Bachelor's thesis

Information and communication technology

2024

Arttu Niemi

Multiple users on shared MR environment

TURKU AMK
TURKU UNIVERSITY OF
APPLIED SCIENCES

Bachelor’s Thesis | Abstract

Turku University of Applied Sciences

Information and communication technology

2024 | 35 pages

Arttu Niemi

Multiple users on shared MR environment

The objective of this thesis was of mixed reality (MR) application for stand-alone virtual reality (VR) glasses. It needed to be able to support multiple users connecting to same environment and use their relative position. The relative position is important in this case as the application is supposed to be used in the same room with others, so the positions and orientations need to match between the real world and virtual world.

The project was developed in Unity. The networking for the project is done using Mirror networking, which is one of the most well know networking libraries for Unity. With Mirror, it is possible to build networking in client-server model where one of the clients also act as server, so there is no need for external server.

The synchronization of users’ positions is done using trilateration and orientations is done using trigonometry. This thesis goes through the mathematics for trilateration and trigonometry that was used, and their implementation in Unity.

The end product is that synchronization script that can be used in multiple projects. Part of the end product is example application using that method of synchronizing position and orientation.

Keywords:

Virtual reality, Mixed reality, Trilateration, Multiplayer

Opinnäytetyö (AMK) | Tiivistelmä

Turun ammattikorkeakoulu

Tieto- ja viestintätekniikka

2024 | 35 sivua

Arttu Niemi

Useita käyttäjiä jaetussa yhdistetyn todellisuuden ympäristössä

Tavoitteena oli kehittä yhdistetyn todellisuuden sovellus, jota voitiin käyttää itsenäisesti käytettävillä virtuaalitodellisuuslaseilla. Sovelluksen tuli tukea useiden käyttäjien yhdistämistä samaan ympäristöön ja käyttää heidän suhteellista sijaintiansa toisiinsa. Suhteellinen sijainti on tässä tapauksessa tärkeää, koska sovellusta on tarkoitus käyttää samassa huoneessa muiden kanssa, joten sijaintien ja suuntauksien on vastattava reaalimaailmassa ja virtuaalimaailmassa.

Projekti kehitettiin Unity-pelimoottorilla. Projektin verkkotoiminta toteutettiin Mirror-verkostointikehyksen avulla, joka on yksi tunnetuimmista Unityn verkostointikehyksistä. Mirrorin avulla on mahdollista rakentaa verkostointi asiakas-palvelin-mallilla, jossa yksi asiakkaista toimii myös palvelimena, joten ulkoista palvelinta ei tarvita.

Käyttäjien sijaintien synkronointi toteutettiin trilateraatio-menetelmällä ja katseiden suuntien synkronointi trigonometrian avulla.

Lopputuloksena saatiin synkronointiskripti, jota voidaan käyttää useissa projekteissa. Osana lopputulosta tehtiin esimerkkisovellus, joka käyttää kyseistä sijainnin ja suuntauksen synkronointimenetelmää.

Asiasanat:

virtuaalinen todellisuus, yhdistetty todellisuus, trilateraatio, moninpeli

Content

Equations

Figures

Pictures

List of abbreviations

Component	One C# script that can be attached to Game Objects
Game Object	Object that is used in Unity
MR	Mixed reality
P2P	Peer-to-Peer, networking architecture
RCP	Remote procedure call
Unity	The game engine used to create the project
URP	Universal render pipeline
VR	Virtual reality

1 Introduction

There is a growing need for new ways to teach skills required to do work, especially in more hands-on jobs, such as mechanics or nurses. The simulation for those kinds of jobs usually does not allow teachers to teach at the same time as students do the exercises to practice these skills, and instead all the teaching is done by the simulation application. A good example is automobile mechanics whose only effective study method is by repairing real cars with the teacher next to providing instructions and fixing real cars can be expensive, so it would be more cost effective if these exercises could be done in virtual space instead.

To address this problem, the objective of this thesis was to develop new application where multiple users can connect to same environment, so there can be teacher overseeing and explaining things as students can see and do practices on their own with virtual objects instead of physical ones. The application was required to run on Meta Quest 3 glasses and use Mirror networking solution. The application was required to have mixed reality (MR) capabilities that those Meta Quest glasses support. Because of that MR, there needed to be a way to synchronize the users' real-life position to their relative position in MR environment. This was because in MR it is important that the users are in same position as their avatar.

The theoretical portion includes the description of coordinate system. The portion also explains matrix equations required for trilateration and trigonometric equations, that was used for synchronization of users' position and rotation.

The practical portion details the implementation of the mathematics in the application. The portion also details the implementation of networking using required networking solution and MR functionality.

The end product of this thesis is an application that can run on Meta Quest 3 glasses in stand-alone mode and have capabilities of connecting multiple of them together and run different scenarios on shared MR environments.

2 Mathematics for location and orientation synchronization

This chapter will introduce the mathematical principles that make the synchronization of users possible.

2.1 Multilateration for position synchronization

Multilateration is a method used to calculate relative position of an object based on its distance from multiple reference points, which locations are known. Most well know of these is trilateration, which uses three of the reference points. Trilateration can be expressed as finding intersection point of three spheres, each centered on a reference point with radius equal to the distance to the object (Thomas et al. 2005).

Examples for trilateration can be found in robotics, kinematics, aeronautics, crystallography, and computer graphics (Thomas et al. 2005). One of the most well-known use case for trilateration is the global positioning system (GPS), where trilateration is used to determine GPS device's location in relation to three satellites.

