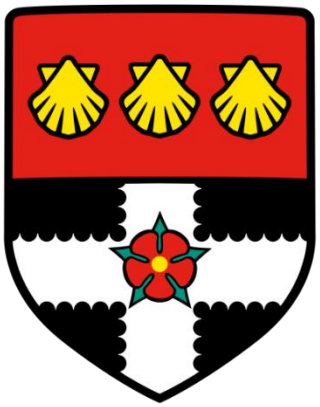


University of Reading

Department of Computer Science

Nightmare –
Multiplayer Networked Horror Game

Aadil Sattar

Supervisor: Shuang-Hua Yang

A report submitted in partial fulfilment of the requirements of

the University of Reading for the degree of

Bachelor of Science in Computer Science

April 23, 2024

Declaration

I, Aadil Sattar, of the Department of Computer Science, University of Reading, confirm that this is my own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly. I give consent to a copy of my report being shared with future students as an exemplar. I give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Aadil Sattar

April 23, 2024

Abstract

This report focuses on the development of a multiplayer networked game called “Nightmare,” designed to explore advanced game mechanics within Unity’s game development framework. The project aimed to create an immersive horror game environment that supports real-time interaction among players over a network, exploring both the technical challenges and player experience enhancements needed to make it work. Research was done on Multiplayer games and Horror games, followed by games in the combined Multiplayer Horror genre, which is a relatively new genre and has been on the rise due to its growth in popularity. The development employed Agile methodologies, focusing on iterative design, continuous integration, and regular testing phases to ensure functionality and performance. The core methods included the utilization of the Mirror networking library for managing complex player interactions and maintaining consistent game state across networked clients, a critical factor given the real-time horror setting. Key results from the project include successful implementation of networked player synchronization, where player inputs and interactions were seamlessly managed across different sessions without perceptible lag or discrepancies. Furthermore, testing validated this, with the final user test coming back very positively. The user test also allowed a basis for future work, and what work still needs to be done on the game before it could be released to public.

Table of Contents

Chapter 1 – Introduction

1.1 – Background

The project involves the development of a multiplayer networked horror game designed to provide an immersive and interactive experience for players. Combining elements of horror and real-time multiplayer interaction, taps into both the rising popularity of horror video games and the expanding capabilities of networked gaming.

The project draws on several key theories and applications such as:

- Psychological Engagement – Horror games leverage theories relating to enhancing player engagement and fear response with regards to psychological theories.

- Unity's Networking Framework – uses Unity's Multiplayer and Networking framework, here ensure smooth and responsive gameplay across different network conditions. This supports synchronous player interactions and real-time game state updates.

- Game Mechanics in Multiplayer Settings – incorporating cooperative elements along with competitive mechanics that require the players to both work together and against each other, while navigating horror elements. I draw on these theories to make the project work.

1.1.1 – Motivation behind Project

The motivation behind developing a multiplayer networked horror game stems from several key observations and trends:

- Growing Popularity of Multiplayer Experiences
- Integration of Social Elements in Gaming
- Increased Interest in Horror as a Genre

I also had the personal reason to creating the project, as I have enjoyed a lot of multiplayer horror experiences lately, and wanted to research how they are made myself.

1.1.2 – Target Audience

The horror game genre has a big variety of audiences that they can be catered for, due to it having such variety in term of styles and subgenres. psychological thrillers that tap into deep, unsettling fears to more straightforward jump-scare-based games that aim for immediate shock value. Consequently, the target audience is quite diverse.

When designing this game, the target audience I had in mind was teenagers and above, so ages 12+, to allow for jumpscares while also keeping the game accessible for all. This involves me needing to balance scare factor, complexity and narrative themes to cater for both the younger audience and anything above.

1.2 – Problem Statement

Despite the growing popularity of multiplayer networked games, many suffer from issues related to scalability, player engagement, and network latency, which can detract from user experience and game performance. This problem relates to me as I personally am an avid gamer and have played games that are online and available.

1.3 – Aims and Objectives

1.3.1 – Aims

The aim of my project is to develop a multiplayer networked immersive horror game environment with a robust infrastructure that ensures a seamless, synchronized gaming experience for all players. I also just want to make a game that is fun to play in the end.

1.3.2 – Objectives

The required objectives to make this work are as follows:

- Robust Multiplayer Server which allows for at least 2 players.
- Each player should have items they can use.
- Have a suitable user interface.
- Have a suitable menu.
- Player/Ghost animations networked for everyone to see
- Ghost Indicators that alert players to nearby supernatural activity.
- Interactive gameplay used for enhancing player immersion.
- Engaging and exploratory design of game world, to encourage exploration through use of rich environments and hidden secrets.
- Death Mechanics, where consequences are added for players death, such as ghost mode or limited respawns to maintain game tension.
-

1.4 – Game Overview and Approach

1.4.1 – Game Overview

“Nightmare” is a multiplayer networked game designed to combine elements of strategy, exploration, and survival within a haunted setting. The game challenges players to navigate through dynamically changing environments, interact with AI-driven ghosts, and cooperate with and compete against other players to achieve the main objective, get rid of as many ghosts as possible. This project aims to explore innovative mechanics in networked game environments and assess player engagement and interaction within these complex systems.

1.4.2 – Approach to Development

My approach involved first doing research on different game mechanics, and then implementing under the Agile Methodology. This Agile Framework allowed me to be constantly iterating the design and mechanics of the game, ensuring progressive refinement. By utilising constant sprint reviews and planning sessions, I was able to make quick adjustments based on real time feedback.

1.4.3 – Approach to Testing

My testing methodology mainly consisted of the network performance to optimise connectivity and responsiveness. I also made sure to get user feedback, which I was able to look at and it provided valuable insights that informed further development and enhancements of the game. This feedback loop was crucial in fine-tuning the gameplay experience to meet user expectations and improve overall satisfaction.

1.4.4 – Approach to Ethics

In the development of my game, ethics played a central role due to the interactive and networked nature of the game. I wanted to game to sport fair balanced gameplay, with little chance to cheat. I wanted to ensure a fair competitive environment for all players.

Data Privacy and security also played a role in how I designed the game to be ethical. Within the game, no data is collected. Only option data is collected using the Google Form surveys I have. Within these surveys, no personal data is collected, and it is all about the game.

1.5 – Organisation of the Report

My report is organised in to 8 chapters, including the one you are currently reading, Chapter 1 being the introduction. Chapter 2 details the literature review, where I review from historical horror and networked games to modern networked horror game design. Chapter 3 highlights the methodology I figured out and followed. Chapter 4 contains the implementation detail of the game I made, including code snippets. Chapter 5 shows off the resultant game, in screenshots. Chapter 6 delves into the resultant game, discussing and analysing it against the objectives. Chapter 7 Includes a conclusion and future works. Chapter 8 is the reflection, looking at personal gains.

Chapter 2 – Literature Review

This literature review explores the development and integration of multiplayer networked games, with a focus on the horror genre. Starting from the inception of video games, we trace the key technological advancements that have shaped this industry. The review emphasizes how multiplayer elements have revolutionized the horror gaming experience, moving from isolated scares to interactive, shared encounters. We also examine the impact of major game engines like Unity and Unreal in creating immersive environments. The goal is to understand the evolution of multiplayer horror games and to anticipate future trends in this engaging field.

2.1 – Historical Context

In this section, I talk about the history of games, along with the introduction of different aspects of gaming, especially relating to multiplayer integration and introducing the horror genre.

2.1.1 – Introduction to Video Games and Horror Games

Video games have been around for a long time, starting as simple creations, but have evolved in to one of the biggest industries in the world. It is estimated that in 2022 alone, it made a whopping 347 billion U.S. dollars in revenue [1]. Dating back all the ways to the late 1940s, they have always been a place of experimentation and innovation, scaling from small singleplayer 2D environments, to 3D. The horror genre of video games is not new either, with attempts dating back to the early 1970s, when a haunted house game was included with the first video game console, which even included some multiplayer attributes.

This constantly changing atmosphere will be the focus of this section of the literature review, where I talk about the historical context relating to video games, with a particular focus on the horror genre of games, expanding on the multiplayer and further talk about networking within that.

2.1.2 – Technological Advancements Impacting Game Development

Video games and their development is generally tied to the technological capabilities of hardware at the time. The first ever “video game” for example, was developed to use the radar displays, also known as cathode-ray tubes [3]. Using the radar screens was them using the best technology that they had at the time. However, it is not regarded as the first ever video game, as it did not run on a computing device. The first “real” video game came a bit over a decade later, in 1958, with the release of “Tennis for Two”.

“Tennis for Two” was released in October of 1958 by a physicist and technician for Brookhaven National Laboratory's annual public exhibition and was said to have been the most popular exhibit there [4]. It introduced a way for two people to play against each other, making it also the first multiplayer video game. The whole system was made up of a small analog computer with an oscilloscope screen, and each player used a knob to change the angle at which the ball was hit. The physicist, William Higinbotham, who worked on it, saw this interest in the interactive display, but didn’t think much of it, and the equipment that was used for the project was repurposed for other means.

The next big leap in video games happened in 1962, when the Massachusetts Institute of Technology got their hands on the DEC PDP-1 minicomputer. Using this computational device, Computer Scientist Steve Russell collaborated with a group of others to make the very first computer game. This game is known as “SpaceWar”, shown on figure 1. The significance of this game is paramount and gave way to a lot of iterations of code, designed by programmers

worldwide. This is due to the fact that systems, like the DEC PDP-1 minicomputer, were becoming increasing available around the world. However, there was a big problem, the price. US$120,000 (equivalent to $1,200,000 in 2023) is how much the system would have cost to install, and so the only development done was at a research scale.

Figure 1 - Playing Spacewar on a DEC PDP-1 [32]

As technology got more and more accessible over the 1960s, we see a rapid development of different games. The development of arcade machines starting in 1966, allowed for a system that was dedicated for a specific game. This took a while to develop, as the parts that were needed to put into this machine were very expensive, making the whole concept financially unviable. Finally, in 1971, Pong was released as the first arcade game, allowing for the user to insert a coin to play. Pong was a game you played against the computer, so required more processing power than games before it.

The new arcade experiences allowed for the introduction of “gaming consoles”, which were made to play a variety of games. This showed the viability of game development, creating a whole industry. As mentioned before, the first gaming console was “The Odyssey”, and this came with a couple games that shared the Pong style gameplay, developed by engineer Ralph Baer. It was designed for peoples living rooms and bought gaming in to households.

The advent of personal computers, dubbed PCs, in the 1980s ushered in a new era of game development. It meant that more people were able to access technologies that weren’t available to them previously, and they were being bought for people homes. Also, the fact that high level development languages such as C were now well developed and had documentation meant there were normal people who were able to start developing and gaining an interest in programming.

In the 1990s, 3D graphics became widely available, with the release of Sony’s PlayStation and the Nintendo N64, which created a whole new set of development criteria for games. There have been rapid changes over this time, and most importantly everything got cheaper for the average consumer. 2000s involved the rise of the internet, which allowed for multiplayer gaming to be done across different households.

Technology has constantly been evolving, changing, but most important it has been improving over time and this has had a direct effect on development of video games, as I have explained in this historical context.

2.1.3 – Evolution of Multiplayer Elements in Games

The first video games that were available widely, like the ones mentioned above, were made as multiplayer games due to limitations of the technology. In "Pong" for example, it was easier to not have the calculate where the ball was going, as this would require more computational power. Instead, it just needed calculate if each of the players had made it to the place the ball was, and then if they had, calculate its trajectory. The movement of the pong paddles was handled by the two people playing, not requiring the computer to figure out where the ball is going.

The release of MUDs (Multi-User Dungeons) was significant in the development of online gaming as we know it. The first one "MUD1", was designed and developed in the University of Essex, and initially allowed for people on that network to play together. It was available locally for people on the Essex network, and was later connected to JANET, the UK research network, making it one of the first games to facilitate interconnected multiplayer gameplay across various institutions, thereby expanding the scope and scale of interactive online gaming. This early model of multiplayer shared rooms became a foundation for what we today call Massively Multiplayer Online Role-Playing Games (MMORPGs). These were extremely popular, and continue to have big user bases, showing the potential there was for multiplayer within gaming.

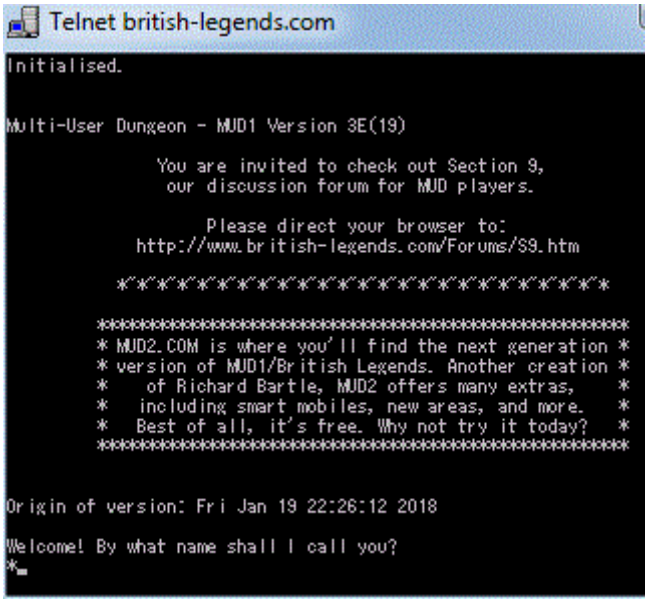


Figure 2 - Screencap of a modern system connecting to the first MUD v3 [33]

The next innovation involved the use of 3D environments to host these multiplayer networked games where players across different networks could play against each other. Some games were initially released with locally multiplayer modes available, but as time went by, they received updates that allowed for players to play across different networks. "Doom (1993)" was a great example of this, starting with an initial 2-4 players being able to play on the same network, but was later updated to use the DWANGO protocol to allow for 4 consecutive players across different networks. It was a bit difficult and required a bit of set up on each of the user's side, but ultimately, it worked [7]. This was the first introduction for many to the concept of 3D shared spaces, and these elements continued to develop.

2.1.4 – Evolution of Game Design in Horror Games

In this section, I will research how game design has evolved historically within the realm of horror games.

As mentioned before, the first "horror game" was included with the inclusion of the horror house overlay. This first attempt's design involved a basic 2D image, where one player hid as a ghost and the other player used hints to find them. It was extremely simplistic, but we see how the developers at the time worked with what they had. Then, in 1981, a game was released for the Atari, simply called "Haunted House", which used a top-down view of the player traversing through different room of a house. It relied on basic mechanics and limited graphics, focusing primarily on survival and exploration in a spooky setting. These both employed simplistic scare tactics with little to no storytelling and relied mainly on imagination.

As technology got better, there was a shift to making these experiences more immersive, using the 3D graphics now available. The introduction of personal computers to household was a welcome one, and they contained enough processing power to run 3D games. In 1982, the first horror game that utilised 3D-ish graphics was called "3D Monster Maze", however it did not contain any imagery that would tie it to traditional horror games [5]. It was developed over 7 months, not by a developer, but by a microprocessor scientist. It technically regarded as three dimensional, however there was no movement in the Y axis, shown in figure 3.

Figure 3 - Screencap from 3D Monster Maze [34]

Game design for Horror Games improved further in the coming years, especially when dedicated gaming consoles came out. The best-selling game on the Sony PlayStation was the cult classic "Resident Evil", which was praised for its graphics, gameplay, sound, and atmosphere, which all enhance the horror experience. The graphics involve the user seeing everything and being able to move around very freely. They did it so well, that some have argued that it is one of the best games of all time. [6]

Design of games evolved greatly for horror games during the period between when they were first released till the early 2000s. I explore more modern iterations later.

2.1.5 – Integration of Multiplayer Elements with the Horror Genre

Multiplayer elements being added to horror games was inevitable, and a welcome addition. They were initially designed as single player experiences, as described above, which were designed around the feeling of being alone, isolating the player and heightening the feeling of vulnerability, but the introduction of multiplayer game modes fundamentally changed how horror games were designed and experienced, creating a layer of interaction that could enhance fear and suspense. Developers and story writers for these games needed to rethink things, as they now needed to provide for shared experiences and incorporate real-time interactions.

One of the first examples of where multiplayer and horror was infused was a game from the Resident Evil series mentioned before. “Resident Evil: Outbreak” allowed up to 4 players at the same time to navigate zombie-filed world of Resident Evil [8]. Released in 2004, it was designed from the ground up to facilitate multiplayer, even going as far as to not even have a proper single player mode, and instead opted for having the player team up with two AIs within the game. This game however had a lot of problems on release, with the European version taking 9 months more than the other versions to come out due to networking issues. [9]

Figure 4 - Screencap of Resident Evil running in 4 player Split screen Coop mode [35]

“Left 4 Dead” came out in 2008 and built on the same principles as “Resident Evil: Outbreak”, a multiplayer horror survival game. Some would say it perfected the niche market it was in, gaining widespread recognition. This, along with the fact it was released on PC means that it is still relevant and has an active player base to this day [10]. It came out with state-of-the-art features, such as the zombie’s ability to adjust based on players’ actions, making each playthrough unpredictable, adding to the fear factor.

Looking at a more recent game, "Dead by Daylight" released in 2016 explores asymmetrical gameplay, where players experience the horror from different perspectives, either as the hunter or the hunted. This approach leverages the unpredictability of human opponents, intensifying the horror and making the gameplay deeply engaging.

Integrating multiplayer elements into horror games has been extremely successful and has broadened the appeal of the genre, subsequently making it popular. The shared experience has deepened the emotional and psychological impact of the gameplay.

2.1.6 – Current Trends and Future Directions

The direction for horror games in general seems extremely positive with lots of development within the video game space being translated to this genre.

Exciting developments include the involvement of emerging technologies like Virtual Reality (VR), which puts the user directly into a 3D version of the world. This is demonstrated in a game from the "Resident Evil" series, "Resident Evil 7: Biohazard", transports the player into eerie environments and confronting them with lifelike scares that feel intensely real due to the 360-degree surround visuals and spatial audio.

AI-driven horror experiences are also a current trend that is extremely promising. A game that did it well, according to critics, was Alien: Isolation [11]. It released in 2015, and used AI to play with the users' psychology, where they escaped an "Alien" that would know where to be all the time. This was highly successful, and the game was highly praised in AI development spaces online.

Currently, a lot is changing in this landscape, and further developments are continuing to happen.

2.2 Networking Technology in Multiplayer Networked Games

Within this section, I explore the technologies required to facilitate networking within multiplayer networked games. This includes examining different network models, such as client-server and peer-to-peer, and looking at their advantages and disadvantages. Additionally, I discuss how games handle issues like latency, using methods such as prediction algorithms and lag compensation to improve player experience. Finally, I cover scalability and load balancing strategies, crucial for managing large numbers of players, particularly through techniques like cloud gaming and distributed architectures. These elements are fundamental in ensuring smooth, responsive gameplay in multiplayer environments.

2.2.1 – Network Models – Client-Server

In a client-server model, all the clients, or in a games' case players, connect to a centralised server that processes all the game data. These are often used in games that have a fixed world, as it allows for the server to change stuff on that world, affect all the clients, like an MMORPG [15].

This model has a lot of benefits, such as the fact that they allow for a better control over the game environment and enhance security measures against cheating [12]. Games such as first-person shooters need this, as it allows them to employ anti-cheat methods, to stop people exploiting on the game, as code needs to be sanitised to the server. Any misuse can be flagged on the server too. Further benefits include the fact that it can make the client software more simplistic, as all the complex calculations need to happen on the server, so the client can just be a shell and use less storage. Central servers can prioritize resources and data management to ensure optimal performance across varying network conditions, offering a better overall quality of service. With proper infrastructure and planning they can curate the experience to support a lot of clients [15].

The disadvantages of this model are that it has higher operational costs and can be expensive to maintain. Server hardware, security, and data centres all incurs these significant ongoing costs. Other costs are also incurred, as more software is needed to be written, as the server and client run different. They need to run well together, but they are vulnerable to centralised failure. If the server goes down, the whole game will crash for everyone. Other disadvantages include the fact that latency can be higher, as player to player interactions need to cross over a server to be shown between players.

2.2.2 – Network Models – Peer-to-Peer (P2P)

In a peer-to-peer model, each peer (client) acts as both a client and a server. This is a common set up for games that are smaller or are locally multiplayer, where each device connects directly to each other.

