Analiza praćenja pogleda u kolaborativnim i kompetitivnim VR igrama

Haramina, Emilia

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 483

ANALYSIS OF GAZE TRACKING IN COLLABORATIVE AND COMPETITIVE VIRTUAL REALITY GAMES

Emilia Haramina

Zagreb, June 2024

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 483

ANALYSIS OF GAZE TRACKING IN COLLABORATIVE AND COMPETITIVE VIRTUAL REALITY GAMES

Emilia Haramina

Zagreb, June 2024

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 04 March 2024

MASTER THESIS ASSIGNMENT No. 483

Student:	**Emilia Haramina (0036526081)**
Study:	Computing
Profile:	Network Science
Mentor:	prof. Lea Skorin-Kapov

Title: **Analysis of Gaze Tracking in Collaborative and Competitive Virtual Reality Games**

Description:

Gaze tracking in virtual reality (VR) involves monitoring and interpreting the direction of a user's gaze within a VR environment. This technology typically utilizes specialized sensors, cameras, or infrared light to accurately track eye movements and determine where the user is looking. In VR games, gaze tracking can provide insights into players' behavior, preferences, strategies, reactions, and decision-making processes. This information can be valuable for player profiling, optimizing user interfaces, content placement, and overall user experience design. Additionally, gaze tracking can be used for features such as gaze-based aiming or interaction. Your task is to develop two simple multiplayer VR games: one that is collaborative and the other competitive. The objective is then to conduct a user study designed to analyze gaze patterns in both developed games. Collected data should be analyzed to investigate how gaze patterns differ across various scenarios, such as in competitive vs. collaborative games, and scenarios differing in implemented interactions.

Submission date: 28 June 2024

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

DIPLOMSKI ZADATAK br. 483

Pristupnica:	**Emilia Haramina (0036526081)**
Studij:	Računarstvo
Profil:	Znanost o mrežama
Mentorica:	prof. dr. sc. Lea Skorin-Kapov

Zadatak: **Analiza praćenja pogleda u kolaborativnim i kompetitivnim VR igrama**

Opis zadatka:

Praćenje pogleda (engl. gaze tracking) u virtualnoj stvarnosti (engl. Virtual Reality, skr. VR) uključuje praćenje i tumačenje smjera korisnikovog pogleda unutar VR okruženja. Ova tehnologija obično koristi specijalizirane senzore, kamere ili infracrveno svjetlo kako bi točno pratila pokrete očiju i odredila gdje korisnik gleda. U VR igrama, praćenje pogleda može pružiti uvid u ponašanje, preferencije, strategije, reakcije i procese donošenja odluka igrača. Te informacije mogu biti korisne za profiliranje igrača, optimizaciju korisničkih sučelja, postavljanje sadržaja i općenito dizajn korisničkog iskustva. Dodatno, praćenje pogleda može se koristiti za ciljanje unutar VR okruženja ili interakcije na temelju pogleda. Vaš zadatak je razviti dvije jednostavne višekorisničke VR igre: jednu koja je kolabortivna, a drugu koje je kompetitivna. Nadalje, cilj je provesti korisničku studiju osmišljenu s ciljem analize uzoraka pogleda tijekom igranja obje razvijene igre. Potrebno je analizirati prikupljene podatke i istražiti kako se uzorci pogleda razlikuju u različitim scenarijima, poput kod kompetitivnih i kolaborativnih igara, te kod scenarija s različitim interakcijama.

Rok za predaju rada: 28. lipnja 2024.

I am sincerely grateful to my mentor, prof. dr. sc. Lea Skorin-Kapov, for her guidance throughout most of my college years and for her unwavering support to push me through exploring many interesting research topics. I would also like to express my gratitude to dr. sc. Sara Vlahović and mag. ing. Mirta Moslavac for their invaluable assistance and unfaltering enthusiasm in assisting me in any circumstance. From my heart, I express my sincerest gratitude to my parents for instilling in me the qualities of perseverance in my life, curiosity in my studies, and resilience in the face of any challenge.

Contents

1 Introduction

Gaze tracking refers to measuring and analyzing what a person is looking at by recording their eye movements [1]. It is a technology commonly used in retail and advertising, but also finds use for gaming and understanding the human brain [2]. The most common way the gaze of a user is tracked is by measuring the difference in the location of the pupil center and the reflection of the cornea provided by an infrared light [3]. This difference changes with the user looking in different directions, and it allows the correct gaze of the user's eye to be calculated via computer vision algorithms.

Virtual reality (VR) allows users to be immersed into a completely virtual world through the use of VR headsets. While it is still a relatively new technology, some VR headsets are able to track a user's gaze. However, in VR, the eyes do not always accurately point to where the user is looking at. Because the VR display is so close to the user's eyes, the gaze of both eyes does not meet at the point of focus of the user like it does in the real world [4]. However, the object a user is looking at can be found by tracing a virtual line from the eye in the calculated direction into the virtual world a user is immersed in. With this virtual line, the point a user is looking at can be found in an artificially made, virtual world. There are VR games that possess mechanics that use gaze tracking for controlling certain objects, or even the player. However, there are not many studies that analyze the gaze tracking of VR users. There are even less studies that analyze the difference of these data between collaborative and competitive game modes.

The analysis of gaze data can provide valuable insights into certain modifications or additions that developers can implement in their multiplayer VR games to direct the player's attention towards specific areas of the environment. Furthermore, the results of this analysis may differ between collaborative and competitive multiplayer VR games. Therefore, in order to gain insight into the gaze tracking data in different game modes

of multiplayer games, an application called **OVRseer** was developed in the scope of this thesis. **OVRseer** is a multiplayer VR application that provides two users with two maps they have to explore and for which they have to remember the layout. While they are playing both the collaborative and the competitive versions of the game, their gaze is recorded. Afterward, a user study is conducted using a general question form users had to fill out and the gaze data received from the recorded sessions is analyzed.

The analyzed gaze data consists of heatmaps of the player's gaze, as well as statistics for how many certain objects were looked at. Then, the similarities and differences between the two multiplayer game modes are investigated in detail. This analysis could prove useful in making multiplayer VR games in the future, as it could show what parts of the map players focus on in collaborative and competitive game modes, respectively. It is expected that in competitive mode, players will split up and explore the map themselves to not give the other player an advantage. On the other hand, players playing collaboratively could split up to cover more parts of the map or explore together to help each other remember parts of the map better. Additionally, players may look at objects of different colors or sizes when playing with or against the other player. To draw the players' attention in these two game modes, the results of this user study could be utilized in the future development of multiplayer VR games.

The main goals of this thesis are as follows:

1. list and review prior research regarding the use of gaze tracking in VR,

2. explain in detail the process of developing a multiplayer VR game that contains both collaborative and competitive versions while recording the gaze of the players, and

3. perform a user study to investigate the differences in gaze tracking between collaborative and competitive game modes in VR.

All of these goals are achieved in the five major chapters of this thesis. After the introductory chapter, the following chapter explains definitions of the key concepts used in this thesis, lists games that utilize gaze tracking technology, and provides an overview of previous work analyzing gaze data in VR. The third chapter describes the design of

a multiplayer VR game in which players explore different maps in multiplayer, then try to find certain places on each map in collaborative and competitive game modes. Afterward, the fourth chapter delves into the process of developing this multiplayer VR game that contains both collaborative and competitive modes while collecting gaze tracking information. The next chapter analyzes the data gained from playing the aforementioned game, as well as the recorded gaze data. Finally, the thesis presents a conclusion containing a thorough summary and overview of the achieved results. Additionally, the list of references, figures, tables, and abbreviations utilized in this thesis are also provided. The form used in the user study and all pictures contained in the developed VR game are also provided and can be found in the appendices.

2 Gaze Tracking in Virtual Reality

The Merriam-Webster dictionary defines VR as "an artificial environment that is experienced through sensory stimuli (such as sights and sounds) provided by a computer and in which one's actions partially determine what happens in the environment" [5]. By dissecting this definition, the main elements VR consists of can be further analyzed and explained. Firstly, the definition describes VR as an "artificial environment", meaning the space the user of VR sees is a completely artificial, three-dimensional (3D) environment simulated using a computer [6]. Following that, the definition says VR is "experienced through sensory stimuli (such as sights and sounds) provided by a computer", which is achieved by the use of various VR devices, such as headsets, earphones, controllers, gloves, bodysuits, and many others [7]. These VR devices allow the user to feel immersed in the artificial environment by simulating stimuli from the real world. Finally, according to the definition, the virtual environment is one "in which one's actions partially determine what happens in the environment". Using various VR devices, the computer can recognize the user's input and alter the virtual environment, which allows the user to interact with the artificial VR world and the objects inside it [8].

The application of VR has already made its way into many real-world scenarios, for example in medicine, media, education, entertainment, and many more [9]. VR technology is utilized in many industries because of the versatility of its content. In healthcare, VR has been used as pain relief for burn injuries, operation preparation, and treatment for mental health issues [10]. Trying on different clothes in a virtual world is a time-effective experience for shoppers. VR can also help architects by making it possible not only to see what a building or space will look like, but also how it will feel. VR has been found to also significantly increase learning retention levels, allowing students to use this technology to learn more effectively [11]. Additionally, VR makes for a more im-

mersive gaming experience, with users being able to see the gaming environment from the character's point of view.

Eye gaze tracking refers to the procedure of measuring and analyzing a person's eye movements to determine where they are looking [1]. This technology is widely used in retail and advertising, where a customer's attention can be analyzed to optimize sales approaches and customer experience [2]. Gaze tracking can also be used in gaming, allowing players to interact with a game using their eyes [1]. Neuroscience and psychology can use this technology to further understand how the brain processes visual information and how people react to different stimuli surrounding them.

The typical way eye gaze tracking is implemented is by continuously measuring the relative difference in the location of the pupil center and the reflection of the cornea provided by an infrared light invisible to the human eye [3]. The data changes depending on where the user is looking, allowing for the gaze of the eye to be calculated with computer vision algorithms. However, in VR, the eyes do not always point exactly where the user is looking. This is due to the VR display being so close in front of the eyes that the gaze of both eyes does not meet at an object at the center point, as is the case in the real world [4]. This incomplete information can be filled by looking at the virtual environment that users are in. By tracing a virtual line from the direction of the user's eye gaze into the virtual world, an object at a central point of the eye gaze can be deduced, and the data about the user's gaze can be collected. A visual of this process can be seen in Figure 2.1. On the left part of the Figure, gaze direction in the real world is shown, where the object a user is looking at is the central point of their eye gaze. On the right, when wearing a VR headset, the user is actually looking at a VR display in front of their eyes. In the virtual world, the user's eyes are placed in their environment according to their head tracking. Due to this, the gaze direction can be calculated by tracing a virtual line from the user's eye representation in the direction of their eye gaze. Some VR headsets that utilize gaze tracking technology include Meta Quest Pro, PlayStation VR2, and HTC Vive Pro Eye.

Gaze tracking was first utilized by having users wear specific contact lenses with a pointer attached to them, with their gaze being shown by the pointer [12]. Since then, gaze tracking technology has gone through many iterations, with it now being lightweight and able to be utilized within a VR headset. When used with VR technol-

ogy, gaze tracking can be paired with the simulated virtual environment to easily observe where the user is looking. In a virtual world, multiple objects can be combined into regions of interest, and in VR, it is easy to determine what regions a user looked at during a session [13].

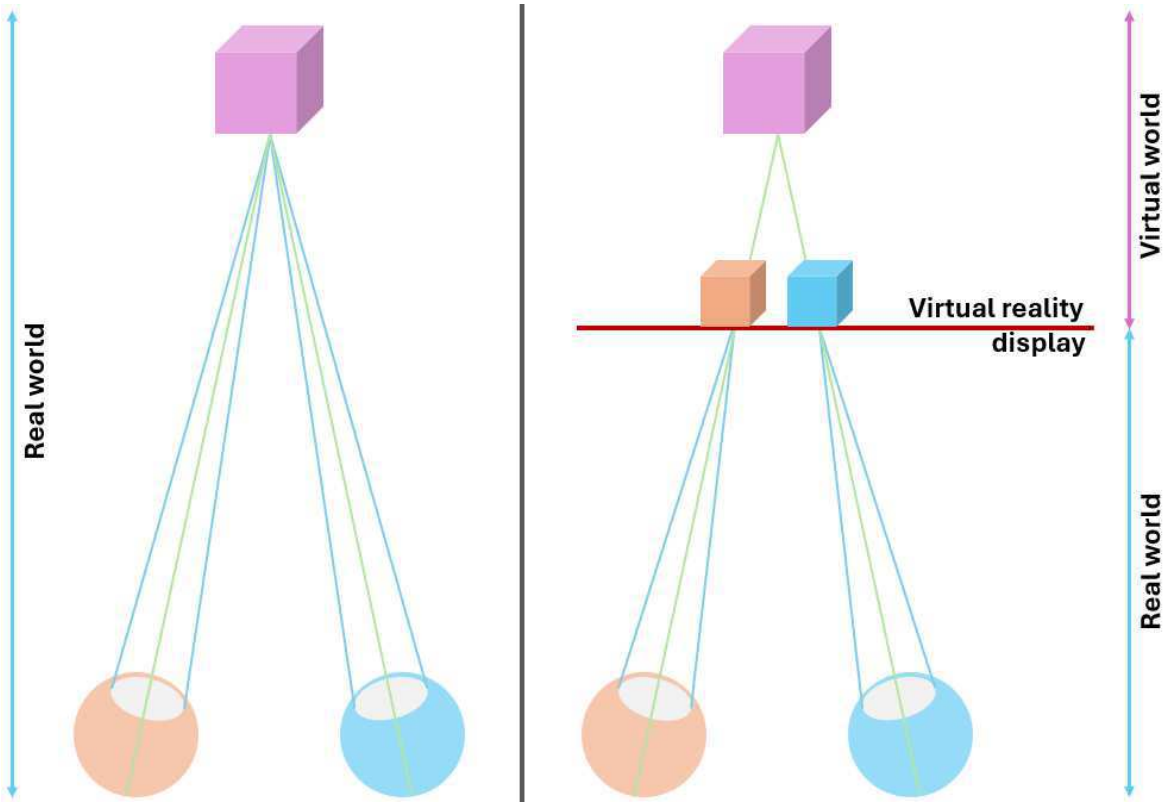

Figure 2.1: Gaze tracking in the real world (left) and in VR (right), adapted from [3]

Some games have already made their mechanics rely solely or partly on gaze tracking or, more commonly, blinking or movements of the user's eyes. For example, the game **Blink** changes the game environment when a player blinks, allowing for unique puzzles in the game [14]. **Before Your Eyes** is another game that utilizes an eye tracking mechanic in which blinking controls the game's story and affects its outcomes [15]. In VR, there are only a handful of games that take advantage of the relatively new gaze tracking technology. One of those games is **Synapse**, in which players can use telekinesis to move different objects or enemies around. However, their gaze dictates which objects or enemies this impacts, with the object or enemy becoming highlighted before the player activates the telekinesis ability [16]. Another VR game that uses eye tracking mechanics is **Switchback VR**, which contains an area of the game in which blinking causes enemies to change positions and eventually attack the player [17].

Some studies utilize user gaze tracking in VR for the sake of performing an analysis of the gathered data. One of those is a study by Duchowski et al., containing a VR application with the environment of an aircraft [12]. In this study, the application was developed to analyze the aircraft inspection training of users. The gaze of the users using the application was tracked to analyze the difference in performance between novices and experts. Another study by Mutasim, Batmaz, and Stuerzlinger analyzed eye-hand coordination training systems using gaze tracking [18]. The collected data presented how much additional time users needed to visually find a target and how much more time had passed before they selected the target.

A study by Hu places users in static and dynamic scenes, tracking their gaze while they look around the scenes [19]. The study found that the users' gaze is correlated to their head rotation velocities. Additionally, in dynamic scenes, the users' gaze was found to have strong correlations to the position of dynamic objects. Burova et al. investigated the usefulness of augmented reality (AR) simulation in VR [20]. They conducted a survey that utilized gaze data from the evaluation. Their results showcase the potential of AR simulations in VR when combined with gaze tracking.

While there are quite a lot of studies covering the overview, setup or implementation of gaze tracking in VR headsets and VR applications ([21], [22], [23], [24], [25], [26]), there are fewer studies that analyze gaze tracking of VR users ([12], [18], [19], [20]). Furthermore, no studies analyzed in the scope of this thesis analyzed the difference in gaze tracking data between collaborative and competitive games. One study that analyzed gaze tracking data in VR is a user study by Clay, König, and König, which presented users with a virtual city they had to explore [13]. While exploring the city, the gaze tracking information of participants was collected. Mainly, the time houses were looked at, which allowed a heatmap of the city to be made. The heatmap showed what parts of the city participants looked at most and how far away they were when looking at them. Afterward, participants were shown a series of pictures containing buildings in the virtual city, then had to answer how well they remembered the building in the pictures and whether they could find their way back to the building. The key results of this study include that most of the virtual city was visited during the given time of exploration and that most houses were seen by more than half of the participants. In addition, bigger

house complexes and more unique houses were gazed at longer than those placed in a row along the street. The study also concluded that the longer a house was looked at, the higher the subjective familiarity rating of a house was, but the correlation was lower than expected. Finally, there was no strong relation found between familiarity and viewing distance of a house. This user study served as a motivation for this Master's thesis.

3 Design of OVRseer

The design of the application necessary to conduct the user study is described in this chapter. **OVRseer** is a multiplayer game played in VR. Players explore two maps in two different game modes while trying to remember as many parts of the maps as possible. During exploration, their gaze is monitored to gather information regarding the objects and parts of the map the players are looking at. Afterward, they are given pictures taken on the map and must answer questions regarding familiarity with those pictures. Finally, for each picture, players have to return to the place where the picture was taken, gaining points depending on how fast they find it. The data obtained from their responses and the duration required to locate where a picture was taken are also preserved. These data can be combined with the previously mentioned gaze data to determine whether the location of a player's gaze has an impact on their familiarity with specific areas of the map. Furthermore, it could suggest a connection between players' gaze and their level of confidence in returning to certain places on the map. The two different game modes are collaborative, in which players work together, and competitive, in which they play against each other. In these two different game modes, the players' gaze may differ, and it would be helpful to know what parts of the map players focus on in each respective version. Using these data, it is possible to incorporate features in future multiplayer VR games that aim to direct players' attention towards specific areas of the virtual environment.

3.1 Game Analysis

OVRseer could be considered a puzzle game, with players having to remember parts of the maps so they could later recall them. The main elements of the game include exploration of the map, answering questions about a certain place on the map, and finding

their way back to where the picture of that place was taken. The theme of one map is modern, boasting an amusement park, while the theme of the other is fantasy, including a medieval village. The art style of both maps is low poly. The game is multiplayer, with two people able to play at the same time. Since the game is primarily developed with the aim of collecting and analyzing user eye gaze data and the gameplay is not long, there is no story players have to follow. **OVRseer** is a 3D game in VR that players play from a first-person perspective. It can be played on standalone VR devices, or on VR devices with a computer connection.

3.2 Map Design

OVRseer is set in two open maps left for players to explore. The first is an amusement park, with various attractions boasting vivid colors spread through the map. In this map, the players are able to walk on structured walkways, not being able to move on the grass. The other map includes a medieval village set in a fantasy-inspired world. The village is full of houses, with some being open for the players to explore inside. On this map, the player can move on any ground surface, but is restricted from moving too far away from the village.

Both the amusement park map and the medieval village map can prove useful when analyzing the player's gaze when exploring them. The vivid and differing colors of the attractions on the amusement park map can provide valuable insights into the colors that players tend to focus on. On the medieval village map, there is a central part of the village and some buildings scattered on the outskirts of the village. It would be helpful to determine whether players are primarily focused on the central area of the map, or if they also explore other parts of the map. Additionally, it should be noted that the data collected on these maps may exhibit variances when playing collaboratively or competitively.

3.3 Gameplay

Both maps contain the same gameplay, with the players starting in an area they can't move from. They are presented with menus, with one of the players able to change the multiplayer game mode and start the game. For players to be able to move using a VR

headset, VR controls are needed. The player can look around by moving their head, and move using their controllers. Players have an avatar which represents them and which can be seen by the other player. Players' avatar design consists of a head tracking their actual head movement. After one player starts the game, players have a certain amount of time to explore the map, instructed to remember as much as possible. Additionally, they have a menu on their left hand, visible when they turn their hand towards themselves. This hand menu shows the time they have left to explore the map.

After the exploration time has passed, players are presented with a set of pictures on their menus, one by one. For each picture, players have to individually answer questions regarding their familiarity of the place in the picture. Then, they have to find the place where the picture was taken. On the hand menu, the current picture and time left for finding where it was taken is shown. Once players find where the picture originated from, they are moved back to their starting area and given points depending on their performance. The faster they find where the picture was taken, the more points they gain. In collaborative mode, players gain points together. On the other hand, in competitive mode, players collect points individually, with the one with more points being the winner. Through the game, the gaze of players for objects of interest and the map as a whole is recorded for analysis.

4 Development of OVRseer

The application developed as a part of this thesis, **OVRseer**, is an immersive multiplayer VR game designed for the analysis of the difference in a player's gaze data between different versions of multiplayer games. The amount of time and total time the players looked at objects around the map, as well as the presence of the other player, is recorded. Additionally, their movement across the map and parts of the map their eyes focused on are also recorded for analysis. The application uses VR technology to create an immersive environment for players to explore. The game can only be played in pairs, with each player using their own VR headset. Two players can play this game on different networks, but for this thesis, both players were connected to the same one. It should be noted that, while any VR headset could be used to play the game, for the user study, at least one player should wear a gaze tracking VR headset. As a part of this thesis, one player used the Meta Quest 2, while the other wore the Meta Quest Pro, with the latter being capable of tracking a player's gaze.

4.1 Used Technologies and Tools

Unity is a game engine developed by *Unity Technologies*. Numerous platforms, including desktop, mobile, console, and VR, are compatible with the engine. It can be used to produce two-dimensional or 3D games, interactive simulations, and other kinds of experiences. Industries other than video gaming, including film, automotive, architecture, engineering, construction, and the military, have also adopted the game engine due to the versatility of content that can be produced using it [27]. The version of Unity used for the development of **OVRseer** is 2023.2.17f.

C# is a versatile, general-purpose, and multi-paradigm programming language. It includes static typing, strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented, and component-oriented programming disciplines. It was designed by *Microsoft* and debuted with *Visual Code* and the *.NET Framework* [28]. The version of C# used for the development of **OVRseer** is 9.0.

The **Meta XR All-in-One SDK** bundles several *Meta* SDKs together. It includes many features like advanced rendering, social, and community building, and provides capabilities to build immersive experiences in both VR and mixed reality [29]. **OVRseer** was developed with version 64.0.0.0 of this SDK.

Photon Unity Networking 2 is a networking solution developed by *Exit Games*. It simplifies the development of multiplayer games in Unity. A globally distributed Photon Cloud hosts Photon Pun 2 games to reduce latency and offer the shortest round-trip timings. The client-server architecture is used [30]. The version of Photon PUN 2 used for the development of **OVRseer** is 2.46.

Cognitive3D is a 3D analytics platform that tracks human behavior in VR/AR simulations and provides useful insight into the data [31]. With a simple integration of their package into Unity and exporting a scene, the player sessions can be replayed with their gaze data shown. A gaze heatmap can be analyzed for sessions, gaze data for specific objects is measured, the data can be aggregated from multiple sessions, and other data from VR sessions can be looked at. The version of the Cognitive3D SDK used in the development of **OVRseer** is 1.4.7. The Pro plan is used for analyzing the data. Some of the gaze data is available in the free version as well, but most of it requires the Pro plan to be viewed.