2.1.1 Relevant equations

The problem in trilateration is finding solution for three quadratic equations. Equation 1 shows the equations that needs to be solved, where $P_i = (x_i, y_i), i = 1, 2, 3$ are the coordinates of the reference point i, and d_i is the distance between the reference point i and the tracked object.

$$\begin{cases} (x - x_1)^2 + (y - y_1)^2 = d_1^2 \\ (x - x_2)^2 + (y - y_2)^2 = d_2^2 \\ (x - x_3)^2 + (y - y_3)^2 = d_3^2 \end{cases}$$

Equation 1. Quadratic equations for trilateration (Modified from Thomas et al. 2005)

2.1.2 Trilateration solution using matrices

In this thesis trilateration was used to solve 2D position of player, so only the x and y positions of reference points were used, and the result has x and y position. Equation 2 shows the algebraic method of solving the three quadratic equations.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2x_1 - 2x_3 & 2y_1 - 2y_3 \\ 2x_2 - 2x_3 & 2y_2 - 2y_3 \end{bmatrix}^{-1} \times \begin{bmatrix} x_1^2 - x_3^2 + y_1^2 - y_3^2 + d_3^2 - d_1^2 \\ x_2^2 - x_3^2 + y_2^2 - y_3^2 + d_3^2 - d_2^2 \end{bmatrix}$$

Equation 2. Algebraic equation to solve trilateration (Modified from Doukhnitch et al. 2008)

These results can be unified between users, when using arbitrary coordinate system instead of Unity's coordinates. In this project the arbitrary coordinates are placed so that one of the reference points is placed at (0, 0) colored red, another one along positive x-axis (+x, 0) colored green, and last one on positive y (±x, +y) colored blue. This way the position of user's character can be placed correctly in relation to reference points. Figure 1 shows how the reference points are placed on that arbitrary coordinate system and the user's relative position.

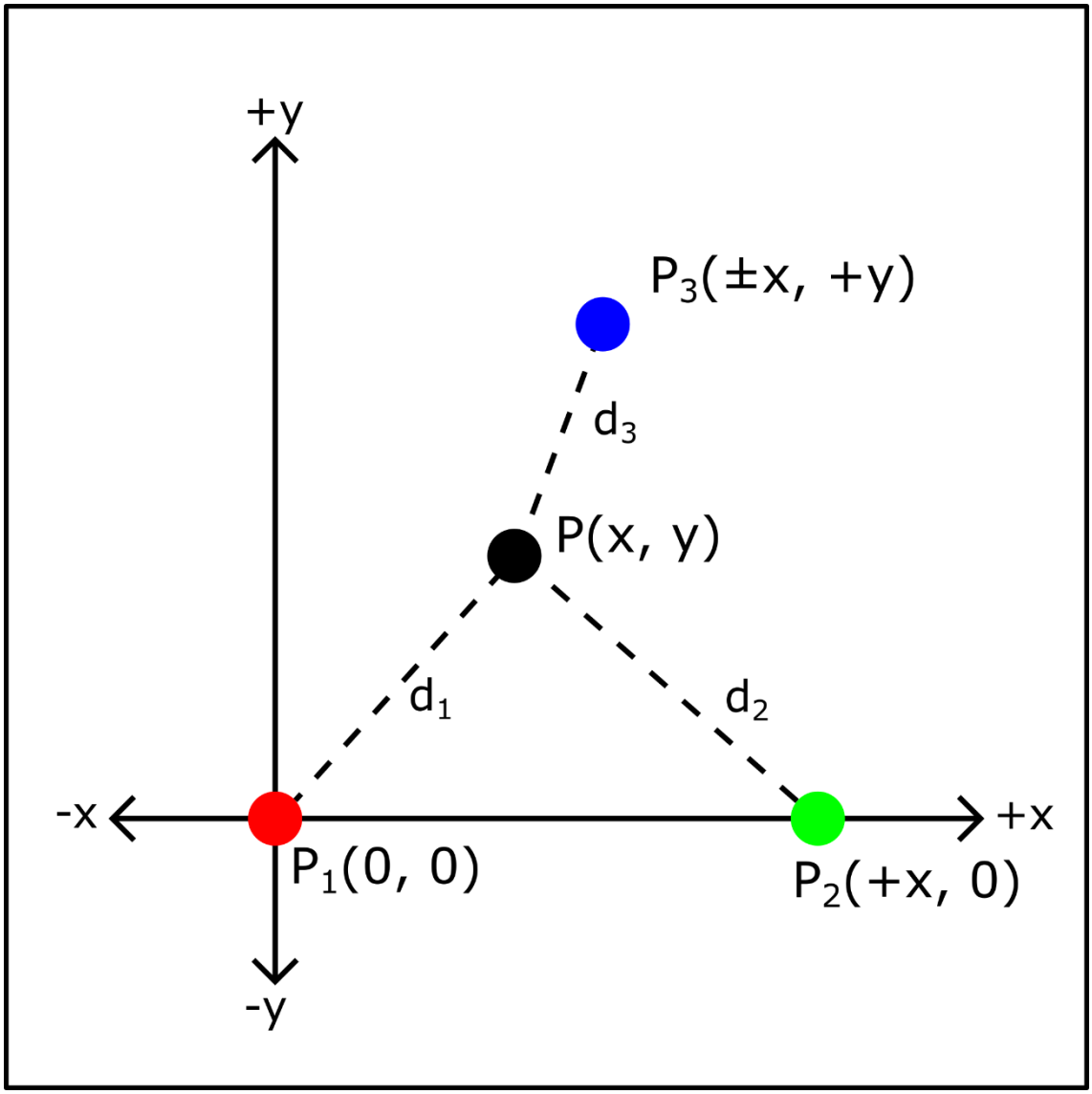


Figure 1. Relation of user's and reference points' location

2.2 Trigonometry for users' orientation synchronization

The user character's orientation in relation to arbitrary coordinate system can be calculated using trigonometry. Equation 3 shows how this can be solved with inverse sine function, where z_i is the z-coordinate of reference point i and d is the distance between them. The z-coordinate uses Unity's coordinates instead of the arbitrary coordinate system's coordinates.

$$\theta = \text{Asin}(\frac{z_2 - z_1}{d})$$

Equation 3. Trigonometric equation to solve user's orientation

Theta θ has a value between –90° and 90°. To get full 360° of rotation it needs to be differentiated if the user is facing towards positive or negative x-axis. This can be done by calculating whether $x_2 - x_1$ is positive or negative. If the result is negative, θ needs to be subtracted from 180° to give full rotation, from –90° to 270°.

3 Creating the project in Unity

Game engines are software frameworks that provide a comprehensive set of tools and technologies for game developers to create interactive experiences. They handle core functionalities like rendering graphics, simulating physics, managing audio, and providing input handling. Game engines can offer simple interfaces for adding customized scripts or objects, or they can come without this interface but still handle the more complex tasks of game development.