P2P networking contains plenty of advantages. The main one is that overall costs are reduced, as no expensive server architecture is required to uphold the game. Two separate versions of the game don't need to be upheld either, as there is one that is distributed and can communicate as both server and client [13]. The heavy lifting can either be spread across all the computers, or they can each be doing the different tasks that only that player needs to see. Another benefit is that there is a theoretical lower latency using this networking model, as they are allowed to directly communicate between each other. Other than that, another advantage is that they are more scalable, if the implementation is correct, an infinite number of players can join. Furthermore, it is more resilient to central failures, as it is decentralised, so if one player gets disconnected, it still has all the other players connected to each other. [16].

It has plenty of advantages, but now I will explore the disadvantages. Firstly, it is very hard to regulate cheating as there is no central server to monitor the game activities, and implementing effective security measures is difficult. Cheating is regarded as where a player tries to manipulate data that they are not entitled to change and is a big problem in P2P networked games, for the reasons discussed above [16]. Another disadvantage relates to performance, and due to the distributed nature of the processing and data handling in P2P networking players with suboptimal systems may negatively affect the gameplay experience for others by causing delays or lag. Going off this point, relating to data handling, we can talk about the data consistency. Due to the decentralised nature, it can be very difficult to synchronise data across all the peers, especially as it grows, as all of the peers need to be able to see the data but there is no authority for it.

2.2.3 – Latency Issues

We have established each of the models, and now I will look at stuff that affects both. Latency is a key factor of online game and describes the amount of delay on a network or Internet connection in milliseconds. It can stem from a numerous reason, which I will discuss [27].

The main source of latency comes from the physical distance between the server and clients in a server-client network model, and each peer if we are talking about a P2P model. The packets need to travel a further distance, which directly affects the time it takes for data to travel back and forth, creating a delay that is fundamentally tied this distance. In server-client architectures, players who are located further from the central server will experience greater latency due to the longer travel time for data packets. Similarly, in a P2P model, the delay can vary widely depending on the distances between each peer in the network, as every player's machine communicates directly with the others without the mediation of a central server [31].

Other factors that affecting and compounding latency:

- Internet Connection Quality – If the quality of the connection between the server and the client, or between the peers is not of sufficient quality, issues can arise in relation to packet transfer. Poor internet service with low bandwidth, high interference, or unstable connections are what classify poor internet connection quality and increases the time it takes for data to be sent and received [31].
- Hardware Limitation – The performance of a player's hardware that they run the game on can significantly impact latency. Older or less powerful devices will take longer to process and send data, contributing to overall latency. Additionally, any of the persons personal networking infrastructure, or their local internet infrastructure can cause fluctuations in the latency. Inadequate or outdated networking hardware like routers and modems can also slow down data transmission.
- Server Capacity – This mainly affects the client-server model, and if a server is overloaded with too many players or is running resource-intensive processes, it can slow down, leading to longer processing times for player actions and thus increasing latency. It can also affect the P2P networking model, as if a lot of people join the game, data synchronisation can become a problem.

2.3 – Game Design and Mechanics in Horror Games

In this section, I discuss current game design and mechanics revolving multiplayer horror games, sometimes exploring the two concepts, horror, and multiplayer, together but also separately of each other.

2.3.1 – Core Gameplay Mechanics

Horror games are generally split into subgenres, which can provide a variety of different mechanics such as exploration, resource management, and puzzle-solving. In a survival/mystery setting, exploration would compel players to navigate eerie environments, enhancing the feeling of vulnerability as they uncover the unknown. Resource management would add a layer of strategic decision-making, with players needing to conserve scarce resources like health items and ammunition. Both of these work together to elevate the stakes during any terrifying encounters there might occur [29]. A setting that would involve puzzle-solving engages players intellectually, intertwining seamlessly with the narrative to unlock new areas or story elements, while also serving as a pacing mechanism to build tension gradually.

2.3.2 – Multiplayer Integration

Integrating multiplayer elements into the horror genre allows for the introduction of a dynamic layer to the experience. There can be two different ways it can be implemented, with cooperative modes or competitive modes. In cooperative modes, players must collaborate to overcome challenges, which can alter the typical solitary fear associated with horror games into a shared, intense anxiety. Competitive modes on the other hand, can add an element of unpredictability as players not only face in-game horrors but also have to think about what is going on in other players minds, especially ones they are playing against. This can create a more intriguing fear of both the known and unknown threats.

2.3.3 – Environmental Design

Most opinions agree that this is the most important part of the game and can make or break the perception people have of the game at launch. The level layout is often claustrophobic, incorporating winding corridors and sudden dead ends that create a sense of entrapment and confusion. Lighting is used sparingly and effectively with dimly lit corridors and flickering lights can obscure threats, again playing on the fear of the unseen. All of this is complemented by the sound design which enhances the atmosphere with eerie or jarring noises that can startle players or make them feel on edge. The combination of these elements ensures that the game environment itself becomes a conduit for terror [28].

2.3.4 – Narrative and Storytelling

A proper narrative can make the game spooky, and how this is shown to the user is using stories. This however is not too closely related to what I plan to produce in this paper, but I wanted to explore it anyways. Generally, horror games make the use of complex story arcs involving themes of isolation, supernatural occurrences, or can use psychological terror to engage players deeply. Another popular horror narrative direction can involve player-driven choices that affect the outcome and can lead to multiple different endings, adding a layer of consequence to every decision and can also increase the replay value. Techniques such as non-linear narratives, unreliable narrators, and immersive lore through found documents or recordings deepen the mystery and suspense, urging players to piece together the story while navigating the other horrors around the map [30].

2.3.5 – Conclusion

Together, these design elements and technological advancements create a rich, engaging game environment where every mechanic and narrative decision is aimed at heightening the horror experience, making horror games a uniquely immersive and emotionally charged genre.

2.4 – Player Experience and Psychology in Horror Games

The player experience is an important part of making my Multiplayer Horror experience work, so in this section I explore the different techniques that are employed. Furthermore, I explore some of the psychological factors that contribute to this experience within the horror games.

2.4.1 – Emotional Engagement

Horror games excel at engaging players emotionally by exploiting psychological principles to evoke intense feelings such as fear, anxiety, and anticipation. Effective techniques include jump scares, which exploit the human startle reflex, and the use of suspense and uncertainty to trigger the brain’s innate fear responses. As mentioned in the design, the environment plays a crucial role, with dim lighting and eerie soundscapes that increase heart rates and enhance the fear of the unknown [26]. Psychologically, the horror games often employ narratives that delve into dark, unsettling themes, mirroring players' deepest fears and anxieties, making the experience feel personal and thus more terrifying.

2.4.2 – Player cooperation and interaction

In multiplayer horror games, player cooperation and interaction significantly impact the psychological experience. Cooperative play in a horror setting often heightens emotional

intensity as players share the fear and may rely on each other for survival, which can intensify the collective response to threats. Competitive elements introduce tension and paranoia, as players not only contend with in-game horrors but also the unpredictability of other players' actions, as mentioned before. This dynamic can lead to a complex emotional state where trust and fear can mingle, amplifying the overall stress and engagement levels.

2.4.3 – Challenge and Skill Balance

Talking about engagement levels, challenge and skill balancing is an important way to maintain this. Games must be challenging enough to keep players on edge but also attainable enough that players feel a sense of accomplishment when they overcome obstacles. This balance is known as the flow state and is pivotal; if the game is too difficult, they will quit, if it isn't rewarding enough, they may get bored and quit [24]. This however is very variable, so player testing needs to be done, or difficulty levels can be added.

2.4.4 – Behavioural Responses

Now, we will explore the behavioural responses created by horror games. These games are meticulously designed to provoke fight-or-flight reactions, challenging players to panic, but then slow down and think critically, while being under high stress [25]. This design manipulates player responses through various means, such as limiting resources or presenting sudden, intense threats, which keeps players in a heightened state of alertness. This continuous tension not only sustains the game's atmosphere but also engages players at a fundamental psychological level. Each decision a player makes is imbued with urgency and consequence, highlighting the critical role of game mechanics in shaping the horror gaming experience.

2.5 – Conclusion

In conclusion, after exploring the different ways horror games curate player experience and affect their psychology, we have found that they differ in the ways that they can have effects on the player. It needs to be the right balance to engage players and keep them playing.

2.5 – Engine Use in Multiplayer Networked Horror Games - Unity and Unreal Engine

This section of my literature review evaluates the use of Unity and Unreal Engine in the development of multiplayer networked horror games, with a specific focus on "Phasmophobia" and "Demonologist." These two games were perfect for me to review, as my requirements for my game line up with how these games work and play. By reviewing these games, along with existing research and developer logs, we aim to understand how the choice of game engine affects game design, performance, and player experience.

2.5.1 – Unity Engine in "Phasmophobia"

Unity is a multi-platform game engine, that can be used for developing and running both 2D and 3D games. It uses C# as its primary programming language, to make everything work. Unity is very popular due to how easily it is to access, and it is free to all. This accessibility is widely appreciated by the indie game development community. Furthermore, Unity has extensive support for virtual reality, which gives it a unique selling point as not many other game engines have this feature. [20]

Research indicates that Unity's flexible asset pipeline and extensive support for various VR hardware make it a popular choice for developers focusing on immersive environments [20]. "Phasmophobia" utilizes Unity to great effect, leveraging its efficient lightmapping and real-time global illumination to enhance the eerie atmosphere crucial for horror gameplay. Figure 5 shows what the game looks like.

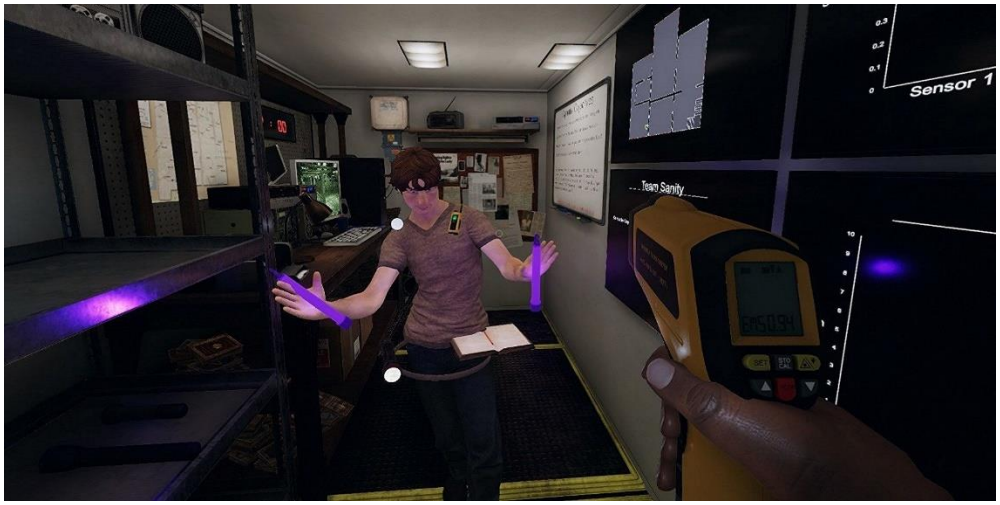


Figure 5 – Phasmophobia [36]

Here is how “Phasmophobia” utilises the resources of Unity:

- **Lightmapping and Real-Time Illumination** – Unity's lightmapping tools allow for detailed pre-calculated lighting of static scenes, allowing for the game to run smoothly when real-time illumination is not required. When real-time illumination is required, the game can change the environment, to account for gameplay elements changing. This can facilitate in shadow and creating an eerie atmosphere.

- **Spatial Audio Engine** – Unity’s audio engine is leveraged to provide spatial audio that reflects the game’s three-dimensional environment. Accurate audio cues are crucial in horror games, enhancing the player's sense of dread through nuanced sounds that are integral to the haunted house experience.

- **Scripting Flexibility with C#** – Unity’s scripting powered by C# allows for the programming of complex game mechanics and AI behaviours. In "Phasmophobia," this capability is crucial for scripting the unpredictable ghost behaviours, enhancing the game's challenge and replay-ability while allowing for easy updates and tweaks based on player feedback.

- **Networking Capabilities** – "Phasmophobia" utilizes Photon Unity Networking (PUN) to manage its multiplayer interactions. Photon enables features like room creation, matchmaking, and efficient network message passing. This integration ensures smooth, stable multiplayer sessions, which are essential for real-time cooperation and interaction among players in the game's immersive horror environment [23].

- **Cross-Platform Capability** – Allows for a greater community of people to play together and is used in Phasmophobia specifically.

2.5.2 – Unreal Engine in "Demonologist"

Unreal Engine is like Unity, being a multi-platform game engine, but is mostly used for 3D games. It uses C++ as its programming language, but also uses blueprints, which are a way of visual scripting, and enhances control and customization while developing games. It is mostly popular for its advanced graphical capabilities, including high-fidelity visuals and dynamic lighting, which appeal especially to AAA developers. It is free to use with a royalty model tied to game revenues, promoting accessibility among developers. Furthermore, it has started recently also supporting VR support, but it is in the very early stages of development. [21]

Unreal Engine is celebrated for its high-fidelity visuals and robust multiplayer framework, which are crucial for creating deeply atmospheric horror experiences. "Demonologist" employs Unreal Engine to capitalize on these strengths, creating a visually stunning and responsive game world that heightens the psychological tension. Figure 6 represent Demonologist and shows off its superior visuals.

Figure 6 – Demonologist [37]

Here is how “Demonologist” utilises the resources of Unreal Engine:

- **Advanced Graphics and Rendering** – "Demonologist" leverages Unreal Engine's superior rendering capabilities, which include high-quality lightmapping and real-time global illumination, to create a visually immersive and atmospheric setting. These features enhance the realism and visual detail of the game environments, playing a critical role in delivering the creepy and unsettling atmosphere essential for horror gameplay. However, they come at a cost to the hardware capabilities.

- **Spatial Audio Engine** – Unreal Engine's audio system supports complex soundscapes and 3D spatial audio, which "Demonologist" uses to build tension and ambiance. The engine allows for fine-tuning audio cues and integrating them seamlessly with game events, adding depth to the horror experience. Sounds like footsteps of other players add to the cohesion and confusion.

- **Blueprint Visual Scripting** – This offers a similar level of scripting flexibility, perhaps even more so than Unity. Blueprints allow developers to quickly prototype and iterate complex game logic and behaviours without deep coding, making it easier to adjust gameplay mechanics and interactions dynamically. Furthermore, C++ can still be used to do scripting anyways if the programmer is more comfortable.

- **Networking Capabilities** – "Demonologist" benefits from Unreal Engine's robust networking capabilities, which are essential for multiplayer interactions. The engine supports large-scale player connectivity and smooth online interactions, which are key for a multiplayer horror game where player coordination can significantly influence gameplay dynamics.[21]

2.5.3 – Comparative analysis and Development Considerations

Choosing between the two engines is normally a consideration depending on how development is done.

Unity is favoured for its ease of use, support for VR and multi-platform deployment, making it ideal for indie developers or smaller projects. Its C# scripting and a vast asset store simplify development, which is beneficial for teams with limited budgets or those needing rapid development cycles.

Looking at Unreal Engine, it is known for its high-fidelity graphics and robust customization options, is suited for projects prioritizing visual quality and technical depth. The engine's Blueprint system helps facilitate complex gameplay mechanics without deep coding knowledge, making it suitable for larger or more technically ambitious projects.

Financially, Unity offers a subscription model with no royalties, while Unreal is free until a project reaches a revenue cap, after which royalties are charged. This makes Unreal attractive for low-budget or experimental projects but requires consideration of long-term costs.

In summary, the choice between Unity and Unreal should be based on your team's expertise, project scale, visual requirements, and financial considerations, ensuring the selected engine matches your development goals and resources [22].

2.6 – Conclusion

In this literature review, I have charted the evolution of multiplayer networked horror games, highlighting how technological advancements and innovative game design have enriched player experiences. As networking technologies and game engines evolve, they continue to push the boundaries of what is possible in horror and networked gaming, making experiences more immersive and emotionally intense. Looking ahead, emerging technologies promise to further enhance the realism and engagement of these games. By understanding the past and present of multiplayer horror games, we can better prepare for other developments that are to occur.

Chapter 3 – Methodology

This chapter outlines the methodological framework underpinning the development of my multiplayer game “Nightmare”. Firstly, a requirements report needs to be proposed and investigated, which allows for a detailed understanding of the essential features, player expectations, and technical needs of the game. Then using this requirements report, I can plan out the methodologies I need to follow in terms of software development, and assist me in creating the game

3.1 – Requirements Report

The requirements report serves as the foundational document that guides all subsequent stages of development, ensuring that the design and functionality align with the defined objectives.

3.1.1 – Functional Requirements

Multiplayer Server Support: The game must facilitate a seamless and synchronized experience for more than 2 players within a server environment. This involves robust back-end infrastructure to ensure that the players can join, interact, and engage in gameplay while experiencing minimal latency and no disconnection issues, promoting a cohesive and dynamic multiplayer ecosystem.

Supernatural Presence Indicators: The implementation of distinct indicators for supernatural entities is crucial to the gameplay. These could manifest as visual anomalies, audio cues, or changes in the environment, providing players with subtle hints of paranormal activity, thereby enhancing the suspense and horror elements of the game.

AI Ghost Navigation: The implementation of an artificial intelligence system is essential for the autonomous navigation of ghost characters within the game's environment. This AI should employ sophisticated algorithms to enable ghosts to move, interact, and respond to player actions in a manner that feels both unpredictable and grounded in the game's internal logic, enhancing the realism and engagement of the gameplay.

Interactive Horror Elements: The game's atmosphere should be enriched with varied elements that contribute to a sense of horror and unease. This could include manipulable objects, dynamic shadows, and ambient sounds that react to player actions or serve to heighten the tension within the game.

AI-Triggered Global Events: The introduction of AI-driven global events can significantly impact the gameplay experience, affecting all players simultaneously. These events should be designed to complement the gameplay. This could perhaps provide a narrative direction to the game, adding layers of complexity and unpredictability to the player experience.

Local Events: The game should balance the events that target individual players and affect the entire player base. Personalized events can enhance the immersive experience for players, making their interactions feel unique and tailored, while global events foster a sense of community and shared challenge.

Haunting Map Design: The design of the game map is fundamental to facilitating an engaging and terrifying exploration experience, whilst also allowing for a level of freedom of movement for the players. The map should be crafted to support ghost haunting mechanics, with attention to

detail that promotes a believable and immersive environment conducive to exploration and discovery.

Gameplay Mechanics: Implementing competitive gameplay mechanics, such as a match timer and scoring system will introduce competitive elements and objectives, motivating players to strategize and perform to the best of their abilities within the allotted time. This system should be designed to make sure that the game is actually fun and should not try to distract from any immersive scary horror elements.

Ghost Searching Capability: A key gameplay mechanic involves the player's ability to search and identify ghosts within the environment. This could be implemented through specialized tools, abilities, or environmental cues, requiring players to actively investigate and engage with their surroundings to uncover supernatural presences. These items should vary in their functions and effectiveness, encouraging players to strategize and make decisions based on their current situation and desired outcomes within the game.

Death Mechanics: The mechanics surrounding player death and transformation into ghosts should be carefully considered, offering a unique perspective on gameplay and interaction within the game world post-death. This could introduce novel gameplay mechanics and strategic considerations for affected players.

Sound Engineering: The strategic use of sound design to mislead or deceive players can significantly enhance the horror experience. Sounds that mimic player actions, ghost movements, or environmental cues can create tension and uncertainty, compelling players to rely on more than just auditory information to navigate the game world.

3.1.2 – Non-Functional Requirements

Low Latency: As mentioned in the multiplayer functional requirement, ensuring low latency is paramount to providing a smooth, real-time multiplayer experience, and is an important requirement. The game's network infrastructure must be optimized to handle data transfer efficiently, minimizing lag and ensuring synchronous gameplay.

Scalability and efficiency: The server architecture must be scalable, capable of accommodating a fluctuating number of players without compromising performance. This involves dynamic resource allocation and efficient load balancing to maintain gameplay quality at varying levels of demand.

Stability: Being reliably stable is key in maintaining player engagement and satisfaction. The game must execute all functions as intended under specified conditions, with mechanisms in place to quickly address and rectify any issues that arise.

Accessibility and Usability: The interface and controls of the game should be intuitive and accessible, allowing players to easily navigate menus, utilize items, and interact with the game environment without unnecessary complexity or confusion. Features that enhance accessibility for a diverse range of players, including those with disabilities, are important for broadening the game's appeal and inclusivity. This may involve customizable control schemes, adjustable difficulty levels, and support for assistive technologies.