Microsoft Visual Studio 2022 is an integrated development environment from Microsoft [32]. Along with supporting numerous programming languages, and many more available through plugins, it contains an integrated debugger. Unity is set up to operate in this environment. **OVRseer** was developed with Microsoft Visual Studio 2022 version 17.7.4.

The **Unity Asset Store** is a marketplace of assets that can be used in a Unity project [33]. The *Meta XR All-in-One SDK* is used to simplify a user's interactions with a virtual

world through VR [34]. To make the development of the multiplayer version easier, *PUN 2 - FREE* is employed [35]. Two maps in **OVRseer** from the Unity Asset Store were also used. One of those maps is *Amusement park 1*[1], which contains attractions set in a cartoon-like amusement park setting [36]. The other map is *Dreamscape Village - Stylized Fantasy Open World*[2], a large medieval fantasy village [37].

4.2 Implementation and Features of OVRseer

In this section, the implementation of **OVRseer** will be discussed. Game features will be described, as well as their implementation using the aforementioned technologies and tools. Firstly, the integration of the Meta XR All-in-One SDK will be explained, allowing players to use VR headsets to interact with the virtual environment. Secondly, the multiplayer implementation will be discussed, which makes it possible for two players to play **OVRseer** together over the Internet. Next, the design of two scenes will be analyzed. Following that, the thesis will explain how gaze tracking was implemented using Cognitive3D. Afterward, the necessary steps to implement the exploration of both maps will be explained. Then, the finding of places where pictures of the map were taken will be discussed in detail. Finally, the thesis will go over how players get points when playing.

4.2.1 Meta XR All-in-One SDK Integration

To allow players to explore the virtual world, their head and hands or controller movement need to be tracked in the virtual space. Players need to see their hands or controllers, interact with the game menus present in the game, and be able to move and turn. To achieve this, the Meta XR All-in-One SDK was used.

VR Player Representation

The *OVRCameraRigInteraction* prefab, seen in Figure 4.1, from the Meta XR All-in-One SDK package was dragged into the Unity hierarchy. This prefab contains objects that have positions and rotations corresponding to the real-life location of the player's head and hands. In addition, the prefab activates and deactivates objects should the player

[1]https://assetstore.unity.com/packages/3d/environments/landscapes/amusement-park-1-235574
[2]https://assetstore.unity.com/packages/3d/environments/fantasy/dreamscape-village-stylized-fantasy-open-world-244797

switch from using their hands to using the headset's controllers and the opposite. With the Meta Quest Pro, however, a player can use one hand and one controller simultaneously, allowing for more freedom of interaction.

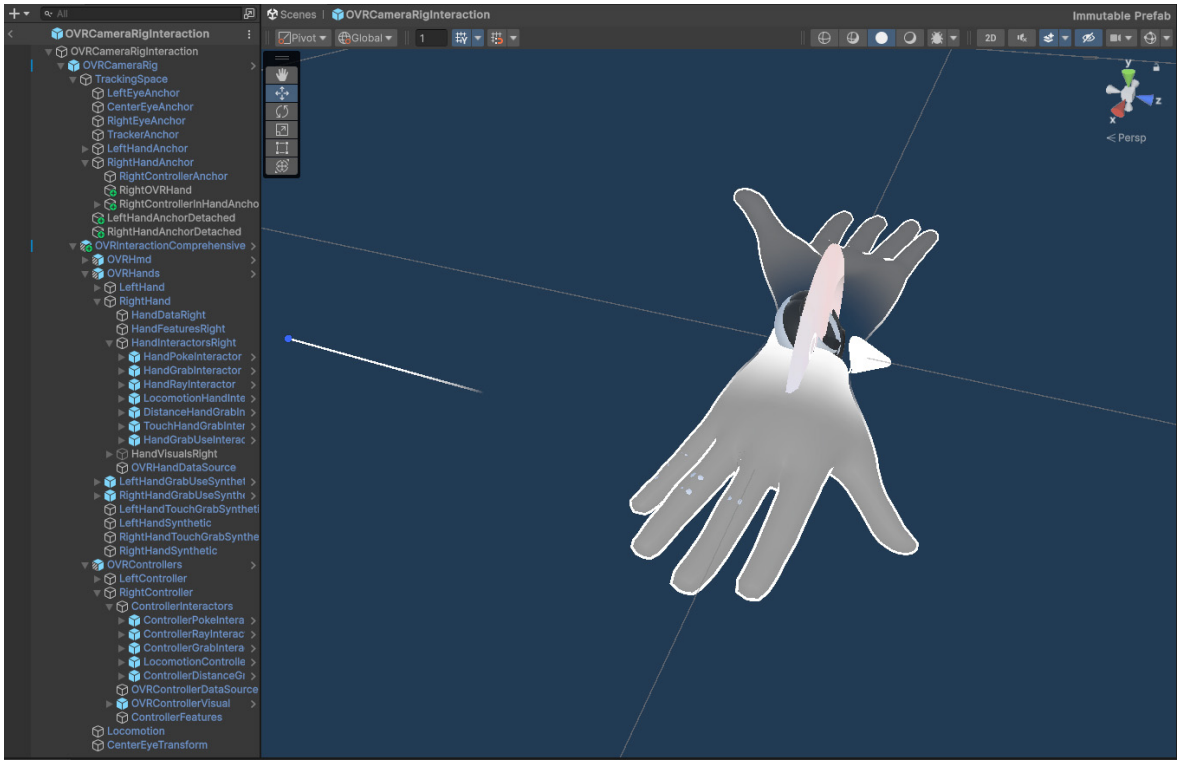

Figure 4.1: The *OVRCameraRigInteraction* prefab from the Meta XR All-in-One SDK

The hands appear as transparent, gray hands with the same joints a human hand has. The hand tracking for the Meta XR All-in-One SDK is fairly accurate, being able to track only the parts of the hands that are directly seen from the headset's point of view. Of course, if a part of the hand is obscured by real-life objects or the other hand, the model in the virtual world will not appear correctly, since the camera cannot track it correctly without being able to see it. Controllers appear the same as the controllers used by the player, with different types of controllers available and automatically changing to the correct model depending on what VR device the player is using. When the player pushes a button or moves the thumbstick of a controller, the same input appears to happen on the virtual controller, as it mimics the real-life controller.

The objects under the parent object *TrackingSpace* correspond to their real-life counterparts, tracking their movement. For example, the *CenterEyeAnchor* object corresponds to the head of the player, while the *RightHandAnchor* object tracks the movement of the player's right hand, whether it is holding a controller or not. Additionally, because the

prefab uses a camera on the *CenterEyeAnchor* object that tracks the player's head position and rotation, the *Main Camera* that is present in the scene by default should be deleted.

VR UI Interaction

The *OVRCameraRigInteraction* prefab also contains interactors that are required for the player to interact with the virtual world. Both the hands and the controllers contain poke, grab, ray, locomotion, and distance grab interactors, while the hands also contain the touch hand grab and hand grab use interactors. However, only the poke, ray, and locomotion interactors are used in the exploration of **OVRseer**, so the way those interactors work will be explained. The poke and ray interactor both allow players to interact with any User Interface (UI) present in the virtual world. The difference is that the poke interactor requires players to directly poke the UI with either their fingers or controllers, while the ray interactor requires players to only point at the UI from a distance with their hands or controllers. To select a UI element from afar, one option is for players to pinch their thumb and index finger while pointing at the UI if they are using their hands. On the other hand, if they are using controllers, they can press the trigger button, which their index finger is laying over. A poke interaction example with UI using hands is seen in Figure 4.2, while ray interaction using controllers can be viewed in Figure 4.3.

Figure 4.2: Poke interaction with the UI using a hand

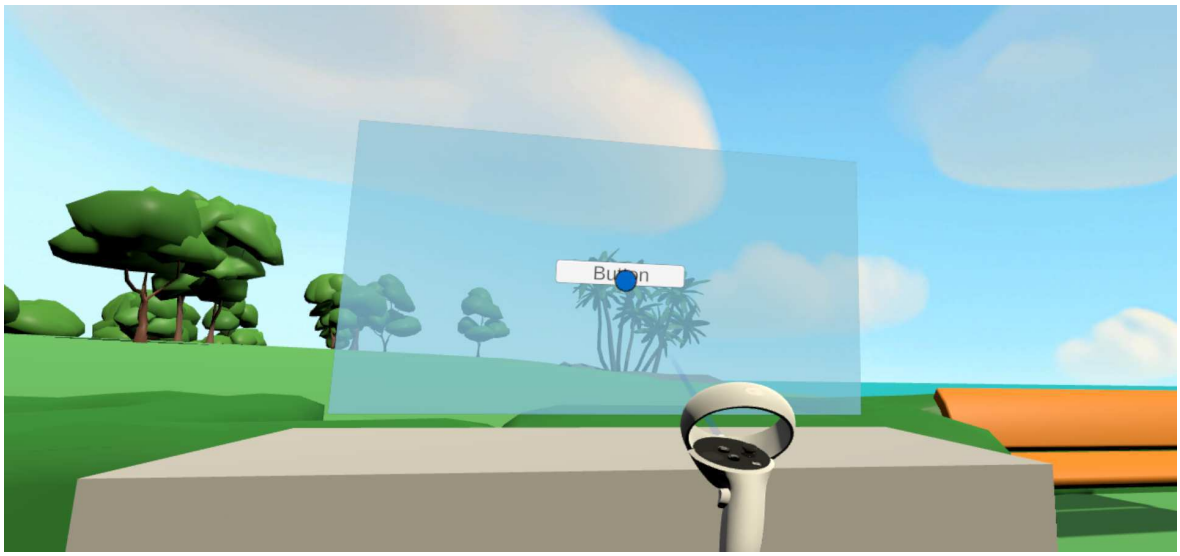

Figure 4.3: Ray interaction with the UI using a controller

For the player to be able to interact with the UI, it has to be modified to work with VR. To do this, the *Canvas Render Mode* property must be set to *World Space*, and the *Pointable Canvas*, *Ray Interactable*, and *Poke Interactable* components from the Meta XR All-in-One SDK package must be added to the canvas object. The *Canvas* property of the *Pointable Canvas* has to reference the *Canvas* component of the same game object. Also, the *Pointable Element* property of both the *Ray Interactable* and *Poke Interactable* components has to be set to the *Pointable Canvas* component. Next, a surface has to be made that allows players to interact with the UI elements contained in it. This surface must be referenced as the *Ray Interactable*'s *Surface* component and the *Poke Interactable*'s *Surface Patch* component. A new, empty game object has to be made, with the components *Plane Surface*, *Clipped Plane Surface* and *Bounds Clipper* attached to it. The *Plane Surface* property of the *Clipped Plane Surface* must reference the *Plane Surface* component on the object, while the *Bounds Clipper* has to be added to the *Clippers* list property. Finally, the *Bounds Clipper* has to be edited using the *Show Surface Visuals* button, and set to the size of the UI used. With these steps done, players can interact with the UI using their hands or controllers.

VR Locomotion

The locomotion interactor has two uses: movement and turning. For moving around, players have to clench a fist with their palm facing up or down and stretch out their thumb and index finger, point to where they want to teleport, and then pinch those two

fingers together. If they are using controllers, players have to move the thumbstick forward, also point to where they want to move, then let go of the thumbstick. In both cases, the player will be teleported to the wanted location. An example of teleportation with the controllers is visible in Figure 4.4.

Figure 4.4: Teleport interaction using a controller

If the VR headsets are connected to a computer via a cable, the turning capabilities of the players are restricted, so they are also able to turn using their hands or controllers. To turn with their hands, players have to make a similar shape with their hands to the one required for teleporting, but their palm has to be turned inward and their thumb and index finger have to be slightly bent. When players do this, arrows pointing to the left or right appear above their hand. By holding their hand in the same position, but moving it to the left or the right, then pinching their thumb and index finger together, players turn 45 degrees in the direction symbolized by the picked arrow. With controllers, players turn by simply pushing either thumbstick to the left or the right, turning 45 degrees in the desired direction. An example of turning with a hand can be seen in Figure 4.5.

Additionally, to allow players to be able to teleport on walkable surfaces and restrict them from teleporting on other objects, all asset prefabs had to be modified. The components *Teleport Interactable*, *Collider Surface*, and *Reticle Data Teleport* were added to all used assets. For the *Collider* property of the *Collider Surface* component, the collider of the object in question was referenced. Then, the *Collider Surface* component was used for the *Surface* property of the *Teleport Interactable* component. For assets that could be

walked on, the *Allow Teleport* property of the *Teleport Interactable* component was set to *true* and the *Reticle Mode* property of the *Reticle Data Teleport* component was set to *Valid Target*. On the other hand, assets that could not be walked on had these properties set to *false* and *Invalid Target* respectively.

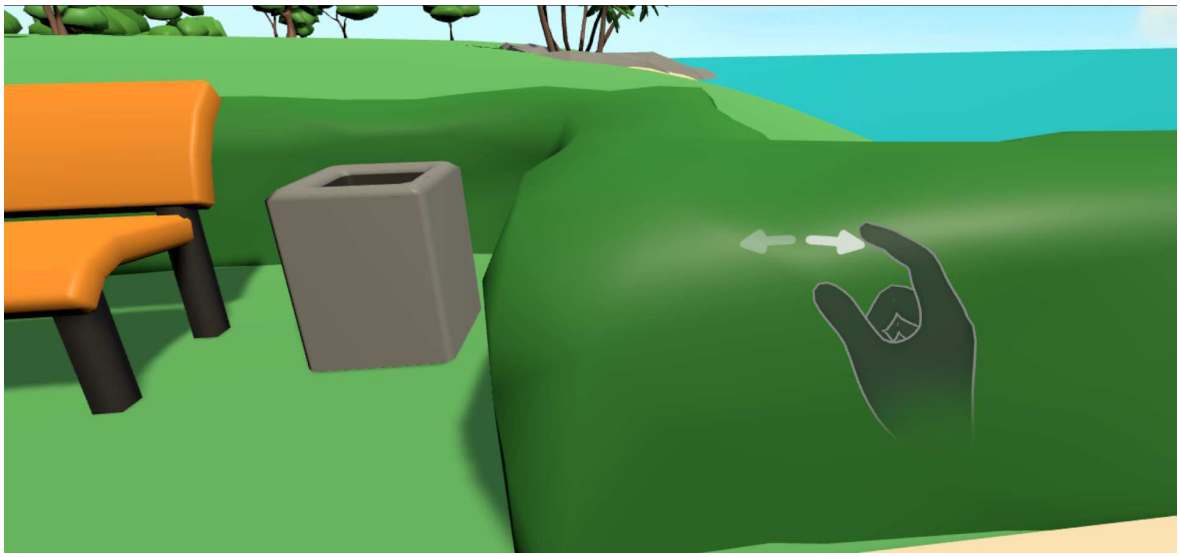

Figure 4.5: Turn interaction using a hand

4.2.2 Scene Design

As previously stated in section 4.1, two maps were used in **OVRseer** to make development easier, *Amusement park 1* and *Dreamscape Village - Stylized Fantasy Open World*, both of which are available in the Unity Asset Store. These maps were chosen because of their similar and large enough size, as well as their diverse assets and attention to detail. Each map makes up a scene, so there are two scenes used in the game. The first scene is called *Map_Amusement_Park*, featuring the amusement park map, while the other, which contains the medieval village map, is called *Map_Dreamscape_Village*.

The first map features a virtual amusement park, which can be seen in Figure 4.6. The most important parts of the map are the many attractions scattered across the map, a castle, a stage, a fountain, a ship, and a tunnel. Players started in the bottom right corner of the map, on the side of the bridge opposite the amusement park. On this map, players can teleport on the roads and the platforms of some attractions, but not on the grass.

Figure 4.6: The amusement park map

The second map represents a medieval village, seen in Figure 4.7. This map's most prominent features are the houses, a statue, a watermill, a dock, an inn, long stairs, a church, and a central big tree. The windmill on the map also looks important, but it is out of the player's reach, so it is not noted as one of the prominent features. It should be noted that due to the distance between the camera and the map when taking a photo of the whole map, many assets in the Figure have low level of detail, while some are not even visible. Up close and from a VR perspective, the medieval village has many details and the assets look much more detailed. On this map, players can teleport on grass and most wooden surfaces, but there are invisible walls players can not teleport through around the village that prevent them from going too far outside the village.

Figure 4.7: The medieval village map

4.2.3 Gaze Tracking

Cognitive3D is used to track a player's gaze while they are playing the game. Implementation of gaze tracking in **OVRseer** is covered in this subsection. Additionally, features of Cognitive3D, as well as all data available when using the software, is also explored.

Cognitive3D Setup in Unity

After importing the Cognitive3D package into the project and making an account on their website[3] users are granted a *Developer Key*. This key is used to connect the developed game to a Cognitive3D *Dashboard*. To set up Cognitive3D in the Unity project, the *Project Setup* has to be opened through the Cognitive3D tab at the top of the Unity project. The window that pops up can be seen in Figure 4.8 and the *Developer Key* from the Cognitive3D *Dashboard* should be copied into the *Developer Key* field. It should be noted that the change in this value is not tracked in some version control systems, such as Git, so developers have to enter the *Developer Key* if they pulled the project from a Git repository.

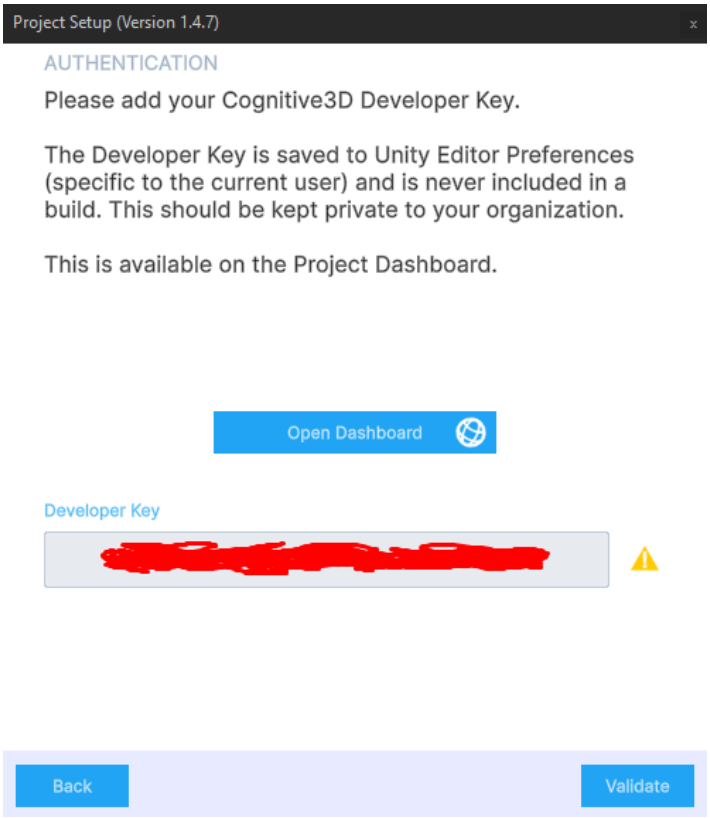

Figure 4.8: Cognitive3D's Project Setup window in the Unity Editor

[3]https://cognitive3d.com

After validating the *Developer Key*, VR SDK choices are presented. For this project, the Oculus Integration 53+ / Meta XR 64+ was chosen, since the Meta All-in-One SDK was used. After that, the Photon PUN 2 checkbox is ticked, since the game uses Photon PUN 2 for enabling multiplayer. Following that, the *Project Setup* window is closed. Afterward, both scenes, *Map_Amusement_Park* and *Map_Dreamscape_Village*, need to be set up through the *Scene Setup* window accessed through the Cognitive3D tab at the top of the Unity project while the scene is open. After going to the next page, the developer is presented with VR components that have to be tracked for gaze tracking to work properly. The *HMD* property should be linked with the *OVRCameraRigInteraction -> OVRCameraRig -> TrackingSpace -> CenterEyeAnchor* object. Similarly, the *Tracking Space* property should reference the *OVRCameraRigInteraction -> OVRCameraRig -> TrackingSpace* object. Finally, the *Left Controller* and *Right Controller* fields should be set up to follow the *OVRCameraRigInteraction -> OVRCameraRig -> TrackingSpace -> LeftHandAnchor* or *RightHandAnchor* object, depending on the controller side. After all properties are correctly referenced, the developer has to Setup the GameObjects.

Afterward, the *Quest Pro Eyetracking* and *Quest Hand Tracking* checkboxes must be ticked. Then, the current scene geometry needs to be exported. When that is done, the final window allows developers to upload the scene geometry, upload the scene thumbnail and upload dynamic meshes, which will be discussed shortly. It should be noted that uploading a new scene geometry makes a new version of the scene on the Cognitive3D web-application. Once everything is uploaded, the scene can be viewed on the Cognitive3D web-application, and gaze data from played sessions can be analyzed.

Additionally, to obtain even more data, objects of interest are made dynamic objects by adding the *Dynamic Object* component. This component gives each object a unique id, which enables them to be tracked on the Cognitive3D web-application. With that set up, gaze data through multiple sessions specifically for each of those objects can be viewed. To upload dynamic objects, developers have to navigate to the *Dynamic Objects* window through the Cognitive3D tab at the top of the Unity project. This window contains the names of all dynamic objects in the scene. One can then either select the Upload All Meshes button or select wanted dynamic objects in the object hierarchy and upload the selected meshes. Once they are uploaded, they can be seen on the Cognitive3D web-

application for that particular scene. It should be noted that dynamic objects can be uploaded and removed from a scene at any time, and do not change the version of the scene in Cognitive3D.

Cognitive3D Dashboards

For the *Dashboards* tab in the Cognitive3D web-application, sessions and their data that are analyzed can be filtered by tags, which will be explained later, by scenes, and by the time they are recorded. The *Project Overview* tab, part of which is seen in Figure 4.9, contains scores from 0 to 100 that measure the participants' comfort and immersion in the application, as well as an aggregate performance of the application. In addition, it counts the total number of unique sessions recorded, the average session duration, and the total session duration in a selected time period. The number of sessions can also be viewed split by the version of an uploaded scene, while the total and average session duration can be seen across all selected scenes.

