyzes all of the different elements of the HUD that are presented to the player

Figure 76 – Game HUD overlay (1/2)

There are three UI elements that need to be mentioned that are present in Figure 76. All of them are outlined in red and have numbered arrows pointing to them. The first one, on the top of the screen, is the Scoreboard ***widget***. It serves several purposes, mimicking the way that scoreboards function in a basketball match in real life. First and foremost, it displays the score of each team, so that players know if they're winning, losing, or in a standstill in the case of a draw. Then, it displays the remaining time in the match, 3 minutes and 52 seconds in this case, so that players can know roughly how much time they still have to play. Below that, it displays the remaining time in the shot clock. In the game situation displayed in the figure, no team has possession of the basketball, so the shot clock will always display the maximum possible number of seconds, 15, but as soon as a player from any team picks up the ball, this number will begin to decrement. If it ever reaches 0, the game will be paused, and possession will be assigned to the opposing team.

The second one, located in the center of the screen, is the crosshair ***widget.*** The main purposes it serves are to give the player a location to direct his or her eyes during times of chaos, and to serve as a default reference point whenever the player is in possession of the ball and decides to aim the basketball. During the development of the project, it was found that omitting this element would lead some people to unconsciously experience a stronger sense of motion sickness, so its inclusion is a must have for the comfort of the player.

The third one, located in the bottom right corner of the screen, is the remaining dashes ***widget***. The main purpose it serves is to accurately communicate to the player the number of dashes that he or she can still perform. This widget is updated whenever a dash is performed and whenever a dash is replenished.

There is still one other HUD element that needs to be mentioned, and it's functioning can be viewed in Figure 77, displayed below

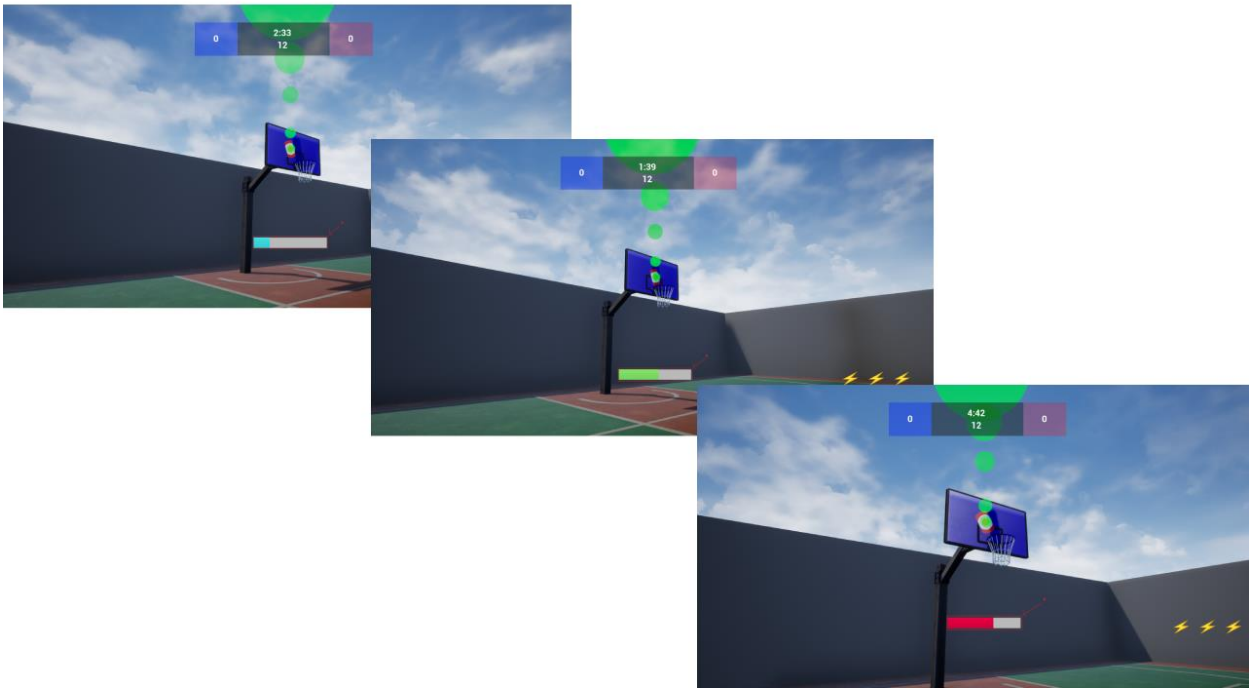


Figure 77 – Game HUD overlay (2/2)