Immersiveness: This is mentioned broadly in the functional requirements, but I will solidify the points here, where I want to create a compelling horror experience. The audio and visual elements must be of high quality and designed to immerse players in the game world. Attention

to detail in these areas can significantly enhance the overall atmosphere and emotional impact of the game.

Security: Protecting the game from cheating, hacking, and other forms of manipulation is essential for fair play and player trust. Implementing robust security measures will help safeguard the integrity of the gameplay experience and user privacy. However, I will not put too much emphasis on this, as it is a casual game.

3.2 – Software Environment and Development Resources

This section outlines the development environment, programming languages and libraries used in the creation of “Nightmare”, based off the requirements reports I have detailed. Furthermore, I highlight their roles in facilitating the design and implementation of various game mechanics and features. The choice of these tools was guided by their robustness, community support, and suitability for developing networked games.

3.2.1 – Development Methodology

Agile methodologies to ensure flexibility in design and development, allowing for iterative testing and refinement of game mechanics. The project was structured around the principles of Agile development, which emphasizes iterative progress through repeated cycles (sprints) of planning, development, testing, and evaluation. These sprints allowed for me to constantly be updating the game, adding different features as needed.

3.2.2 – Development Environment

The core platform for developing my game was Unity, one of the most widely used game engines in the industry. Unity offers comprehensive tools for both 2D and 3D game development, including physics, animation, and rendering capabilities, which were crucial for creating the immersive environments of the game, as discussed in the literature review. The engine's well-documented API and active developer community significantly expedited the development process. The reasons I chose it over Unreal Engine was that, after researching them in the literature review, is because I did not need all of the visual fidelity. Other than that, I found that Unity games ran better on systems that had lower specifications. Also, the fact that I already had experience working with Unity was a big push.

Integrated with Unity, Visual Studio served as the primary IDE for writing and debugging the game's C# scripts. It provided powerful features such as IntelliSense, debugging, and direct integration with Unity's asset workflows, which enhanced coding efficiency and accuracy. Compared to Unreal Engine’s blueprints, I preferred this, as I was familiar with visual studio.

3.2.3 – Programming Language

All server and client logic, including gameplay mechanics, UI interactions, and network management, were implemented using C#, a language known for its robustness and extensive feature set in object-oriented programming. C#'s compatibility with Unity and its ease of use for complex calculations and system design made it the ideal choice for this project.

3.2.4 – Libraries and Frameworks

Mirror Networking Library was chosen as the networking library to handle all aspects of network communication and synchronization in my game. Mirror is a powerful and user-friendly network library for Unity, designed as a successor to UNet, the deprecated Unity networking library. I was familiar with UNet before it was deprecated, and it is why I chose to use Mirror. It simplifies the process of creating networked multiplayer games by handling complex networking tasks like command and RPC calls, and state synchronization. In “Nightmare”, Mirror was used to synchronize player movements, game state, and real-time interactions across various client sessions, ensuring a seamless multiplayer experience. As the server and client commands are all contained in the same code, I didn’t need perform anything extra, with regards to

The Unity Profiler was employed to monitor and analyse the game’s performance across various systems. This included rendering, scripting, memory usage, and asset loading times. The Unity Profiler is integral for ensuring that the game runs efficiently on all supported platforms, providing developers with the data necessary to optimize both the front-end and back-end of the game.

Along with the Unity Profiler I used a network profiler, known as Mirage, for optimizing network performance and ensuring efficient data transmission between clients and servers. This tool is instrumental in identifying bottlenecks and inefficiencies in network communication. By providing detailed insights into packet size, network calls, and latency, the Mirage Profiler allowed me to fine-tune network operations, crucial for maintaining smooth gameplay in the multiplayer environment.

3.2.5 – Version Control

GitHub was used for version control, and allowed me to systematically track changes, giving me secure storage of all development assets, along with a robust backup system. This practice was vital for maintaining a structured development timeline and ensured that all increments of the game build were versioned and recoverable.

3.2.6 – Assets and Prefabs

To give my game a professional look, a strategic decision was made to incorporate high-quality, ready-made assets and prefabs from the Synty Store. This approach provided a lot of advantages significantly accelerated the development timeline and enhanced the visual appearance of the game.

Using the prefabricated assets from Synty, the process of designing complex environments, such as the haunted house setting in the game, was faster than if I had designed all the walls myself. It also looks a lot better, as it the designs were made by professionals.

The Player Prefabs that were provided within the Synty asset packs worked with Unity’s built-in animator, and I was able to fetch some pre rigged animations online from Mixamo.com

3.3 – Testing Methodology

3.3.1 – Game testing

Game testing was an integral part of each development sprint. Due to the iterative nature of the agile methodology, game testing was done at every stage of the development process, allowing for immediate feedback and prompt incorporation of changes. This continuous testing loop significantly enhanced the game's quality and adherence to user requirements.

3.3.2 – Network testing

Given the multiplayer networked nature of the game, network testing was crucial to ensure stable and seamless gameplay across various network conditions. Latency testing was the main focus of this and

Furthermore, testing was done on the game's ability to handle dropped connections and reconnects without impacting gameplay experience.

3.3.3 – Player experience testing

To find out how the player felt about the game, I used a Google Form to allow for me to see how players were responding to my game. The questions were as follows:

1. On a scale of 1(a lot) to 5(none) how much lag did you experience?
2. How would you rate the visual and audio quality of the game from 1 (poor) to 5 (excellent)?
3. Rate your overall satisfaction with the social interaction features of the game, from 1 (very unsatisfied) to 5 (very satisfied).
4. Were you satisfied the variations in the items available for the playable to use to find the ghost, on a scale of 1 (very unsatisfied) to 5 (very satisfied).
5. On a scale of 1(worst) to 5(best) how was the overall experience?
6. What improvements would you suggest for enhancing the game experience?

The result of this questionaire I look at in the discussion and analysis chapter.

3.4 – System Architecture and Design

This is the final part before I move on to Implementation, I would like to provide the system architecture and the design I have, so I built this UML Diagram, shown in figure 7 .This is to conclude the finding that I have made from this methodology and designed this. The rationale for each function is defined within the implementation chapter.

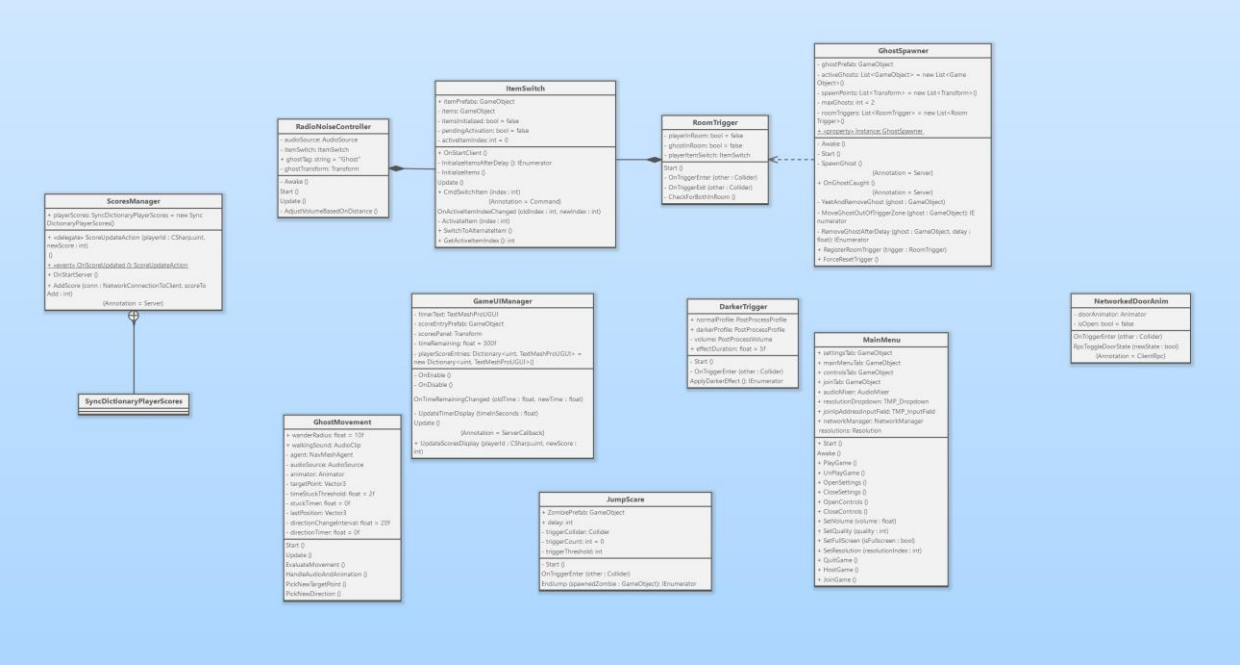


Figure 7 - UML diagram demonstate relationship between classes

Chapter 4 – Implementation

In this section, I demonstrate the implementation of my multiplayer networked game, describing the core mechanics, the stuff I did for synchronization, and the networking. Furthermore, I show my testing and describe challenges faced, with their solutions.

4.1 – Game Mechanics

Within game mechanics, I detail how I implemented different with relation to what part of the script they are in, along with code snippets.

4.1.1 – PlayerScript.cs

This is a crucial script as it allows the player to control their character. It provides them functionality such as movement, camera control, and interactions within the game environment. This script uses Unity's input system to translate player commands into actions, such as moving, jumping, and interacting with game elements like ghosts.

Firstly, we discuss player movement. Player movement in a multiplayer game needs to be smooth and responsive. In my implementation, it is driven by the player's input through keyboard commands (WASD for directional movement and spacebar for jumping). The script translates these inputs into motion using Unity's physics engine by applying forces for movement and jumping. Jumping first uses the function isGrounded to make sure the player is on the ground, and if they are, allow the player to jump. If it didn't check it would make the player fly if they held down space. Walking is shown in figure 8 and jumping is shown in figure 9.

```
// Movement using WASD and arrow keys
float moveZ = Input.GetAxis("Vertical") * Time.deltaTime * currentMoveSpeed;
float moveX = Input.GetAxis("Horizontal") * Time.deltaTime * currentMoveSpeed;

// Move the player
transform.Translate(new Vector3(moveX, 0f, moveZ));
```

Figure 8 - Movement Code

```
// Check if the player is grounded
isGrounded = Physics.Raycast(transform.position, Vector3.down, 0.1f);

// Jumping
if (Input.GetKeyDown(KeyCode.Space) && isGrounded)
{
    GetComponent<Rigidbody>().AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
}
```

Figure 9 - Jumping mechanic

The camera movement is an important feature that allows the player to look around, within a limited range. It also rotates the player prefab so it is always facing the same way as the camera. This is shown in figure 10.

```
// Mouse input for rotation
float mouseX = Input.GetAxis("Mouse X") * mouseSensitivity;
float mouseY = Input.GetAxis("Mouse Y") * mouseSensitivity;

// Rotate the player horizontally based on mouse input
transform.Rotate(Vector3.up, mouseX);

// Calculate vertical rotation incrementally
rotationX -= mouseY;
rotationX = Mathf.Clamp(rotationX, -90f, 90f); // Limit vertical rotation to prevent flipping
```

Figure 10 - Camera and player rotation

Animation is also handled in the PlayerScript, shown in figure 11, where the movement speed of the player is calculated, and if they are going over a certain speed, the walking animation is played. Figure 12 shows the animator that is placed on each player, that makes them play the animations when the Boolean IsWalking is set to True, and vice versa when it is set to false.

```
// Update the IsWalking parameter in the Animator
animator.SetBool("IsWalking", Mathf.Abs(moveX) > 0.01f || Mathf.Abs(moveZ) > 0.01f);
```

Figure 11 - The walking animation activator, depending on movement speed

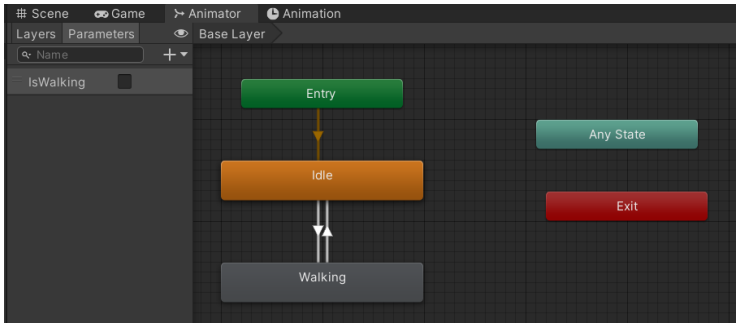


Figure 12 - Unity animator.

These next methods work in conjunction with other classes and methods to perform the functions needed to make different game mechanics work. One of the first methods that runs is one that is required for the interface to function and display scores. Shown in figure 13, it sets up to display the scores, and I will talk more about that later, when talking about the implementation of the scoresManager class. I needed to set it up as a `[Command]` which instructs the Mirror Network Manager to perform this command on the server, making sure that the most recent scores are available on the players screen as soon as they join.

```
[Command]
1 reference
private void CmdRequestScores()
{
    ScoresManager scoresManager = FindObjectOfType<ScoresManager>();
    if (scoresManager != null)
    {
        scoresManager.AddScore(connectionToClient, 0);
    }
    scoresManager.SendCurrentScoresTo(connectionToClient);
}
```

Figure 13 - User requesting score function.

The method `SetIsNearGhost(bool)` shown in figure 14 is referenced with the RoomTrigger.cs script mentioned later, and makes sure that the player and the ghost that they want to catch is in the same room.

```
public void SetIsNearGhost(bool isNear)
{
    isNearGhost = isNear;
}
```

Figure 14 - Setter method for isNearGhost()

The next method is being checked constantly, as it within the Update method the PlayerScript. `CheckForScoringOpportunity()`, shown in figure 15 makes sure that as soon as all the conditions are met, the method `CmdCatchGhost()` can be called. It also makes sure that the player cannot spam the button and earn extra points while the animation of the catching of the ghost takes place, using the `isActionOnCooldown` Boolean.

```
1 reference
private void CheckForScoringOpportunity()
{
    if (Input.GetKeyDown(KeyCode.G) && isNearGhost && !isActionOnCooldown)
    {
        CmdCatchGhost(); // Send a command to the server to catch the ghost
        isActionOnCooldown = true; // Start cooldown on the client side immediately
    }
}
```

Figure 15

The method `CmdCatchGhost()` needs to happen on the server so that another ghost can be spawned in a different place, and there is only one at any given time, on all clients. Therefore I make the use of `[Command]` which instructs the Mirror Network Manager to perform this command on the server. The code for this is shown in figure 16.

```
[Command]
1 reference
private void CmdCatchGhost()
{
    // Assuming the ghost catching logic might involve score increase and
    // potentially other server-side effects
    if (isNearGhost) // Ensure isNearGhost or equivalent logic is validated server-side
    {
        ScoresManager scoresManager = FindObjectOfType<ScoresManager>();
        if (scoresManager != null)
        {
            scoresManager.AddScore(connectionToClient, 1);
        }
        ghostSpawner.OnGhostCaught(); // Notify the GhostSpawner that a ghost has been caugh


        RpcHandleGhostCaught(); // Call a ClientRpc to handle client-side effects
    }
}
```

Figure 16

When the ghost is caught, animation and effects needs to happen on the clients side too, which is what the `RpcHandleGhostCaught()` within `CmdCatchGhost()` is for, which utilises `[ClientRpc]` to perform the actions on the players. Figure 17 demonstrates this method, which also contains the coroutine `WaitAndReset()` discussed later.

```
[ClientRpc]
1 reference
private void RpcHandleGhostCaught()
{
    ghostSpawner.OnGhostCaught(); //
    StartCoroutine(WaitAndReset());
}
```

Figure 17

`WaitAndReset()` works to implement the cooldown functionality, mentioned in the `CheckForScoringOpportunity()` and prevents the user from clicking on the catch ghost button (G). This is important as it prevents the player from spamming the G button and gaining too many points while the ghost is being exorcised/removed from the game. It is shown in figure 18.

```
1 reference
IEnumerator WaitAndReset()
{
    yield return new WaitForSeconds(15); // Wait for 15 seconds

    isActionOnCooldown = false; // Allow the action to be triggered again
    isNearGhost = false; // Reset proximity flag
}
```

Figure 18

4.1.2 – ItemSwitch,cs

This is also placed on the player prefab and is integral to managing the items a player can carry and actively use within the game. It handles the instantiation, activation, and switching of items based on player inputs and utilizes Unity's Mirror networking to synchronize item states across all clients, maintaining consistency in the multiplayer environment.

Upon the client's start, the script initializes the items after a brief delay to ensure all necessary game components are loaded properly. This is done using a coroutine that waits for a short period before activating the initial item setup. Figure 19 shows this initialisation, along with the `OnStartClient()` that is built into Mirror's `NetworkBehaviour` class, which this class inherits from. This delay is required so that the player can spawn in holding the first item.

```
3 references
public override void OnStartClient()
{
    base.OnStartClient();
    StartCoroutine(InitializeItemsAfterDelay());
}

1 reference
private IEnumerator InitializeItemsAfterDelay()
{
    yield return new WaitForSeconds(0.1f); // Wait for a brief moment

    InitializeItems();

    // Immediately activate the initial item if it's not the first item
    if (activeItemIndex != 0)
    {
        ActivateItem(activeItemIndex);
    }
    else if (items.Length > 0)
    {
        // For the first item, force activation to ensure visibility
        pendingActivation = true;
        ActivateItem(0); // Directly activate the first item
    }
}
```

Figure 19

The `InitializeItemsAfterDelay()` calls the `InitializeItems()` shown in figure 20, which spawn the items from the prefabs list that I can edit easily within Unity, adding and changing items as needed. Each item is positioned relative to the player, and specific items might have unique orientations based on their intended use. This setup process also initially disables the items to prevent them from appearing before they are actively selected.

```
1 reference
private void InitializeItems()
{
    items = new GameObject[itemPrefabs.Length];
    for (int i = 0; i < itemPrefabs.Length; i++)
    {
        GameObject item = Instantiate(itemPrefabs[i], transform.position + transform.forward * (i + 1), transform.rotation);
        item.transform.SetParent(transform);
        Vector3 offset = new Vector3(0.25f, 0.95f, 0.5f);
        item.transform.localPosition = offset;
        if (i != 4 && i != 5)
        {
            item.transform.localRotation = Quaternion.Euler(0, 180, 0);
        }
        else
        {
            // For 4th and 5th elements, keep the rotation as Quaternion.identity (no rotation)
            item.transform.localRotation = Quaternion.identity;
        }
        item.SetActive(false); // Initially disable the item
        items[i] = item;
    }

    itemsInitialized = true;

    if (pendingActivation)
    {
        ActivateItem(activeItemIndex);
        pendingActivation = false;
    }
}
```

Figure 20

Within the `Update()` is where the input from the user is grabbed, shown in figure 21. Each key press corresponds to a different item in the inventory, and the script handles these inputs to update the active item index.

```
Unity Message | 0 references
void Update()
{
    if (Input.GetKeyDown(KeyCode.Alpha1)) { CmdSwitchItem(0); }
    else if (Input.GetKeyDown(KeyCode.Alpha2)) { CmdSwitchItem(2); }
    else if (Input.GetKeyDown(KeyCode.Alpha3)) { CmdSwitchItem(4); }
    else if (Input.GetKeyDown(KeyCode.F)) { CmdSwitchItem(5); }
    else if (Input.GetKeyDown(KeyCode.Alpha0)) { CmdSwitchItem(-1); }
}
```

Figure 21

When the player presses down on the numbers 1-3, it calls the method `CmdSwitchItem(int index)`, shown in figure 22, where the integer is the number of the list that you want which performed on the server to ensure that the item change is reflected across all clients, and so other players can see your item has changed.

```
[Command]
6 references
public void CmdSwitchItem(int index)
{
    activeItemIndex = index;
}
```

Figure 22

The script uses a SyncVar hook to call a method whenever the active item index changes. This method, OnActiveItemIndexChanged, ensures that the newly selected item is activated (made visible) and all others are deactivated, and all the other players can see that it has changed. This mechanism is crucial for ensuring that players see only the currently active item, enhancing realism and clarity. This `SyncVar` is shown in figure 23 and the code that it uses is shown in figure 24.

```
[SyncVar(hook = nameof(OnActiveItemIndexChanged))]
```

Figure 23

```
1 reference
void OnActiveItemIndexChanged(int oldIndex, int newIndex)
{
    if (itemsInitialized)
    {
        ActivateItem(newIndex);
    }
    else
    {
        pendingActivation = true;
    }
}

4 references
private void ActivateItem(int index)
{
    foreach (GameObject item in items)
    {
        if (item != null) item.SetActive(false);
    }

    if (items == null || index < -1 || index >= items.Length)
    {
        Debug.LogError("ActivateItem: Attempt to access uninitialized array or out of range index.");
        return;
    }

    if (index == -1)
    {
        return;
    }
    else { items[index].SetActive(true); }

}
```

Figure 24

The following method shown in figure 25, is `SwitchToAlternateItem()` used by the the RoomTrigger class, to change the item to an alternative version of it. This is used for when the ghost is in the same room as the player, it makes the flame on the candle go out.

```
1 reference
public void SwitchToAlternateItem()
{
    if (activeItemIndex == 0) // Check if the player is holding item 1 (index 0)
    {
        int alternateIndex = (activeItemIndex + 1) % itemPrefabs.Length; // Example logic to switch to the next item
        CmdSwitchItem(alternateIndex); // Command to switch item
    }
}
1 reference
public int GetActiveItemIndex()
{
    return activeItemIndex;
}
```

Figure 25

4.1.3 – RadioNoiseController,cs

This script is a specialized component attached to a radio item in the game. It dynamically controls the audio output based on the radio's proximity to ghosts, creating an engaging and responsive player experience. The script uses Unity's audio system to manipulate sound volume and playback in real-time. It doesn't have any networking connected to it, as it is primarily for use by the person holding it.