One of these game engines is called Unity, which was published in 2005 by Unity Technologies (Unity Technologies). Unity offers simple ways to develop 2D and 3D games using their primary scripting API, which is used with C# using Mono (Unity Technologies). Mono was a software framework designed to run .NET Framework software on Linux-based operating systems. Later it was extended to also run on other kinds of operating systems like Unix-based or embedded systems (The Mono Project). Unity also offers Unity Asset Store where users can share or sell their assets for development. These assets include 3D models and tools to extend Unity's functionality among other things (Unity Asset Store).

Unity's core development revolves around "Game Objects." These are the building blocks of any game made with Unity. Game Objects can be anything that is in the scene like environment, the computer-controlled characters, or the player character. Game Objects can have various components attached to it, such as mesh rendering for rendering the 3D object's shape, colliders for physics interactions, or material for visual appearance. These components are created using C# programming language, and Unity gives you the possibility to write your own components to make the Game Objects do something that is not provided by built-in components. Game Objects are placed in scenes that are then loaded one at a time. ("Game Objects")

I decided to use Unity for two reasons. The first reason was the multi-platform support that Mono offers. This was required because the program had to run on Meta Quest 3 glasses which uses an android-based operating system. Another

reason was that a before mentioned asset store, where I could get required addons for networking and XR development.

3.1 Adding the extended reality capabilities

3.1.1 What is extended reality

Extended reality (XR) is umbrella term that encapsulates virtual reality (VR), augmented reality (AR) and mixed reality (MR) under it. These technologies involve the integration of virtual elements into the physical world, but at varying degrees. (Milgram and Kishino)

Virtual reality

Virtual reality works by creating virtual environments where players can interact with objects and the objects interact with the environment. Advantages of VR include maximum immersion. The downside is that users get detached from physical reality.

Augmented reality

Augmented reality works by augmenting physical reality. This can be done for example by adding 3D objects to the physical environment that can be viewed by camera feed. These objects cannot interact with the physical environment but can interact with user input. The advantage of AR is that users always have some attachment to physical reality. The downside is that immersion is lacking because virtual and physical objects cannot interact with each other.

Mixed reality

Mixed reality is mixture of VR and AR, which tries to take advantage of both of those technologies. Main feature of MR is that it offers more immersive experience than AR but also attachment to physical reality that the VR does not offer. This is achieved by making the virtual objects interact with physical objects, for example the object disappearing if it is supposed to be behind corner.

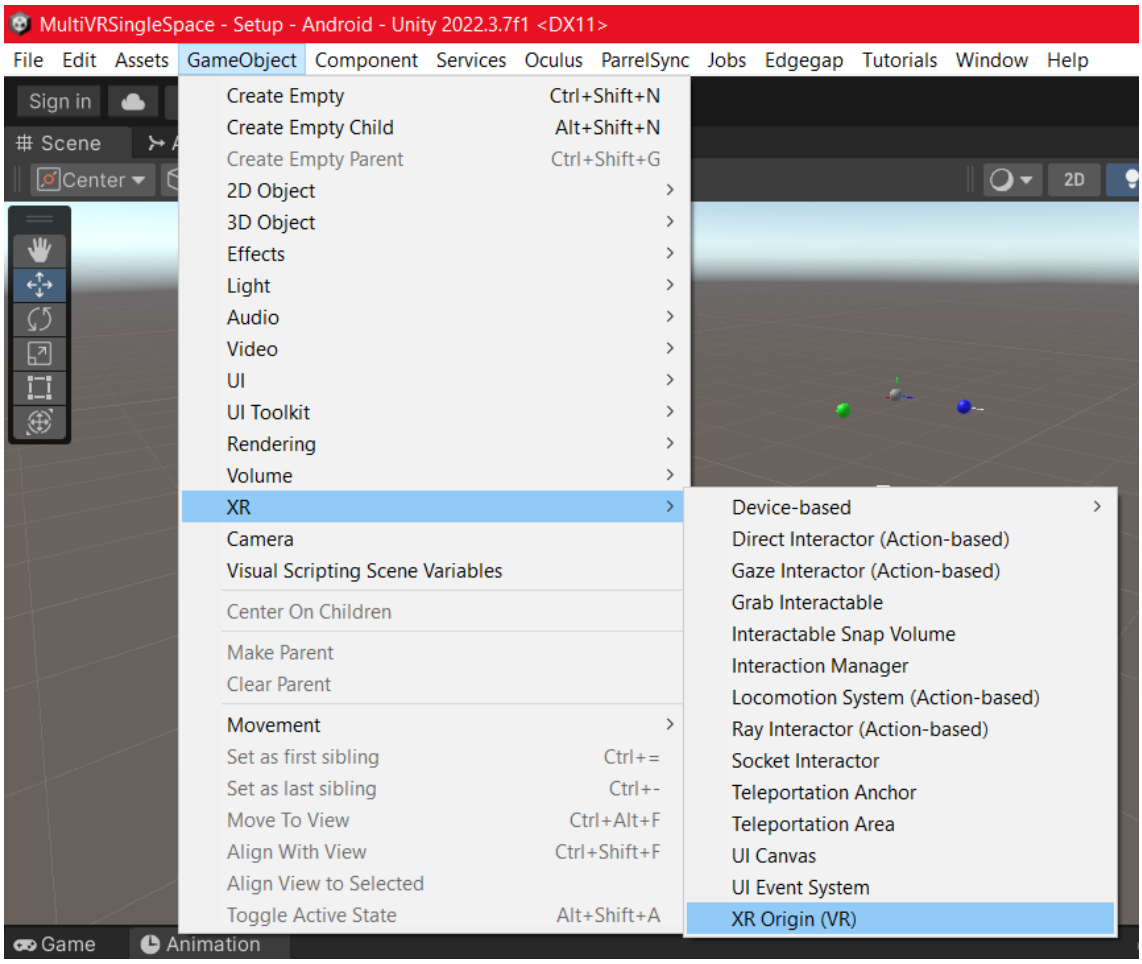
3.1.2 Adding VR controls using OpenXR

The original scope of the project was to be a VR application, but the scope was later expanded to also include MR. In Unity VR development, there are generally two approaches for XR libraries that can be used. One of these choices is Meta's XR SDK, which is specifically designed for Meta's own VR glasses. The other choice is the OpenXR library, which is designed to support as many different VR products as possible and not just Meta's VR glasses. (Unity Technologies, "XR Interaction Subsystems")

I decided to use OpenXR as it offers support for more VR glasses, which generally enhances user accessibility. Had the MR requirement been known from the start, I would have chosen Meta's XR SDK, as OpenXR does not have the same level of access to Quest glasses cameras, which is required for MR.