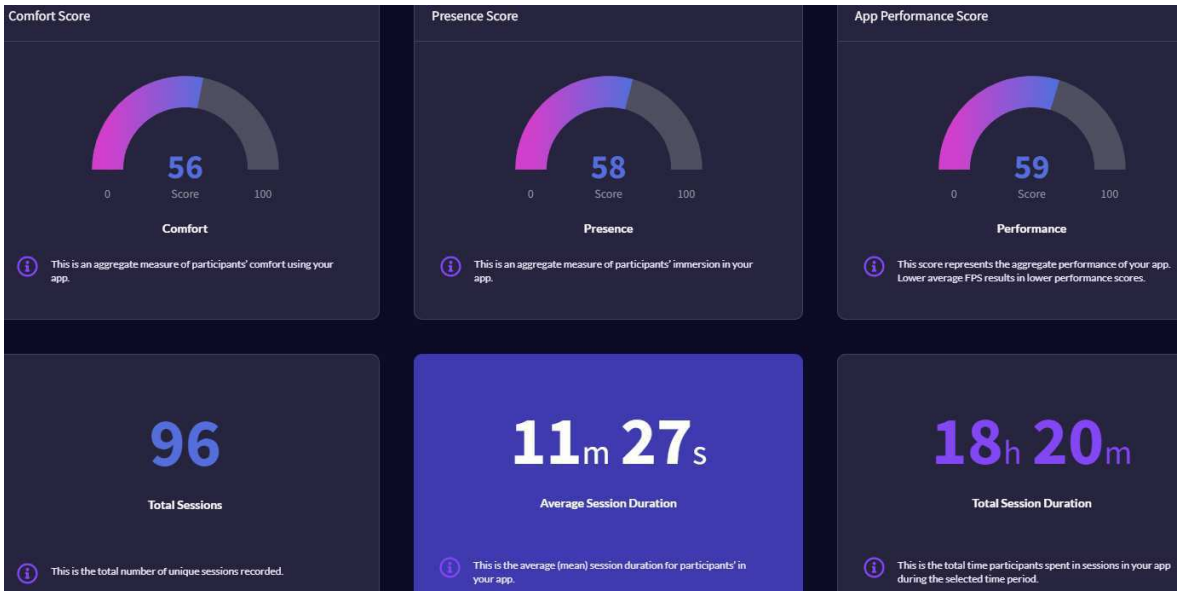

Figure 4.9: Part of Cognitive3D's *Project Overview* window on the web-application

In the *App Performance* tab from the *Dashboards* tab, part of which can be viewed in Figure 4.10, the aggregate performance of the application is visible once again, but the percentage of session time above 60 and 72 Hz can also be seen. The average frame rate for the application can also be analyzed in total and by a version of the scene. Battery efficiency data is also available, but no relevant data for it has been shown in any of the sessions recorded for this thesis. Finally, this window generates a top-down view of a

particular scene and shows on which parts of the map the refresh rate most frequently dropped below a selected refresh rate. In Figure 4.10, an example with the medieval village map and 72 Hz is shown. The more red an area of the map is, the more frequently the refresh rate dropped below the 72 Hz mark.

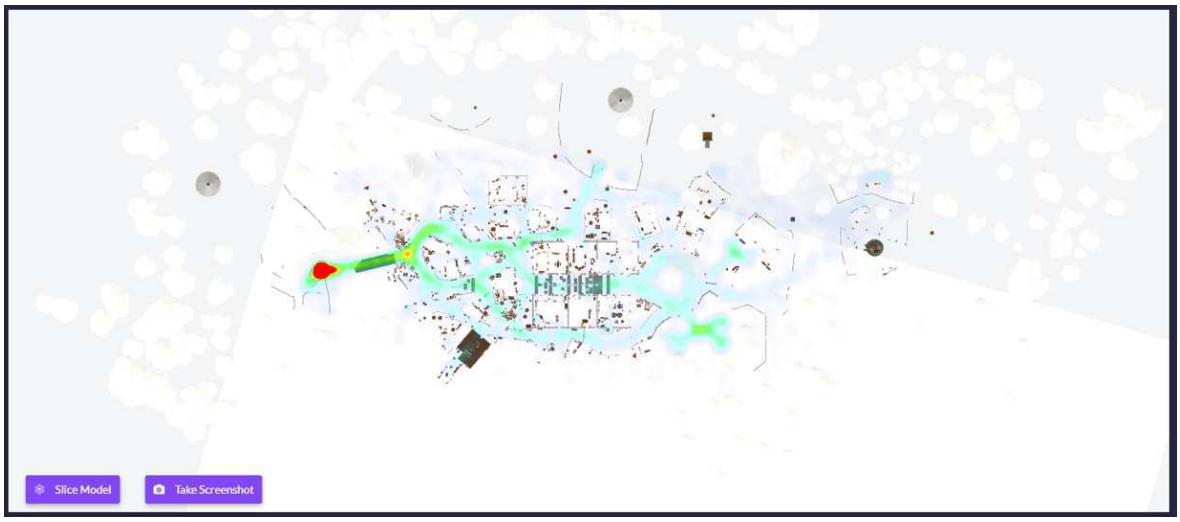

Figure 4.10: Part of Cognitive3D's *App Performance* window on the web-application

Other available *Dashboards* tab data, which is not relevant for the user study of this thesis, includes analyzing monthly, weekly, and daily active devices, new devices, device retention, and average session times for the application in the *Live Operations* tab. The *Demographics* tab shows the average playable space, the highest traffic countries, and sessions by geography. In the *Spatial Optimization* tab, developers can find detailed insights into the comfort and presence of the application. It also includes data on whether the users playing were sitting or standing, the orientation of their headset while they were playing, controller ergonomics, and a top-down map view of where users were exiting the experience in a selected scene.

Cognitive3D Scenes

On the *Scenes* tab, one can pick a scene they have uploaded from the Unity Editor. All data here will be displayed on the amusement park scene. For each scene, sessions and their data can be filtered by tags, the time they were recorded, and the version of the uploaded scene. The *Scene Summary* window also displays the dates the selected scene was created and updated on, as well as the latest version of the scene. It also shows how much time has passed since the last session was recorded, the total session duration, the

average scene session duration, and the total number of scene sessions, as seen in Figure 4.11. Additionally, all recorded sessions can be seen in this window, as can be viewed in Figure 4.12. Each session is given a name consisting of an adjective, a color, and an animal name, as well as the city it was recorded in, to more easily remember any session if needed. The session length and the date of the session can also be viewed.

Figure 4.11: Cognitive3D's *Scene Summary* window in the *Scene Viewer* on the web-application

Figure 4.12: Examples of recorded sessions in Cognitive3D

Tags can be edited for sessions, and for this thesis, the tags *pro*, *quest2*, *collaborativ* (shortened from the word collaborative because of character limit), *competitive*, *procoll*, *procom*, and *study* were added. The *pro* and *quest2* tags were added for sessions played on the Meta Quest Pro and Meta Quest 2, respectively. The *collaborativ* and *competitive* tags were added to sessions that were played in collaborative or competitive game modes, also respectively. Additionally, the *study* tag was added to all sessions that were recorded as part of the user study, to differentiate them from sessions that were recorded while testing the game. Finally, the *procoll* tag was added for all sessions that were played collaboratively on the Quest Meta Pro, while the *procom* tag was added to those that were played competitively on the device. Finally, the scene session report and individual session data can be downloaded and a session can be replayed on the *Scene Explorer*, which will be explained later on.

Cognitive3D Scene Viewer

In the *Scene Viewer*, the selected scene can be viewed in both 3D and top-down view. For the selected scene, queries can be run for gaze, eye fixations, user positions, and events across the selected sessions. The results of these queries appear as transparent colored squares in the places the selected data appears. In the top left of the *Scene Viewer*, the color of the squares is explained, with a higher number meaning more of the selected data was found at that position. Examples of these queries can be seen in Figure 4.13, where a query for gaze is run, and in Figure 4.14, in which a user position query results are shown.

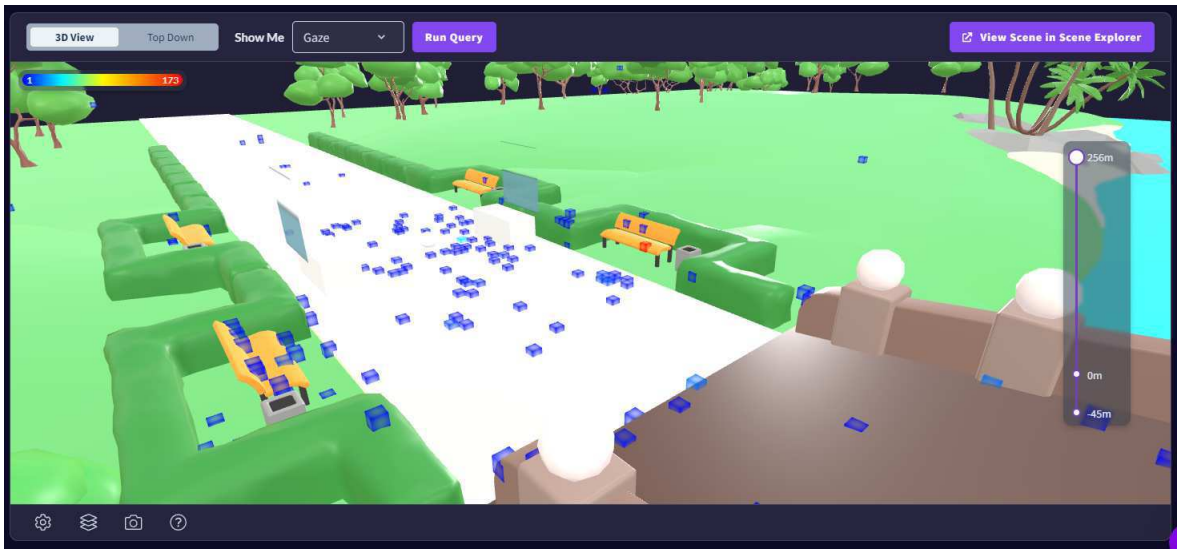

Figure 4.13: Gaze query on Cognitive3D's *Scene Viewer*

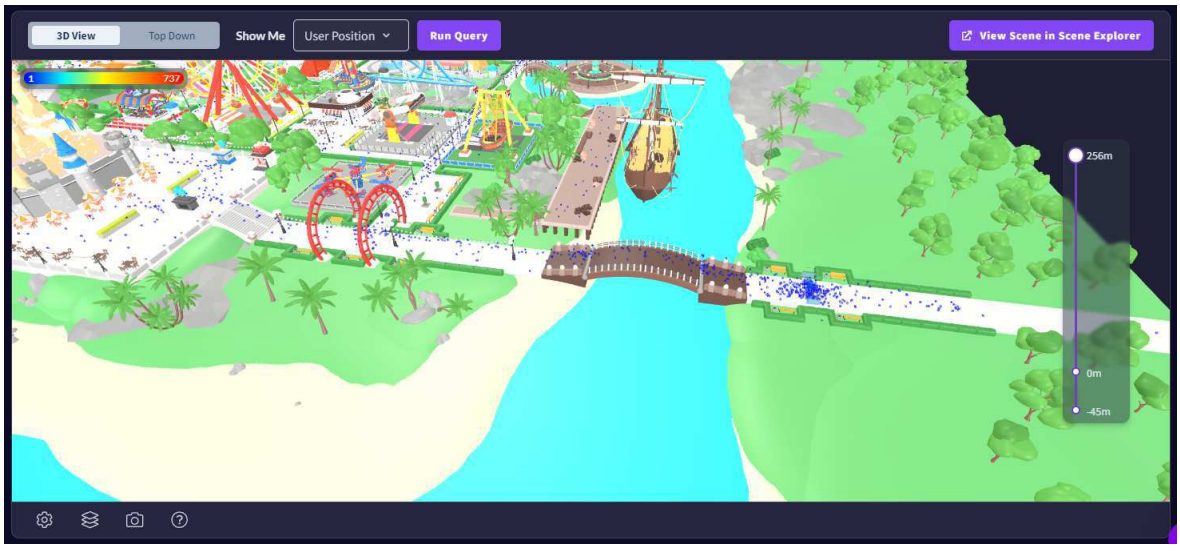

Figure 4.14: User position query on Cognitive3D's *Scene Viewer*

Cognitive3D Object Explorer

In the *Object Explorer*, a list of all dynamic objects can be found. By clicking on one of them, a 3D view of the object appears. Gaze and eye fixation queries can be run on these objects, with visualization available both in the form of a heatmap and in the form of cube aggregation. An example of a gaze heatmap on a dynamic object can be viewed in Figure 4.15, which features the ship from the amusement park map. Additionally, gaze or eye fixation metrics can be analyzed for each dynamic object. Firstly, the rank of the dynamic object as opposed to other dynamic objects in terms of gaze or eye fixations is stated. The metrics also show the average gaze or eye fixation data per session, and the number of total sessions the dynamic object appears in. Then, more detailed metrics can be analyzed for each dynamic object. These metrics include the average gaze instance duration, count, and time, as well as the total gaze count and time. It also states the number of sessions that include any gaze on that object and the ratio of sessions that contain gaze on that object. More available data includes the average time to first gaze at the object, and the average gaze sequence for that object.

Figure 4.15: Dynamic object gaze heatmap in the *Object Explorer*

Cognitive3D Scene Explorer

Finally, in the *Scene Explorer*, one can explore the scene in 3D, similarly to the *Scene Viewer*. However, recorded sessions can also be replayed in the *Scene Viewer*, and an example of this can be seen in Figure 4.16, where the session *Direct Sepia Pheasant from Zagreb* is replayed. The session replay shows the player moving on the map, and the recording time can be manipulated at the bottom of the Figure. On the left of the Figure,

the player is seen. The robotic head moves along with the Cognitive3D player head, which is inside the robotic head on the Figure, because it was given the *Dynamic Object* component. The robotic head has some lag when moving, so the Cognitive3D player head is sometimes seen, with the robotic head following behind, especially if the player is teleporting very fast. In the top left of the Figure, a view of the player's controllers can be seen, and all their button presses during the session recording can be viewed. The controllers also follow the movements of the player during the recording.

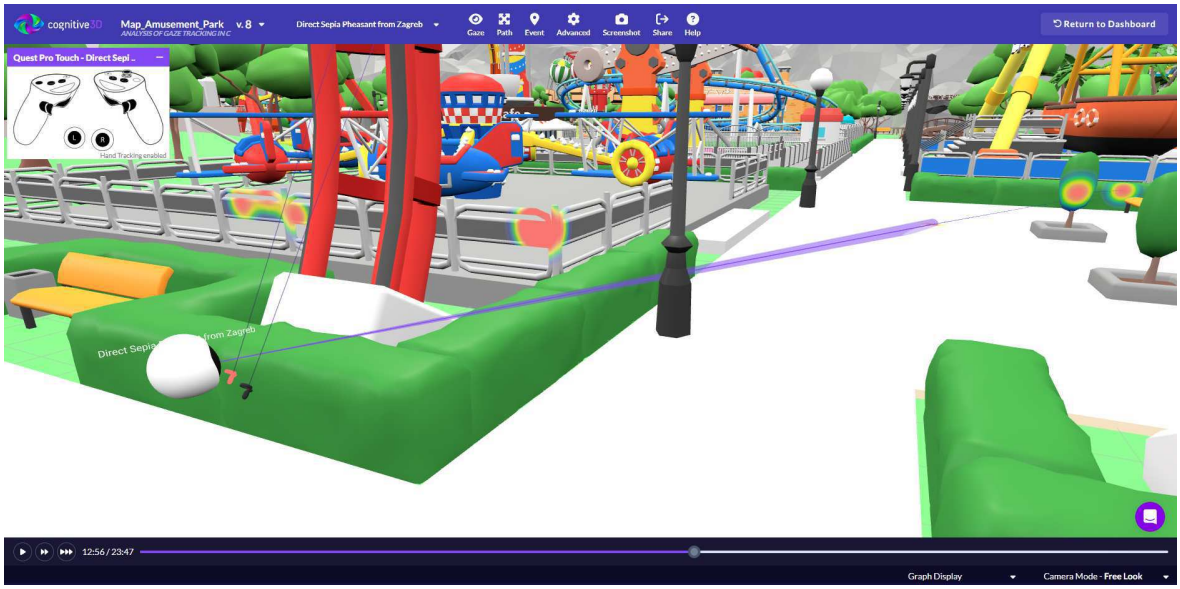

Figure 4.16: One session viewed in the *Scene Explorer*

The gaze vector coming from the player's head can be turned off or changed to a line instead of a cone through the *Gaze* tab at the top of the *Scene Explorer*. Additionally, a gaze heatmap can be seen in the Figure, but it is currently set to an animated display, which means the data appears when the player looks at it in the recording, but disappears shortly after. In the *Gaze* tab, the heatmap size and intensity can be changed. The gaze heatmap can also be set to an aggregated view, also through the *Gaze* tab, and an example of the results for a session can be seen in Figure 4.17. Parts of the map that the players' gaze fixated on can also be viewed by choosing *Eye Fixation* instead of *Gaze* visualization. An example of this can be seen in Figure 4.18, where the eye fixation visualization are seen as purple circles, with their size being larger if players fixated on them more. Eye fixations can also be connected to dynamic objects. The path the player took can be viewed with the *Path* tab, being able to switch between tracking the floor and tracking the head. A players' recorded path following their head can be seen in Figure 4.19. Dynamic

objects can be clicked in the 3D map and their average gaze and eye fixation data and data for the selected session can be viewed in the *Scene Explorer*. An example of dynamic object data in the *Scene Explorer* can be viewed in Figure 4.20. In addition, multiple sessions can be viewed at once in the *Scene Explorer* by selecting them at the top of the window, and their aggregated data can then be viewed.

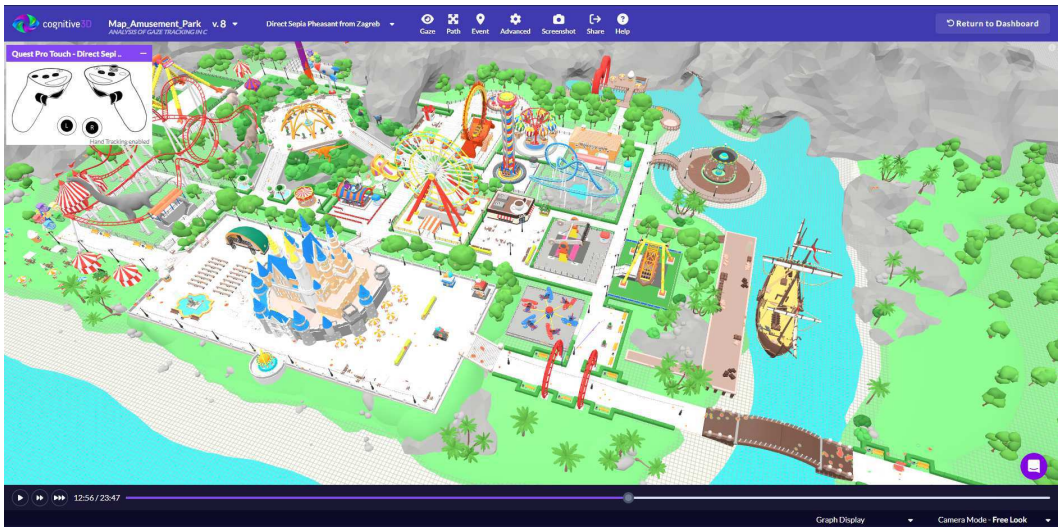

Figure 4.17: Aggregated gaze heatmap for one session in the *Scene Explorer*

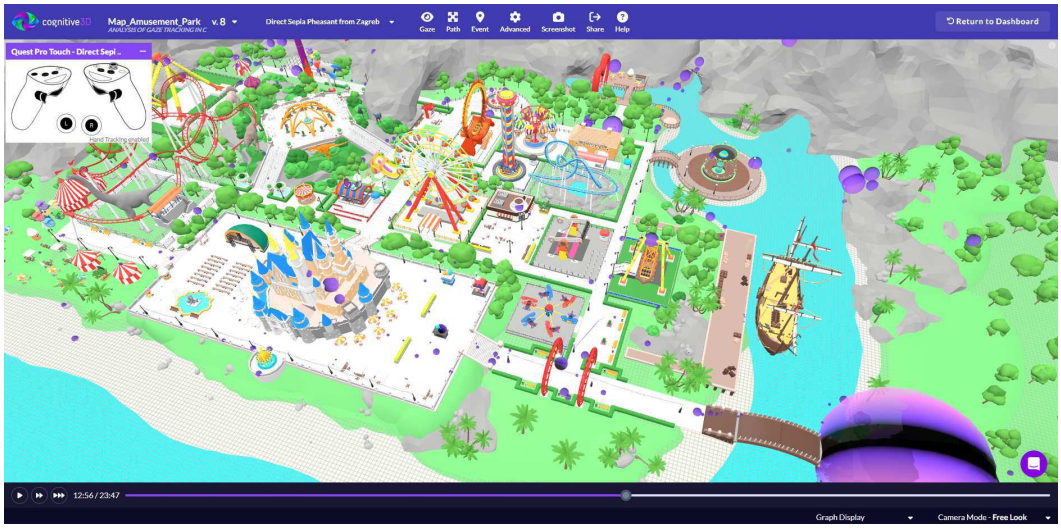

Figure 4.18: Eye fixations for one session in the *Scene Explorer*

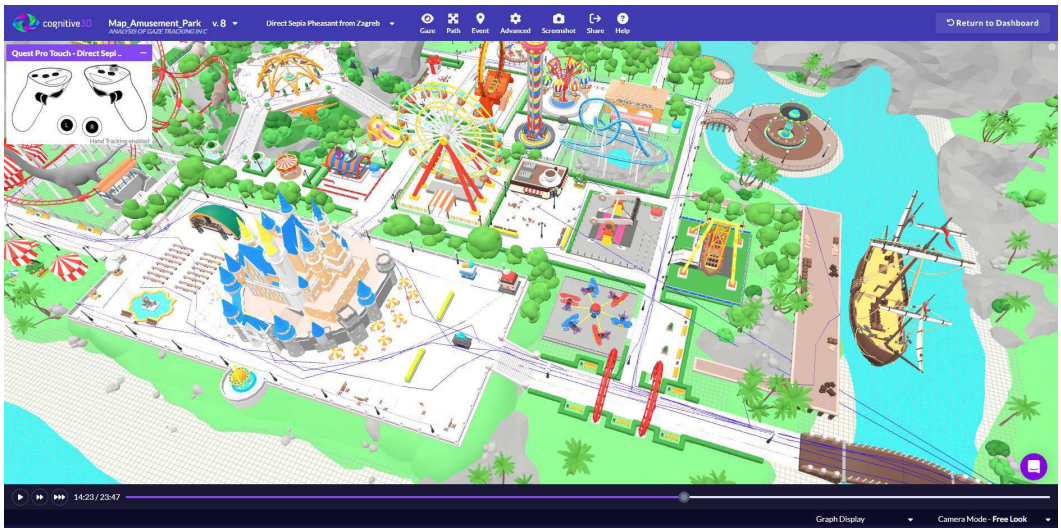

Figure 4.19: Path for one session in the *Scene Explorer*

Figure 4.20: Dynamic object data for one session in the *Scene Explorer*

4.2.4 Multiplayer Implementation

To make the developed game collaborative and competitive, multiple players need to be able to play the game together, at the same time. To make this possible, Photon PUN 2 was used, enabling the game to be multiplayer.

Photon Server Setup

Firstly, a new app was made on the Photon web-application[4]. The application type was set to *Multiplayer Game*, while the Photon SDK was set to *Pun*. The application name was simply set to *Gaze Tracking*. An example of a correctly set up Photon app can be

[4]https://dashboard.photonengine.com

seen in Figure 4.21. The *App ID* is a unique key, so it was removed from the Figure, and it is needed to connect the Unity project to the Photon app.

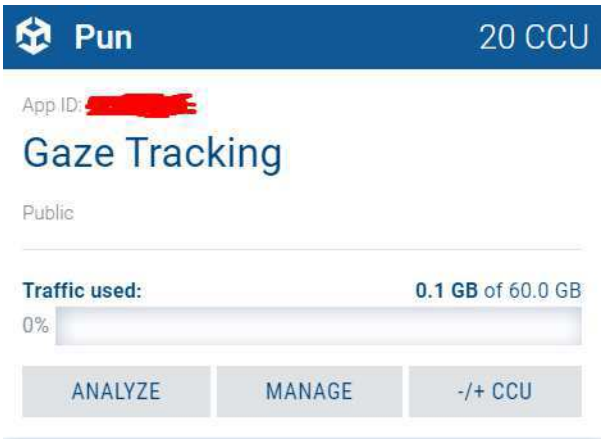

Figure 4.21: Example of a Photon application

Next, to connect the Photon application to the Unity project, the *PhotonServerSettings.asset* had to be modified. The asset can be found at *Assets -> Photon -> PhotonUnityNetworking -> Resources* after importing Photon PUN 2 into the Unity project. All properties of the asset can be viewed in Figure 4.22. Firstly, the *App Id PUN* property must be set to the same *App ID* that is given for the created application on the Photon *Dashboard*. Doing this connects the Unity project to that application and the server given for that application. Additionally, the *Fixed Region* property was set to the best region found while testing. In this case, it was the *eu* region, but this field varies depending on the location of the developer. Sometimes, this field does not need to be set, as Unity always picks the region with the lowest ping. However, when developing the application for this thesis, the location seemed to be right in between two regions. This caused a problem when testing the game, as one instance would be in one region, while the other would be in another, so they were unable to join the same server.