Firstly, it needs to initialise itself, and connect with all the components needed. Shown in figure 26, it requires the `AudioSource` and the `itemSwitch` components. The `AudioSource` setup ensures that the radio is ready to adjust its playback settings based on game conditions right from the start, and the `itemSwitch` using the `GetComponentInParent<ItemSwitch>()` that the radio only affects the game environment when it is selected by the player, and only that player can hear it.

```
Unity Message | 0 references
private void Awake()
{
    audioSource = GetComponent<AudioSource>();
}

Unity Message | 0 references
void Start()
{
    itemSwitch = GetComponentInParent<ItemSwitch>();
}
```

Figure 26

The update method, shown in figure 27, contains all of the logic behind the dynamic audio adjustment, where it continuously checks the distance between the radio and the nearest ghost. If the ghost is within a specific range and the radio is the active item, the volume is adjusted to enhance the feeling of proximity and presence. It also makes sure that the noise only happens when the ActiveItem is the Radio.

```
Unity Message | 0 references
void Update()
{
    // Find the ghost in the scene by tag
    GameObject ghost = GameObject.FindGameObjectWithTag(ghostTag);
    if (ghost != null)
    {
        ghostTransform = ghost.transform;
    }
    if (ghostTransform == null) return;

    if (itemSwitch != null && itemSwitch.GetActiveItemIndex() == 2)
    {
        AdjustVolumeBasedOnDistance();
    }
    else
    {
        // Mute the radio if it's not the active item
        audioSource.volume = 0;
    }
}
```

Figure 27

This script, figure 28, is used to calculate the volume of the audio based on how close the ghost is to the radio. This calculation uses a linear interpolation of the distance to set the volume, with the sound reaching maximum volume at a minimum threshold distance and decreasing as the ghost moves away.

```
private void AdjustVolumeBasedOnDistance()
{
    float distance = Vector3.Distance(transform.position, ghostTransform.position);
    float maxVolumeDistance = 5f; // Distance at which the volume is at its highest
    float minVolumeDistance = 20f; // Distance at which volume starts to decrease

    // Normalize the distance within our volume range and invert it
    float volume = 1.0f - Mathf.Clamp01((distance - maxVolumeDistance) / (minVolumeDistance - maxVolumeDistance));

    audioSource.volume = volume;
    if (!audioSource.isPlaying)
        audioSource.Play();
}
```

Figure 28

4.1.4 – GhostSpawner,cs

The GhostSpawner.cs script handles the spawning, management, and removal of ghosts within the game environment. It is designed to maintain a balanced gameplay by ensuring that ghosts are generated and deactivated in response to various game events. This script uses Unity's networking capabilities via Mirror to synchronize ghost activities across all clients in a multiplayer setting.

Upon initialisation of the game, where the `Awake()` method is called, shown in figure 29, the script identifies potential spawn points for ghosts by tagging them with the tag `"NPCSpawnPoint"` within the Unity Editor. These points are stored in a list which the spawner will later use to randomly place new ghosts in the game world. These SpawnPoint are just empty 3D objects with a tag, but it allowed me to place them wherever they were needed.

```
Unity Message | 0 references
private void Awake()
{

    GameObject[] points = GameObject.FindGameObjectsWithTag("NPCSpawnPoint");
    foreach (GameObject point in points)
    {
        spawnPoints.Add(point.transform);
    }
}
```

Figure 29

The `SpawnGhost()` method actively controls the number of ghosts in the game based on predefined limits. Shown in figure 30, it randomly selects a spawn point from the available list to instantiate ghosts, ensuring that the game environment remains dynamic.

```
[Server]
2 references
private void SpawnGhost()
{
    Debug.Log("Attempting to spawn ghost");

    if (activeGhosts.Count >= maxGhosts || spawnPoints.Count == 0) return;

    int spawnPointIndex = Random.Range(0, spawnPoints.Count);
    GameObject ghost = Instantiate(ghostPrefab, spawnPoints[spawnPointIndex].position, Quaternion.identity);
    NetworkServer.Spawn(ghost);
    activeGhosts.Add(ghost);
}
```

Figure 30

When a ghost is captured or an interaction occurs that requires a ghost to be removed the script manages this by deactivating the ghost and potentially spawning a new one if the maximum limit allows, shown in figure 31. This ensures that the gameplay remains engaging and balanced. The removal of the ghost is done using `YeetAndRemoveGhost(GameObject ghost)` method, which can pinpoint which ghost is the one in the same room.

```
[Server]
2 references
public void OnGhostCaught()
{
    if (activeGhosts.Count > 0)
    {
        GameObject toRemove = activeGhosts[0];
        activeGhosts.RemoveAt(0);

        YeetAndRemoveGhost(toRemove);
        ForceResetTrigger();

    }
    if (activeGhosts.Count < maxGhosts)
    {
        SpawnGhost();
    }
}
```

Figure 31

This method, shown in figure 32, not only removes the ghost from the game environment but also ensures that any associated components, such as AI navigation and colliders, are properly dismantled to avoid residual game artifacts. This is done to allow for the animation to occur on the ghost, where it gets launched through the walls, so doesn't require the colliders or any associated scripts anymore.

```
1 reference
private void YeetAndRemoveGhost(GameObject ghost)
{
    // Remove specific scripts or components in the desired order
    GhostMovement movementScript = ghost.GetComponent<GhostMovement>();
    if (movementScript != null) { Destroy(movementScript); }

    // Remove the NavMeshAgent
    NavMeshAgent navAgent = ghost.GetComponent<NavMeshAgent>();
    if (navAgent != null)
    {
        Destroy(navAgent);
    }

    // Remove the BoxCollider
    BoxCollider boxCollider = ghost.GetComponent<BoxCollider>();
    if (boxCollider != null)
    {
        Destroy(boxCollider);
    }

    // Start a coroutine to move the ghost out of the trigger zone
    StartCoroutine(MoveGhostOutOfTriggerZone(ghost));

    // Start a coroutine to remove the ghost after a delay, giving it time to exit the trigger zone
    StartCoroutine(RemoveGhostAfterDelay(ghost, 5f)); // Adjust the delay as needed
}
```

Figure 32

The following IEnumerator methods, in figure 33, `MoveGhostOutOfTriggerZone(GameObject ghost) and RemoveGhostAfterDelay(GameObject ghost, float delay),` are both called within the method, and demonstrate the two steps that happen when removing the ghost. The first step involves applying a force on the ghost that causes them to be chucked away, and the second allows for the ghost to be removed so that another can take its place.

```
1 reference
private IEnumerator MoveGhostOutOfTriggerZone(GameObject ghost)
{
    float moveDuration = 5f; // Duration in seconds over which the ghost will move out of the trigger zone
    float startTime = Time.time;

    // Calculate the target position outside of the trigger zone
    Vector3 targetPosition = ghost.transform.position + new Vector3(0, 10, 0);

    while (Time.time < startTime + moveDuration)
    {
        // Linearly interpolate the position over time
        ghost.transform.position = Vector3.Lerp(ghost.transform.position, targetPosition, (Time.time - startTime) / moveDuration);
        yield return null; // Wait for the next frame
    }
}

1 reference
private IEnumerator RemoveGhostAfterDelay(GameObject ghost, float delay)
{
    yield return new WaitForSeconds(delay);

    if (ghost != null)
    {
        Destroy(ghost);
    }
}
```

Figure 33

The following methods, in figure 34, are used due to the `roomTriggers triggerCollider` not being the most reliable. They do not successfully perform the `OnTriggerExit` methods, so I needed to make these methods to make sure the logic worked successfully. The `ForceResetTrigger()` is called within the

```
1 reference
public void RegisterRoomTrigger(RoomTrigger trigger)
{
    if (!roomTriggers.Contains(trigger))
    {
        roomTriggers.Add(trigger);
    }
}
1 reference
public void ForceResetTrigger()
{
    Collider triggerCollider = GetComponent<Collider>();
    if (triggerCollider != null)
    {
        triggerCollider.enabled = false;
        triggerCollider.enabled = true;
    }
}
```

Figure 34

4.1.5 – RoomTrigger,cs

This script is put on each of the 30 unique zones the ghost can spawn in. It contextually manages the player and ghost interactions based on their presence within designated room boundaries. This script is instrumental in facilitating gameplay mechanics that depend on the proximity of characters to specific game elements, enhancing both the challenge and realism of the environment.

Within the `Start()` method, shown in figure 35, the script locates and registers itself with the GhostSpawner, ensuring it is recognized and managed as part of the broader ghost management system. This step is crucial for making sure the roomTriggers can be reset, as shown in the final part of the ghostSpawner script

```
Unity Message | 0 references
void Start()
{
    GhostSpawner spawner = FindObjectOfType<GhostSpawner>();
    if (spawner != null)
    {
        spawner.RegisterRoomTrigger(this);
    }
    else
    {
        Debug.LogWarning("GhostSpawner not found in the scene.");
    }
}
```

Figure 35

The core functionality of the RoomTrigger.cs script is to detect when players or ghosts enter or leave the room, shown in figure 36. This detection is achieved using Unity's Collider component, where the script listens for OnTriggerEnter and OnTriggerExit events. OnTriggerEnter is very reliable, however OnTriggerExit is not, so I implemented a different version shown in GhostSpawner.

```
Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (other.CompareTag("Player"))
    {
        playerInRoom = true;
        playerItemSwitch = other.GetComponent<ItemSwitch>(); // Get the ItemSwitch component
        CheckForBothInRoom();
    }
    else if (other.CompareTag("Ghost"))
    {
        ghostInRoom = true;
        CheckForBothInRoom();
    }
}

Unity Message | 0 references
private void OnTriggerExit(Collider other)
{
    if (other.CompareTag("Player"))
    {
        playerInRoom = false;
        playerItemSwitch = null; // Clear the reference
    }
    else if (other.CompareTag("Ghost"))
    {
        ghostInRoom = false;
    }
}
```

Figure 36

In figure 37, the method `CheckForBothInRoom()` allows for the detection of what happens when both the players are in the same room. Firstly, it switches the item the player is holding to its alternate version, which I have implemented as being the candle going from lit to not lit. Then the Boolean in the PlayerScript, `SetIsNearGhost(Bool)` is set to true.

Figure 37

4.1.6 – GhostMovement,cs

This script is essential for automating the ghost movement within the game world utilizing Unity's `NavMeshAgent` for artificially intelligent pathfinding. The script is designed to handle complex movement patterns, including wandering within a defined radius, responding to being stuck, and integrating sound and animation based on movement state.

This part of the script, shown in figure 38, makes sure that the required components, NavMeshAgent for navigation, AudioSource for sound effects, and Animator for animations, are added.

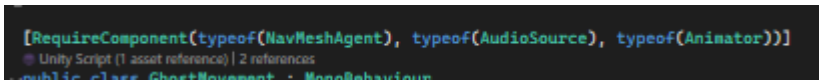


Figure 38

The `Start()` here initialises all the components required, and the `Update()` method starts all of the scripting required for the ghost to walk around intelligently. This is seen in figure 39.

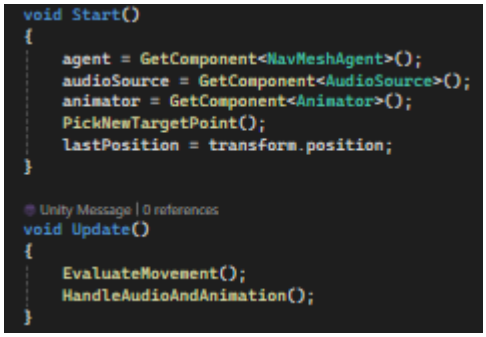


Figure 39

The EvaluateMovement() function, shown in figure 40, is a crucial part of the script, managing the ghost's navigation logic. It tracks how much time has passed to trigger direction changes periodically and checks if the ghost reaches its destination to pick a new target point. It also monitors whether the ghost has moved significantly and if not, it considers the ghost stuck and changes direction.

```
1 reference
void EvaluateMovement()
{
    directionTimer += Time.deltaTime;
    if (directionTimer >= directionChangeInterval)
    {
        PickNewDirection(); // Force a new direction every 20 seconds
        directionTimer = 0f;
    }

    if (!agent.pathPending && agent.remainingDistance < 0.5f)
    {
        PickNewTargetPoint();
    }

    if (Vector3.Distance(transform.position, lastPosition) < 0.01f) // Check if the ghost hasn't moved much
    {
        stuckTimer += Time.deltaTime;
        if (stuckTimer >= timeStuckThreshold) // Ghost is considered stuck
        {
            PickNewDirection(); // Pick a new forward-facing direction
            stuckTimer = 0f;
        }
    }
    else
    {
        stuckTimer = 0f; // Reset the stuck timer if the ghost is moving
    }
    lastPosition = transform.position; // Update the last known position
}
```

Figure 40

The `HandleAudioAndAnimation()` method, shown in figure 41, synchronizes the ghost's walking sounds and animations with its movement speed. If the ghost is moving faster than a minimal threshold, it plays the walking sound and activates the walking animation. If the ghost stops or slows down significantly, it stops the sound and deactivates the animation. The animator is the same as the one seen in figure 12.

```
1 reference
void HandleAudioAndAnimation()
{
    float speed = agent.velocity.magnitude;
    if (speed > 0.1f && !audioSource.isPlaying)
    {
        audioSource.clip = walkingSound;
        audioSource.Play();
    }
    else if (speed <= 0.1f && audioSource.isPlaying)
    {
        audioSource.Stop();
    }

    animator.SetBool("IsWalking", speed > 0.1f);
}
```

Figure 41

The `PickNewTargetPoint()` and `PickNewDirection()` methods shown in figure 42 allow the ghost to navigate the environment dynamically and can ensure the ghost does not follow a predictable path, thereby enhancing the gameplay challenge. These methods calculate new destinations based on random directions within a specified radius, using Unity's NavMesh.SamplePosition() to find viable points on the NavMesh, which is cooked beforehand.

```
2 references
void PickNewTargetPoint()
{
    Vector3 randomDirection = Random.insideUnitSphere * wanderRadius + transform.position;
    NavMesh.SamplePosition(randomDirection, out NavMeshHit hit, wanderRadius, 1);
    targetPoint = hit.position;
    agent.SetDestination(targetPoint);
}

2 references
void PickNewDirection()
{
    float angle = Random.Range(-45, 45); // Adjust the range as needed
    Vector3 forward = transform.forward;
    Vector3 newDirection = Quaternion.Euler(0, angle, 0) * forward * wanderRadius;
    Vector3 newPosition = transform.position + newDirection;
    NavMesh.SamplePosition(newPosition, out NavMeshHit hit, wanderRadius, 1);
    targetPoint = hit.position;
    agent.SetDestination(targetPoint);
}
```

Figure 42

4.1.7 – ScoresManager.cs

The ScoresManager class is used for tracking player scores within the networked environment. It uses Mirror's SyncDictionary to synchronize score data across clients, ensuring that all players have up-to-date score information.

Firstly, the score manager is initialised, as soon as the server starts, and registers callbacks and network message handlers to handle scores efficiently across the network, shown I figure 43.

```
6 references
public override void OnStartServer()
{
    base.OnStartServer();
    playerScores.Callback += OnPlayerScoreChanged;
    NetworkServer.RegisterHandler<RequestScoresMessage>(OnRequestScores);

}
```

Figure 43

The `AddScore()` shown in figure 44 allows the addition of a specified score to a player's existing score or creates a new entry if the player does not already have one. As discussed before, Mirror's SyncDictionary which automatically synchronizes changes with clients.

```
[Server]
2 references
public void AddScore(NetworkConnectionToClient conn, int scoreToAdd)
{
    var playerId = conn.identity.netId;

    if (playerScores.ContainsKey(playerId))
    {
        playerScores[playerId] += scoreToAdd;
    }
    else
    {
        playerScores.Add(playerId, scoreToAdd);
    }
}
```

Figure 44

A callback method triggered by the `SyncDictionary` whenever a score entry is added, changed, or removed. This method, shown in figure 45, is designed to invoke further actions such as changing UI components.

```
// This callback is triggered whenever the SyncDictionary changes.
2 references
private void OnPlayerScoreChanged(SyncDictionary<uint, int>.Operation op, uint playerId, int score)
{
    // Use an event to notify about the score change.
    OnScoreUpdated?.Invoke(playerId, score);
}
```

Figure 45

This figure 46 demonstrates the method `InitializePlayerScore()` and was made to set the initial scores for players when they connect to the server, ensuring every player has a score entry.

```
[Server]
1 reference
public void InitializePlayerScore(NetworkConnectionToClient conn, int initialScore)
{
    var playerId = conn.identity.netId;

    // Check if the player already has a score entry; if not, initialize with the given initialScore
    if (!playerScores.ContainsKey(playerId))
    {
        playerScores.Add(playerId, initialScore);

        RpcUpdateClientScores(playerId, initialScore);
    }
}
```

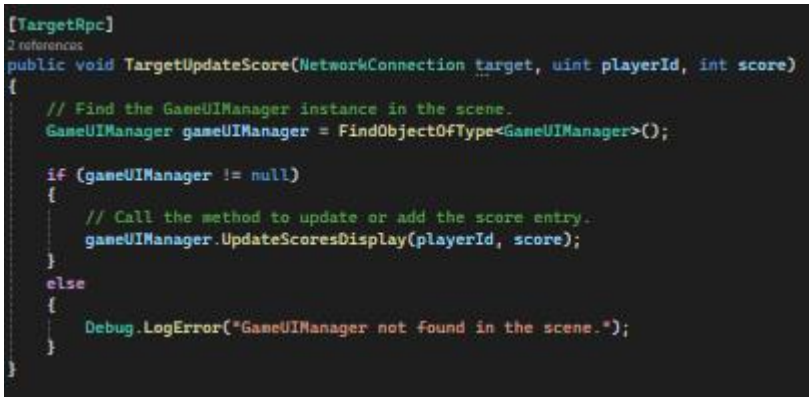
Figure 46

`OnRequestScores()` method responds to client requests for scores by sending the current scores to the requesting client and is useful in scenarios where client need to reconnect or join late.

```
1 reference
private void OnRequestScores(NetworkConnection conn, RequestScoresMessage msg)
{
    foreach (var score in playerScores)
    {
        // Send each score to the requesting client
        TargetUpdateScore(conn, score.Key, score.Value);
    }
}
```

Figure 47

This `TargetUpdateScore()` method, that is shown in figure, updates a specific client with score information. This method is called to ensure a client has up-to-date score data, particularly useful after handling specific requests. Shown in figure 48.

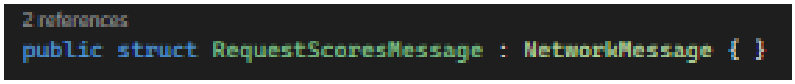


```
[TargetRpc]
2 references
public void TargetUpdateScore(NetworkConnection target, uint playerId, int score)
{
    // Find the GameUIManager instance in the scene.
    GameUIManager gameUIManager = FindObjectOfType<GameUIManager>();

    if (gameUIManager != null)
    {
        // Call the method to update or add the score entry.
        gameUIManager.UpdateScoresDisplay(playerId, score);
    }
    else
    {
        Debug.LogError("GameUIManager not found in the scene.");
    }
}
```

Figure 48

Finally, The following struct, shown in figure 49, is used to define a network message type that can be sent from clients requesting current score data.