The development began by creating a Unity project with a VR template. The template automatically includes the required Unity packages for VR development. The template also has custom project settings that are better suited for VR applications. These changes include using the Universal Render Pipeline (URP) instead of Unity's standard one, which is the recommended choice for VR applications. Also included in the template are custom configurations for either low-power devices or standard ones.

When the project using the VR template was created, the first thing was to add a VR rig to the scene. A VR rig is Unity's Game Object that contains the required components for VR interaction. That rig's job is to handle the correct camera placement in the scene relative to users' movement and allow users to interact with the Game Objects placed in the scene. This rig can be added from GameObject > XR > XR Origin (VR), which is shown in Picture 1.



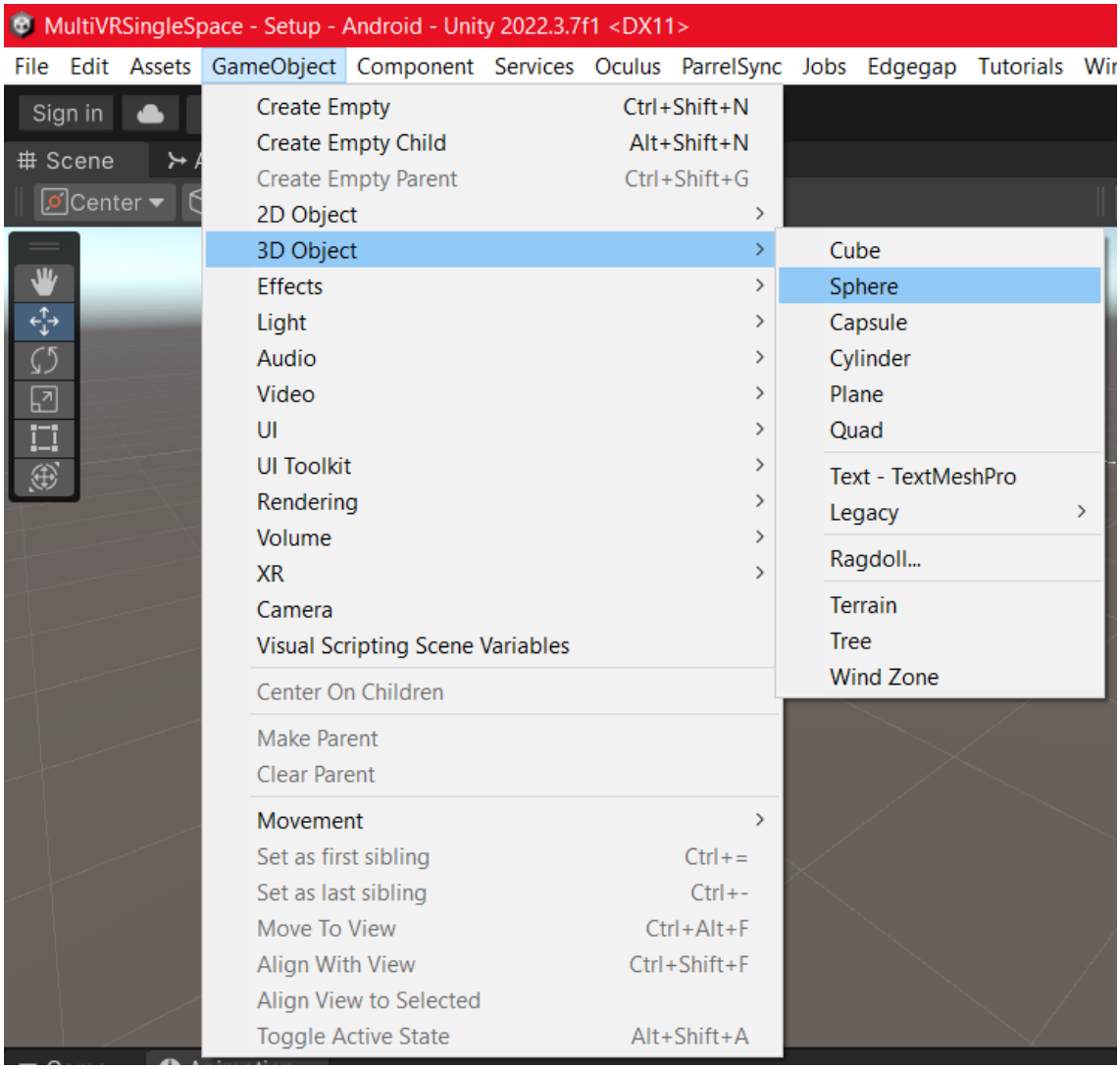
Picture 1. Dropdown menu for adding VR rig

After the rig is created, the action references need to be added. Those action references are added to Game Objects for both hands, which are found as child objects of "Camera Offset," which is a child object of "XR Origin (XR Rig)." Both hand objects, named "Left Controller" and "Right Controller," have a component named "XR Controller (Action-Based)," which has reference fields for position and rotation references, and input references for every input. The references

can be added by checking the reference box and then dragging the correct input action reference to the reference field or finding it from the list when pressing the right side of the field. These references are part of the “Input Action Asset,” which is included in the VR template. Each reference is a reference to a specific button or joystick axis on the controller. These references are required for the script to know when the physical input was provided.