Player Presence

For players to be able to see each other on the same server, a player's presence had to be developed. The prefab for a networked player can be seen in Figure 4.23 and needs to be placed in a folder named *Resources* in the *Assets* folder. In this project, the *Player* prefab was placed in the *Resources -> NetworkedPrefabs* folder. It contains a robotic head representing the player's head in an easily viewable and gender-neutral way. The robotic head was made bigger in the virtual environment than the size of a real-life head, so players

could identify each other easier and from a greater distance. A *Photon View* component was added to the *Player* object, which synchronizes the object across the network, allowing other players to see a specific player's movement. Because the player's head can move and rotate, the *Head* object was given a *Photon Transform View* component, with position and rotation synchronization checked. The scale of the robotic head never changed, so scale synchronization was not checked in the *Photon Transform View* component.

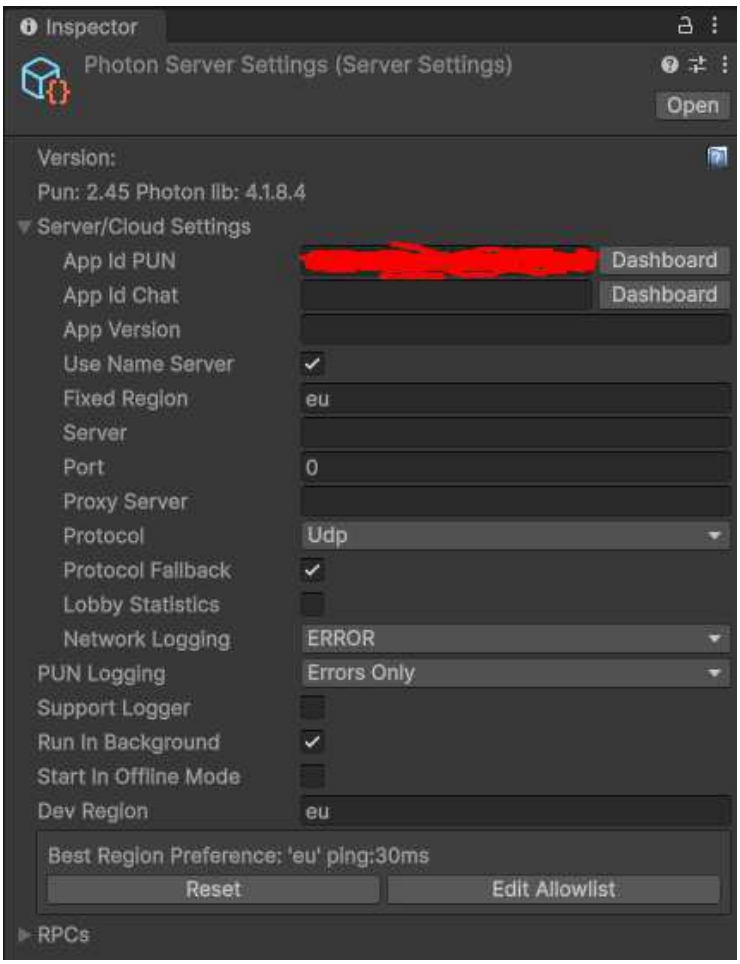

Figure 4.22: Properties of the *PhotonServerSettings* asset

To make the robotic head follow the movements of the player's actual head, a *VRTracker* script was created and added onto the *Player* object. The code responsible for setting the player's robotic head and finding their real-life head position and rotation can be seen in Figure 4.24. The *head* variable is set to the *Head* object of the *Player* prefab inside the Unity Editor. The *photonView* variable is set to the *Photon View* component on the current object inside the *Start* function. However, the *headRig* variable has to be found in the scene, since it is not a part of the *Player* prefab, but of the *OVRCameraRig* object. The function *FindAnyObjectByType* finds objects on the scene by a given type, in this

case *OVRCameraRig*. After the *OVRCameraRig* object is found, the *Find* function goes through its children, specifically the *TrackingSpace* object of the *OVRCameraRig*, until it arrives at the *CenterEyeAnchor* object, which tracks the player's head movements.

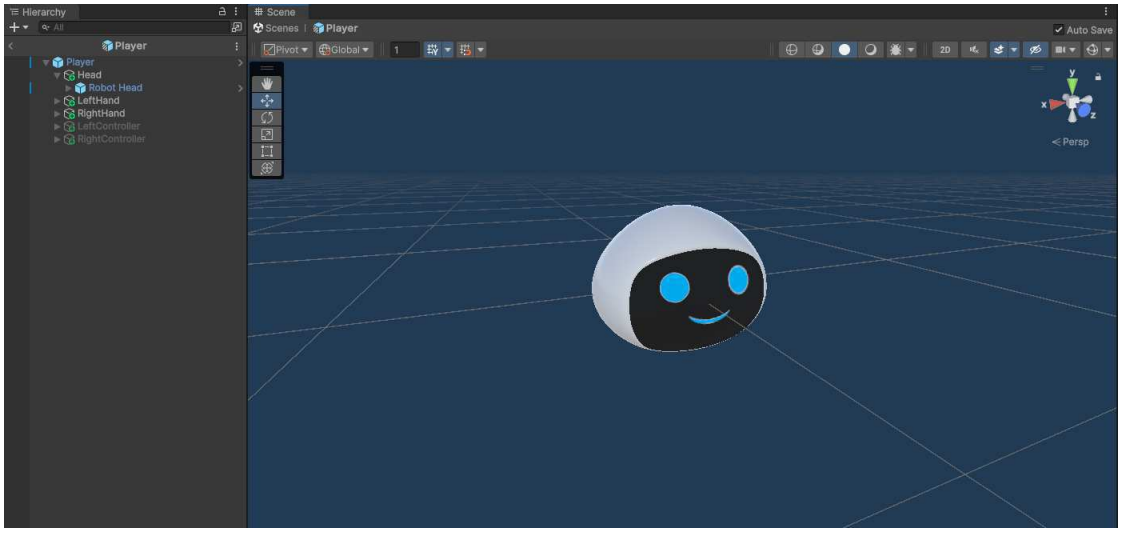

Figure 4.23: The *Player* prefab, representing a networked player's presence

```
public Transform head;
private PhotonView photonView;
private Transform headRig;
void Start()
{
    photonView = GetComponent<PhotonView>();
    OVRCameraRig rig = FindAnyObjectByType<OVRCameraRig>();
    headRig = rig.transform.Find("TrackingSpace/CenterEyeAnchor");
    if (photonView.IsMine)
    {
        foreach (var item in GetComponentsInChildren<Renderer>())
            item.enabled = false;
        foreach (var item in GetComponentsInChildren<Canvas>())
            item.enabled = false;
    }
}
```

Figure 4.24: The variables and the *Start* function in the *VRTracker* script

It should be noted that the player would be able to see their own head this way, which would interfere with their vision of the virtual environment. To avoid this, if the owner of the *Photon View* component on the *Player* object is the player for whom the game is currently running this script, their head is made transparent. This is *achieved* by disabling the *Renderer* and *Canvas* components on all children of the *Head* object. The Renderer component is responsible for rendering the robot head, while the *Canvas* component allows the facial expression of the robot to be visible.

To move the robotic head according to the player's movement, the code in Figure 4.25 is utilized. The head is only moved if the player is the actual owner of the player presence, or, in other words, if this player prefab is tracking their movement, and not of the other player. The head movement is mapped through the *MapPosition* function, which copies the position and rotation of the *CenterEyeAnchor*, tracking the player's head, onto the *Head* object of the *Player* prefab. With this code, the player's head movement is synchronized across the network, and the other player sees their head moving in real-time in their own game.

```
1  void Update()
2  {
3      if (photonView.IsMine)
4          MapPosition(head, headRig);
5  }
6  void MapPosition(Transform target, Transform rigTransform)
7  {
8      target.position = rigTransform.position;
9      target.rotation = rigTransform.rotation;
10 }
```

Figure 4.25: The *Update* and *MapPosition* functions in the *VRTracker* script

Photon Room Joining

To join the same room, players first need to connect to the same server, and then to a lobby. Connecting to the server is done through the *NetworkManager* script in an empty game object. This script inherits the *MonoBehaviourPunCallbacks* script from Photon PUN 2, allowing it to override callback functions that are called when the player connects to, for example, the server. The *NetworkManager* script contains many game objects that are parents of UI elements. These UI elements are activated and deactivated as the player goes through different connection phases. These UI elements change the text the player sees, as can be viewed in Figure 4.26.

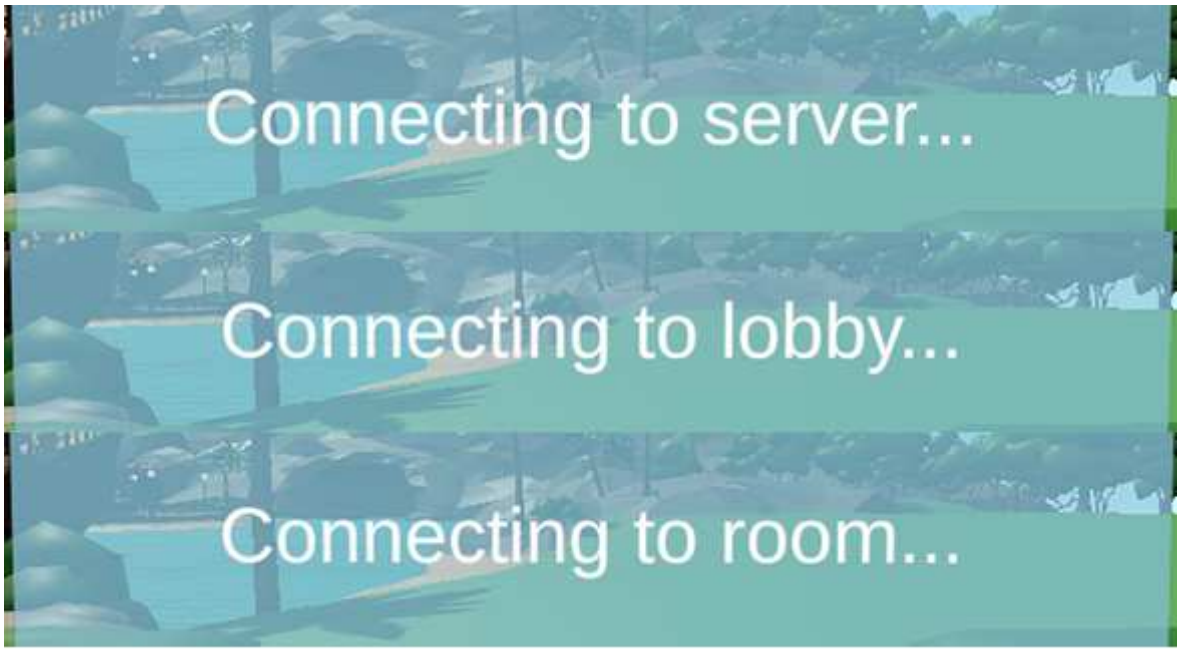

Figure 4.26: The different UI elements a player sees while going through all phases of connecting with another player

Firstly, in the *Start* function, the player connects to the server through the *PhotonNetwork.ConnectUsingSettings* function. The *OnConnectedToMaster* function is called when a player successfully joins a server. This function is overridden to add two lines. The first is setting the *AutomaticallySyncScene* property of *PhotonNetwork* to *true*. This synchronizes all player's games to match the active scene to the one active for the Photon room owner. In other words, when the room owner loads a scene, that same scene will load for all other players as well, if they set the property to *true*. The other line, *PhotonNetwork.JoinLobby*, makes the player join a Photon lobby. This code can be seen in Figure 4.27.

```
void Start()
{
    foreach (var connectingToServerText in connectingToServerTexts)
        connectingToServerText.SetActive(true);
    PhotonNetwork.ConnectUsingSettings();
}
public override void OnConnectedToMaster()
{
    foreach (var connectingToServerText in connectingToServerTexts)
        connectingToServerText.SetActive(false);
    PhotonNetwork.AutomaticallySyncScene = true;
    foreach (var connectingToLobbyText in connectingToLobbyTexts)
        connectingToLobbyText.SetActive(true);
    PhotonNetwork.JoinLobby();
}
```

Figure 4.27: The *Start* and *OnConnectedToMaster* functions in the *NetworkManager* script

Afterward, in the overridden *OnJoinedLobby* function that gets called once the player successfully joins a lobby, the player joins or creates a room with *PhotonNetwork*'s *JoinOrCreateRoom* function. Usually, this function gets called when players create lobbies, allowing others to join them. However, because **OVRseer** is developed exclusively for analyzing data and will only be played in a laboratory setting in a pair, only one room was made to ever exist in the game and players join it automatically. To avoid complications, the room is simply called *Room*, and the maximum amount of players that can join it is 2, as can be viewed in Figure 4.28.

```
public override void OnJoinedLobby()
{
    foreach (var connectingToLobbyText in connectingToLobbyTexts)
        connectingToLobbyText.SetActive(false);
    foreach (var connectingToRoomText in connectingToRoomTexts)
        connectingToRoomText.SetActive(true);
```

```
    PhotonNetwork.JoinOrCreateRoom("Room", new RoomOptions() { MaxPlayers = 2
        }, null);
}
```

Figure 4.28: The *OnJoinedLobby* function in the *NetworkManager* script

Player Instantiation

Once a player joins the room, the overridden *OnJoinedRoom* function gets called, seen in Figure 4.29. In it, the *Player* prefab for the player running the scripts gets instantiated across all instances of players who are in the same room. This is done with the *PhotonNetwork.Instantiate* function. Additionally, if there is only one player in the room, meaning this is the first player to join the room, a text appears saying they have to wait for the other player to join. However, if there are two players in the room, the game menu is shown. This also happens if the player is the first one to join and another player joins the room, calling the overridden *OnPlayerEnteredRoom* function.

```
public override void OnJoinedRoom()
{
    foreach (var connectingToRoomText in connectingToRoomTexts)
        connectingToRoomText.SetActive(false);
    player = PhotonNetwork.Instantiate("NetworkedPrefabs/Player", new
        Vector3(0, 0, 0), Quaternion.identity, 0);
    if (PhotonNetwork.CurrentRoom.PlayerCount == 1)
        foreach (var waitingForPlayerText in waitingForPlayerTexts)
            waitingForPlayerText.SetActive(true);
    else
        ShowGameMenu();
}
public override void OnPlayerEnteredRoom(Player newPlayer)
{
    foreach (var waitingForPlayerText in waitingForPlayerTexts)
    {
        waitingForPlayerText.SetActive(false);
```

```
    }
    ShowGameMenu();
}
```

Figure 4.29: The *OnJoinedRoom* and *OnPlayerEnteredRoom* functions in the *NetworkManager* script

Multiplayer Menus

The *ShowGameMenu* function in Figure 4.29 displays all elements of the game menu visible to the player before starting a game. There are two menus in one game, one for each player. The owner of the room has a dropdown menu which allows them to pick whether they will be playing the collaborative or competitive version of the game, and to start the game using a button. The other player only sees text saying the owner is starting the game. These two player menus can be seen in Figure 4.30. For the other player to not be able to change the game mode and start the game, the *ShowGameMenu* function disables the game mode dropdown menu. The function also sets the start button interactive field to *false* if the player is not the owner of the room, as can be seen in Figure 4.31. This removes the dropdown menu and grays out the start button for that player.

Figure 4.30: The game menus for the owner of the room (left) and the other player (right)

```
public void ShowGameMenu()
{
    ownerStartingText.SetActive(true);
    gamemodePicker.SetActive(true);
    startButton.SetActive(true);
    if (!PhotonNetwork.IsMasterClient)
    {
```

```
        gamemodePicker.SetActive(false);
        startButton.GetComponent<Button>().interactable = false;
    }
}
```

Figure 4.31: The *ShowGameMenu* function in the *NetworkManager* script

The *NetworkManager* script also has another function, *RemoveNetworkText*, seen in Figure 4.32, that removes all game menu text. This function gets called later on, when the owner of the room starts the game, which will be discussed in the subsection 4.2.5. The *handMenuNetworkText* variable references text that is written as a placeholder on a hand menu until the game starts, pointing out that a timer and pictures will be shown to the player later on. This hand menu will be covered in detail in subsections 4.2.5 and 4.2.6.

```
public void RemoveNetworkText()
{
    ownerStartingText.SetActive(false);
    gamemodePicker.SetActive(false);
    startButton.SetActive(false);
    handMenuNetworkText.SetActive(false);
}
```

Figure 4.32: The *RemoveNetworkText* function in the *NetworkManager* script

RPC Functions

Additionally, using Photon PUN 2, some functions used in other scripts were marked with *[PunRPC]*. These functions are called through RPC calls, which calls them across all instances of the game. With this, synchronization of functions for all players can be achieved. When an RPC function is analyzed in subsections 4.2.5, 4.2.6, and 4.2.7, it will specifically be emphasized it is marked with *[PunRPC]*.

4.2.5 Map Exploration

To restrict players from exploring the map before the other player joins and before starting the game, an empty game object called *StartAreaColliders* was created. Its children contain the *BoxCollider* component, and their colliders were placed in such a way that they surround the area where players first spawn in. Additionally, the children were made unable to teleport through or no, which restricts players from leaving their spawn area. An empty game object named *GameManager* was created in the scene, and a *GameManager* was created and added as a component to that game object. Once both players started the game and joined the room through the network, the owner of the room can pick what game mode they will be playing on the game menu. The choices they can choose are collaborative and competitive on the game menu seen in Figure 4.30. The chosen game mode, and future game data, are written in a file that can be read after the game is over. The file is given the name *GazeTracking_*, followed by the current date and time, and it is saved in the Unity project's persistent data path (*Users/current user/AppData/LocalLow/DefaultCompany/Project name*).

Afterward, the owner of the room starts the game by pressing the *Start Game* button, which is also on the game menu. Clicking this button calls the function *StartGame* of the *GameManager* component, which can be viewed in Figure 4.33. This function first checks whether the player who called it is the owner of the room, and if they are, calls four RPC functions that will run for both players. The first is the *RemoveNetworkText* function in the *GameManager*, which simply calls the function of the same name in the *NetworkManager* script, which was already seen in Figure 4.32 and explained before. The second one is *DeactivateStartAreaColliders*, which deactivates the colliders restricting players from moving outside their spawn area.

```
1 public void StartGame()
2 {
3     if (PhotonNetwork.IsMasterClient)
4     {
5         _photonView.RPC("RemoveNetworkText", RpcTarget.All);
6         _photonView.RPC("DeactivateStartAreaColliders", RpcTarget.All);
7         _photonView.RPC("StartExplorationTimer", RpcTarget.All);
```

```
        _photonView.RPC("SetGamemode", RpcTarget.All, (Gamemode)
            gamemodePicker.value);
    }
}
```

Figure 4.33: The *StartGame* function in the *GameManager* script

The next function that is called is *StartExplorationTimer*, which utilizes the *Timer* component that was added to an empty game object named *Timer* by calling its function of the same name that can be seen in Figure 4.34. This function first goes through all game objects that contain text that should be shown during the time players are exploring the map. This includes text on the game menu, visible in Figure 4.35 and text on the players' hand menu, which will be covered shortly. Then, it sets necessary *bool* variables that enabled the timer to start counting down the exploration time, which is set to 600 seconds, which equates to 10 minutes, in the Unity editor. The last function, *SetGamemode*, sets the gamemode variable of the *GameManager* script to either collaborative or competitive, depending on which was chosen in the game menu.

```
public void StartExplorationTimer()
{
    foreach (var timerText in explorationTimerTexts)
        timerText.SetActive(true);
    started = true;
    finished = false;
    exploration = true;
}
```

Figure 4.34: The *StartExplorationTimer* function in the *Timer* script

In the *Update* function of the *Timer* script, time counts down from the set amount until it reaches zero, as can be seen in Figure 4.36. This function firstly checks whether the timer is started, and if it is, updates the UI which shows the timer countdown through the *ShowOnGUI* function, which can be seen in Figure 4.37. To update the UI, it converts the time left to a formatted string showing minutes and seconds left, then sets the text of

all timer countdown game objects to that time. Additionally, if the timer has started, it is checked whether it finished. If the timer did not finish, the time between two *Update* function calls is subtracted from the timer variable containing the time left for the timer. Once this variable becomes lower than zero, it means the set amount of time passed, and the timer is set to 0 seconds to avoid it appearing negative on the UI text.

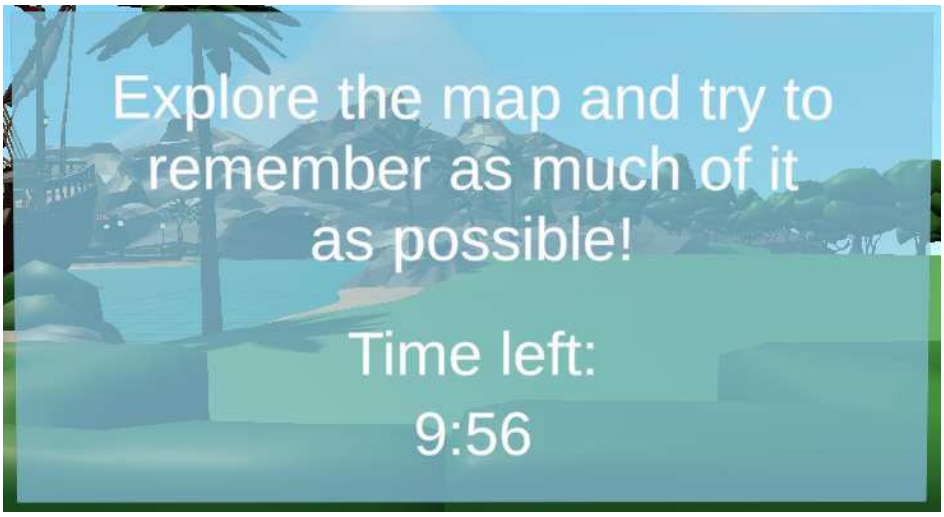

Figure 4.35: The text that appears on the game menus while players are exploring the map, along with the timer counting down the time they have left

```
void Update()
{
    if (started)
    {
        if (!finished)
        {
            timer -= Time.deltaTime;
            if (timer <= 0.0f)
            {
                timer = 0.0f;
                if (exploration && PhotonNetwork.IsMasterClient)
                    _photonView.RPC("ExplorationEnded", RpcTarget.All);
                else if (!exploration && !gameManager.pictureFound)
                    PictureNotFound();
            }
        }
        ShowOnGUI();
```

```
    }
}
```

Figure 4.36: The *Update* function in the *Timer* script

```
void ShowOnGUI()
{
    int minutes = Mathf.FloorToInt(timer / 60f);
    int seconds = Mathf.FloorToInt(timer - minutes * 60);
    string timerText = string.Format("{0:00}:{1:00}", minutes, seconds);
    foreach (var timeTextField in timeTextFields)
        timeTextField.text = timerText;
}
```

Figure 4.37: The *ShowOnGUI* function in the *Timer* script

If players are currently in the first part of the game, exploring the map and the players calling this function is the room owner, the function *ExplorationEnded* is called for both players through an RPC call. Whether players are exploring the map is decided through the *exploration* variable. This function, shown in Figure 4.38, stops the timer, ends the exploration period, and calls the *StartQuestions* function of the *GameManager* script, which will be covered in subsection4.2.6.

```
[PunRPC]
public void ExplorationEnded()
{
    finished = true;
    timer = 0.0f;
    exploration = false;
    gameManager.StartQuestions();
}
```

Figure 4.38: The RPC function *ExplorationEnded* in the *Timer* script

For players to be able to see how much time they have left without having to return to their starting area, a hand menu on the left hand is implemented. Before players start the game, the hand menu only contains placeholder text. In the exploration period, the hand menu displays the time left for players to explore the map, shown in Figure 4.39.