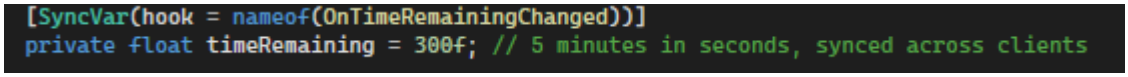
```
2 references
public struct RequestScoresMessage : NetworkMessage { }
```

Figure 49

4.1.8 – GameUIManager.cs

The GameUIManager.cs script manages the game interface elements, such as displaying scores and a countdown timer, in a networked environment using Unity and Mirror Networking. This script is essential for keeping players informed about the game state in real-time.

Firstly, I wanted to evaluate the SyncVar used for the timer, shown in figure 50, where it synchronises the remaining game time across all clients, ensuring every player sees the same countdown timer. The hook attribute method OnTimeRemainingChanged is called whenever timeRemaining changes, which updates the timer display accordingly.



```
[SyncVar(hook = nameof(OnTimeRemainingChanged))]
private float timeRemaining = 300f; // 5 minutes in seconds, synced across clients
```

Figure 50

In figure 51, we see the use of the hook mentioned above, along with the formatting of the timer for the display, seen in the method `UpdateTimerDisplay()`.

```
// Hook method called when timeRemaining changes
1 reference
void OnTimeRemainingChanged(float oldTime, float newTime)
{
    UpdateTimerDisplay(newTime);
}

1 reference
private void UpdateTimerDisplay(float timeInSeconds)
{
    // Format the time as MM:SS
    int minutes = Mathf.FloorToInt(timeInSeconds / 60);
    int seconds = Mathf.FloorToInt(timeInSeconds % 60);
    timerText.text = string.Format("{0:00}:{1:00}", minutes, seconds);
}
```

Figure 51

Figure 52 show the countdown logic to decrement the timeRemaining SyncVar, and the ServerCallback mean it is executed only on the server. This approach minimizes network overhead and ensures that only the server has authority over time management.

```
[ServerCallback]
Unity Message | 0 references
void Update()
{
    if (timeRemaining > 0)
    {
        timeRemaining -= Time.deltaTime;
    }
}
```

Figure 52

Next, shown in figure 53, we look at the use of `OnEnable()` and `OnDisable()` to hook on to the ScoreManager, so it doesn't have to constantly be polling, and still reflect the latest score, as it only updates the scoreboard when the scores update, shown in the .

```
Unity Message | 0 references
private void OnEnable()
{
    ScoresManager.OnScoreUpdated += UpdateScoresDisplay;
}

Unity Message | 0 references
private void OnDisable()
{
    ScoresManager.OnScoreUpdated -= UpdateScoresDisplay;
}
```

Figure 53

Finally, in figure 54 it shows the method `UpdateScoresDisplay()` that is used to handle the creation and updating of any score entries. If a new player's score needs to be displayed, it instantiates a new score entry prefab, adds it to the scoresPanel, and updates the relevant score text. This dynamic approach allows for scalability in player management.

```
3 references
public void UpdateScoresDisplay(uint playerId, int newScore)
{
    if (!playerScoreEntries.ContainsKey(playerId))
    {
        // Instantiate a new score entry and set it as a child of scoresPanel
        GameObject scoreEntryGO = Instantiate(scoreEntryPrefab, scoresPanel);
        TextMeshProUGUI scoreText = scoreEntryGO.GetComponent<TextMeshProUGUI>();

        // Add to dictionary for quick access
        playerScoreEntries[playerId] = scoreText;
    }

    // Update the score text
    playerScoreEntries[playerId].text = $"Player {playerId}: {newScore}";
}
```

Figure 54

4.1.9 – MainMenu.cs

The MainMenu.cs script serves as a comprehensive interface controller for the main menu of my game, incorporating elements of network setup, audio and display settings, and scene management using Mirror for networking. It manages various user interactions such as setting resolutions, adjusting volumes, and joining or hosting games.

Firstly, as seen in figure 55 each of the different tabs have their own GameObject, containing all of the elements required. This organisation made it easier to code for an implement the menu

```
public GameObject settingsTab;
public GameObject mainMenuTab;
public GameObject controlsTab;
public GameObject joinTab;
```

Figure 55

The `Start()` method seen in figure 56 initialises the resolution settings and populates the dropdown menu for the resolutions. It also sets the default IP for the join tab, allowing for easier starting of games. Furthermore, it sets all the tabs to inactive other than the `mainMenuTab` allowing for it to start on the right page.

```
Unity Message | 0 references
public void Start()
{

    resolutions = Screen.resolutions;

    resolutionDropdown.ClearOptions();

    // Set the input field's text to "localhost"
    joinIpAddressInputField.text = "localhost";

    List<string> options = new List<string>();

    int currentResolutionIndex = 0;
    for(int i = 0; i < resolutions.Length; i++)
    {
        string option = resolutions[i].width + " x " + resolutions[i].height;
        options.Add(option);

        if (resolutions[i].width == Screen.currentResolution.width &&
            resolutions[i].height == Screen.currentResolution.height)
        {
            currentResolutionIndex = i;
        }
    }

    resolutionDropdown.AddOptions(options);
    resolutionDropdown.value = currentResolutionIndex;
    resolutionDropdown.RefreshShownValue();

    mainMenuTab.SetActive(true);
    settingsTab.SetActive(false);
    controlsTab.SetActive(false);
    joinTab.SetActive(false);
}
```

Figure 56

There are a couple of these tab open/closers, so I wont screenshot them all, but they all look like figure 57 provided, and made the programming a lot easier, and ensure that navigating the menu is intuitive and clear.

```
0 references
public void OpenSettings()
{
    settingsTab.SetActive(true);
    mainMenuTab.SetActive(false);
}

0 references
public void CloseSettings()
{
    mainMenuTab.SetActive(true);
    settingsTab.SetActive(false);
}
```

Figure 57

Figure 58 demonstrates the settings that could be changed, using the Unity built-in options, along with the resolution, which required a lot more work to set up. This functionality is crucial for allowing players to customize their gaming experience, and allowing different tiers of hardware to be supported

```
0 references
public void SetVolume (float volume)
{
    audioMixer.SetFloat("Volume", volume);
}

0 references
public void SetQuality(int quality)
{
    QualitySettings.SetQualityLevel(quality);
}

0 references
public void SetFullScreen(bool isFullscreen)
{
    Screen.fullScreen = isFullscreen;
}

0 references
public void SetResolution(int resolutionIndex)
{
    Resolution resolution = resolutions[resolutionIndex];
    Screen.SetResolution(resolution.width, resolution.height, Screen.fullScreen);
}
```

Figure 58

These methods, shown in figure 59, provide the functions to start hosting a game or to join a game using a specified IP address, transitioning the player to the online scene specified in the NetworkManager.

```
0 references
public void HostGame()
{
    networkManager.StartHost();
    UnitySceneManager.LoadScene(networkManager.onlineScene);
}

0 references
public void JoinGame()
{
    if (joinIpAddressInputField != null)
    {
        networkManager.networkAddress = joinIpAddressInputField.text;
        networkManager.StartClient();
        UnitySceneManager.LoadScene(networkManager.onlineScene);
    }
    else
    {
        Debug.LogError("Join IP Address Input Field is not assigned.");
    }
}
```

Figure 59

Finally, one of the simplest functions, but is a solid requirement, the quit button, shown in figure 60.

```
0 references
public void QuitGame ()
{
    Debug.Log("Quit");
    Application.Quit();

}
```

Figure 60

4.1.10 – SingleDoorAnim.cs

This was one of my shortest scripts and enabled the doors to open whenever a player walked up to them. A trigger collider, along with the script seen in figure 61, was placed on the doors that allowed it to sense when a player came up to the door, triggering it to open.

```
using UnityEngine;
using Mirror;

Unity Script (9 asset references) | 0 references
public class NetworkedDoorAnim : NetworkBehaviour
{
    [SerializeField] private Animator doorAnimator;
    private bool isOpen = false;

    Unity Message | 0 references
    void OnTriggerEnter(Collider other)
    {
        if (!isServer)
            return;

        if (other.CompareTag("Player"))
        {
            isOpen = true;
            RpcToggleDoorState(isOpen);
        }
    }

    [ClientRpc]
    1 reference
    void RpcToggleDoorState(bool newState)
    {
        if (newState)
        {
            doorAnimator.SetTrigger("Open");
        }
    }
}
```

Figure 61

4.2 – Networking and Interactions

In this section, I explore how I have used networking to support the functionality of my game, as it is a multiplayer networked game.

4.2.1 – Player Movement and Animation

Animations are important as they give life to the players and the NPCs. The players and NPCs have both idle and walking animations as shown in figure 12, and uses animations provided from mixamo.com, which allows for less hectic development. Figure 62 shows the idle animation clip, which uses Unity's built-in animation window to work.

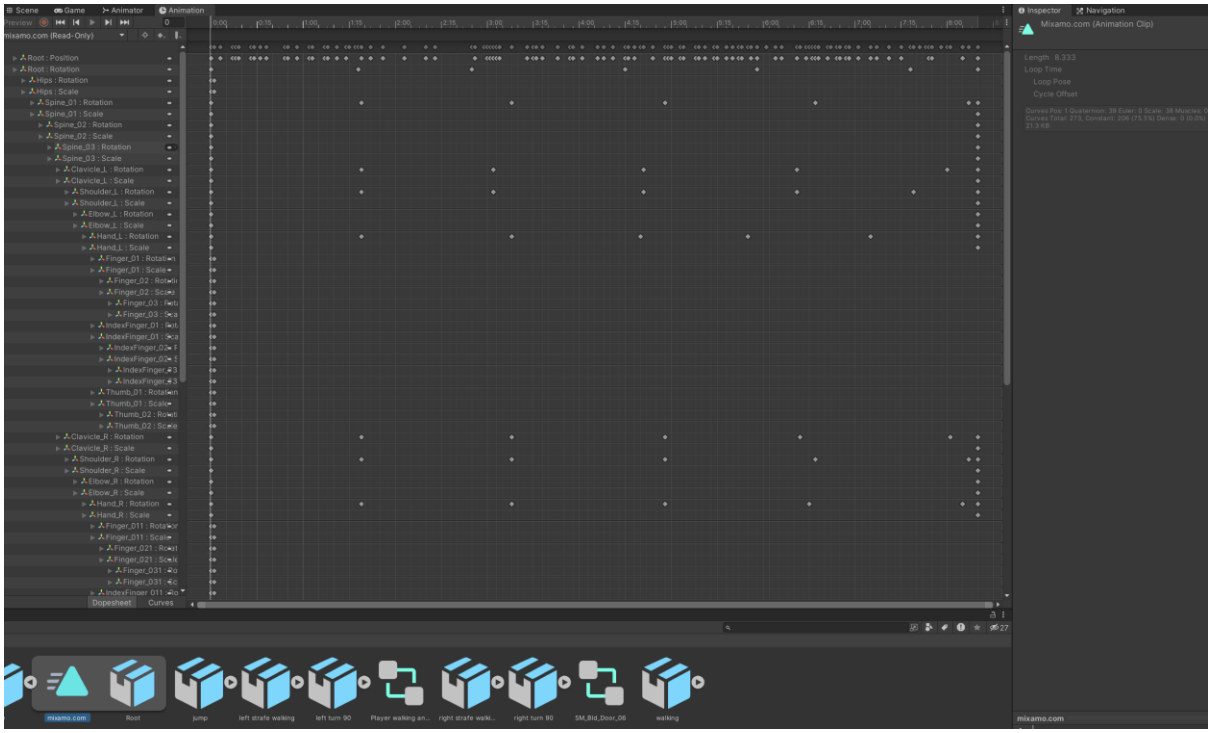


Figure 62

Using Mirror Networking, I was able to make the animations and show the player moving over the network to all players using the NetworkAnimator and NetworkTransform, shown in figure 63 and figure 64 respectively. A similar method was used for door opening animation.

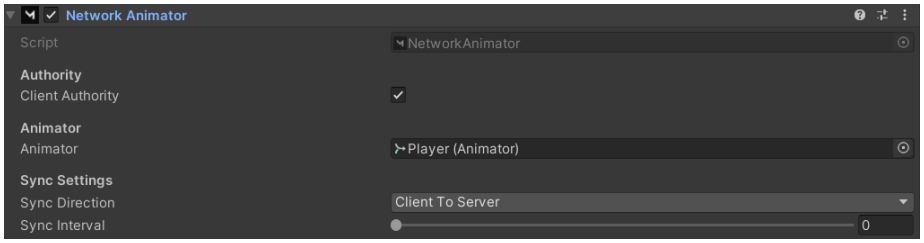


Figure 63

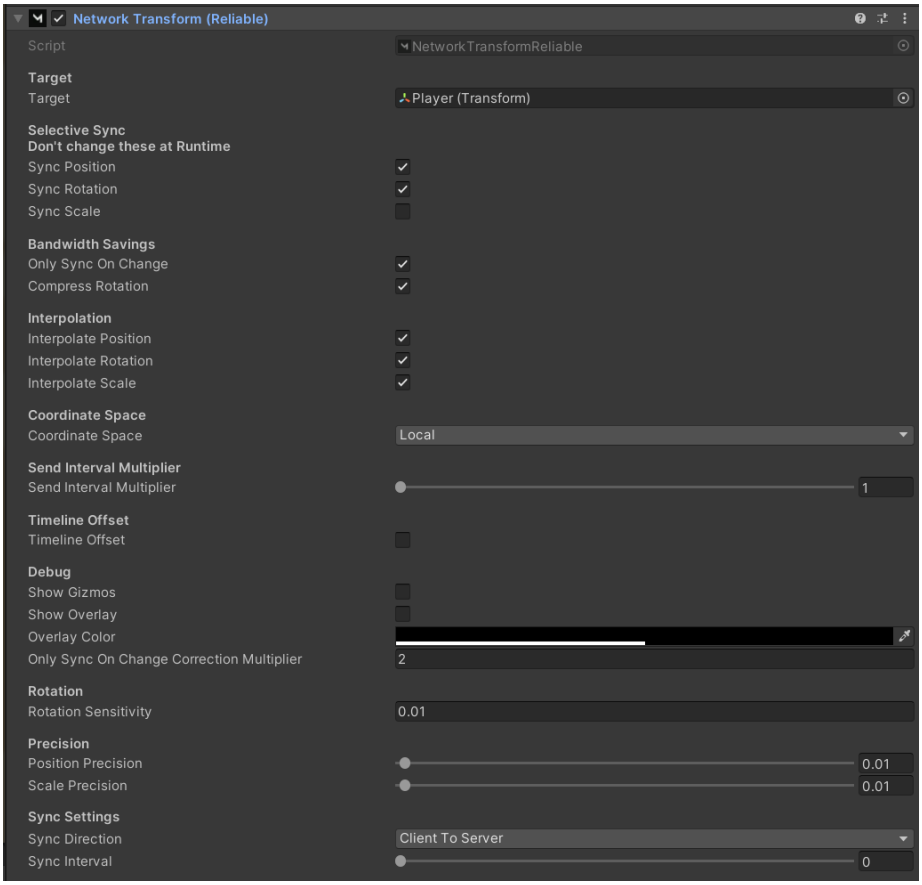


Figure 64

4.2.2 – Item and Item Switching

This was a tricky one, but I was able to do it by making the items children of the prefab of the player, and so it inherited all the aspects of the parent. To facilitate the item switching required a lot more work, and I will talk more about it in the challenges faced.

4.2.3 – NPC Movement and Global/Local Events

NPC Movement also used the Mirror Networking network animator and network transform that was used for the player and works very smoothly. To implement the global events was easy, as the items just had to affect anything with the tag player, however, local was hard to implement as it meant separating what each player was being affected by, without using a tag. I decided to use the netid that came with Mirror Networking, and it worked really well.

4.2.4 – Score Manager/ User Interface

The score manager/ game UI uses plenty of networking to work, as it requires the constant update of scores from multiple players in real-time and needed to set the right time. To ensure accuracy and minimize latency, I optimized the networking calls and used efficient data serialization. This approach helps in synchronizing the scores across various clients seamlessly, even in scenarios with high network traffic or varying connection speeds. Additionally, implementing robust error handling and retry mechanisms further ensures that all players' scores are updated reliably and promptly.

4.2.5 – Network Protocols and Topology

The main network protocol I opted to use was Threaded KCP, which is well-suited for real-time applications requiring low latency and high throughput. KCP (KCP UDP Protocol) is a reliable UDP-based protocol that outperforms TCP in scenarios where speed and efficiency are critical. Additionally, I implemented Mirror using a Peer-to-Peer topology, where one player acts as a host, each peer connects directly to others. This is a popular solution amongst indie games

4.3 – Challenges Faced

4.3.1 – Items not syncing properly.

One of the biggest challenges I faced was where the items weren't syncing between players properly, and once one player would change their item, it would change for everyone. I solved this with the use of the SyncDictionary. Then using what I learnt, I was able to set up the ScoresManager.

4.3.2 – Game crashing when client tried to catch ghost.

This was THE biggest challenge, as I was very new to Mirror Networking. I managed to fix it, and the problem was with how I had set up the RTC. The incorrect configuration was causing the game to crash whenever it tried to handle real-time communications. After reconfiguring the RTC settings and ensuring proper synchronization with the server, the game became stable and the crashes stopped.

4.3.3 – OnTriggerExit() not working as expected.

This caused me a big headache, but it turned out it wasn't even my fault. It was due to the fact that OnTriggerExit is unreliable in certain scenarios, such as when objects move too quickly or deactivate before exiting the trigger zone. To resolve this, I solved the problem by resetting the trigger boxes ensuring that exit events are handled correctly even when OnTriggerExit fails to fire. This approach helped stabilize the trigger logic and made the game mechanics more reliable.

4.3.4 – SetQuality() not working as expected.

Changing setting with SetQuality would cause issues with the game's performance and visuals, particularly with the lighting getting bugged out. This problem arose because changes in quality settings didn't properly reinitialize all graphical elements. To fix this, I ensured that all lighting settings were explicitly reset and recalculated whenever SetQuality() was called. This involved adding additional code to handle dynamic lighting updates, which helped maintain consistent lighting quality across different settings and resolved the glitches that were occurring.

Chapter 5 – Results

In this chapter, I demonstrate the results of my game implementation, showing off what I have done.