Next the interaction manager needs to be created. The manager's job is to act as an intermediary between the interactors and interactables. Interactors, for example, can be the VR rig, whereas the interactables are the objects that those interactors interact with, like a Game Object that a user wants to grab and move around. There is no premade Game Object for an interaction manager, but it can be added by first creating an empty Game Object and then adding an “XR Interaction Manager” component to it. (Unity Technologies, “Setting up XR Interaction”)

The last thing to add is interactable objects. This is done by creating a Game Object with “Mesh,” “Mesh Renderer,” and “<3D shape or mesh> Collider” components, which can be found in GameObject > 3D > <Desired Shape>. For example, when creating a sphere object, it has a Sphere (Mesh) component and a Sphere Collider component. Then the components “Rigidbody” and “XR Grab Interactable” need to be added to the Game Object. Picture 2 shows the dropdown menu for creating sphere Game Object. Next, the Interaction Manager needs to be added to the XR Grab Interactable component. (Unity Technologies, “Making objects grabbable”)



Picture 2. Dropdown menu for creating sphere Game Objects

3.1.3 Adding MR support using Meta XR SDK

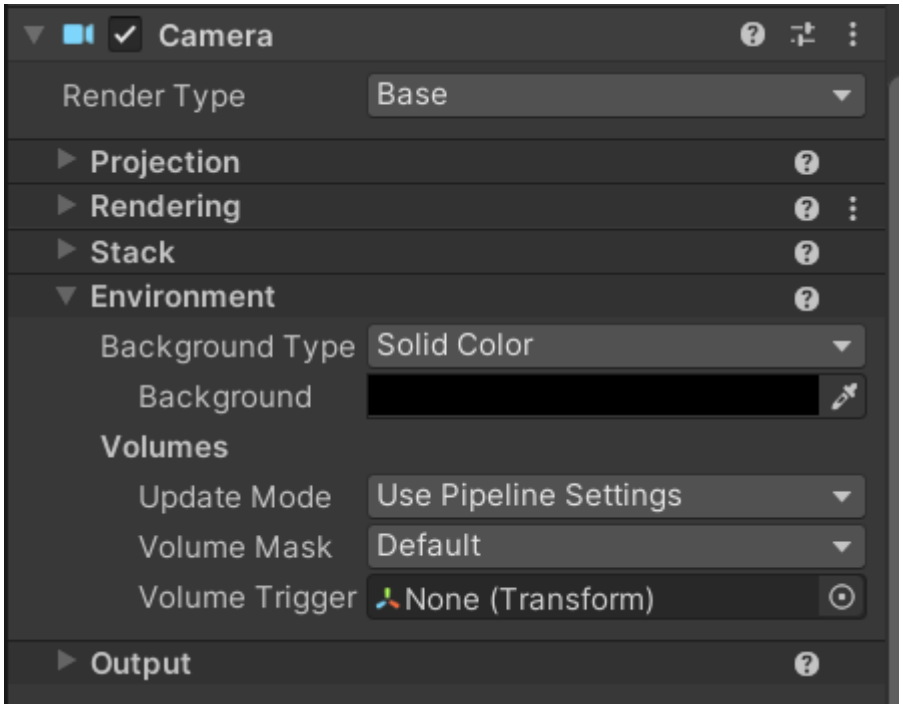
Meta XR SDK allows the use of Passthrough, a feature of Meta Quest glasses that shows the camera feed within the headset. Passthrough can replace the skybox with the camera feed, but that feed is not otherwise directly accessible. It can be used concurrently with OpenXR. (Meta, “Passthrough”)

To use the passthrough in Unity with OpenXR, the first thing to do is create an empty Game Object and then add “OVR Passthrough Layer” and “OVR Manager” components to it. After that is done, a few settings on the OVR

Manager must be changed. The first settings to change are in the Tracking section of settings, Tracking Origin Type to “Floor level” as otherwise the VR rig’s camera gets placed in wrong place when the application is started, and unchecking the Allow Recenter and Late Controller Update checkboxes because they can mess with VR rig’s or it’s controller’s position or rotation. Use Position Tracking and Use IPD Position Tracking needs to stay checked as otherwise the camera will not track users head movements. These settings are shown in Picture 3.

Picture 3. Settings for OVR Manager to make it work with OpenXR

Then Enable Passthrough checkbox should be marked, so the Meta glasses can use the passthrough feature. Lastly the Background should be changed to solid color and the color to black. The background setting can be found in Main Camera’s camera-component under “Environment”, shown in the Picture 4. This passthrough Game Object is needed in each scene that uses passthrough.



Picture 4. Camera setting for changing the background to solid black

3.2 Adding the networking using Mirror

3.2.1 What is Mirror

There are two commonly used network architectures in game development. The networking can be either done with client-server architecture or peer-to-peer architecture (P2P).

Client-server architecture is based on a model where there is single centralized server, and multiple clients connect to that server. The advantage of using client-server architecture is that it can be limited to how much client application can affect its users' data. Example of this is that if the client-server networking can be setup so that each client calculates user character's movement and just reports the new position to server or the client can only report the user inputs and server does the moving and reports back to client the new position. In this example, someone could take advantage of the looser security on that data and move their character illegally on client side and the new position just gets

reported to server. There is also a real security advantage as all traffic goes through the central server, so it limits the possibility of attacking others connected to the same multiplayer.

P2P instead is based on model where every user application handles their own user and then report the changes to other users. There is no centralized server which lowers the running costs of networking but allows malicious users to target other users as they are directly connected to each other.

Mirror networking library is an Unity package that helps adding multiplayer functionality to Unity projects. Mirror was created to be an open-source alternative to Unity's own UNET, which got discontinued because the codebase was massive and had many bugs. Mirror uses code that is based on original UNET but has then modified to have better performance and fix the bugs that the original UNET had. Mirror provides tools for multiplayer development that are easy to use, such as SyncVars which are variables that are automatically serialized or remote procedure calls (RPC) which allows function execution on other machines. Mirror is primarily designed around client-server architecture but has also a level of flexibility to implement aspects of P2P. Mirror also offers implementation of host server, which is instance of application that is same time client and server. This way there is no need for an external server and the device running the client can also run the server. (Mirror Networking)

3.2.2 Adding networking to project

I used Mirror networking at the commissioner's request. I had no prior experience using Mirror, but I had completed projects using different networking solutions.