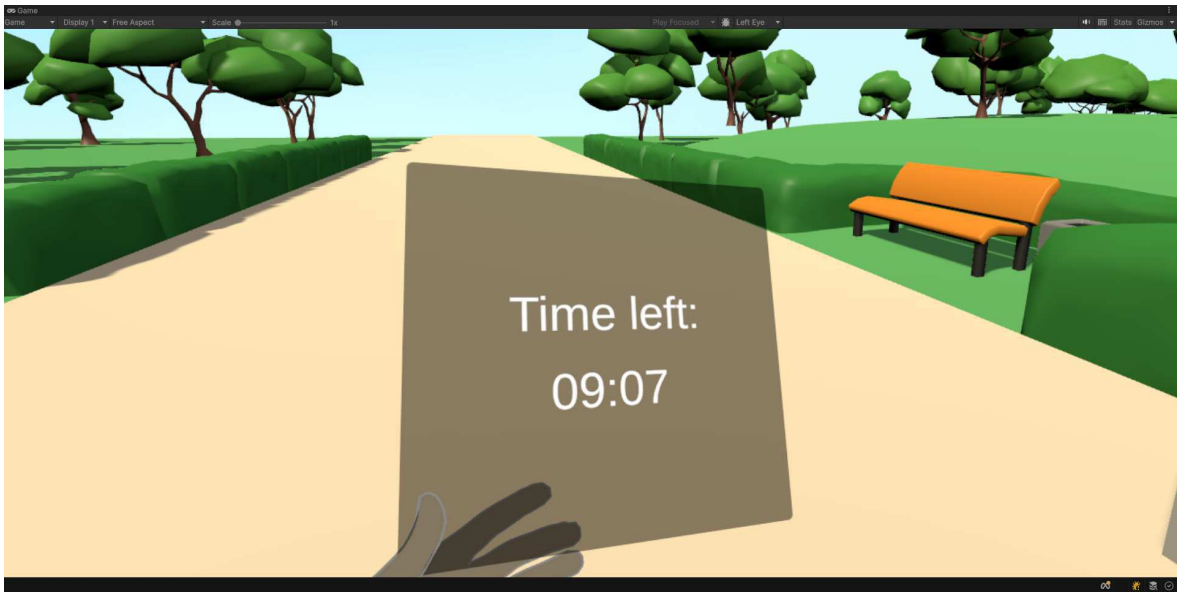

Figure 4.39: The player hand menu displaying the amount of time left to explore the map

The hand menu was implemented using the *HandMenuController* script, which can be seen in Figure 4.40. The *handPoint* and *controllerPoint* variables reference empty game objects with a *RectTransform* component attached to them. These game objects were moved to a position in which the menu can easily be seen from a player's point of view when they are using their hands or using controllers, respectively. The *gazeInteractorHandPoint* and *gazeInteractorControllerPoint* variables also reference empty objects, but with a *Transform* component, and that are positioned in the way that the menu appears only when a player turns their hand or controller inward towards themselves. The *handMenu* variable references the canvas of the hand menu. The *gazeInteractor* variable is set to reference an *XRGazeInteractor* component of a newly created gaze object.

```
1 public RectTransform handPoint;
2 public RectTransform controllerPoint;
3 public Transform gazeInteractorHandPoint;
4 public Transform gazeInteractorControllerPoint;
5 public RectTransform handMenu;
6 public Transform gazeInteractor;
```

```
private SkinnedMeshRenderer leftHand;
private SkinnedMeshRenderer rightHand;
void Start()
{
    OVRCameraRig rig = FindAnyObjectByType<OVRCameraRig>();
    leftHand = rig.transform
        .Find("OVRInteractionComprehensive/.../l_handMeshNode")
        .GetComponent<SkinnedMeshRenderer>();
    rightHand = rig.transform
        .Find("OVRInteractionComprehensive/.../r_handMeshNode")
        .GetComponent<SkinnedMeshRenderer>();
}
void Update()
{
    if (leftHand.enabled || rightHand.enabled)
    {
        handMenu.position = handPoint.position;
        handMenu.rotation = handPoint.rotation;
        gazeInteractor.position = gazeInteractorHandPoint.position;
        gazeInteractor.rotation = gazeInteractorHandPoint.rotation;
    }
    else
    {
        handMenu.position = controllerPoint.position;
        handMenu.rotation = controllerPoint.rotation;
        gazeInteractor.position = gazeInteractorControllerPoint.position;
        gazeInteractor.rotation = gazeInteractorControllerPoint.rotation;
    }
}
```

Figure 4.40: The *Start* and *Update* functions in the *HandMenuController* script

The script first sets the *leftHand* and *rightHand* variables to the *SkinnedMeshRenderer* components of the Meta XR All-in-One SDK hands in the *Start* function. This is so the script is able to check whether these components are turned on, signifying whether the user is using their hands instead of controllers. Then, the *Update* function periodically checks whether there is one hand present using these components, meaning the player is using their hands, or not, meaning the player is using controllers. Depending on whether the player is using hands or controllers, the *handMenu* position and rotation, as well as the *gazeInteractor* position and rotation are changed to their hand or controller versions, respectively.

To make the hand menu appear and disappear when turned towards and away from the player, as well as fade in and out smoothly, the hand menu canvas was given the *CanvasGroup* component. An empty object named *Sphere* was put as a child of the *CenterEyeAnchor* game object of the *OVRCameraRig* and given the *XRSimpleInteractable* component, for which the *Allow Gaze Interaction* property was set to true. Then, the *XRInteractableAffordanceStateProvider* component was added as well, with the *Interactable Source* property set to the game object's *XRSimpleInteractable*. A new *Float Affordance Theme* asset was created and called *HandMenuAffordance*. Next, a *Float Affordance Receiver* component was added to the *Sphere* game object. Its *Affordance State Provider* property was set to the before added component, while the *Affordance Theme Datum* was set to the newly created asset. Finally, the *CanvasGroup.alpha* from the hand menu canvas was set on the *Value Updated* event.

4.2.6 Picture Finding

Once the exploration period of the game is over, the *StartQuestion* function in the *GameManager* script is called, as seen in Figure 4.38. This function disables all exploration timer texts, activates all picture related menus, and calls the function *NextPictureNumber*, as can be seen in Figure 4.41. The picture related menus appear on both the players' game menus, like the example in Figure 4.39, featuring a picture taken somewhere on the map. A picture related menu also appears on the hand menu, seen in Figure 4.42, with the same picture as the ones on the game menus. All available pictures can be seen in Appendix B.

```
public void StartQuestions()
{
    foreach (var timerText in exploreTimerTexts)
        timerText.SetActive(false);
    foreach (var pictureMenu in pictureMenus)
        pictureMenu.SetActive(true);
    handMenuPictureText.SetActive(true);
    NextPictureNumber();
}
```

Figure 4.41: The *StartQuestions* function in the *GameManager* script

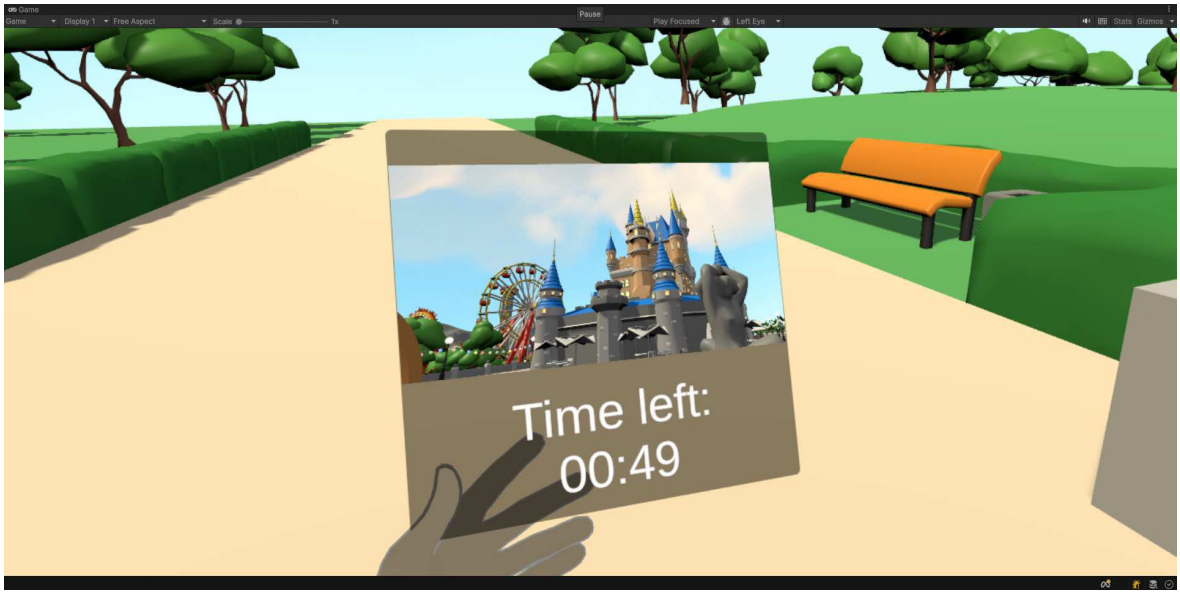

Figure 4.42: The player hand menu displaying a picture taken on the map

The *NextPictureNumber* function that can be viewed in Figure 4.43 first pauses the timer by calling the *Timer* script's *PauseTimer* function. This simply sets the started variable to false, making the timer unable to count down in the *Update* function. Then, it disables the text that shows a player has to wait for the other player to finish finding the picture. This is useful when this function is called after the other player finds a picture the current player has already found. Finally, if the player currently running this script is the owner of the room, the number of pictures that were not yet chosen is saved in the *pictureNumber* variable. Then, a random number is generated from 0 to the number of pictures not yet picked, excluding that number. The RPC function *NextPicture* is

then called for all players, with the generated random number sent as a parameter of the function.

```
public void NextPictureNumber()
{
    timer.PauseTimer();
    foreach (var otherPlayerWaitingText in otherPlayerWaitingTexts)
        otherPlayerWaitingText.SetActive(false);
    if (PhotonNetwork.IsMasterClient)
    {
        int pictureNumber = pictures.Count;
        int randomNumber = Random.Range(0, pictureNumber - 1);
        _photonView.RPC("NextPicture", RpcTarget.All, randomNumber);
    }
}
```

Figure 4.43: The *NextPictureNumber* function in the *GameManager* script

In the RPC function *NextPicture*, seen in Figure 4.44, all UI containing text is first disabled. Then, variables *answersSubmitted*, *pictureCollidersFound*, and *pictureFound* are set to their default values before players answer questions about a picture and try to find where it was taken. Next, depending on whether the player running the script is the owner of the room or not, the player's game menu shows questions about the pictures that they have to answer. The other game menu only shows the other player is answering their questions. The questions players are asked are whether they remember the place on the picture very well and whether they could find their way back to the place on the picture, with an example visible in Figure 4.45. The answers are in the form of dropdown menus, with answers available from *1 (Don't agree at all)* to *5 (Strongly agree)*. The start area colliders are activated again, restricting the players from leaving the starting area until they answer questions about the given picture, and the player is teleported back to their spawn point. Finally, all answers are then set to their default value, *1 (Don't agree at all)*.

```
[PunRPC]
public void NextPicture(int randomNumber)
{
    foreach (var pictureTimerText in pictureTimerTexts)
        pictureTimerText.SetActive(false);
    answersSubmitted = 0;
    pictureCollidersFound = 0;
    pictureFound = false;
    if (PhotonNetwork.IsMasterClient)
    {
        questions[0].SetActive(true);
        otherPlayerAnswers[0].SetActive(false);
        otherPlayerAnswers[1].SetActive(true);
    }
    else
    {
        questions[1].SetActive(true);
        otherPlayerAnswers[1].SetActive(false);
        otherPlayerAnswers[0].SetActive(true);
    }
    startAreaColliders.SetActive(true);
    OVRCameraRig.transform.position = spawnPoint.position;
    foreach (var answer in answers1)
        answer.value = 0;
    foreach (var answer in answers2)
        answer.value = 0;
    RawImage picture = pictures[randomNumber];
    pictures.Remove(picture);
    GameObject pictureCollider = pictureColliders[randomNumber];
    pictureColliders.Remove(pictureCollider);
    lastPictureCollider = pictureCollider.gameObject;
    handMenuPicture.texture = picture.texture;
    foreach (var menuPicture in menuPictures)
        menuPicture.texture = picture.texture;
```

```
35      pictureCollider.GetComponent<MeshRenderer>().enabled = true;
36      pictureCollider.GetComponent<BoxCollider>().enabled = true;
37      sr.WriteLine();
38      sr.WriteLine("Picture " + (picturesCopy.IndexOf(picture) + 1) + ":");
39  }
```

Figure 4.44: The RPC function*NextPicture* in the *GameManager* script

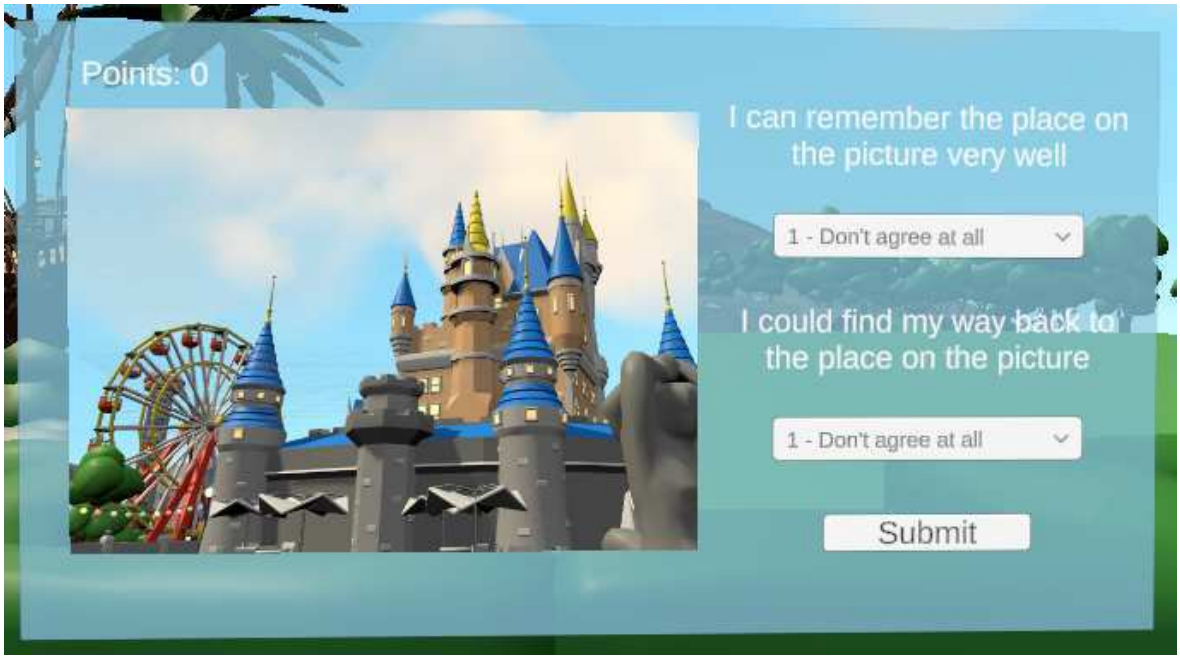

Figure 4.45: The game menu displaying a picture and questions related to the picture

In every scene, there are ten pictures, and for each picture, there is a cube game object in the place where the picture was taken, like the one shown in Figure 4.46. The cube has a *BoxCollider* component and a transparent, green material. The material was made opaque enough to be seen when players knew where it was located. It was also transparent enough so it could not be easily seen from far away, and players actually had to remember where the place in the picture was located. In the RPC function *NextPicture*, seen in Figure 4.44, the picture symbolizing the given random number parameter and the picture's corresponding cube were saved and removed from their lists so they would not be picked again. The picked cube was also saved, so it could be deactivated once the picture finding time is up. The hand menu and game menu pictures were changed to show the picked picture, and the cube was made visible. Finally, the picture's index was saved in the session's file, so the data could be analyzed later on.

Figure 4.46: The transparent, green cube in the place a picture was taken

The players now had to answer the two questions presented to them, visible in Figure 4.45. Once players click the *Submit* button, the *SubmitAnswers* function in the *GameManager* script is called. In this function, the player's answers are also stored in the session's file and the questions are disabled. A new text also appears instead of the questions, saying the other player is answering their questions, as can be seen in Figure 4.47. Additionally, this function calls the RPC function *AnswerSubmitted* for both players.

```
1  public void SubmitAnswers()
2  {
3      sr.WriteLine("I can remember the place on the picture very well: ");
4      if (PhotonNetwork.IsMasterClient)
5          sr.WriteLine(answers1[0].value + 1);
6      else
7          sr.WriteLine(answers1[1].value + 1);
8      sr.WriteLine("I could find my way back to the place on the picture: ");
9      if (PhotonNetwork.IsMasterClient)
10         sr.WriteLine(answers2[0].value + 1);
11     else
12         sr.WriteLine(answers2[1].value + 1);
13     if (PhotonNetwork.IsMasterClient)
14     {
15         questions[0].SetActive(false);
```

```
        otherPlayerAnswers[0].SetActive(true);
    }
    else
    {
        questions[1].SetActive(false);
        otherPlayerAnswers[1].SetActive(true);
    }
    _photonView.RPC("AnswerSubmitted", RpcTarget.All);
}
```

Figure 4.47: The *SubmitAnswers* function in the *GameManager* script

The RPC function *AnswerSubmitted*, seen in Figure 4.48, simply raises the *answersSubmitted* variable by one. Additionally, once the variable reaches two, meaning both players had submitted their answers, the *FindPictureCollider* is called.

```
[PunRPC]
public void AnswerSubmitted()
{
    answersSubmitted++;
    if (answersSubmitted == 2)
        FindPictureCollider();
}
```

Figure 4.48: The *AnswerSubmitted* function in the *GameManager* script

Then, the *FindPictureCollider* function disables the current game menu text for both players and shows the timer, as seen in Figure 4.49. The colliders around the player's spawn point are also disabled, so they can move freely. Finally, the timer is started again through the *Timer* script's *StartPictureTimer* function, but only for the length of finding the picture, which is set to 60 seconds or one minute. This function, seen in Figure 4.50, sets the timer to the given time and enables all text that shows timers for the picture finding period of the game. It also sets all variables needed for the timer to count down.

```
public void FindPictureCollider()
{
    foreach (var otherPlayerAnswer in otherPlayerAnswers)
        otherPlayerAnswer.SetActive(false);
    startAreaColliders.SetActive(false);
    foreach (var pictureTimerText in pictureTimerTexts)
        pictureTimerText.SetActive(true);
    timer.StartPictureTimer(timeToFindPicture);
}
```

Figure 4.49: The *FindPictureCollider* function in the *GameManager* script

```
public void StartPictureTimer(float time)
{
    timer = time;
    foreach (var timerText in pictureTimerTexts)
        timerText.SetActive(true);
    started = true;
    finished = false;
}
```

Figure 4.50: The *StartPictureTimer* function in the *Timer* script

At this point, two outcomes are possible for each player. One is that the player finds the cube corresponding to the given pictures, and the second one is that one minute passes without the player finding the cube. To cover the first outcome, each picture's cube is given the *PictureCollider* script, which checks for collisions and makes the cube transparent and removes its collider at the start of the game, during exploration time. In Figure 4.51 the *OnCollisionEnter* function can be seen, which checks whether the collision was done with an object on layer 8, which is the *Hand* layer given to the player's left hand. If it was, the *GameManager* script's *PlayerFoundPictureCollider* function is called, and the cube is disabled and made invisible.

```
private void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.layer == 8)
    {
        gameManager.PlayerFoundPictureCollider();
        gameObject.SetActive(false);
        GetComponent<MeshRenderer>().enabled = false;
        GetComponent<BoxCollider>().enabled = false;
    }
}
```

Figure 4.51: The *OnCollisionEnter* function in the *PictureCollider* script

The *PlayerFoundPictureCollider* function, part of which is seen in Figure 4.52, activates the start area colliders, so the player can not leave the spawn area anymore and teleports the player to their spawn point. It also removes the timer text from the game menus and shows new text on the game menus which says they have to wait for the other player to find the picture. Additionally, the *pictureFound* variable is set to false, rendering the Timer variable unable to finish counting down. However, the timer keeps going and is visible on the hand menu, so players can still see how much time the other player has left to find the picture's cube. Finally, the RPC function *PictureColliderFound* is called for both players.

```
public void PlayerFoundPictureCollider()
{
    startAreaColliders.SetActive(true);
    OVRCameraRig.transform.position = spawnPoint.position;
    foreach (var pictureTimerText in pictureTimerTexts)
        pictureTimerText.SetActive(false);
    foreach (var otherPlayerWaitingText in otherPlayerWaitingTexts)
        otherPlayerWaitingText.SetActive(true);
    pictureFound = true;
    ...
```

```
11      _photonView.RPC("PictureColliderFound", RpcTarget.All, pointsAwarded,
            PhotonNetwork.IsMasterClient);
12      ...
13  }
```

Figure 4.52: The gameplay part of the *PlayerFoundPictureCollider* function in the *GameManager* script

Otherwise, if the timer runs out, the *PictureNotFound* function of the *Timer* script is called, as can be seen in Figure 4.36 in line 14. It is called because it is not currently the exploration period and the picture has not yet been found. The *PictureNotFound* function, seen in Figure 4.53, stops the timer and sets it to 0 seconds, so negative time would not be shown. It also calls the *PictureNotFound* function of the *GameManager* script. In the *GameManager* script's *PictureNotFound* function, which is visible in Figure 4.54, the current active cube is deactivated. Additionally, the game menu texts displaying the timers are deactivated and the RPC function *PictureColliderFound* is called for both players.

```
1  void PictureNotFound()
2  {
3      finished = true;
4      timer = 0.0f;
5      gameManager.PictureNotFound();
6  }
```

Figure 4.53: The *PictureNotFound* function in the *Timer* script

```
1  public void PictureNotFound()
2  {
3      lastPictureCollider.SetActive(false);
4      foreach (var pictureTimerText in pictureTimerTexts)
5          pictureTimerText.SetActive(false);
6      ...
```

```
7       _photonView.RPC("PictureColliderFound", RpcTarget.All, 0,
            PhotonNetwork.IsMasterClient);
8   }
```

Figure 4.54: The gameplay part of the *PictureNotFound* function in the *GameManager* script

The RPC function *PictureColliderFound* will be covered in detail in subsection 4.2.7, but it is important to note that this function calls the function *NextPictureNumber* from the *GameManager* script, seen in Figure 4.43, if there are still pictures players have not gone through, which resets the cube finding process. If players have gone through all the pictures, the function ends the game by calling the function *EndGame*, also in the *GameManager* script. The *EndGame* function pauses the timer and teleports the player to their spawn point, then shows their final point standing.