The HUD element in question is the shot power indicator bar, that exists to communicate to the player the shot strength that they are applying to any shot that they attempt. The shot strength is one of the most important variables in the shooting system, so this element is of extreme importance. As we can see, the contents of the bar are colored differently depending on whether the shot power is less than necessary, inside the optimal range, or more than necessary. The blue, green and red colors were chosen because they are easily distinguishable between themselves, and also ease player information recognition and retention, due to their universal element. It is also important to notice that, in an effort to prevent the player HUD from being needlessly cluttered with elements, the shot power indicator bar is only visible while the player is attempting a shot, becoming invisible immediately after the player shoots the ball.

6 Evaluation

6.1 Evaluation Parameters

The current chapter intends to present to the reader the parameters by which the success of the implementation was evaluated. The main source of data that led to the contents of this chapter was the realization of play sessions with a group of play testers, all playing the same build of the project. Players engaged in either 1v1 or 2v2 matches and were then asked to answer a questionnaire with questions related to their experience with the game. Players were encouraged to play more than one session if they desired to do so. In total, 37 distinct questionnaires and their answer data was collected.

Being that the main focus of this proof-of-concept delivery is to determine the practical potential of a basketball videogame in a First-Person Perspective within a multiplayer setting, the gauges by which this potential is measured will always be, at least in some part, relative to the user. Players that don't normally play or even like sports games will tend to not be as excited about the gameplay as players that regularly play sports games, regardless of the sport itself.

Efforts were made, however, to include within the questionnaires questions that were independent of videogame genre preference, and that intended solely to evaluate the experience of the players on a technical perspective.

The questions that were presented to the play testers were as follows:

1. How regularly do you play videogames?
2. What is your favorite videogame genre?
3. How many matches did you play in the play session you participated in?
4. What game mode did you play?
5. How would you rate your experience with HoopZ?
6. How would you rate the originality of the gameplay?
7. How would you rate the accessibility of the gameplay?
8. How would you rate the aiming mechanics?
9. How would you rate the shooting mechanics?
10. How would you rate the movement mechanics?
11. How would you rate the game's art style?
12. How would you rate the intuitiveness of the game controls?
13. How adequate was the match duration?
14. Would you play HoopZ again?
15. Do you have any constructive feedback regarding any aspect of the game? If so, what is it?

Questions 5, 6, 7, 8, 9, 10, 11 and 12 asked the play testers to provide an answer in the range of 1 to 10. Question 1 provided the following options:

- Never played any
- Yearly
- Monthly

- Weekly
- Daily

Question 2 provided the following options:

- Action-Adventure
- Racing
- Sports
- Simulation
- MMORPGs
- MOBAs
- First Person Shooters
- Other

Question 3 provided the following options:

- 1
- 2
- 3
- Other

Question 4 provide the following options:

- 1v1
- 2v2
- Both

Question 13 provided the following options:

- Too short
- Too long
- Just right

Question 14 provided the following options:

- Yes
- No

6.2 Evaluation Results

The current chapter displays the data obtained from the questionnaires, described in the previous chapter, collected from the play testers.

On the first question “How regularly do you play videogames?”:

- No one answered that they never played any videogames
- 2 people answered they played videogames yearly
- 5 people answered they played videogames monthly
- 19 people answered they played videogames weekly
- 11 people answered they played videogames daily

On the second question “What is your favorite videogame genre?”:

- 7 people answered they prefer Action-Adventure games
- 4 people answered they prefer Racing games
- 7 people answered they prefer Sports games
- 1 person answered they prefer Simulation games
- 2 people answered they prefer MMORPGs
- 6 people answered they prefer MOBA games
- 10 people answered they prefer First Person Shooter games
- No one chose any other type of games

On the third question “How many matches did you play in play session you participated in?”:

- 3 people answered they played 1 match
- 10 people answered they played 2 matches
- 16 people answered they played 3 matches
- 8 people answered they played more than 3 matches

On the fourth question “What game mode did you play?”:

- 2 people answered they only played 1v1 matches
- 21 people answered they only played 2v2 matches
- 14 people answered they played both games modes

On the fifth question “How would you rate your experience with HoopZ?”:

- 4 people rated their experience a 6 out of 10
- 2 people rated their experience a 7 out of 10
- 13 people rated their experience an 8 out of 10
- 11 people rated their experience a 9 out of 10
- 7 people rated their experience a 10 out of 10

On the sixth question “How would you rate the originality of the gameplay?”:

- 10 people rated the originality of the gameplay an 8 out of 10
- 11 people rated the originality of the gameplay a 9 out of 10
- 16 people rated the originality of the gameplay a 10 out of 10

On the seventh question “How would you rate the accessibility of the gameplay?”:

- 4 people rated the accessibility of the gameplay a 3 out of 10
- 6 people rated the accessibility of the gameplay a 5 out of 10
- 11 people rated the accessibility of the gameplay a 6 out of 10
- 5 people rated the accessibility of the gameplay a 7 out of 10
- 7 people rated the accessibility of the gameplay an 8 out of 10
- 4 people rated the accessibility of the gameplay a 9 out of 10

On the eight question “How would you rate the aiming mechanics?”:

- 2 people rated the aiming mechanics a 4 out of 10
- 4 people rated the aiming mechanics a 5 out of 10
- 8 people rated the aiming mechanics a 6 out of 10
- 11 people rated the aiming mechanics a 7 out of 10
- 8 people rated the aiming mechanics an 8 out of 10
- 4 people rated the aiming mechanics a 9 out of 10

On the ninth question “How would you rate the shooting mechanics?”:

- 1 person rated the shooting mechanics a 5 out of 10
- 5 people rated the shooting mechanics a 6 out of 10
- 4 people rated the shooting mechanics a 7 out of 10
- 10 people rated the shooting mechanics an 8 out of 10
- 8 people rated the shooting mechanics a 9 out of 10
- 9 people rated the shooting mechanics a 10 out of 10

On the tenth question “How would you rate the movement mechanics?”:

- 5 people rated the movement mechanics a 4 out of 10
- 8 people rated the movement mechanics a 5 out of 10
- 14 people rated the movement mechanics a 6 out of 10
- 9 people rated the movement mechanics a 7 out of 10
- 1 person rated the movement mechanics an 8 out of 10

On the eleventh question “How would you rate the game’s art style?”:

- 6 people rated the game’s art style a 4 out of 10
- 9 people rated the game’s art style a 5 out of 10
- 12 people rated the game’s art style a 6 out of 10
- 10 people rated the game’s art style a 7 out of 10

On the twelfth question “How would you rate the intuitiveness of the game controls?”:

- 5 people rated the intuitiveness a 4 out of 10
- 6 people rated the intuitiveness a 5 out of 10
- 5 people rated the intuitiveness a 6 out of 10
- 9 people rated the intuitiveness a 7 out of 10
- 7 people rated the intuitiveness an 8 out of 10
- 5 people rated the intuitiveness a 10 out of 10

On the thirteenth question “How adequate was the match duration?”:

- 8 people rated the duration of the match as Too Short
- 2 people rated the duration of the match as Too Long
- 27 people rated the duration of the match as Just Right

On the fourteenth question “Would you play HoopZ again?”:

- 35 people answered Yes
- 2 people answered No

On the fifteenth question ”Do you have any constructive feedback regarding any aspect of the game? If so, what is it?” most people chose to not add any feedback. However, the ones that did all focused on the following points:

- The aiming system didn’t allow for easy aiming of the basketball when the player is on either side of the hoop, instead of the front of the hoop.
- There were no sound effects for the basketball when players were dribbling, or when the ball collided with the ground, which detracted from the immersion of the experience
- There was no tutorial or screen that showed what the controls of the game were, so people could only ask the developer whenever they had any doubts on what button did what
- The animations for the player movement were unrealistic
- There is no animation for when the player is shooting the basketball, so players had a hard time predicting when other players were attempting a shot.

7 Conclusion

The current chapter intends to present the final conclusions that have been gathered after developing all the work that was presented to the reader throughout this document. Within it, considerations about the project and the success of its development are going to be discussed.

Videogames are an art form. The good ones at least. They contain within them a piece of the soul of the people that make them. They are the result of the incredibly difficult labor of people that devote themselves everyday to tell stories that haven't been told or couldn't be told in any other way. They make us laugh, they make us mad, they make us cry. From the moment the very first pixel graced the display of the very first monitor, videogames were an inescapable certainty. As we place ourselves firmly inside the temporal adulthood of the medium as an art form, it becomes important to realize that, like all art forms, the only path forward is that of innovation, of renewal, of rebirth. Only then can videogames continue to bring joy and happiness into the world, connecting players of different backgrounds, different countries, different ethnicities, and different walks of life in their joint love for all things videogames.