After installing the Mirror networking Unity package, the first step was to create a Game Object to function as the network manager. As there is no pre-made Game Object for this purpose, an empty Game Object was created, and the "Network Manager" and "Kcp Transport" components were added. The Network Manager manages the application's networking aspects and communicates with

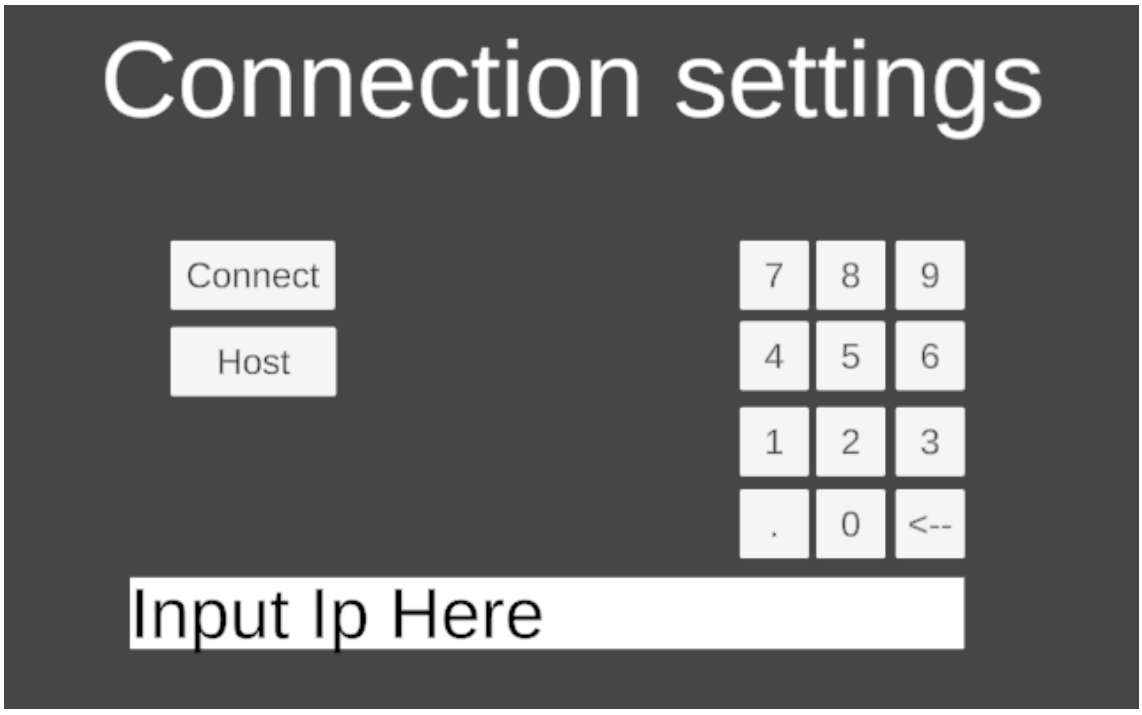
the transport protocol. KCP transport is Mirror's default protocol, although alternatives can be used. After creating the Game Object, the chosen transport must be assigned to the Network Manager component's transport field.

The next step was to create a new Scene for online use. This scene is automatically loaded when the server starts and should be the initial scene users join. Upon creation, this scene is added to the Network Manager component's online scene field. Concurrently, the offline scene should also be added to the Network Manager component. The offline scene, while less critical than the online scene, is simply the scene loaded when a user disconnects from the server.

After adding the scenes, a user character for network use was created. This character is spawned in the scene when a user connects to the server and is assigned to that user. The user character is a prefab Game Object that the Network Manager can then spawn in the scene. In this project, the user character is a modified VR rig incorporating networking components. The "Network Identity" and "Network Transform (Reliable)" components were added to the VR rig Game Object. The Network Identity component manages the Game Object's network identity and provides a globally unique identifier for that object across connected clients. It also manages the authority over the Game Object's state, determining whether the client or server controls it. While Network Identity initially assigns authority to the server, it can be granted to a specific client. This is typically done for Game Objects controlled by a client, such as the user character. The Network Transform component synchronizes the Game Object's transform across connected clients. While this synchronization is performed by the server by default, it can be configured to use client-side values. In VR applications, allowing the client to control the VR rig and report state changes to the server is generally preferable, achieved by changing the Network Transform's sync direction to "Client To Server." The same Network Transform component, with the same "Client To Server" sync direction, was also added to the VR rig's child objects: "Main Camera," "LeftHand Controller," and "RightHand Controller." After these modifications, the

Game Object was saved as a prefab and added to the Network Manager's player prefab field.

A method for users to start a server or connect to one was then implemented using a world-space Unity Canvas accessible via VR controls. The canvas contains buttons for connecting to a server, starting a server, and a numpad for entering the server's IP address, shown in Picture 5. The "StartHost()" method of the Network Manager is called by the server start button, while the connect button first sets the Network Manager's "networkAddress" variable to the server's IP address (provided by the user) and then calls the Network Manager's "StartClient()" method. The canvas buttons have "Box Collider" and "XR Simple Interactable" components. The desired method was then added to the XR Simple Interactable component's Interactable Events' Select Interactable list to trigger the method upon interaction. For a method to be added to this component, it must be public.