4.2.7 Player Points

Players get points for each picture depending on the time left when they find the place a picture was taken. The faster they find it, the more points they get. If one minute runs out before they find that place, they get no points for that picture. The points for each picture start at 1000, and linearly decrease as time passes, down to 0. When playing collaborative mode, the points each player receives are divided by two, but players collect points together. In competitive mode, players earn points individually, and the one with more points performed better in finding the places where the pictures were taken. For both modes, both players need to find the place where the picture was taken, or the timer needs to count down to zero before moving on to the next picture.

To start, the previously described functions *PlayerFoundPictureCollider*, in Figure 4.52, and *PictureNotFound*, in Figure 4.54, both contain parts of the code that deal with the player's points, but were not shown in those figures. This part of the code for the *PlayerFoundPictureCollider* function in the *GameManager* script can be seen in Figure 4.55. In the code, points are awarded according to how fast the player finds the picture origin, with the points starting at 1000 and lowering as more time passes. If the players are playing collaboratively, the points are halved.

```
public void PlayerFoundPictureCollider()
{
    ...
    int pointsAwarded = (int) (timer.timer / timeToFindPicture * 1000);
    if (gamemode == Gamemode.COLLABORATIVE)
        pointsAwarded /= 2;
    _photonView.RPC("PictureColliderFound", RpcTarget.All, pointsAwarded,
        PhotonNetwork.IsMasterClient);
    sr.WriteLine("Time: " + (timeToFindPicture - timer.timer));
    sr.WriteLine("Points: " + pointsAwarded);
}
```

Figure 4.55: The player points part of the *PlayerFoundPictureCollider* function in the *GameManager* script

Then, the RPC function *PictureColliderFound*, which will be explained shortly, is called for both players, with the number of points awarded and whether the current player is the room owner as parameters. Finally, the time spent looking for where the picture was taken, as well as the points gained, are written in the current session's file. Similarly, in the *PictureNotFound* function in the *GameManager* script seen in Figure 4.56, the same RPC function is called. The time and points are written down in the session's file, but the time is set to 60 seconds, and the points awarded are zero.

```
public void PictureNotFound()
{
    ...
    sr.WriteLine("Time: " + timeToFindPicture + " (Not found)");
    sr.WriteLine("Points: " + 0f);
    _photonView.RPC("PictureColliderFound", RpcTarget.All, 0,
        PhotonNetwork.IsMasterClient);
}
```

Figure 4.56: The player points part of the *PictureNotFound* function in the *GameManager* script

In the *PictureColliderFound* function in the *GameManager* script, which can be seen in Figure 4.57, firstly the *pictureCollidersFound* variable is raised by 1. Then, the points awarded are added either to one of the two players, depending on which player called the RPC function, or added to a pool of points for both players. This differs based on whether the players are playing the competitive or collaborative version of the game. Then, the *UpdatePlayerPoints* function is called, which simply updates both player's points on the game menu, as seen in Figure 4.58. These points are visible in the top left corner of both player's game menus, as can be seen in Figure 4.45. If both players found the picture's cube or the time ran out, which would make the value of the *pictureCollidersFound* variable equal to two, the text saying players have to wait for the other player deactivates. Finally, if there are more pictures left to go through, the next picture is chosen and shown through the *NextPictureNumber* function. Otherwise, the game ends with the *EndGame* function in the *GameManager* script.

```
[PunRPC]
public void PictureColliderFound(int pointsAwarded, bool isMasterClient)
{
    pictureCollidersFound++;
    if (gamemode == Gamemode.COMPETITIVE)
    {
        if (isMasterClient)
            playerOnePoints += pointsAwarded;
        else
            playerTwoPoints += pointsAwarded;
    }
    else
    {
        togetherPlayerPoints += pointsAwarded;
        playerOnePoints = togetherPlayerPoints;
        playerTwoPoints = togetherPlayerPoints;
    }
    UpdatePlayerPoints();
    if (pictureCollidersFound == 2)
    {
```

```
21          foreach (var otherPlayerWaitingText in otherPlayerWaitingTexts)
22              otherPlayerWaitingText.SetActive(false);
23          if (pictures.Count > 0)
24              NextPictureNumber();
25          else
26              EndGame();
27      }
28  }
```

Figure 4.57: The *PictureColliderFound* function in the *GameManager* script

```
1  public void UpdatePlayerPoints()
2  {
3      points[0].text = playerOnePoints.ToString();
4      points[1].text = playerTwoPoints.ToString();
5  }
```

Figure 4.58: The *UpdatePlayerPoints* function in the *GameManager* script

The *EndGame* function, visible in Figure 4.59, pauses the timer and teleports the player back to their spawn point. Then, the game menu is set to show only the final scores of each player, with them having the same number of points if they played collaboratively, and different scores if they played competitively. An example of a player's final score can be seen in Figure 4.60. Finally, the total points of the player running the script are written in the current session's file.

```
1  void EndGame()
2  {
3      timer.PauseTimer();
4      OVRCameraRig.transform.position = spawnPoint.position;
5      foreach (var pictureMenu in pictureMenus)
6          pictureMenu.SetActive(false);
7      foreach (var gameEndMenu in gameEndMenus)
8          gameEndMenu.SetActive(true);
```

```
    foreach (var otherPlayerWaitingText in otherPlayerWaitingTexts)
        otherPlayerWaitingText.SetActive(false);
    finalPoints[0].text = playerOnePoints.ToString();
    finalPoints[1].text = playerTwoPoints.ToString();
    if (PhotonNetwork.IsMasterClient)
    {
        sr.WriteLine();
        sr.WriteLine("Total points: " + playerOnePoints);
    }
    else
    {
        sr.WriteLine();
        sr.WriteLine("Total points: " + playerTwoPoints);
    }
    sr.Close();
}
```

Figure 4.59: The *EndGame* function in the *GameManager* script

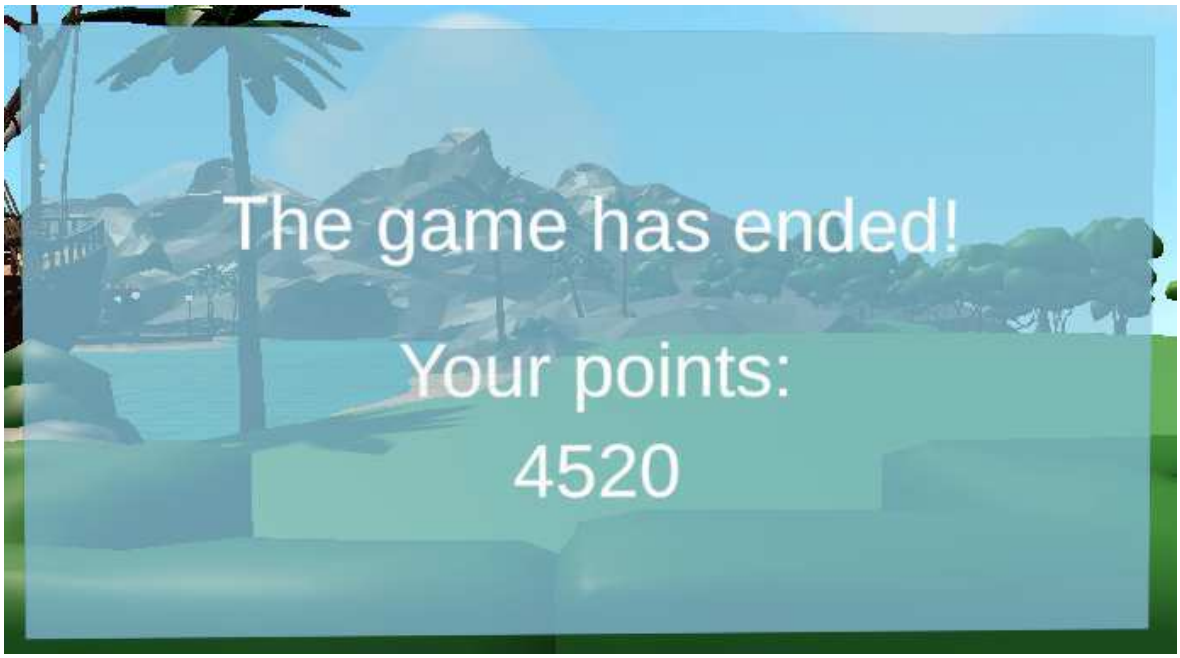

Figure 4.60: A player's final score shown at the end of the game

4.3 Limitations

Although **OVRseer** is a prototype, there are some limitations and possible future additions which will be listed in this section. Most of them are bugs or limitations with the gaze tracking data gained from Cognitive3D. Some are limitations with the headsets used in the user study, or a possible addition to make the multiplayer experience more immersive for players.

4.3.1 No Scene to Scene Sessions

The original idea for **OVRseer** was to have a main menu in which players could navigate to one map or the other. Once players finished the game, they would return to the main menu and be able to play again. However, the version of Cognitive3D used in this implementation does not support recording multiple sessions for different scenes in one go. For example, when players were in the beginning scene, the session for that scene would be recorded. However, when they loaded the amusement park scene, the sessions for that scene would not be recorded. Because of this, the amusement park scene and the medieval village scene were made into two separate builds, and the builds were run one by one.

4.3.2 Missing Shaders Data

The *Dreamscape Village - Stylized Fantasy Open World* asset contains non-standard shaders for the terrain and the foliage, as well as a custom heightmap solution. Due to this, the map's terrain appears gray and flat in the *Scene Explorer*, which is visible in Figure 4.61. When talking to the Cognitive3D team about this, they offered a solution in which the custom shader support can be added, but many custom shaders were used, so this was not added due to lack of time. Additionally, the Cognitive3D web-application contains a guide for adding custom shader support, but, as of writing this thesis, it is outdated and therefore cannot be used. However, because gaze is caught on colliders, the gaze data on this map is still shown correctly, and the most important parts of the map, such as the houses and the statue, are still shown clearly.

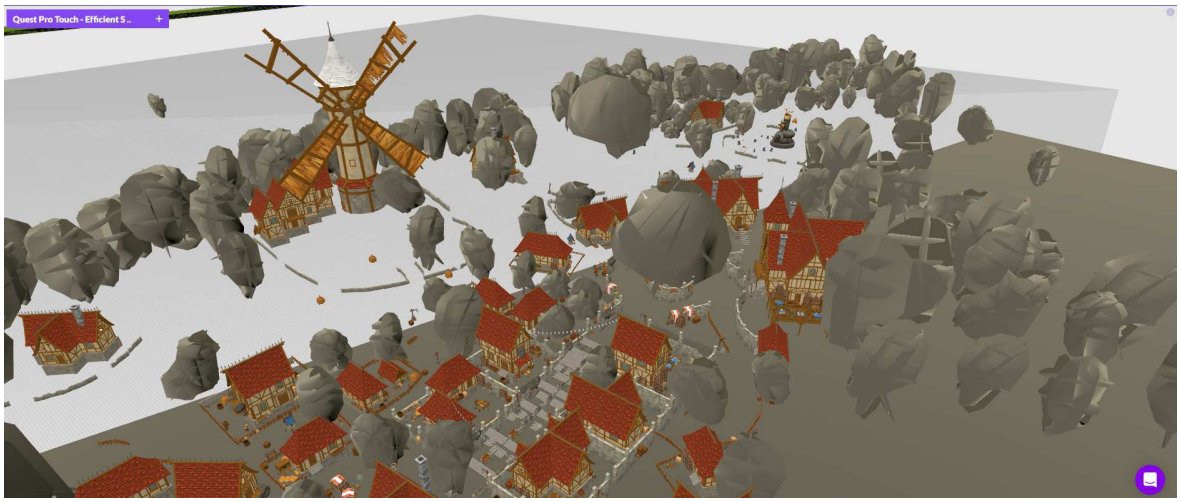

Figure 4.61: The medieval village map appears mostly gray and flat in Cognitive3D's *Scene Explorer*

4.3.3 Players Looking at Their Controllers

Unfortunately, most of the sessions ended up having players spend a significant portion of their time dedicated to gazing at their controllers, at least according to Cognitive3D. As can be seen in Figure 4.62, the left controller of the player appears red in the *Scene Explorer*, meaning that the player looked at it often. When looking at participants of these sessions while they were being recorded in real life, however, this does not appear to be the case. This results in less gaze on objects present on the map, so less data to be analyzed overall. There are a few reasons that may be the source of this problem. The first could simply be that the Meta Quest Pro is not accurate enough at tracking a user's gaze, or players moved the headset after eye tracking calibration had already been done, resulting in inaccurate readings. Another possible reason is that the Cognitive3D SDK does not correctly show the user's gaze within Unity. A potential fix is possible to bypass at least a part of this issue, and that is to place the controllers and hands on a new layer in Unity, then disable gaze from hitting objects in that layer. This can be done through the *Dynamic Object Layer Mask* property in the Cognitive3D *Preferences* file, which can be opened through the Cognitive3D tab at the top of the Unity editor.

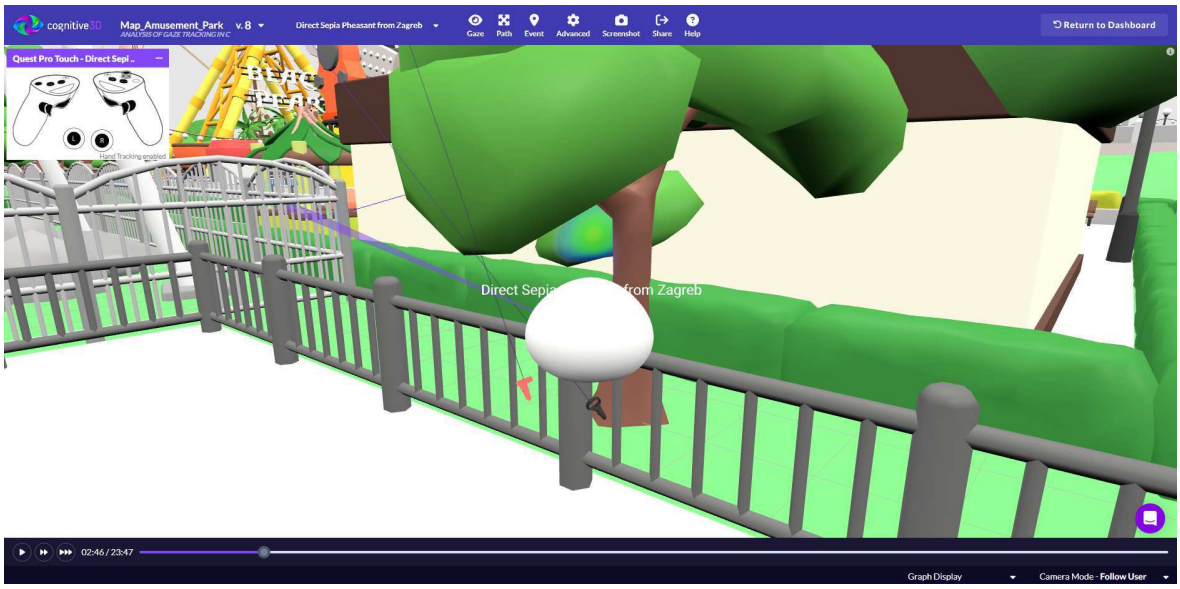

Figure 4.62: A player's left controller appears red in Cognitive3D's *Scene Explorer* because it was gazed at often

4.3.4 Slow Aggregated Gaze Data

When looking at animated gaze data in Cognitive3D's *Scene Explorer*, the recording appears smooth. However, when the data is switched to an aggregated view, the camera movement appears very slow, almost as if it is lagging. While the cause of this is unknown, it may be that the aggregated gaze data is calculated every frame. However, a solution for this could be to just calculate the aggregated gaze data once. After that, either the materials could be colored with gaze data at once or the aggregated gaze data could be sent directly again, without having to be calculated again.

4.3.5 Slow Loading of Multiple Sessions

The amusement park scene file size is about 20 MB, while the medieval village scene file size is nearly 300 MB. Cognitive3D's *Scene Explorer* loads both scenes fairly slowly, with the medieval village scene loading slower because it is bigger. However, to view a heatmap of the participant's gaze data, multiple sessions needs to be chosen in the *Scene Explorer* or the heatmap needs to be viewed through the *Scene Viewer*. The *Scene Viewer* does not display a lot of data if there are only a few sessions, like it is the case with this thesis. Unfortunately, loading multiple sessions in the *Scene Explorer* is extremely slow. A few sessions were left to load for multiple hours, but ultimately did not load at all. Because of this, it is not possible to view an aggregated heatmap of the player's gaze data.

4.3.6 Meta Quest Pro Comfort

Seven out of eight participants wearing the Meta Quest Pro during the user study had to take it off in the middle of the study or had headaches after wearing it because it felt uncomfortable. These problems might have caused disturbances in the gaze data. All participants completed the user study fully. Some participants took a short break, up to 10 minutes, after one map to ease their headache, then completed the eye tracking calibration once more. The main problem lies in the part of the headset which sits on the forehead, as participants temporarily had very red marks and indents on that part of their forehead. In the future, it would be a good idea to explore different headsets that can track gaze or ease the comfort of use for the Meta Quest Pro.

4.3.7 Networked Hands

In the future, it would be useful to implement networked hands and controllers, so players could see each other's hands movements instead of just the other player's head. The *Player* prefab's component *VRTracker* contains code that places hands and controllers in the right position and rotates them by tracking the player's hand or controller movements. Additionally, it switches between hands and controllers depending on which the controller is using. However, animating hands over the network was not implemented, so the aforementioned code is not used in the current version of the game. In the future, this could be implemented to improve the user experience.

5 User Study

The goal of this thesis was to investigate the differences in gaze tracking data between collaborative and competitive VR games. The prior described VR game **OVRseer**, which was examined in detail in chapter 4, is used as an example of a VR game with both collaborative and competitive game modes. The game contains elements that track the player's gaze data if they are using a VR headset that supports gaze tracking. Afterward, the session recording is uploaded to Cognitive3D, where the data can be analyzed.

5.1 Methodology

This user study examines gaze data gathered from participants playing the multiplayer VR game **OVRseer**, responses to questions provided in the game, and general questions. These data are then analyzed to gain insight into which parts of the map participants focus on in the collaborative and competitive game modes. The results may prove helpful in the future implementation of multiplayer VR games by determining how to focus the visual attention of participants on certain areas of the virtual environment.

Gaze tracking is utilized in **OVRseer** to obtain information about how much and how long a participant is looking at something in the virtual world. These data are analyzed to determine which areas participants are most focused on. Throughout the course of the game, participants are presented with two sentences for each picture displayed on each map. These sentences represent the participant's subjective assessment of their familiarity with the depicted picture. Players must rate how true the sentences are for them for each picture. The first is *"I can remember the place on the picture very well"*, while the second is *"I could find my way back to the picture"*. Each of these can be rated a score from 1 to 5, with 1 being *Don't agree at all* and 5 being *Strongly agree*. These ratings may have a correlation with how long participants have spent looking at the objects shown on

the picture. Additionally, participants must answer several general questions in a form. The data gathered in this form, such as gaming experience and VR experience, may also have an impact on a participant's gaze pattern or performance in the game.

The study was done on pairs of participants. It was conducted in a designated laboratory room, MUEXlab[1]. One participant wore a Meta Quest 2 headset, while the other participant wore a Meta Quest Pro. Because of the nature of these headsets, only the gaze of the latter participant was tracked, since the Meta Quest 2 does not support gaze tracking, but other data for both participants was recorded. Both headsets were connected to computers with cables, so the session's file detailing both participant's points could be accessed later on. Both participants were sitting down on chairs for the entire duration of the study. Because the game is networked, the computers were connected to the Internet via Ethernet.

Each study trial lasted around an hour, with a pair of participants being tested. After the participants arrived, they were directed to their computers, where they were tasked to fill out the first part of the form, which included questions about general information such as age, gender, gaming experience, and VR experience. Afterward, the participants put on their respective VR headsets and were given instructions on how to play the game. The participants who wore the Meta Quest Pro additionally had to calibrate their gaze, so the gaze data would be as correct as possible. Their first map was then loaded, and the participants were given movement instructions and a moment to interact with each other in the game. The participants were also told they could ask any questions about the gaze rules or interactions, as long as the answer would not provide them with an unfair advantage in terms of points. After this, the participants played one map, then the other, firstly in one game mode, then the other. It should be noted that the order of the maps and the order of the game modes were randomized, so the maps or the game modes have as little impact as possible on one another. After playing on both maps, the participants left, and the test administrator filled out the second part of the form for both participants. Both files of both sessions were also saved.

[1]https://muexlab.fer.hr

For the purpose of this user study, a simple form consisting of 10 questions was put together. The questions consisted of short text questions (for typing age and points), multiple-choice questions (both single and multiple answers, some with an open-ended "other" option which they could fill themselves), and binary questions (Yes/No and choosing one of the two maps). The entire form is given in Appendix A. The first part of the form contains questions about the participant's age, gender, weekly hours spent on video games, weekly hours spend on virtual reality, and platforms that they used to play video games on in the last year. These questions were answered by participants. The second part of the form serves for the recording of objective metrics, such as the played map per game mode, points, and which player won the game. These questions were filed out by the test administrator after participants finished playing the game. They had to be filled out for both the collaborative and the competitive game mode.

The user study involved 16 participants, with 9 of them identifying as a woman, and 7 of them identifying as a man. The age range of the participants spanned from 19 to 26, with the average age being 23. When asked about their weekly time spent on video games, 2 participants reported not playing video games, 7 participants stated they spent 5 or fewer hours weekly. 3 participants reported spending 10 or fewer hours a week, 2 participants spend 20 or fewer hours weekly, and 2 participants reported to spending 40 or fewer hours weekly. When asked to report their weekly time spent on VR, 11 participants stated they do not use VR, while the other 5 participants reported to using VR for 5 or fewer hours weekly. Out of all the platform options available for playing video games, when asked about the experience in the last year, 14 participants stated they played on personal computers, and 12 participants reported playing on mobile platforms. Eight participants stated they used consoles, eight participants reported they use VR, and four participants stated they used handheld consoles.

5.2 Results and Discussion

In this section, the results of the gathered form data will be analyzed and discussed. Then, the same will be done for data from the questions and points of each game session. Afterward, the gaze tracking data will be analyzed, along with correlations to the previous data. Finally, the key findings for the data will be presented.

5.2.1 Form Data

Because the participants in this study identified either as a woman or a man, only those genders can be analyzed, but there was no significant difference in performance between women and men. When analyzing the number of hours participants spent on video games and VR weekly, there was also no conclusive evidence to show those who spent more, or fewer hours performed better than others. For the two pairs in which one participant spent many more hours a week on video games, the participant with more hours spent accumulated more points. In three pairs where one participant spent slightly more hours on video games weekly, the participant with fewer hours spent won the game. In other pairs, participants stated they spent the same amount of time on video games. When looking at VR experience, only one pair noted a difference in the time they spend on VR in a week, with both pairs still having little experience. The participant with less VR experience gathered more points. Other than these pairs, in one more pair, a participant noted they have played VR before, and lost, while the other participants had not played VR before. This data could point to the game's controls being different and worse than other VR games. However, for all this data, more participants are required before considering their implications in future development.

When considering the points gathered, the amount of points participants gathered between collaborative and competitive game modes are roughly the same, with the average being 3193.75 (standard deviation (SD): 1129.58) for collaborative and 3250.44 (SD: 1469.76) for competitive. The collaborative mode has a slightly higher standard deviation compared to the competitive mode. The total average points for both maps and both game modes was 3222.09 (SD: 1289.76). However, the average points and their standard deviations for the medieval village map were much higher than those for the amusement park map. The average points earned for collaborative play was 2764.00 (SD: 1404.20) for the amusement park map, while it was 3623.50 (SD: 583.30) for the medieval village map. For the competitive game mode, the average points participants earned on the amusement park map was 2977.38 (SD: 1488.03), while it was 3523.50 (1498.10) for the medieval village map.