5.1 – Main Menu

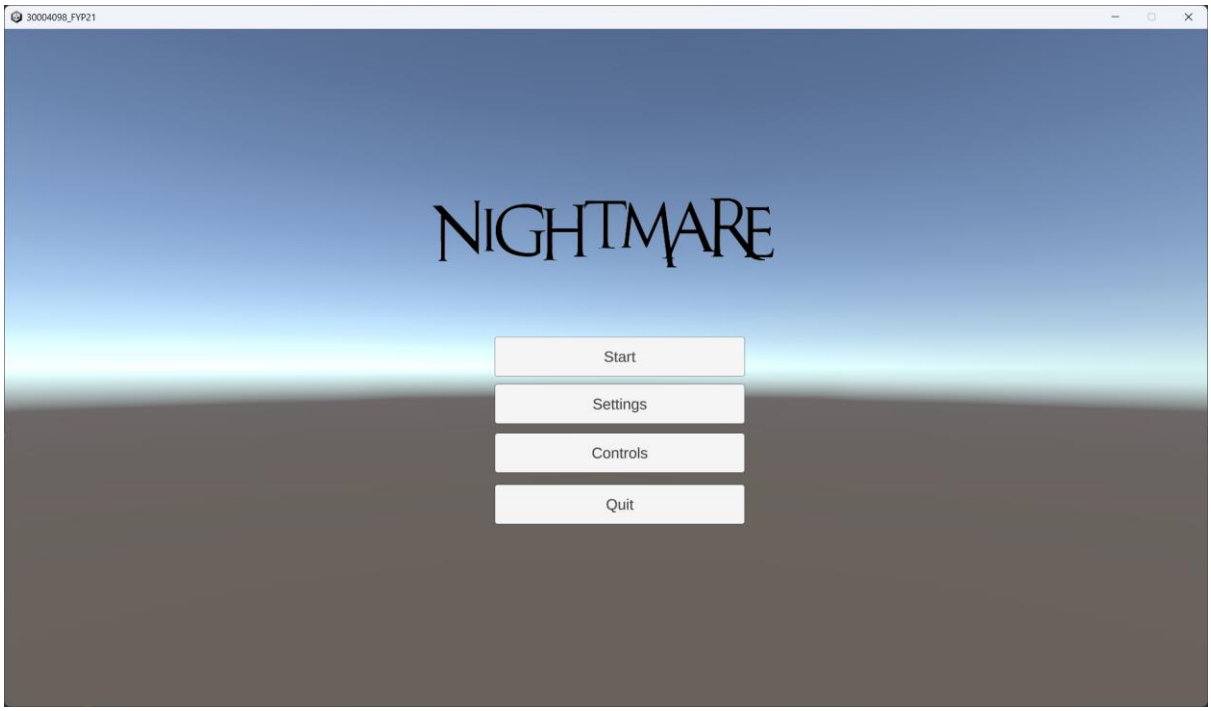


Figure 65 - Main Menu Screen

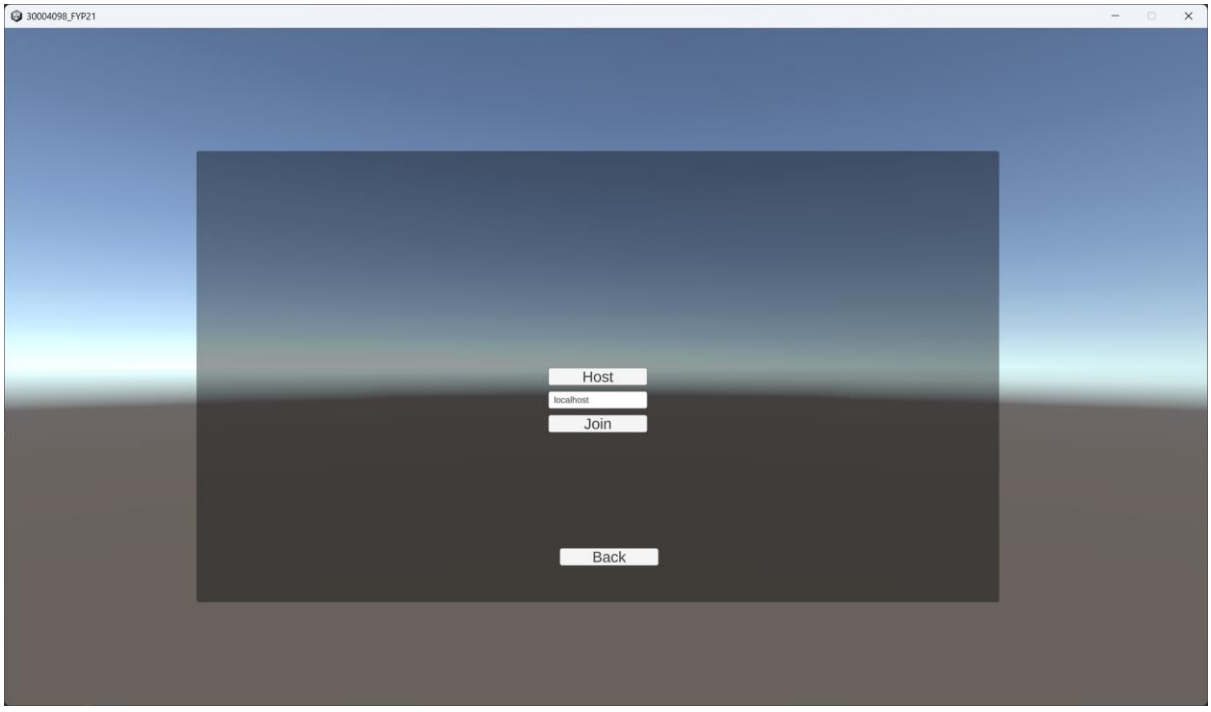


Figure 66 - When start button pressed.

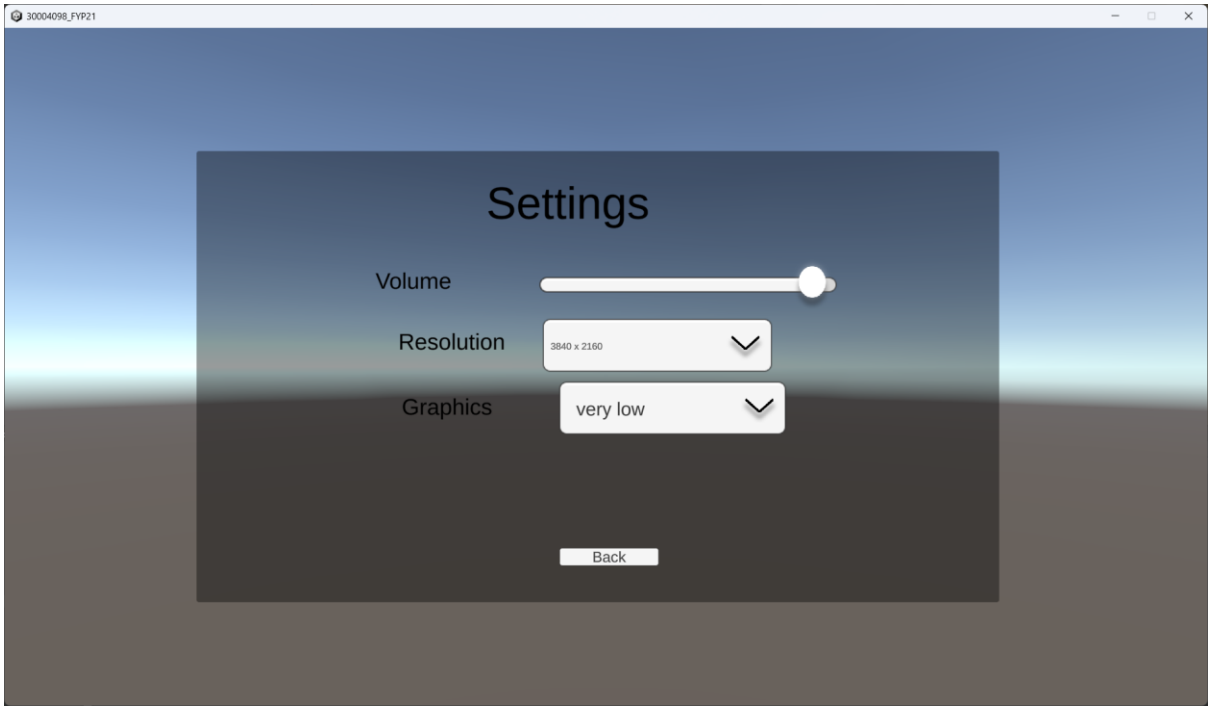


Figure 67 - When Setting button pressed

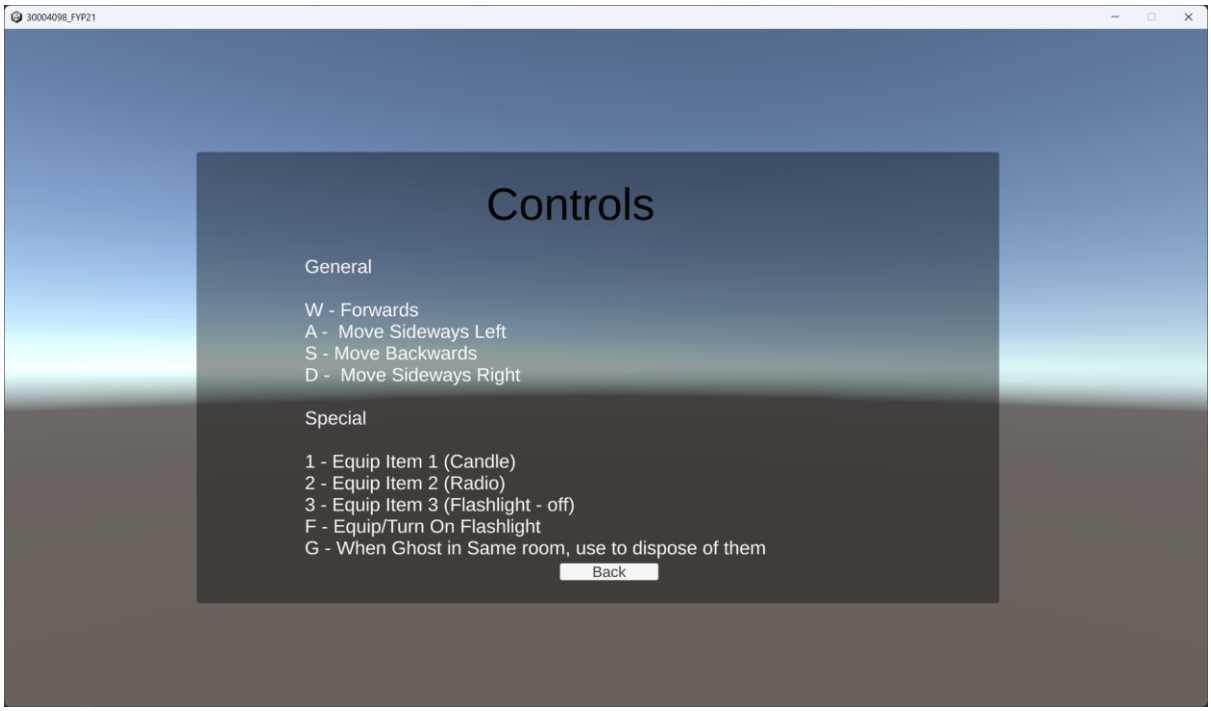


Figure 68 - when Controls button pressed.

5.2 – Game

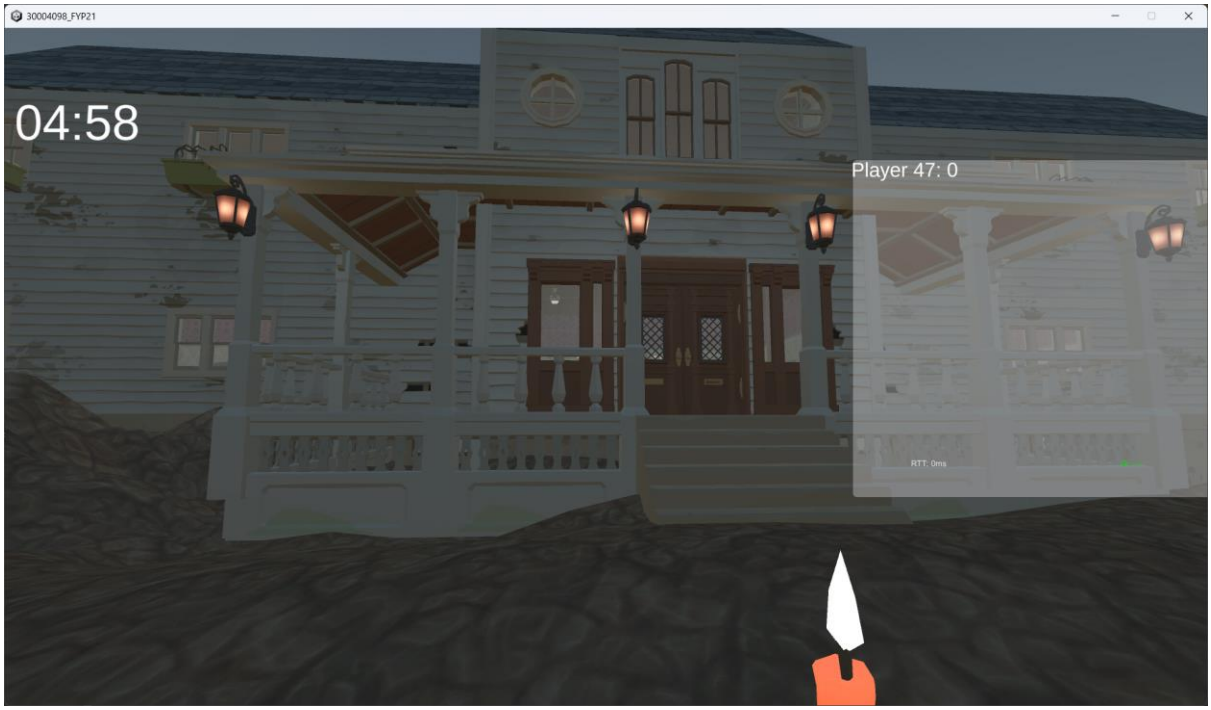


Figure 69 - Loading in to the game.

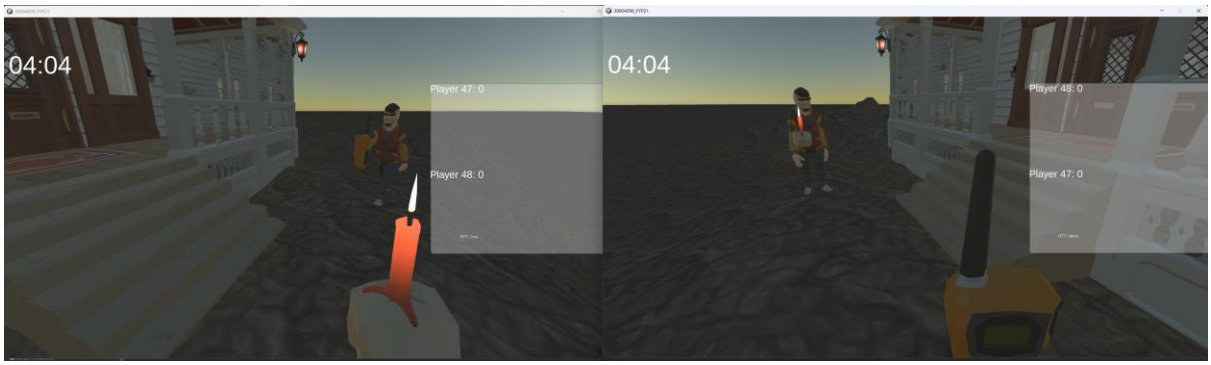


Figure 70 - Two player screens side to side, to show multiplayer capabilities.

Figure 71 - inside the house.

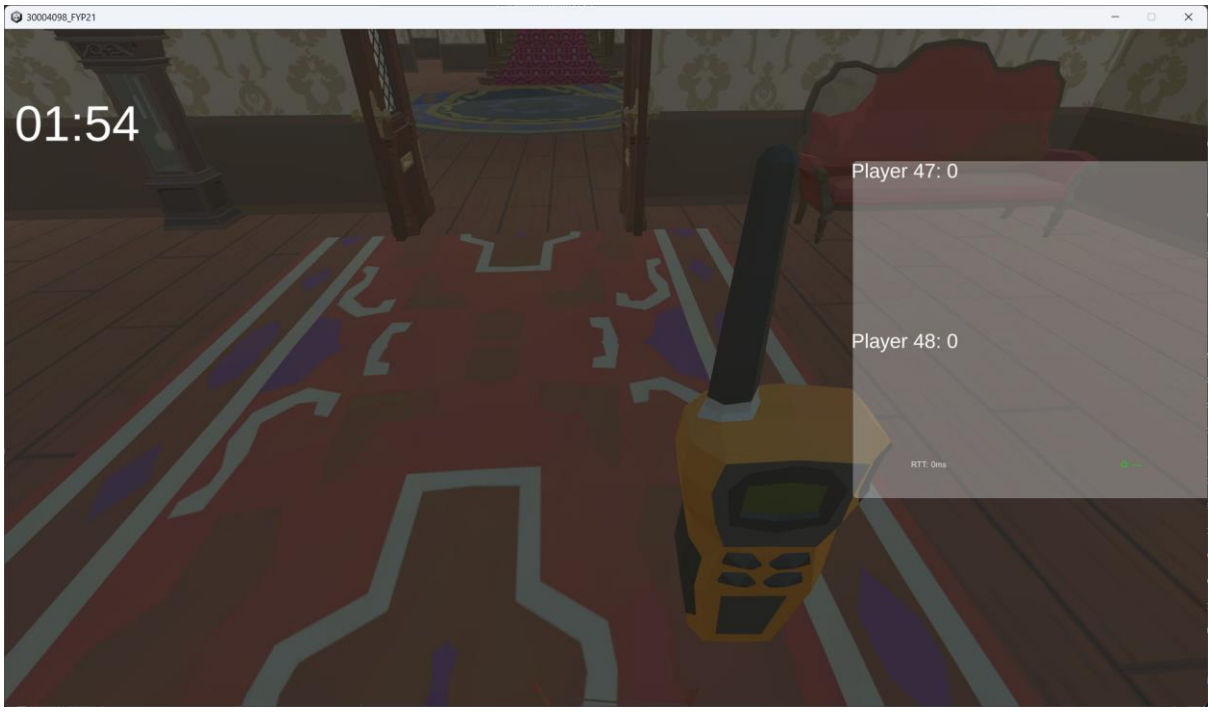


Figure 72 - Observing the radio, which increases in volume as you get closer to a ghost.

Figure 73 - Pressing F turns on flashlight.

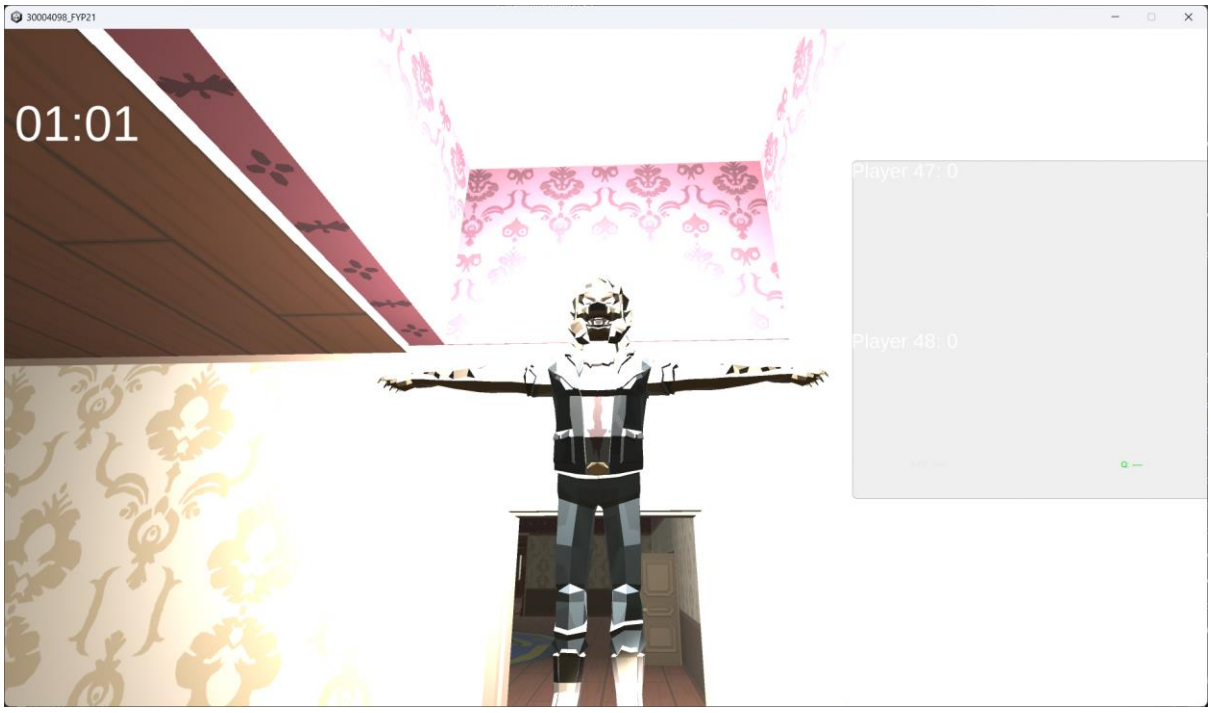


Figure 74 - A global jumpscare, triggered by observing player movement.