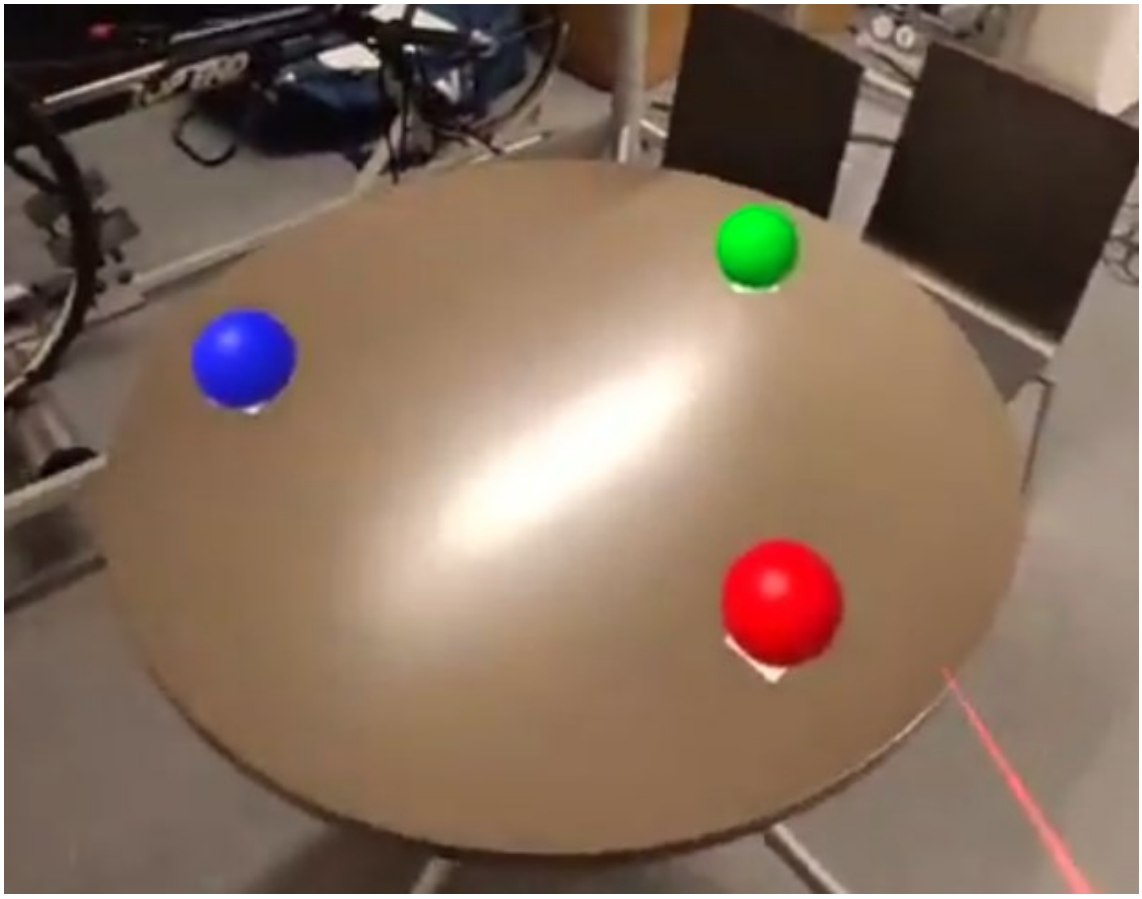
Picture 5. Canvas for users to connect to the server or start one

Finally, specific components of non-local user VR rigs were disabled to prevent input conflicts. This was accomplished using a script that disables the "XR

Origin” component in the VR rig Game Object and the “Camera,” “Audio Listener,” and “Tracked Pose Driver” components in the “Main Camera” child Game Object for each VR rig not controlled by the local user within the “Start” method (a default Unity method called once when the Game Object is instantiated).

3.3 How the position and rotation of users are synchronized

To implement trilateration, a custom script was created. This required three reference points to track an object’s position. These points were implemented as interactable spheres, as described in Chapter 3.1.2. To fix their positions after user placement, the “IsKinematic” checkbox in the Rigidbody component of these Game Objects was enabled. Kinematic Rigidbody objects are unaffected by Unity’s physics engine, meaning they are not subject to gravity or interactions with other physics objects. The “Throw On Detach” checkbox in the XR Grab Interactable component was also disabled. “Throw On Detach” allows VR users to throw interactable objects by swinging and releasing them. Enabling this with a kinematic Rigidbody would cause an error, as the throwing action attempts to apply force to a kinematic object, which is not permitted. These Game Objects were created as spheres with a 0.1-unit diameter. These Game Objects can be placed at predetermined locations in physical space, ensuring consistent placement across all users. These Game Objects and their placement inside the application is shown in Picture 6, the white things under the spheres are physical markers so every user can place the trackers in the same position. To maximize synchronization accuracy, these reference points should be positioned as far apart as possible to minimize relative error.



Picture 6. Game Objects used as reference points inside the application

An empty Game Object with a custom script was created to persist across scene loads and move to the correct position when a user connects to or starts a server. Using Game Objects that persist across scene loads is a simple method for transferring data during scene transitions. This persistence was implemented by calling the “DontDestroyOnLoad(gameObject)” method within the script’s Start method. This method moves the specified Game Object from the currently loaded scene to a scene that is never unloaded.

The script requires a public synchronization method that can be called by other scripts. This method first applies custom coordinates to each reference point to ensure consistent conventions across all clients. The first point is assigned the coordinates (0,0). The second point is placed on the positive x-axis at the same distance from the first point as they are in the scene. The coordinates of the third point are then calculated using trigonometry, based on the coordinates of

the first two points and their distances to the third point. This calculation yields two solutions, one of which can be discarded as it will have a negative y-value, assuming the reference object is placed correctly (resulting in a positive y-value).

After calculating the custom coordinates, the correct pose of the VR rig was calculated using Equation 2 and Equation 3 (no source provided for equations, assuming internal to the project). A pose in Unity is a variable that includes a 3D vector for position and a quaternion for rotation. Math.NET Numerics library was used to perform the matrix multiplication in Equation 2 using the custom coordinates and distances from the VR rig. Equation 3 was implemented directly in code. The resulting pose was then applied to the Game Object's transform, positioning it where the client's VR rig should be in the online scene. While not fully optimized, the code's performance is sufficient, as the synchronized pose is calculated only once, not continuously updated.

The script in the online version of the VR rig was then modified to move to the correct pose upon loading. This was achieved by applying the synchronization object's pose to the VR rig's transform within the script's Start method. This application is limited to the local VR rig using the boolean "isLocalPlayer."

4 Creating demo application to be used as teaching aid

This chapter details the development of a demonstration application using the technology of synchronizing users' position and rotation in shared VR or MR environments. The application demonstrates how this technology can assist in teaching. It is designed as an online questionnaire where users view a short 3D animation about administering first aid and then answer questions about the animation. The animations illustrate the steps of cardiopulmonary resuscitation (CPR), followed by related questions. Such applications allow users to explore the nuances of practical skills independently, while enabling instructors to easily highlight these nuances, all at a reduced cost due to the elimination of physical tools.