The total average points gained for the amusement park map, including both the collaborative and the competitive game mode, was 2870.69 (SD: 1402.00), while the medieval village average was 3573.50 (SD: 1099.45). This difference was not expected, especially since almost all participants thought the medieval village map was going to be more difficult. They said the amusement park map was easier to remember because of the vibrant, differing colors around the whole map and the structural walkways, while the medieval village map appeared much more monotone. The participants' reasoning for thinking the medieval village map was going to be harder is because a lot of assets were repeated. Alongside that, the exploration allowed participants much more freedom since it was more like an open-world game, so they thought they would not be able to remember all parts of the map. Additionally, the standard deviation for the amusement park map was higher than that of the medieval village map.

5.2.2 Session Questions and Points Data

The average score participants gave for how well they can remember the place in the picture for the amusement park was 3.63 (SD: 0.48), while the medieval village score was 3.48 (SD: 0.39), as seen in Table 5.1. Similarly, when asked could they find their way back to the place in the picture, participants gave an average score of 3.39 (SD: 0.42) for the amusement park map and a score of 3.37 (SD: 0.45) for the medieval village map. This means participants were more confident about their map knowledge in the amusement park map than in the medieval village map. However, participants earned more points, or, in other words, found the origin of the picture faster, on the medieval village map than on the amusement park map, with a difference of 4.36. This data is conflicting, especially because participants commented how the amusement park map was easier to remember because of the bright colors and structural walkways, as opposed to the more dull colors and open-world elements of the medieval village map.

Additionally, as can be seen in Table 5.1, the average collaborative score participants gave for remembering the place on the picture in the amusement park map was 3.95 (SD: 0.55), while it was lower, 3.31 (SD: 0.51), for the competitive game mode. The same goes for finding their way back, with an average score of 3.86 (SD: 0.55) for collaborative, and 2.93 (SD: 0.38) for competitive. On the amusement park map, participants were more confident when playing collaboratively with another participant. However, the

difference in the time they found the picture origins, and the points they got, was very slim, with participants finding them 1.30 seconds slower in the collaborative version. It should be noted that the points for the collaborative version are halved, since participants gain half the points for each find because they collect points together.

Table 5.1: Average participant scores and time to find picture origins for each map and game mode

	I can remember the place on this picture very well	**I could find my way back to the place on the picture**	**Time to find (seconds)**	**Points**
Amusement park both	3.63	3.39	42.73	217.97
Medieval village both	3.48	3.37	38.37	263.35
Amusement park collaborative	3.95	3.86	43.38	138.20
Medieval village collaborative	3.21	3.23	37.50	181.18
Amusement park competitive	3.31	2.93	42.08	297.74
Medieval village competitive	3.74	3.51	39.24	345.53

On the other hand, participants were more confident in their map knowledge in the competitive version on the medieval village map, but were also slightly faster to find the picture origins in the other game mode. They were 1.74 seconds slower in the competitive version. Participants rated remembering the place on the picture a 3.21 (SD: 0.55) for the collaborative version and 3.74 (SD: 0.56) for the competitive one. Similarly, for finding their way back, participants gave a score of 3.23 (SD: 0.51) for the collaborative game mode and 3.51 (SD: 0.49) for the competitive one. Participants ended up giving higher scores on the amusement park map for the collaborative game mode, but lower scores for the same map on the competitive game mode. However, they ended up finding picture origins faster on the medieval village map overall than on the other map.

For the amusement park map, the pictures participants had the most trouble finding can be seen in Figures B.3, B.4, B.6 and B.9. The cube for the first of these pictures appears very transparent due to the surrounding colors, and it is on the stairs, so it is easy to miss, and 10 participants missed it. The second picture is close to the first one, and was also not found by 10 participants, and they are both in a place on the map many participants explored little. The third and fourth are both in places that are not on the main road, but closer to the attractions themselves, however, attractions are visible in both pictures, so participants should have been able to navigate using those. The third picture's cube was not found by 10 participants, while the fourth picture's cube was missed by 9 participants.

The pictures which participants performed best at in the amusement park map were the ones in Figures B.2 and B.8. The former was not found by only 1 participant, while the latter was not found by 2 participants, but 7 participants found it within 20 seconds. This is a bit surprising for the first picture, as it was on the far edge of the map. The second picture had a clear view of only one attraction, which perhaps helped participants by not being overloaded with a lot of different objects and colors in one picture.

When looking only at the collaborative sessions, the pictures participants had most trouble navigating are the ones in Figures B.3, B.4, and B.9, with the same reasons as stated before. The first one was missed by 6 participants, while the second and third ones could not be found by 5 participants. The ones participants found the fastest are seen in Figures B.1 and B.8. The reason for the latter is stated before, and it was found within 20 seconds by 1 participant and missed by only 1 participant. The former is near the castle, a central point of the map, and participants commonly used it to navigate each other. Because of this, it is likely participants knew this part of the map very well. This picture's cube was found by 3 participants within 20 seconds and only not found by 1.

In competitive sessions, the picture for which participants missed the cube the most can be seen in Figures B.3, B.4, B.6, and B.9, the same ones as when looking at the overall sessions. The first and last of these pictures' origins were not found by 4 participants, while 5 participants missed the other two. The ones participants found the easiest were the ones in Figures B.8 and B.10. The reasoning for the former is explained above, while the latter features the ship that can be seen at the starting area. Both picture's origins were found within 20 seconds by 4 participants and only missed by 1.

On the other hand, the worst performing pictures in the medieval village map were the ones in Figures B.16, B.17, and B.20. The first two were both not found by 8 participants, but are located in the middle point of the map, so it is not clear why they were missed by so many participants. The last picture was very close to Figure B.19, which was not found by only 2 participants, while 13 participants missed the last one. Both places are on the far edge of the map, and the less frequently found one is located inside a structure, which may have been overlooked by participants (i.e., there is a possibility that participants failed to notice its entrance). However, the worse performing picture was commonly mistaken to be in the inn in the middle of the map, which is not the case, so participants would often waste time searching there.

The only picture origin found by all participants is for the picture in Figure B.18, which is taken in the aforementioned inn in the middle of the map. The picture in Figure B.11 was not found by only 1 participant, but 13 participants found it within 20 seconds. The reason for this could be because it is very close to the starting area, and the windmill in the background of the picture was frequently used as help for navigation. Additionally, the origin of the picture in Figure B.12 was missed by only 2 participants, and found within 20 seconds by 6 participants. This is a bit contradicting because many participants said they did not know where this was taken when first presented with the picture. However, this picture was taken near the starting area, and participants perhaps navigated using the parts of the picture that were familiar to them.

By filtering only by collaborative sessions, the worst performing pictures are the ones seen in Figures B.17 and B.20, with the same reasoning as stated before. The first picture was missed by 4 participants, while 6 participants could not find the second picture's origin. Pictures in Figures B.11 and B.12 were both found by all participants. Both pictures were found within 20 seconds by 6 and 3 participants, respectively. The reason these pictures performed so well is explained before.

In only the competitive sessions, the pictures for which the origins were not found most frequently can be seen in Figures B.16 and B.20. They are also among the worst performing pictures in the overall medieval village pictures for both versions. The former was not found by 6 participants, while 7 participants did not find the latter. The picture for which most participants found the origin is the one in Figure B.11. Again, this is

probably because of the closeness of the picture's origin to the starting area. The picture's cube was only missed by 1 participant, while 7 participants found it within 20 seconds.

Key Findings

To summarize, participants were more confident in the knowledge of the amusement park map. However, they performed better when playing on the medieval village map, finding the origins of pictures faster, and therefore earning more points. On the amusement park map, participants were more confident in their map knowledge in collaborative mode, but performed slightly better in competitive mode. On the medieval village map, it was the opposite case, with participants being more confident in the competitive game mode, but performing a bit better in the collaborative one.

In the amusement park map, players had the most trouble with the pictures seen in Figures B.3, B.4, and B.9. The first has a cube that appears very transparent because of the surrounding colors. The second and the first were both in an area of the map that was not explored a lot by the participants. The third, however, has an attraction participants could use to navigate. Even though some participants found the attraction itself, they could not figure out where the picture origin was. The best performing pictures in the amusement park map are the pictures in Figures B.2, B.8, and B.10. The first of these is on the edge of the map, so it is a bit surprising most participants managed to find the origin. The second has a clear view of only one attraction, which could have helped participants by not overloading them with information. The last is a picture of the ship that can be seen from the starting area, so it was hard to miss.

When looking at the medieval village map, the worst performing pictures are the ones seen in Figures B.16, B.17, and B.20. Both the first and the second pictures were taken at the middle point of the map, so it is unclear why so many participants had trouble finding them. The third one is located on the far back end of the map, in an opened structure. However, some participants may have missed the entrance of the structure, causing them to not be able to find the origin of the picture. The pictures which most players had an easy time finding are the ones in Figures B.11, B.12, and B.18. The first and second picture area very close to the starting area. The last was taken in an inn in the middle of the map, which all players had visited.

5.2.3 Gaze Tracking Data

Gaze tracking data will be analyzed through the Cognitive3D web-application. The gaze tracking data will be analyzed only for participants who wore the Meta Quest Pro headset, as the headset worn by the other participant, the Meta Quest 2, does not support gaze tracking. The gaze tracking from the latter headset comes from a line coming out directly from the head of the participant in Cognitive3D, which is not equal to where the participants are actually looking. It should be noted that gaze data in the *Scene Viewer* is not that visible, and only a few cubes are visible, as seen in Figure 5.1. The reason for this could be that it requires more sessions in order to show gaze well, so gaze tracking data will only be analyzed through the *Object Explorer* and the *Scene Explorer*. In the *Object Explorer*, something similar occurs, but the gaze heatmap is still visible on some dynamic objects. This includes dynamic objects which have had participants looking at them for longer times than other dynamic objects when looking at all sessions with the Meta Quest Pro. However, when looking at, for example, only the collaborative sessions with the Meta Quest Pro, the heatmaps can also barely be seen. Metrics for dynamic objects are available no matter how many sessions are filtered.

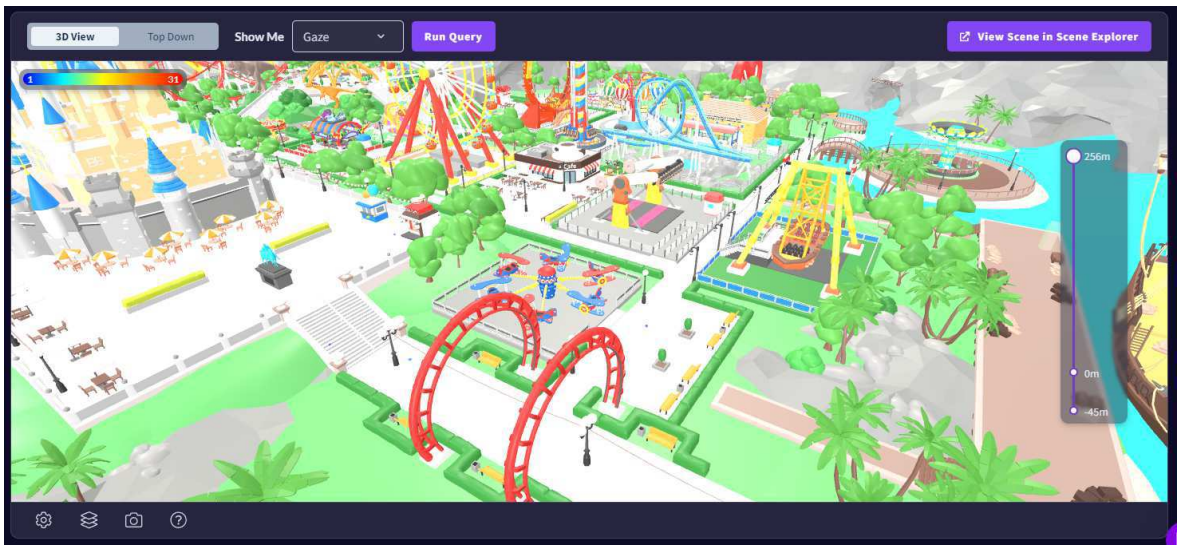

Figure 5.1: Only a few cubes are visible in the *Scene Viewer* due to lack of sessions

Dynamic objects with the highest average gaze time for collaborative, competitive, and both those game modes, and the data available for those objects, are visible in Table 5.2. The dynamic object with the highest average gaze time for both game modes in the amusement park map is the castle. This is to be expected, because it is a large, tall object useful for navigation. Additionally, it had a tunnel going through its center that

the participant can go into. The average gaze count for the castle is 33.50 times, while the average gaze time is 10.72 seconds. The second is the Ferris wheel in the middle of the map, also a tall object, with an average gaze count of 15.25 and an average gaze time of 3.46 seconds. The third is the red roller coaster spanning across most of the back part of the amusement park, with an average gaze count of 20.00 and the average gaze time of 3.34 seconds. The gaze heatmaps for these dynamic objects can be viewed in Figure 5.2. Other dynamic objects with high average gaze time include the orange building, the pirate ship, the boat swing, and the drop tower. It should be noted that most of these dynamic objects, except for the castle, boast warm colors, such as reds and browns. However, most of them, including the castle, are very big objects, so it is expected that the participants look at them.

When looking only at collaborative sessions, the castle is also the object with the highest average gaze time, this time of 13.03 seconds, and it has an average gaze count of 41.75. The second dynamic object is, once again, the Ferris wheel, with an average gaze count of 20.00, and an average gaze time of 5.28 seconds. The third is the orange building to the right of the map that has an average gaze count of 12.75 and an average gaze time of 4.13. Other high-ranking dynamic objects in terms of average gaze time include the red roller coaster, the pirate ship, the swing carousel, the boat swing, and the blue roller coaster. While most of these dynamic objects also have warm color palettes, some of them, such as the castle, the swing carousel, and the blue roller coaster, are mainly filled with cold colors, such as blues.

Figure 5.2: Gaze heatmaps of the castle (left), Ferris wheel (middle), and roller coaster (right) in the amusement park map in both game modes

Table 5.2: Gaze tracking data for dynamic objects with highest average gaze time for each mode in the amusement park map

	Castle	**Ferris wheel**	**Red roller coaster**	**Orange building**	**Drop Tower**
Both modes average gaze count	33.5	15.25	20.00	10.00	8.88
Both modes average gaze time (seconds)	10.72	3.225	37.5005	2.81	2.25
Both modes gaze ratio (%)	100.00	87.50	100.00	87.5	75.00
Collaborative mode average gaze count	41.75	20.00	23.25	12.75	9.00
Collaborative mode average gaze time (seconds)	13.03	5.28	4.10	4.13	2.20
Collaborative mode gaze ratio (%)	100.00	100.00	100.00	100.00	75.00
Competitive mode average gaze count	25.25	10.50	16.75	7.25	8.75
Competitive mode average gaze time (seconds)	8.43	1.65	2.58	1.50	2.30
Competitive mode gaze ratio (%)	100.00	75.00	100.00	75.00	75.00

In only the competitive sessions, the castle was also at the top of the list of dynamic objects by average gaze time. It has an average gaze count of 25.25 and an average gaze time of 8.43 seconds. The next dynamic object is the red roller coaster, with an average gaze count of 16.75 and an average gaze time of 2.58 seconds. The third object was the drop tower, with an average gaze count of 8.75 and an average gaze time of 2.30 seconds. Other highly ranked dynamic objects by average game time are the yellow, spherical platform, the fountain near the Ferris wheel, the café, the Ferris wheel, and the surprises gift stall. Only the castle and the fountain are objects with cold colors, while others are

mainly colored in warm colors, such as reds and browns.

The castle has the highest average gaze time of all dynamic objects through all sessions, possibly due to its size and because it in front of the participants' spawn point. The Ferris wheel and the red roller coaster are high on the list for all modes, probably also because of their size. It should be noted that while in both game modes participants gazed at warm colored objects more often, in collaborative mode, objects with colder colors caught their attention more often. On the other hand, in competitive mode, participants looked at objects with colder colors less frequently. The color red is typically associated with danger, power, and importance [38]. Blue is regularly paired with feelings of calmness, reliability, and productivity [39]. One reason why participants may be looking at blue objects more when playing collaboratively is because playing with the other participant puts them at ease as they work together. This in turn could make them notice more blue objects on the map. On the other hand, playing against the other participant may give them a sense of urgency or importance to gather as many points as possible to win. Looking at red objects may also heighten their feeling of competitiveness.

When comparing these data to the pictures that performed best and worst, a few correlations can be found. Firstly, in pictures that performed the worst for both game modes, which can be seen in Figures B.3, B.4, B.6, and B.9, some of the objects with higher average gaze times can be seen, but they are not the central points of the pictures. In the first one, none of these dynamic objects, which could have been useful to players for navigation, can be seen. The second and third ones contain the drop tower, but participants found it difficult to navigate to where the pictures were actually taken. This could possibly be due to the symmetric nature of the drop tower, so participants could not use only that attraction for help. The third picture additionally contains the orange building, but it is part of the background, and many people may not have noticed it. The last picture features the Ferris wheel, but it is also not the central point of the picture.

On the other hand, the pictures found by most participants are the ones seen in Figures B.2 and B.8. The former contains the red roller coaster as a major part of the picture. The latter features the ship on the water in the background of the picture. However, even though it is on the background, participants could have connected the ship to the water it floats on. This could have ended up making the origin of this picture easier to find.

In only the collaborative sessions, the pictures in Figures B.3, B.4, and B.9 also performed poorly, for the same reasons as stated before. However, it should be noted that the drop tower had less gaze time in collaborative mode than when looking at both modes. Although the difference is not very significant, this could reinforce the reason participants have trouble with finding the origin of the picture in Figure B.4. It should also be noted that in collaborative mode, the orange building had much higher average gaze time. Additionally, participants did not have as much trouble with the picture in Figure B.6 when playing collaboratively. The reason could potentially be that they could navigate better due to looking at the orange building more and remembering that part of the map better.

The pictures participants found most easily are the ones in Figures B.1 and B.8. The former contains the castle and the Ferris wheel, which had higher average gaze count and higher average gaze time in collaborative mode. This could point to participants remembering the castle and the Ferris wheel better, and thus finding the origin of the picture more easily. The latter contains also performs well when looking at both modes, but it should be noted that it contains the swing carousel. This attraction has the highest average gaze time in collaborative mode than when looking at both modes, so this could be a reason why participants found it easier in that game mode.

In competitive sessions, participants had problems with finding the origins of the pictures in Figures B.3, B.4, B.6, and B.9. It should be noted that some of the objects in these pictures are ranked high by their average gaze time, such as the drop tower, the yellow, spherical platform, and the Ferris wheel. However, even though these dynamic objects have higher average gaze time than others in competitive mode, these times are still fairly small when compared to the average gaze time in collaborative mode. For most dynamic objects, their average gaze time was higher when participants played collaboratively. Because of that, the reasoning for these pictures performing poorly is the same as when looking at all sessions. Additionally, although the picture in Figure B.6 was not listed as one of the badly performing ones in collaborative mode, it should be noted it had similar results in both game modes. However, in collaborative, there are pictures that performed worse.

The pictures that performed best in competitive mode are the ones in Figures B.8 and B.10. They both feature attractions that are either on or next to the sea, so participants could remember the sea because it is next to their spawn point. Other than that, there is no data pointing to what could make these pictures perform well in competitive mode. It should also be noted that there were more finds under 20 seconds in competitive mode than in collaborative mode. This could point to participants taking the points more seriously and trying to beat the opponent, as opposed to working together in collaborative mode.

Gaze data for dynamic objects with the highest average gaze time collaborative, competitive, and both game modes for the medieval village map can be seen in Table 5.3. When looking at both collaborative and competitive sessions for the medieval village map, the highest ranking dynamic object by the average gaze time is the inn close to the center of the map. Like the castle in the amusement park map, this building is tall and can be a useful navigation tool. However, the inn also has a large indoor space that can be explored, which in turn makes the participant gaze at it more if they explore it completely. The inn has an average gaze count of 56.40 and an average gaze time of 38.60 seconds. The object with the second highest average gaze time is a large house next to the inn, also containing an interior that can be explored. This building will be called the large house from here on out. The large house has an average gaze count of 19.40 and an average gaze time of 10.19 seconds. The third one is a house close to the starting area, with a part of it hanging over the road. This building will be called the overhead house. The overhead house has an average gaze count of 11.40 and an average gaze time of 3.93 seconds. The gaze heatmaps for these dynamic objects are seen in Figure 5.3. Dynamic objects with high average gaze time other than the mentioned top three include the small houses next to and behind the overhead house. Additionally, the watermill house, a larger house behind the overhead house, and a small house on the left part of the map also have high average gaze times. Just like in the amusement park map, most of these dynamic objects are tall buildings, which allow the participants to navigate easier. In addition, the two dynamic objects with the highest average gaze time have indoor spaces for participants to explore, adding to their gaze time.

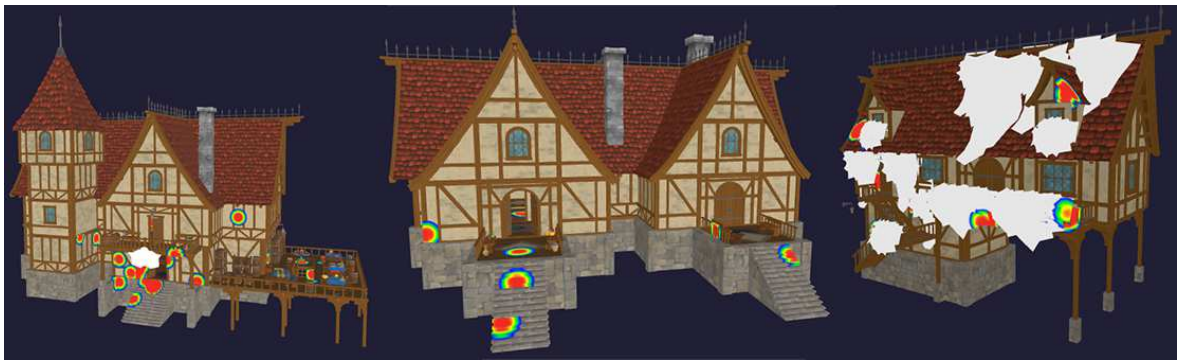

Figure 5.3: Gaze heatmaps of the inn (left), large house (middle), and overhead house (right) in the medieval village map in both game modes

If the dynamic objects are filtered to only gaze data in collaborative sessions, the inn is also the dynamic object with the highest average gaze time. The average gaze count for the inn is 83.50, while the average gaze time is 57.33. Just like when analyzing all sessions, the second dynamic object by the average gaze time is the large house next to the inn. It has an average gaze count of 38.00 and an average gaze time of 22.95 seconds. The third object with the highest average gaze time is the church at the far back of the map. The church, like the inn and the large house, also contains an interior. It has an average gaze count of 7.25 and an average gaze time of 3.30 seconds. Other dynamic objects with high average gaze time include the small houses next to and behind the overhead house, a larger house behind the overhead house, the warrior statue, and a small house on the left part of the map. It seems like when playing collaboratively, participants tended to look at the interiors of buildings and explore the far back part of the map more.