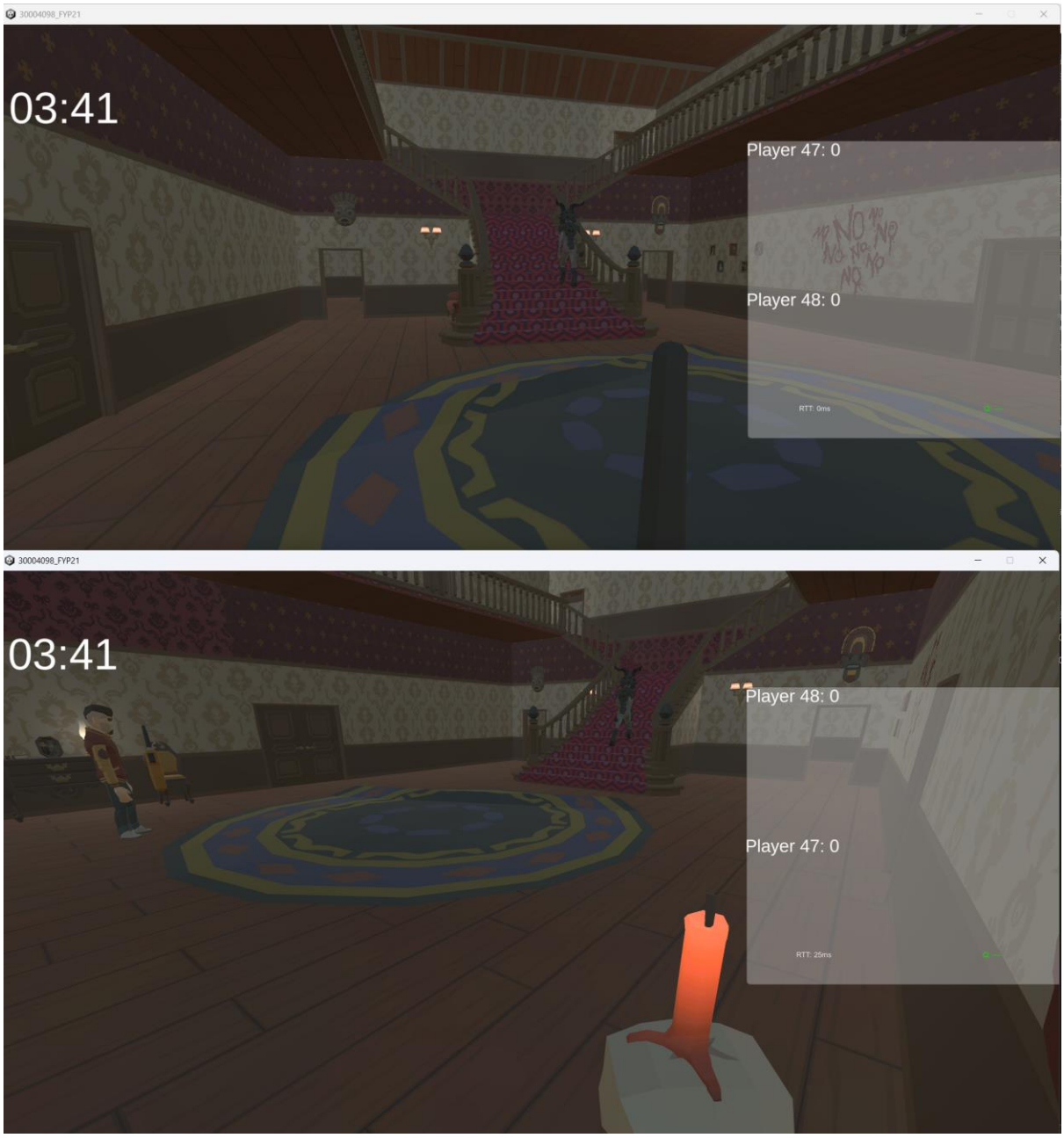


Figure 75 - Two different instances of game, showing the ghost, and the players holding their items.

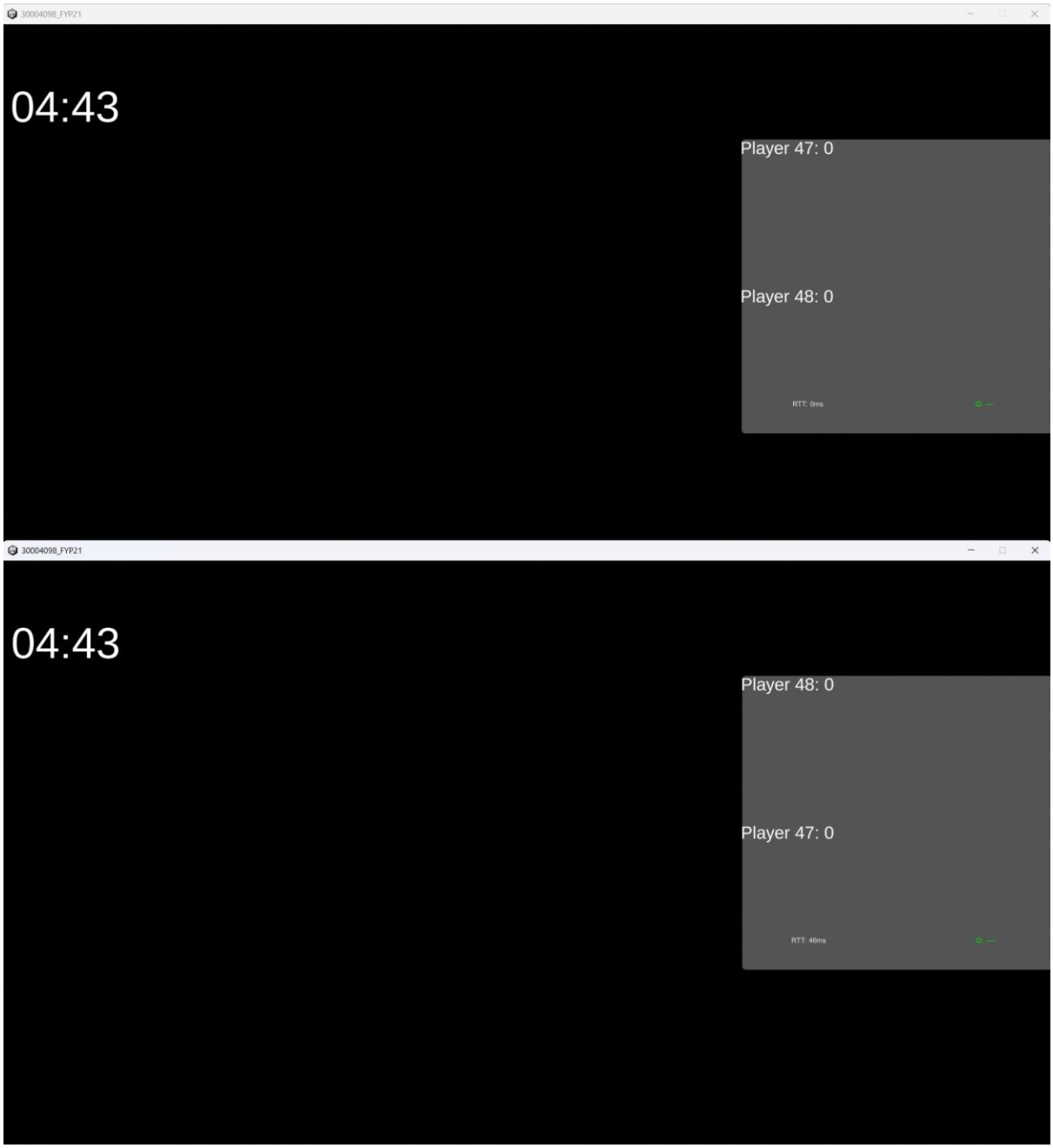


Figure 76 - Global effects can occur, where the players can go blind if stepping in the wrong place

Figure 77 - Top player is in same room as Ghost, so their candle is blown out.

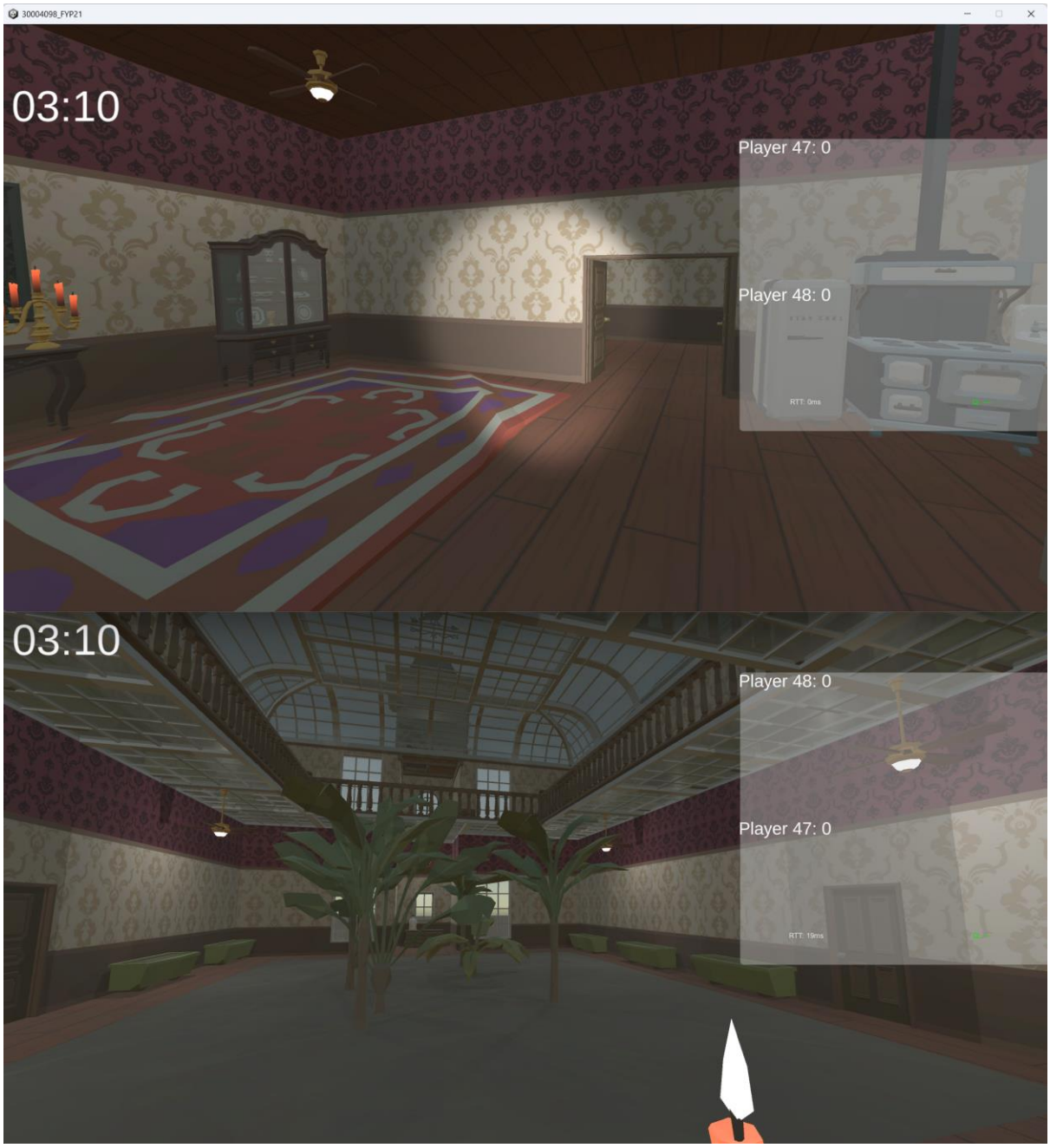


Figure 78 Lots of rooms to explore, and the networking works!

Figure 79 - Before Catching Ghost

Figure 80 - After Catching Ghost, the score increases by one and the ghost flies out of the room

5.3 – Map Design

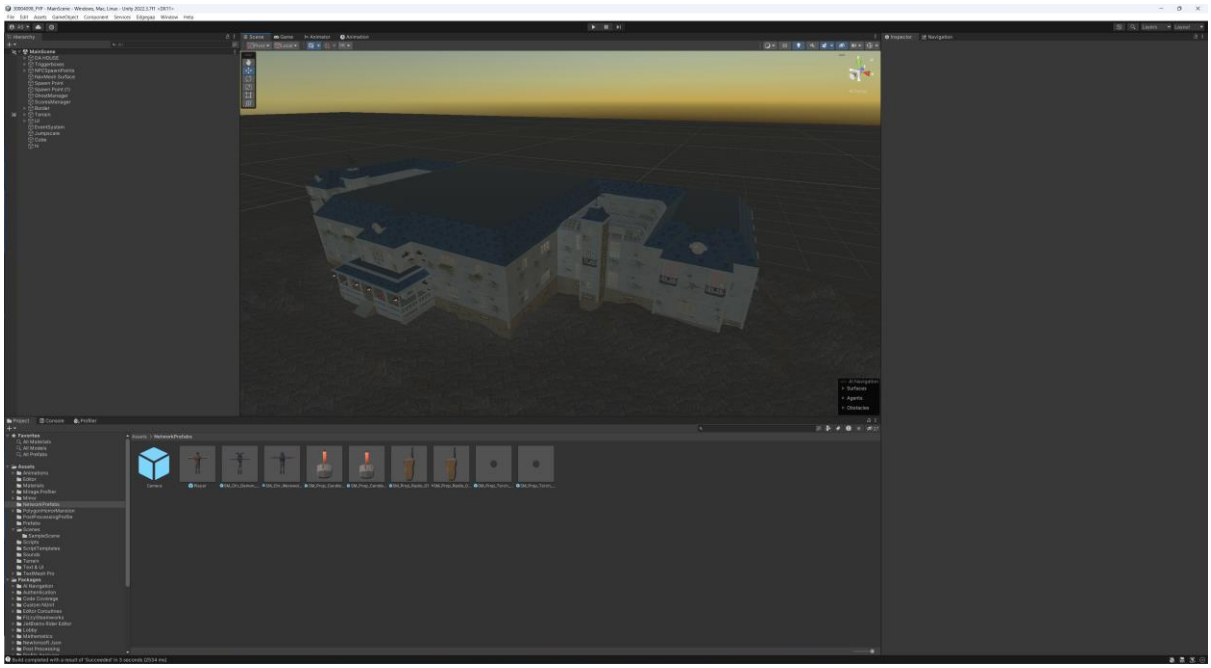


Figure 81 -Zoomed out view of house

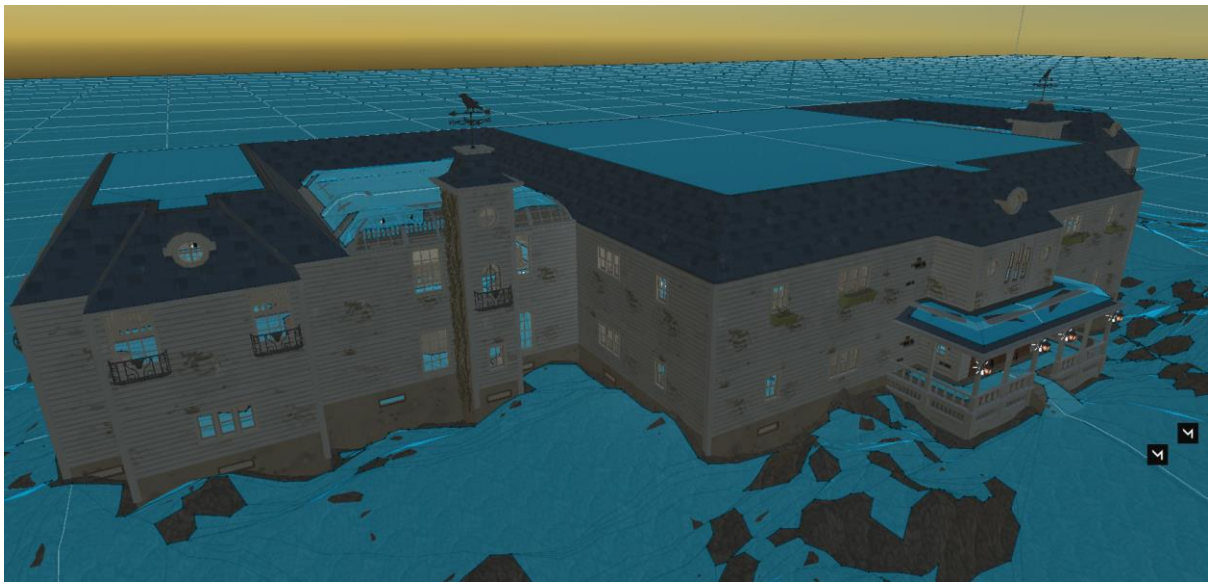


Figure 82 - AI NavMesh where the Ghost can walk around, it can't spawn on roof, but could walk on it.

Chapter 6 – Discussion and Analysis

In this chapter, I discuss and analyse the completed application, comparing against my objectives. Furthermore, I look at the quantitative results obtained from network testing and the qualitative results obtained from the Google Form survey done during the playtest of my game.

6.1 – Initial Objectives vs Final Solution

Initial Objectives	Completed?	Discussion
Robust Multiplayer Server which allows for at least 2 players	✓	Theoretically, an infinite number of players can join, but I have limited it to just 4 to allow for more cohesive gameplay
Each player should have items they can use.	✓	Each player has 3 items they can switch between, and each has unique functionality.
Have a suitable user interface	✓	The interface contains both a timer and score, contributing to the competitive nature of my game
Have a suitable menu	✓	Contains a suitable and easily navigable menu, allowing for better accessibility
Player/Ghost animations networked for everyone to see	✓	All animation are networked across any client who joins, and they all happen at the same time across them
Ghost Indicators that alert players to nearby supernatural activity	✓	Using Audio feedback, you can find the ghost using the Radio, and using visual feedback from the candle, you can figure out what room the ghost is in
Interactive gameplay	✓	There are jumpscares throughout the map, the doors are openable, and you can interact with game elements using the items given.
Engaging and exploratory design	✓	A big map with 26 unique rooms and 30 spawn zones for the ghost, with lost to explore.
Death Mechanics	❌	Unfortunately, I was unable to add these mechanics, as I was unable to get the ghost characters to interact with the players

Table 1 – Initial Objectives vs Final Solution, and discussion

6.2 – Quantitative Results Analysis

I wanted to use the different networking protocols that the Mirror Networking Library had to offer to provide me with some insight into which one would work best for me. Table 1 shows the results of this analysis, where I changes the protocols used a couple times

Protocol	**Average Ping (ms)**
KCP Transport (UDP)	29
Threaded KCP Transport (UDP)	25
Simple Web Transport (WebSocket - TCP)	40
Mirror Telepathy (TCP)	32

Table 2 - Average Ping depending on Protocol.

Analysing this, we see that there is not much difference between the pings and doesn't really matter for my use case. TCP is more reliable overall, and visually, it was found that there was a little less jitter relating to the transformation of different objects. In the end, I opted to use threaded KCP transport, which works on UDP, but provides more stability, while working on lower pings then TCP.

If the game was a fast-paced shooter for example, the 10ms of difference would be a massive improvement. Also, this test was run on a local machine to a local machine and if it were run over the internet we would have seen more conclusive results.

6.3 – Quantitative Results Analysis

In this section, I analyse and discuss results from a Google Forms survey I asked tester of my game to complete.

6.3.1 – Q1 – Experience with Lag

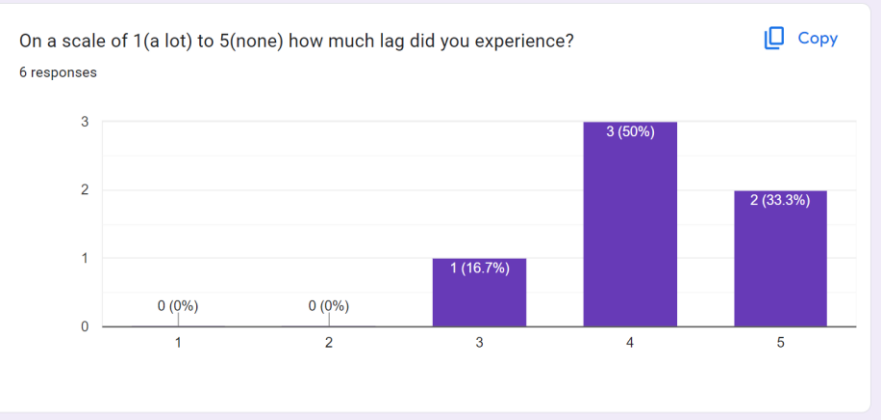


Figure 83

The figure above, figure 83, we observer the feedback from players in relation to the question "On a scale of 1(a lot) to 5(none) how much lag did you experience?" The result for these gave a mean of 4.17, indicating minimal lag experienced by most players. Most responses are 4 and 5, showing that the majority of players did not experience significant lag issues, which is a positive sign for network performance.