HoopZ is a humble attempt by a one-man developer team to achieve this lofty goal. Creativity is only a quarter of the problem. Execution is much, much more difficult. Using the knowledge attained during 3 years as a professional game developer, and 25 years as a gamer, I have tried my best and worked my hardest to execute a small part of the vision that led to the creation of this project. It does not constitute a full game by no means, but rather a proof of concept that shows that a full game is not only possible, but probably very successful if its development is given the time and attention that it deserves.

Some aspects of it work precisely as intended, while some others still need some more work to feel just right. The data that was gathered in chapter 6, although originating from an admittedly small sample size, can already point towards some of the projects best and worst characteristics. The originality of the gameplay and the execution of the shooting and aiming systems, for instance, constitute a resolute success in establishing a valid experience that people enjoy playing. On the other hand, aspects like the art style and the accessibility of the gameplay require further attention, mostly due to the constraints of time that this project was subject to. But overall, the fact that over 94% of the people that played it felt the desire to play it again is an unequivocal testament of the intrinsic value of this project, and guarantee that it's worthwhile to keep working on it in order to see what the final version of it might look like.

For future work, endeavors such as improving the game's animations and visuals, as well as adding more mechanics and game modes are a must. Possibly, the game could even venture out and attempt not only to simulate basketball as it stands in the world today, but also other, sillier versions of it, like adding environmental hazards such as trapdoors or cannons, or pitting more than two teams against one another, maybe with only one hoop that moves around the map. Maybe even some sort of MOBA hybrid, where player characters have different powers that can interact with each other in novel ways while still having to worry about placing the basketball inside of the hoop. It is also instrumental to the success of any prospective final version of the game to add some sort of character and level creation

toolset, to allow the players the chance to create their own avatars and their own maps, with their own rules and intricacies. The possibilities are truly limitless.

As a final note, it is my honest belief that this project can and should be used as a stepping stone into something greater. If nothing else, just for the fun of treading new, untouched land in the realms of videogame design. I can only hope that the work speaks for itself.

References

Anderson, J.C., Jain, D.C. and Chintagunta, P.K. (1993), “Customer value assessment in business markets”, Journal of Business-to-Business Marketing, Vol. 1 No. 1, pp. 3-29.

Gamestop, 2022a. [Online]

Available at: https://www.gamespot.com/articles/dark-souls-3-has-sold-over-10-million-copies-pushi/1100-6477468/

[Accessed 6th of February 2022]

Gro¨nroos, C. (1997), “Value-driven relationship marketing: from products to resources and competencies”, Journal of Marketing Management, Vol. 13 No. 5, pp. 407-19.

Metacritic, 2022a. [Online]

Available at: https://www.metacritic.com/game/playstation-5/nba-2k22

[Accessed 20 February 2022]

Metacritic, 2022b. [Online]

Available at: https://www.metacritic.com/game/pc/dark-souls-iii

[Accessed 20 February 2022]

Metacritic, 2022c. [Online]

Available at: https://www.metacritic.com/game/playstation-4/uncharted-the-lost-legacy

[Accessed 20 February 2022]

Metacritic, 2022d. [Online]

Available at: https://www.metacritic.com/game/pc/counter-strike-global-offensive

[Accessed 20 February 2022]

Metacritic, 2022e. [Online]

Available at: https://www.metacritic.com/game/playstation-4/fifa-22

[Accessed 20 February 2022]

BusinessOfApps, 2022a. [Online]

Available at: https://www.businessofapps.com/data/fortnite-statistics/

Nicola, S., Ferreira, E. P. & Ferreira, J. P., 2012. A novel framework for modeling value for the customer, an essay on negotiation. International Journal of Information Technology \& Decision Making, 11(3), pp. 661-703.

Wilson, D.T. and Jantrania, S. (1995), “Understanding the value of a relationship”, Asia-Australia Marketing Journal, Vol. 2 No. 1, pp. 55-66.

Oxland, Kevin (2004). Gameplay and design, pp. 240

Bethke, Eric (2003). Game development and production.