In competitive sessions, the inn also has the highest average gaze time, 26.12 seconds, and an average gaze count of 38.33. The overhead house is second, with an average gaze count of 15.83 and an average gaze time of 5.92 seconds. The third object with the highest average gaze time is a small house next to the overhead house, with an average gaze count of 17.50 and an average gaze time of 3.57 seconds. Dynamic objects which also have high average gaze time include the watermill house, a small house behind the overhead house, and a larger house behind the overhead house. In addition, a small house on the left part of the map, the large house, and a small house next to the far windmill also have high average gaze time. From this data, it seems that participants focused most of their time exploring the starting part of the map. As can be seen from the two buildings with the highest average gaze time, the bigger buildings caught their attention as well.

Table 5.3: Gaze tracking data for dynamic objects with highest average gaze time for each mode in the medieval village map

	Inn	**Large house**	**Overhead house**	**Church**	**Small house next to overhead house**
Both modes average gaze count	56.40	19.40	11.40	3.50	15.70
Both modes average gaze time (seconds)	38.60	10.19	3.93	1.48	3.12
Both modes gaze ratio (%)	100.00	90.00	100.00	40.00	80.00
Collaborative mode average gaze count	83.50	38.00	4.75	7.25	13.00
Collaborative mode average gaze time (seconds)	57.33	22.95	0.95	3.30	2.45
Collaborative mode gaze ratio (%)	100.00	100.00	100.00	75.00	50.00
Competitive mode average gaze count	38.33	7.00	15.83	1.00	17.50
Competitive mode average gaze time (seconds)	26.12	1.68	5.92	0.27	3.57
Competitive mode gaze ratio (%)	100.00	83.33	100.00	16.67	100.00

In both collaborative and competitive version of the game, participants were drawn to bigger, taller buildings. They also spent a lot of time exploring the interior of buildings. However, participants spent more time exploring interiors in collaborative sessions, both by the average gaze count and the average gaze time. Collaborative sessions also had the church in their top dynamic objects by average gaze time, while this was not the case with competitive sessions. The church also has an interior space to explore, although smaller than the inn and the large house. Additionally, in collaborative mode, it appears that participants spent more time exploring the back part of the map compared to the

competitive mode. The warrior and the statue are both at the back of the map, and they had high average gaze time in collaborative sessions, but not in competitive sessions. A reason for this could be that when playing collaborative, participants tend to split up to cover more of the map and only meet up near the end of the game or on accident. In competitive sessions, participants tended to have more gaze time on the objects near the middle of the map. This could be because they had to explore the whole map themselves and could not rely on the other participant for help. In turn, when playing collaboratively, participants could explore the outskirts of the map more.

In the medieval village map, the worst performing pictures for both game modes are the ones in Figures B.16, B.17, and B.20. The first of these contains both the inn and the large house, both of which had high average gaze time, but they are far away from the origin. Perhaps participants only viewed these buildings from up close, so they did not remember the part of the map where this picture was taken from. The second picture was taken between the inn and the large house, but they are not seen on the picture itself, so this could be the reason for its bad performance. The last picture was taken in the church, which is not among the higher rated dynamic objects. The church was also only looked at in only 40% of sessions, so a big reason why it performed so poorly is possibly because participants did not even know about it.

The pictures in Figures B.11, B.12, and B.18 performed the best among participants. In the first picture, a small house behind the overhead house can be seen, and this building had a high average gaze time. However, it is also likely players managed to find the origin of this picture due to its proximity to the starting area or by navigating using the windmill in the background. The second picture contains both the overhead house and the large house behind it, both boasting high average gaze times. This could have potentially helped participants find them faster, but this picture is also close to the spawn point. The last of these pictures is taken inside the inn, which had the highest average gaze time among all dynamic objects. Because of this, participants could have remembered it very well by looking at it for a long time.

In the collaborative sessions, participants had the most problems with the pictures seen in Figures B.17 and B.20. The reasoning for both are explained earlier, but it is surprising to see the latter perform so bad, as the church was the dynamic object with

the third highest average gaze time. However, many participants confused the inside of the church with the inside of the inn, so this could be an explanation. The picture in Figure B.16 performed better in collaborative than when looking at all sessions, and a reason for this could be that the inn was looked at more often in collaborative.

The pictures participants had the least trouble with are the ones in Figures B.11 and B.12. Possible reasons why for both of these are described before. It should be noted that, while the picture in Figure B.18 is not one of the best performing, every participant still managed to find it, so there was no problem with the inn building. Other pictures in the collaborative mode were simply found faster by the participants.

When looking at only the competitive sessions, the worst performing pictures are the ones seen in Figures B.16 and B.20. The reasons for both performing poorly are described before, and dynamic objects visible on the photos all have lower average gaze times in competitive mode than in collaborative. Additionally, it should be noted that the picture in Figure B.16 was missed by the same amount of people in both the collaborative and the competitive game modes.

The picture that performed best is the one in Figure B.11. This could be due to it being close to the starting area, but the overhead house had significantly higher average gaze time in competitive mode as opposed to the collaborative one. Participants may have found the origin of this picture easier due to looking at the overhead house visible on the picture for longer.

Key Findings

To summarize the findings brought by the gaze tracking data, players definitely seemed drawn to bigger and taller objects on both maps, as they probably helped them with navigation. The data concludes that participants seemed to have looked at objects with colder colors more when playing collaboratively as opposed to playing competitively. In competitive mode, participants tended to look more at warmer colored objects. However, more research needs to be done to determine whether this is a coincidence. Participants also have higher average gaze times for buildings that have interiors. While a reason for this could be that these building have a larger amount of space to look at, it could also be that these spaces intrigue participants, so they choose to spend more time in them. This

could especially be true for collaborative sessions, as in that game mode, buildings with interiors to explore had higher average gaze times as opposed to competitive sessions. It also appears that when playing collaboratively, participants tended to look more evenly around the whole map. On the other hand, in competitive mode, it appears participants focused their attention more on the beginning and middle of the map.

Additionally, almost all dynamic objects had higher average gaze times in collaborative mode. Finally, the analyzing of the gaze data was coupled with how well participants found the origins of the given pictures. It seems that participants had fewer problems finding where a picture was taken if they spent more time looking at objects featured on that picture. In other words, they would find the origin of a picture easier if objects present in the picture had high average gaze time. However, further research and more participants are required before concluding the results of the analyzed data.

5.2.4 Practical Implication of Test Results

If one wants to draw the attention of players of multiplayer VR games to certain objects, there appear to be some ways to do so. Firstly, players tend to be drawn to bigger and taller objects, so implementing those could potentially sway the player's gaze towards them. Players also seem to be drawn to exploring the interiors, so adding them to buildings could increase the time players spend looking at them. This could work even better in collaborative games. Additionally, participants tend to look at more parts of the map in collaborative games, so open-world environments could be a better choice for collaborative games, if the type of the game allows it. On the other hand, players tend to focus their gaze more on only the central part of the map in competitive games. Because of this, if the gaze of a player is important in a competitive game, a small map may prove better. Players in competitive games seem to look at warm colors more often than cold ones. Due to this, adding warmer colored objects may draw the player's attention more to those objects. Finally, if the point of a game is for players to remember something, the longer they look at it, the better they may remember it. Because of this, it could be important to draw the player's attention to whatever object or part of the map they might have to remember in the future.

6 Conclusion

In this thesis, the design and implementation of **OVRseer**, a networked multiplayer VR game for analyzing the players' gaze, is presented. The game contains two different maps, an amusement park and a medieval village. Players are instructed to explore the maps, then try to find certain places on them. The game is played in pairs, either collaboratively or competitively. The gaze data can then be analyzed with the use of the Cognitive3D web-application. Even though a few problems are present in the game and the gaze analyzing software contains some limitation, this game provides a potential base for analyzing gaze data in multiplayer VR games. Future improvements for the game should address these issues. Additionally, investigation of other VR gaze tracking software or the improvement of Cognitive3D could prove useful for future analysis.

The conducted user study provides insights into the gaze pattern differences of players when playing collaboratively or competitively. On the amusement park map, slight differences were found in the colors of objects players were looking at. On the other hand, on the medieval village map, players tended to look at different parts of the map when playing on differing game modes. However, larger and taller buildings were most gazed at by the majority of players, as those objects can be utilized for easier navigation. Additionally, players tended to more frequently find where a picture was taken if they have spent longer looking at the objects visible on the picture.

However, the user study suffers from a small sample size. With more participants, the results of this study could be analyzed even further in future research. In addition, more data could be gathered and then analyzed, both subjective and objective measures. Despite these limitations, the study highlights some differences in gaze patterns of players when playing together with or against another player. It also contains suggestions on how to utilize the gaze data results to create better VR games for players.

References

[1] Datagen. Eye Gaze Tracking: Applications, Techniques, and Key Metrics. Last accessed on 08/04/2024. [Online]. Available: https://datagen.tech/guides/face-recognition/eye-gaze-tracking/#

[2] Eyeware. Top 7 3D Eye Tracking Use Cases Applications. Last accessed on 08/04/2024. [Online]. Available: https://eyeware.tech/blog/top-7-use-cases-for-3d-eye-tracking/

[3] B. Farnsworth, "What is Eye Tracking and How Does it Work?" *iMotions*, last accessed on 31/05/2024. [Online]. Available: https://imotions.com/blog/learning/best-practice/eye-tracking-work/

[4] ——, "What is VR Eye Tracking? [And How Does it Work?]," *iMotions*, last accessed on 31/05/2024. [Online]. Available: https://imotions.com/blog/learning/best-practice/vr-eye-tracking/

[5] Merriam-Webster. Virtual Reality. Last accessed on 07/04/2024. [Online]. Available: https://www.merriam-webster.com/dictionary/virtual%20reality

[6] Virtual Reality Society. What is Virtual Reality? Last accessed on 07/04/2024. [Online]. Available: https://www.vrs.org.uk/virtual-reality/what-is-virtual-reality.html

[7] H. E. Lowood, "virtual reality," *Britannica*, last accessed on 07/04/2024. [Online]. Available: https://www.britannica.com/technology/virtual-reality

[8] G. I. Zheng J. M., Chan K. W., "Virtual reality," *Ieee Potentials*, vol. 17, no. 2, pp. 20–23, 1998.

[9] Iberdola. Virtual Reality: another world within sight. Last accessed on 07/04/2024. [Online]. Available: https://www.iberdrola.com/innovation/virtual-reality

[10] S. Thompson, "VR Applications: Key Industries already using Virtual Reality," *VirtualSpeech*, last accessed on 07/04/2024. [Online]. Available: https://virtualspeech.com/blog/vr-applications

[11] G. Beqiri, "Experiential Learning and Kolb's Learning Styles," *VirtualSpeech*, last accessed on 07/04/2024. [Online]. Available: https://virtualspeech.com/blog/experiential-learning-vr

[12] A. T. Duchowski, V. Shivashankaraiah, T. Rawls, A. K. Gramopadhye, B. J. Melloy, and B. Kanki, "Binocular eye tracking in virtual reality for inspection training," in *Proceedings of the 2000 Symposium on Eye Tracking Research & Applications*, ser. ETRA '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 89–96. https://doi.org/10.1145/355017.355031

[13] V. Clay, P. König, and S. Koenig, "Eye tracking in virtual reality," *Journal of eye movement research*, vol. 12, no. 1, 2019.

[14] Steam. Blink. Last accessed on 10/04/2024. [Online]. Available: https://store.steampowered.com/app/447210/Blink/

[15] ——. Before Your Eyes. Last accessed on 10/04/2024. [Online]. Available: https://store.steampowered.com/app/1082430/Before_Your_Eyes/

[16] I. Higton, "Synapse - a pure power fantasy that epitomises everything great about virtual reality," *Eurogamer*, last accessed on 10/04/2024. [Online]. Available: https://www.eurogamer.net/synapse-a-pure-power-fantasy-that-epitomises-everything-great-about-virtual-reality

[17] J. Erl, "PSVR 2 eye tracking makes VR horror even scarier," *Mixed News*, last accessed on 10/04/2024. [Online]. Available: https://mixed-news.com/en/psvr-2-eye-tracking-makes-vr-horror-even-scarier/

[18] A. K. Mutasim, W. Stuerzlinger, and A. U. Batmaz, "Gaze Tracking for Eye-Hand Coordination Training Systems in Virtual Reality," in *Extended Abstracts of the 2020*

CHI Conference on Human Factors in Computing Systems, 2020, pp. 1–9.

[19] Z. Hu, "Gaze Analysis and Prediction in Virtual Reality," in *2020 IEEE conference on virtual reality and 3d user interfaces abstracts and workshops (VRW)*. IEEE, 2020, pp. 543–544.

[20] A. Burova, J. Mäkelä, J. Hakulinen, T. Keskinen, H. Heinonen, S. Siltanen, and M. Turunen, "Utilizing VR and Gaze Tracking to Develop AR Solutions for Industrial Maintenance," in *Proceedings of the 2020 CHI conference on human factors in computing systems*, 2020, pp. 1–13.

[21] K. Qian, T. Arichi, A. Price, S. Dall'Orso, J. Eden, Y. Noh, K. Rhode, E. Burdet, M. Neil, A. D. Edwards, and J. V. Hajnal, "An eye tracking based virtual reality system for use inside magnetic resonance imaging systems," *Scientific Reports*, vol. 11, no. 16301, 2021.

[22] K. Holmqvist, M. Nyström, R. Andersson, R. Dewhurst, H. Jarodzka, and J. Van de Weijer, *Eye tracking: A comprehensive guide to methods and measures*. oup Oxford, 2011.

[23] H. J. Joo and H. Y. Jeong, "A study on eye-tracking-based Interface for VR/AR education platform," *Multimedia Tools and Applications*, vol. 79, no. 23, pp. 16 719–16 730, 2020.

[24] Z. Zhu and Q. Ji, "Novel Eye Gaze Tracking Techniques Under Natural Head Movement," *IEEE Transactions on biomedical engineering*, vol. 54, no. 12, pp. 2246–2260, 2007.

[25] S. L. Matthews, A. Uribe-Quevedo, and A. Theodorou, "Rendering Optimizations for Virtual Reality Using Eye-Tracking," in *2020 22nd symposium on virtual and augmented reality (SVR)*. IEEE, 2020, pp. 398–405.

[26] M. Cognolato, M. Atzori, and H. Müller, "Head-mounted eye gaze tracking devices: An overview of modern devices and recent advances," *Journal of rehabilitation and assistive technologies engineering*, vol. 5, p. 2055668318773991, 2018.

[27] Wikipedia. Unity (game engine). Last accessed on 31/05/2024. [Online]. Available: https://en.wikipedia.org/wiki/Unity_(game_engine)

[28] ——. C Sharp (programming language). Last accessed on 31/05/2024. [Online]. Available: https://en.wikipedia.org/wiki/C_Sharp_(programming_language)

[29] Meta Quest. Import Meta XR SDKs in Unity Package Manager. Last accessed on 31/05/2024. [Online]. Available: https://developer.oculus.com/documentation/unity/unity-package-manager/

[30] Photon Engine. Pun. Last accessed on 31/05/2024. [Online]. Available: https://www.photonengine.com/pun

[31] Cognitive3D. Cognitive3D. (2024) (SDK version 1.4.7). Last accessed on 06/02/2024. [Computer Software]. Available: https://cognitive3d.com/.

[32] Wikipedia. Microsoft Visual Studio. Last accessed on 31/05/2024. [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_Visual_Studio

[33] Unity. Asset Store. Last accessed on 31/05/2024. [Online]. Available: https://assetstore.unity.com/

[34] Unity Asset Store. Meta XR All-in-One SDK. Last accessed on 31/05/2024. [Online]. Available: https://assetstore.unity.com/packages/tools/integration/meta-xr-all-in-one-sdk-269657#description

[35] ——. PUN 2 - FREE. Last accessed on 31/05/2024. [Online]. Available: https://assetstore.unity.com/packages/tools/network/pun-2-free-119922

[36] ——. Amusement park 1. Last accessed on 31/05/2024. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/landscapes/amusement-park-1-235574

[37] ——. Dreamscape Village - Stylized Fantasy Open World. Last accessed on 31/05/2024. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/fantasy/dreamscape-village-stylized-fantasy-open-world-244797

[38] K. Cherry. Red Color Psychology. Last accessed on 21/06/2024. [Online]. Available: https://www.verywellmind.com/the-color-psychology-of-red-2795821

[39] ——. The Color Blue: Meaning and Color Psychology. Last accessed on 21/06/2024. [Online]. Available: https://www.verywellmind.com/the-color-psychology-of-blue-2795815

List of Figures

List of Tables

Abbreviations

AR Augmented reality

3D Three-dimensional

UI User Interface

VR Virtual reality

Appendix A: User Study Form

User study for Master's Thesis "Analysis of Gaze Tracking in Collaborative and Competitive Virtual Reality Games"

The information and responses gathered by this user data will be used solely for research reasons as part of the Master's Thesis "Analysis of Gaze Tracking in Collaborative and Competitive Virtual Reality Games". Because all data will be evaluated collectively, your personal information will be kept anonymous.

* Indicates required question

General questions

1. Age in years: *

_____

2. Gender: *

Mark only one oval.

- ◯ Woman
- ◯ Man
- ◯ Transgender woman
- ◯ Transgender man
- ◯ Prefer not to answer
- ◯ Other: _____

3. How many hours a week do you spend on video games? *

Mark only one oval.

- ◯ I don't play video games
- ◯ 5 or less hours a week
- ◯ 10 or less hours a week
- ◯ 20 or less hours a week
- ◯ 40 or less hours a week
- ◯ More than 40 hours a week

4. How many hours a week do you spend on virtual reality? *

Mark only one oval.

- ◯ I don't use virtual reality
- ◯ 5 or less hours a week
- ◯ 10 or less hours a week
- ◯ 20 or less hours a week
- ◯ 40 or less hours a week
- ◯ More than 40 hours a week

5. Please check all platforms that you have played video games on in the last year:

Check all that apply.

- ☐ PC
- ☐ Mobile
- ☐ Console
- ☐ Virtual Reality
- ☐ Handheld Consoles
- ☐ Other: _____

Playing the game

Stop here and wait for the examiner to explain the game.

6. (COLLABORATIVE) What map did you play on? *

Mark only one oval.

- ◯ Amusement Park
- ◯ Dreamscape Village

7. (COLLABORATIVE) How many points did you get? *

_____

8. (COMPETITIVE) What map did you play on? *

Mark only one oval.

- ◯ Amusement Park
- ◯ Dreamscape Village

9. (COMPETITIVE) How many points did you get? *

_____

10. (COMPETITIVE) Did you win? *

Mark only one oval.

- ◯ Yes
- ◯ No

Appendix B: Pictures Players Need to Find the Origin of

Figure B.1: The first picture in the amusement park map

Figure B.2: The second picture in the amusement park map

Figure B.3: The third picture in the amusement park map

Figure B.4: The fourth picture in the amusement park map

Figure B.5: The fifth picture in the amusement park map

Figure B.6: The sixth picture in the amusement park map

Figure B.7: The seventh picture in the amusement park map

Figure B.8: The eight picture in the amusement park map

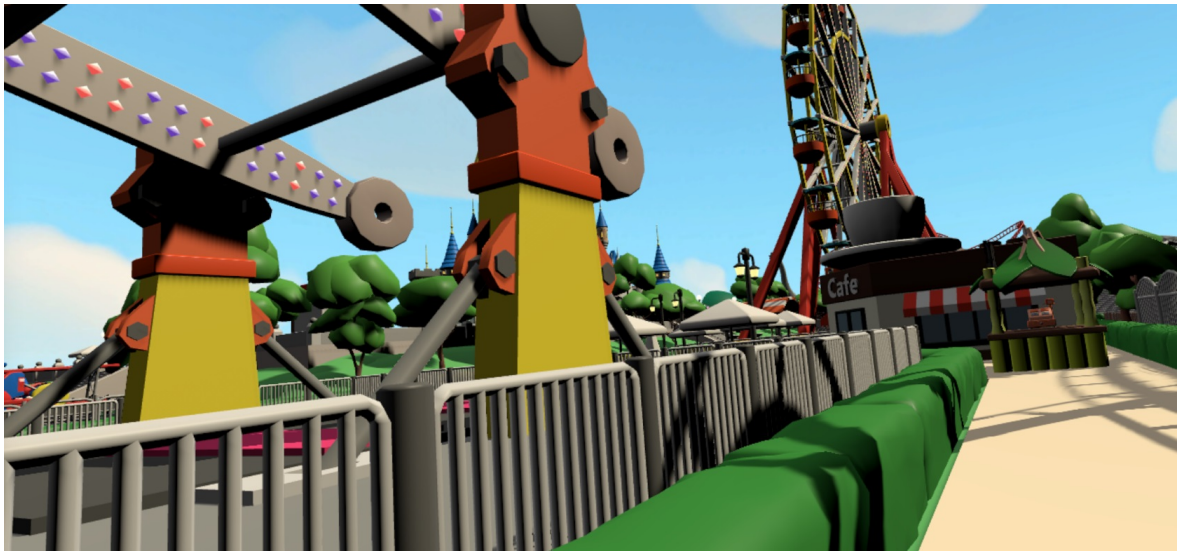

Figure B.9: The ninth picture in the amusement park map

Figure B.10: The tenth picture in the amusement park map

Figure B.11: The first picture in the medieval village map

Figure B.12: The second picture in the medieval village map

Figure B.13: The third picture in the medieval village map

Figure B.14: The fourth picture in the medieval village map

Figure B.15: The fifth picture in the medieval village map

Figure B.16: The sixth picture in the medieval village map

Figure B.17: The seventh picture in the medieval village map

Figure B.18: The eight picture in the medieval village map

Figure B.19: The ninth picture in the medieval village map

Figure B.20: The tenth picture in the medieval village map

Abstract

Analysis of Gaze Tracking in Collaborative and Competitive Virtual Reality Games

Emilia Haramina

This Master's thesis delves into the development of a VR game named **OVRseer** and presents a user study (N=16) to assess the difference in gaze patterns between collaborative and competitive game modes. Players have time to explore two different maps and remember as much of them as possible. Players gather points by returning to various places on the map. The amount of points received depends on their speed to find the required places. On one map, they play collaboratively, and on the other, competitively. When playing collaboratively, they score points and work together to gain as many points. In the competitive game mode, players gain points independently of each other, with their goal being to have more points than the other player. Afterward, the gaze tracking technology present in the game is utilized to analyze the difference in their gaze behavior between the two game modes.

Keywords: VR; Virtual Reality; Gaze Tracking; VR Games; Multiplayer; Collaborative; Competitive

Sažetak

Analiza praćenja pogleda u kolaborativnim i kompetitivnim VR igrama

Emilia Haramina

Ovaj diplomski rad bavi se razvojem VR igre **OVRseer** i predstavlja korisničku studiju (N=16) kako bi se procijenila razlika u uzorcima pogleda između kolaborativnog i kompetitivnog načina igre. Igrači imaju vremena istražiti dvije različite mape i zapamtiti ih što je bolje moguće. Zatim igrači skupljaju bodove vraćajući se na različita mjesta na mapi. Broj osvojenih bodova ovisi njihovoj brzini pronalaska traženih mjesta. Na jednoj mapi igraju kolaborativno, a na drugoj kompetitivno. Kada igraju kolaborativno, dobivaju bodove zajedno i surađuju kako bi skupili što više bodova. U kompetitivnoj verziji igre, igrači dobivaju bodove neovisno jedan o drugom, a cilj im je imati više bodova od drugog igrača. Nakon toga, tehnologija praćenja pogleda prisutna u igri koristi se za analizu razlike u njihovim uzorcima pogleda između dvije verzije igre.

Ključne riječi: VR; virtualna stvarnost; praćenje pogleda; VR igre; višekorisničko; kolaborativno; kompetitivno