6.3.2 – Q2 – Visual and Audio Quality

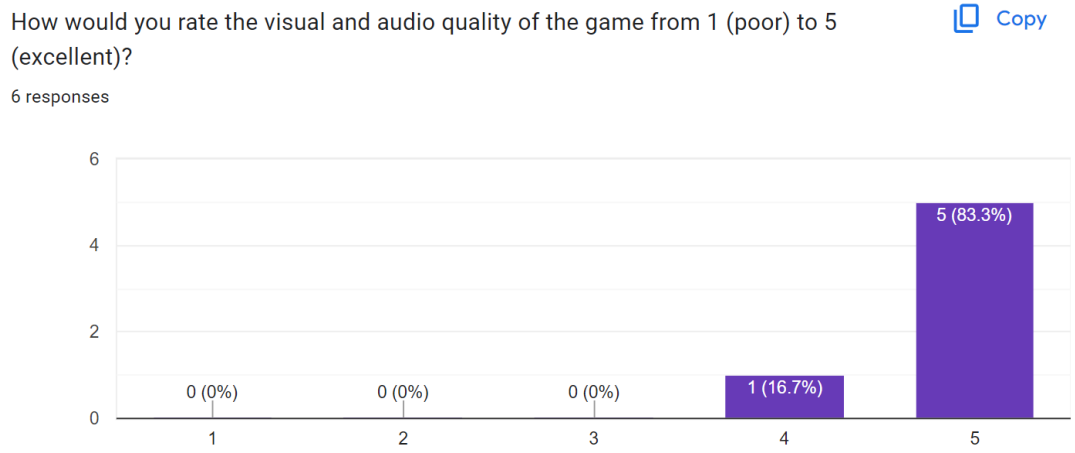


Figure 84

Figure 84 looks at the results of the question "How would you rate the visual and audio quality of the game from 1 (poor) to 5 (excellent)?" This gave the results that gave made up a mean score of Mean: 4.83, suggesting that most players found the visual and audio quality to be excellent. Furthermore, nearly all responses were the highest rating of 5, highlighting a strong positive reception of the game's aesthetics and sound design.

6.3.3 – Q3 – Social Interaction Features

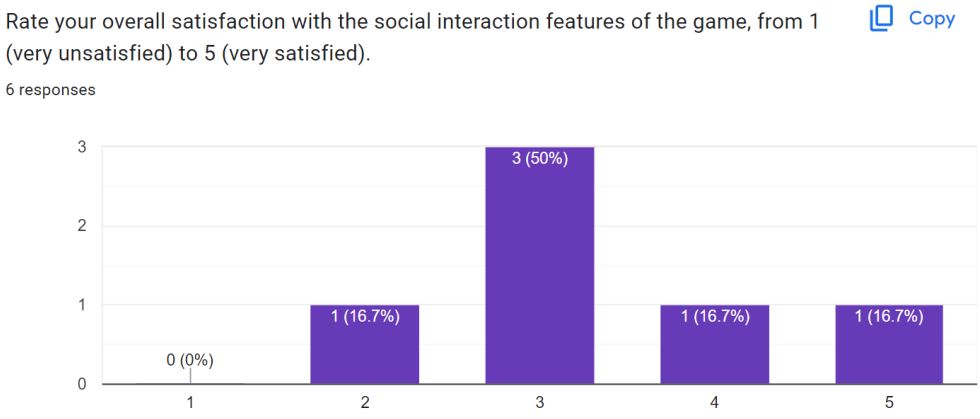


Figure 85

Looking at the next question, "Rate your overall satisfaction with the social interaction features of the game, from 1 (very unsatisfied) to 5 (very satisfied)", shown in figure 85, this had the lowest mean of the lot, with a 3.33, which is moderately positive but shows room for improvement in enhancing player interaction. The ratings vary more in this category, ranging from 2 to 5, which could indicate differing expectations or experiences with social features. This because other games, like Phasmophobia, have more features like proximity chat.

6.3.4 – Q4 – Item Variations

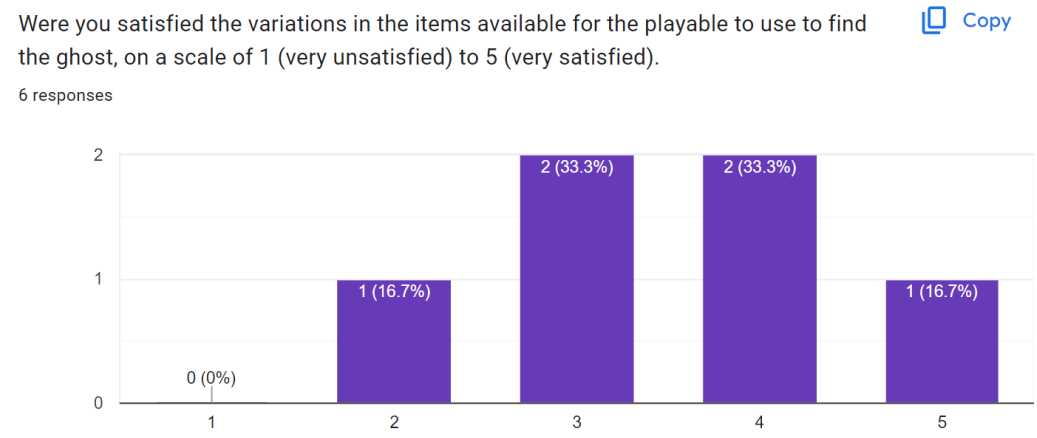


Figure 86

The results of the question "Were you satisfied the variations in the items available for the playable to use to find the ghost, on a scale of 1 (very unsatisfied) to 5 (very satisfied)", shown in figure 86, give a mean value of 3.5, similar to social features, and indicates a slightly above average satisfaction but suggests that diversity or utility of items could be improved. This too has a spread from 2 to 5, and maybe due to the reasons given above, as it doesn't have as many items as other games of this genre.

6.3.5 – Q5 – Overall Experience

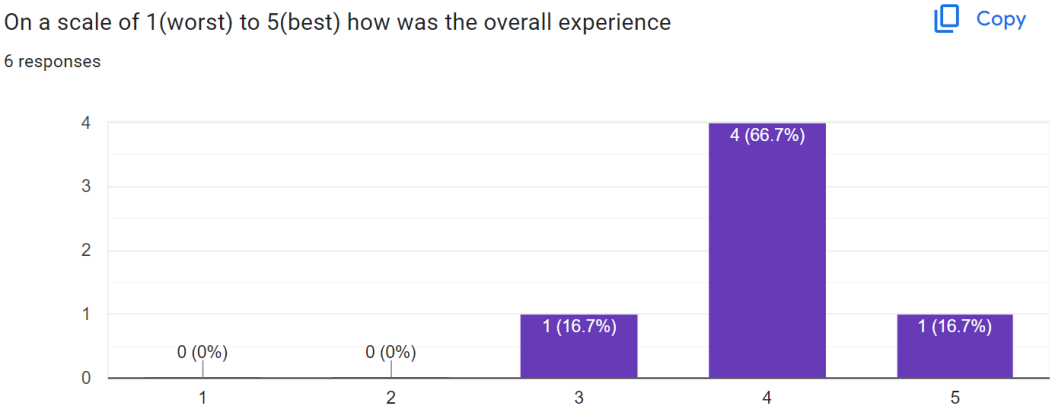


Figure 87

"On a scale of 1(worst) to 5(best) how was the overall experience", shown in figure 87, gives a mean score 4.0, indicating generally positive experiences among players. This shows that the project is overall a success in terms of player experience.

6.3.6 – Q6 – Semantic Feedback

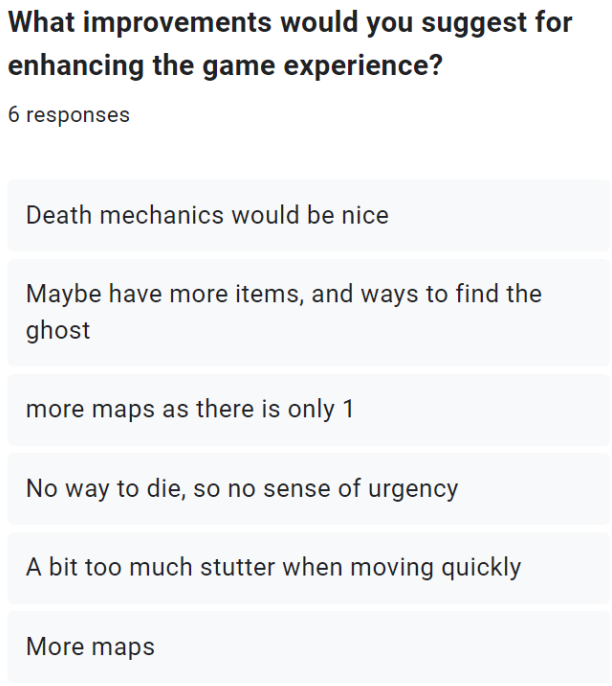


Figure 88

The final question I used for semantic feedback, asking the open-ended question "What improvements would you suggest for enhancing the game experience?", shown in figure 88. Analysing these, we find that multiple respondents mentioned the need for death mechanics or highlighted the lack of it. Also, multiple respondents wanted more maps and more items, to make the game more varied. Only one person mentioned performance issues.

6.4 – Discussion for Improvement

To improve the game, I would need to follow these points:

- **Enhancing Game Dynamics**: Introducing death mechanics might make the game more challenging and thrilling, which could appeal to players looking for more intense gameplay.

- **Expanding Game Content**: Adding more maps and items would likely satisfy players seeking variety and depth in their gaming experience.

- **Optimizing Performance**: Addressing performance issues such as stuttering during movement is essential for maintaining a smooth and enjoyable player experience.

Chapter 7 – Conclusion and Future Work

7.1 – Conclusion

In conclusion, the project in the end completed what it set out to do, getting a result that aligned with the aims set out in the introduction. A multiplayer networked immersive horror game was produced complete with robust infrastructure. It not only met the technical goals of creating a playable and engaging game environment but also advanced the understanding of network interactions within the context of Unity's game development framework.

This project successfully implemented various key features, by creating methods, that allowed for real-time player synchronization, efficient network communication, and dynamic interaction systems. These elements were crucial in ensuring that the game not only functioned seamlessly across a network but also provided an engaging and immersive experience for players.

Using the Unity Game engine, along with the Mirror Networking library, the project was able to prototype and assess changes in the code in real time, and this experience allowed for knowledge to be gained when tackling both expected and unforeseen challenges during the development process. From debugging complex network issues to refining gameplay mechanics for better user engagement, each step allowed for constant improvement, allowing for a more finished outcome.

Moreover, the project served as a practical application of theoretical concepts discussed in academic literature, bridging the gap between the concepts discussed in the literature review and real-world application. Implementing different horror techniques that allow the game to pray on the players psychology, was in big part due to this research. It demonstrated the practical challenges and solutions in implementing theories of networked interactions and user engagement in a game setting.

Player feedback, shown in the discussion, also helped shape the project, giving a constant loop that assisted with the agile methodology used. This iterative design process allowed for a better look at the players psychology, understanding their preferences, behaviours, and overall experience with the game. This feedback loop was instrumental in adjusting game mechanics, improving user interfaces, and refining the overall gameplay experience to better meet player expectations.

7.2 – Future Work

Future work on my project would involve the addition of a lobby, and connecting this lobby through Steam, which would need to be researched, as there are other services that can be used for this. Proximity chat would also be a good thing to research, as it allows for players to communicate with each other more easily. Furthermore, adding death mechanics, which were in the plans, would be another next course of action.

Other than just the practical item that can be added, code clean up and documentation would be a big priority, to allow for easier code reading. Reducing some of the methods to make them more concise, while refracting others is also important, to make it easier call them at different parts of the code.

Chapter 8 – Reflection

Looking back at when I started this project, I was very excited to use Unity in a practical way, making a networked game that I could play in the end. In hindsight, this excitement was way too much for such a hard and winded endeavour. However, I came out on the other side with the ability to understand networking within gaming and Unity. This project has given me a lot of insight into the development of games, and the issues that can occur, especially as a solo developer.

Reflecting on the development, this is stage which consumed most of the time I spent on this project. I needed to utilise a lot of new functions that I hadn't seen before within Unity, and the learning of a whole new library, the networking library Mirror. Throughout the development process, I also learned the importance of thorough testing and iterative improvements. Balancing the gameplay, ensuring server stability, and optimizing network performance were continuous tasks that demanded persistent attention. Each iteration brought its own set of challenges, from debugging unexpected behaviours to refining gameplay mechanics to improve the user experience.

The biggest challenge that I wasn't able to overcome was the death mechanic, I tried implementing them in a number of ways while developing, but in the end, I gave up and removed it all together. If I had more time, I think I would have been able to complete it.

The main thing this project highlighted for me was the need for effective time management and prioritisation, relating to my last point. I should have used some sort of software, maybe even just in GitHub, to track issues, and so I could move between thing more easily.

If I had done this project differently, I would have used a different networking library, and would've built it to support the Server-Client topologu.

In the end I came out as a more confident programmer, with an actual game to prove for it.

References

[1] - J. Clement, “Topic: Video game industry,” *Statista*, Jan. 10, 2024. https://www.statista.com/topics/868/video-games/#topicOverview

[2] - C. Garcia, “The Haunted House,” *Computer History Museum*, Oct. 31, 2012. https://computerhistory.org/blog/the-haunted-house/

[3] - J. T. T. Goldsmith and M. E. Ray, “Cathode-ray Tube Amusement Device,” *Google Patents*, Jan. 25, 1947. https://patents.google.com/patent/US2455992

[4] - E. Tretkoff, “October 1958: Physicist Invents First Video Game,” *www.aps.org*, Oct. 2008. https://www.aps.org/publications/apsnews/200810/physicshistory.cfm#:~:text=In%20October%201958%2C%20Physicist%20William

[5] - Z. Zwiezen, “3D Monster Maze Was the Very First Horror Game,” *Kotaku*, Oct. 17, 2020. https://kotaku.com/3d-monster-maze-was-the-very-first-horror-game-1845384173

[6] - G. Smith, “The Greatest and Most Influential Video Games of All Time,” *web.archive.org*, Oct. 25, 2020. https://web.archive.org/web/20201025213541/https://collider.com/galleries/greatest-video-games-of-all-time/ (accessed Apr. 17, 2024).

[7] - U. Team, “1993-1994: DOOM and DWANGO - UGO.com,” *web.archive.org*, Oct. 14, 2012. https://web.archive.org/web/20121014095704/http://www.ugo.com/games/history-of-online-gaming-1993-1994 (accessed Apr. 18, 2024).

[8] - S. A. Marriott, “Resident Evil: Outbreak - Overview - Allgame,” *web.archive.org*, Nov. 14, 2014. https://web.archive.org/web/20141114174648/http://www.allgame.com/game.php?id=43489 (accessed Apr. 18, 2024).

[9] - K. Reed, “Resident Evil Outbreak,” *Eurogamer.net*, Aug. 13, 2004. Accessed: Apr. 18, 2024. [Online]. Available: https://www.eurogamer.net/r-residenteviloutbreak-ps2

[10] - J. Trinca, “15 Years on, Left 4 Dead Still Rules the Genre It Created,” *VG247*, Nov. 17, 2023. https://www.vg247.com/left-4-dead-15-years-on (accessed Apr. 18, 2024).

[11] - T. Thompson and 2017, “The Perfect Organism: the AI of Alien: Isolation,” *Game Developer*, Oct. 31, 2017. https://www.gamedeveloper.com/design/the-perfect-organism-the-ai-of-alien-isolation

[12] - G. Gambetta, “Client-Server Game Architecture - Gabriel Gambetta,” *Gabrielgambetta.com*, 2019. https://www.gabrielgambetta.com/client-server-game-architecture.html

[13] - Iordanis Koutsopoulos, Leandros Tassiulas, and Lazaros Gkatzikis, “Client and Server Games in peer-to-peer Networks,” Jul. 2009, doi: https://doi.org/10.1109/iwqos.2009.5201412 .

[14] - N. E. Baughman, M. Liberatore, and B. N. Levine, “Cheat-Proof Playout for Centralized and Peer-to-Peer Gaming,” *IEEE/ACM Transactions on Networking*, vol. 15, no. 1, pp. 1–13, Feb. 2007, doi: https://doi.org/10.1109/tnet.2006.886289 .

[15] - S. A. Hamid, R. A. Abdulrahman, and D. R. A. Khamees, “What Is Client-Server System: Architecture, Issues and Challenge of Client -Server System (Review),” *Recent Trends in Cloud Computing and Web Engineering*, vol. 2, no. 1, pp. 1–6, Feb. 2020, doi: https://doi.org/10.5281/zenodo.3673071 .

[16] - T. Hampel, T. Bopp, and R. Hinn, “A peer-to-peer architecture for massive multiplayer online games,” *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games - NetGames ’06*, 2006, doi: https://doi.org/10.1145/1230040.1230058 .

[17] - J. Smed and H. Hakonen, *Algorithms and Networking for Computer Games*. John Wiley & Sons, 2017. Accessed: Apr. 21, 2024. [Online]. Available: https://books.google.co.uk/books?hl=en&lr=&id=ayQmDwAAQBAJ&oi=fnd&pg=PR15&dq=networking+prediction+algorithm+games&ots=i72FKnXVX8&sig=7V6e2XOzIM24NVmYZNIRzQuEAnE#v=onepage&q=networking%20prediction%20algorithm%20games&f=false

[18] - E. Larsson, “Movement Prediction Algorithms for High Latency Games,” *DIVA*, Aug. 11, 2016. https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A952076 (accessed Apr. 21, 2024).

[19] - S. Liu, X. Xu, and M. Claypool, “A Survey and Taxonomy of Latency Compensation Techniques for Network Computer Games,” *ACM Computing Surveys*, Feb. 2022, doi: https://doi.org/10.1145/3519023 .

[20] - A. Hussain, H. Shakeel, F. Hussain, N. Uddin, and T. L. Ghouri, “Unity Game Development Engine: a Technical Survey,” *ResearchGate*, Jul. 2020. https://www.researchgate.net/publication/348917348_Unity_Game_Development_Engine_A_Technical_Survey

[21] - P. Hattonpublished, “Unreal Engine 5 review,” *Creative Bloq*, Mar. 25, 2023. https://www.creativebloq.com/reviews/unreal-engine-5-review

[22] - H. A. J. Al Lawati, “The Path of UNITY or the Path of UNREAL? A Comparative Study on Suitability for Game Development,” *Journal of Student Research*, Jul. 2020, doi: https://doi.org/10.47611/jsr.vi.976 .

[23] - “Voice chat,” *Phasmophobia Wiki*. https://phasmophobia.fandom.com/wiki/Voice_chat#:~:text=Player%20chat%20is%20used%20to (accessed Apr. 21, 2024).

[24] - W. Admiraal, J. Huizenga, S. Akkerman, and G. ten Dam, “The concept of flow in collaborative game-based learning,” *Computers in Human Behavior*, vol. 27, no. 3, pp. 1185–1194, May 2011, doi: https://doi.org/10.1016/j.chb.2010.12.013 .

[25] - N. Coyle, “The Psychology of Horror Games,” *Psychology and Video Games*, Nov. 19, 2020. https://platinumparagon.info/psychology-of-horror-games/

[26] - G. Author, “Saying Isn’t Feeling: Evoking Emotional Engagement in Players,” *Gnome Stew*, Feb. 03, 2014. https://gnomestew.com/saying-isnt-feeling-evoking-emotional-engagement-in-players/ (accessed Apr. 21, 2024).

[27] - O. Alawode, “Understanding Internet Latency: Why It Matters to You,” *Nomad Internet*, Feb. 29, 2024. https://nomadinternet.com/blogs/countrynomad/understanding-internet-latency-why-it-matters-to-you (accessed Apr. 21, 2024).

[28] - K. Tokarev and A. Dragonis, “Creating Realistic Environments for a Horror Game,” *80.lv*, Jun. 10, 2019. https://80.lv/articles/creating-realistic-environments-for-a-horror-game/

[29] - M. Abbas, “Unlocking the Psychology behind Horror Game Design,” *Cubix Blogs*, Mar. 19, 2024. https://www.cubix.co/blog/psychology-behind-horror-game-design (accessed Apr. 21, 2024).

[30] - G. Bonfim, “The Importance of Narrative in Horror Games,” *Dailymobs*, Jul. 17, 2023. https://dailymobs.com/en/the-importance-of-narrative-in-horror-games/ (accessed Apr. 21, 2024).

[31] - “Network Latency: Understanding the Impact of Latency on Network Performance,” *Kentipedia*, Jul. 21, 2023. https://www.kentik.com/kentipedia/network-latency-understanding-impacts-on-network-performance/

Image Sources

[32] https://www.interactive.org/news/112018_spacewar_pioneer.asp

[33] https://en.wikipedia.org/wiki/File:MUD1_screenshot.gif

[34] https://en.wikipedia.org/wiki/File:3D-monster-maze-T-rex-2-steps-away.png

[35] https://www.reddit.com/r/localmultiplayergames/comments/a7ieie/

[36] https://gamerant.com/phasmophobia-screenshot-teases-update/

[37] https://www.gamegrin.com/directory/game/demonologist/images/

Appendix

GitHub Link - https://github.com/aadilrsattar/Nightmare-Game-FYP