4.1 Creating synchronized canvas for questionnaire

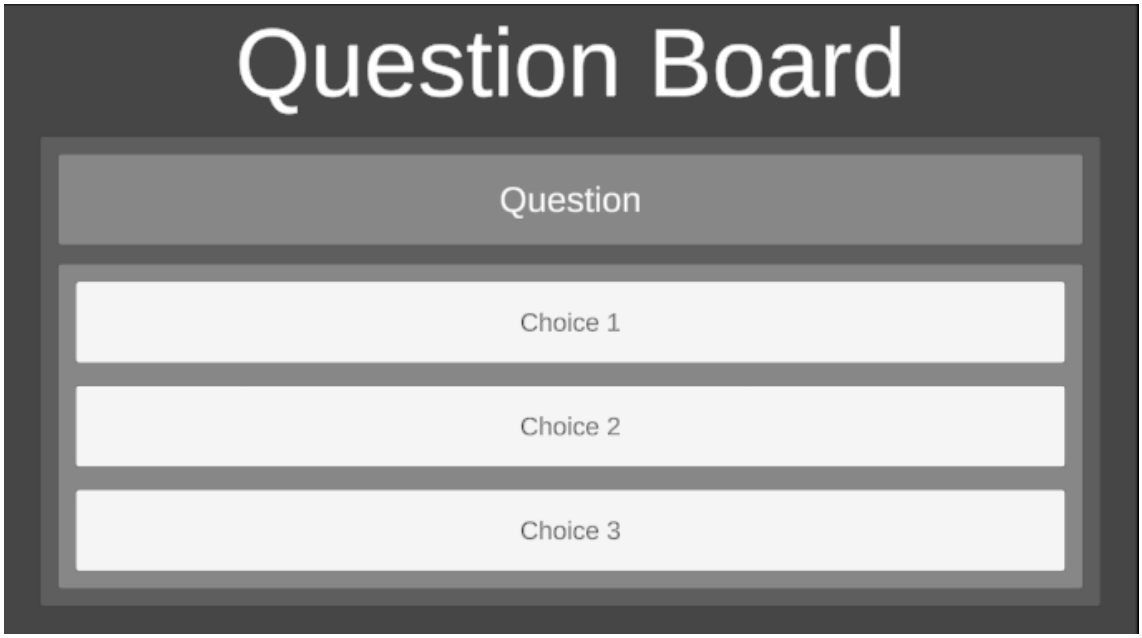
A canvas was created to display and answer the questionnaire. The questions were designed as multiple-choice, with results showing the number of selections for each choice and the correct answer after all responses were submitted.

To synchronize answers across the network, remote procedure calls (RPCs) were implemented to register user answers on the server. RPCs are created using standard C# method syntax with the addition of the "[Command]" or "[ClientRpc]" attribute before the method's return type. Commands execute code on the server, initiated by a client, while ClientRpcs execute code on clients, initiated by either the server or another client. Mirror requires specific prefixes for RPC method names: "CMD..." for Commands and "RPC..." for ClientRpcs. Commands, by default, require authorization, although this can be disabled to allow unauthorized server-side code execution. While generally not recommended, this is useful when all accessing sources are client applications. Unauthorized access is granted by adding "(requiresAuthority = false)" after the command attribute within the square brackets.

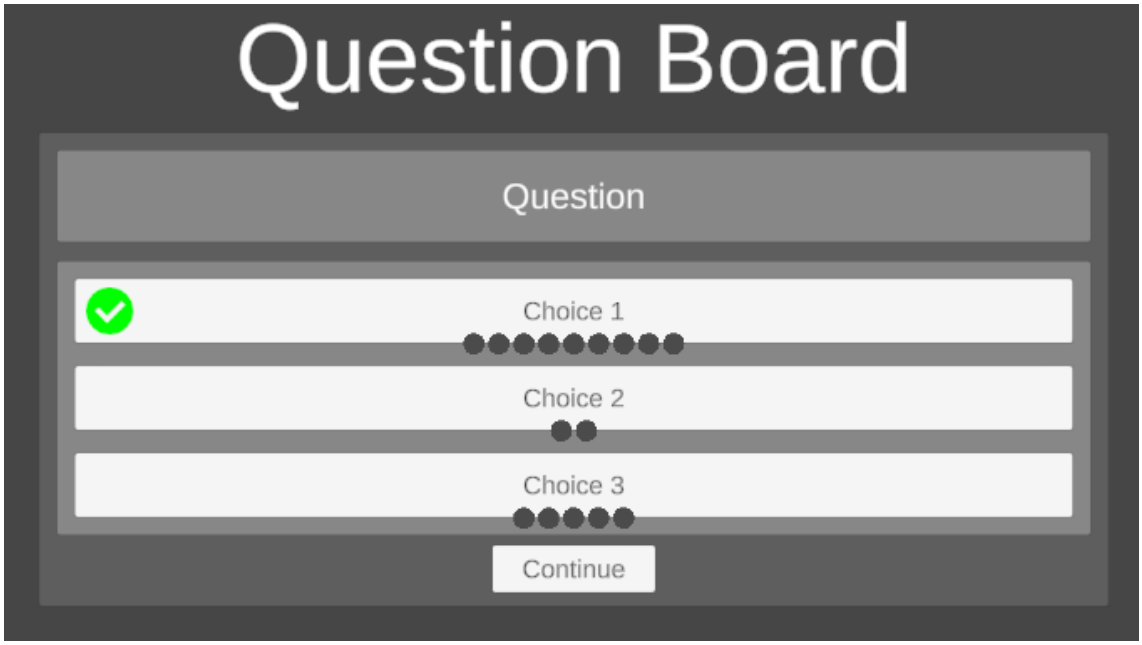
A struct was created to represent questions, containing the question text, all answer choices, the ID of the correct answer, and a list to store all submitted answers. Structs are used to create custom data types similar to classes but with some key differences. Structs cannot inherit from other structs or classes in the same way classes can. Additionally, classes are references to objects in memory, whereas structs store data directly in the variable's memory location. This makes structs a better choice for data types that remain relatively unchanged during runtime. To enable synchronization, the struct was made serializable by adding the "[Serializable]" attribute before its declaration.

The next step involved creating the various canvas screens: a starting screen (shown in Picture 7), a question screen (shown in Picture 8), and an answer display screen (shown in Picture 9). The question and answer display screens were designed generically, with text dynamically populated by code, eliminating the need for separate screens for each question.

Picture 7. Canvas look for starting the first animation



Picture 8. Canvas look for answering the question



Picture 9. Canvas look for showing the all the answered and correct one

After creating the necessary screens, code was written to manage transitions between them. The host client controls all stage transitions using a button displayed only to the host, except for the transition to the question screen after the animation finishes, which is shown to all clients. The button's visibility is

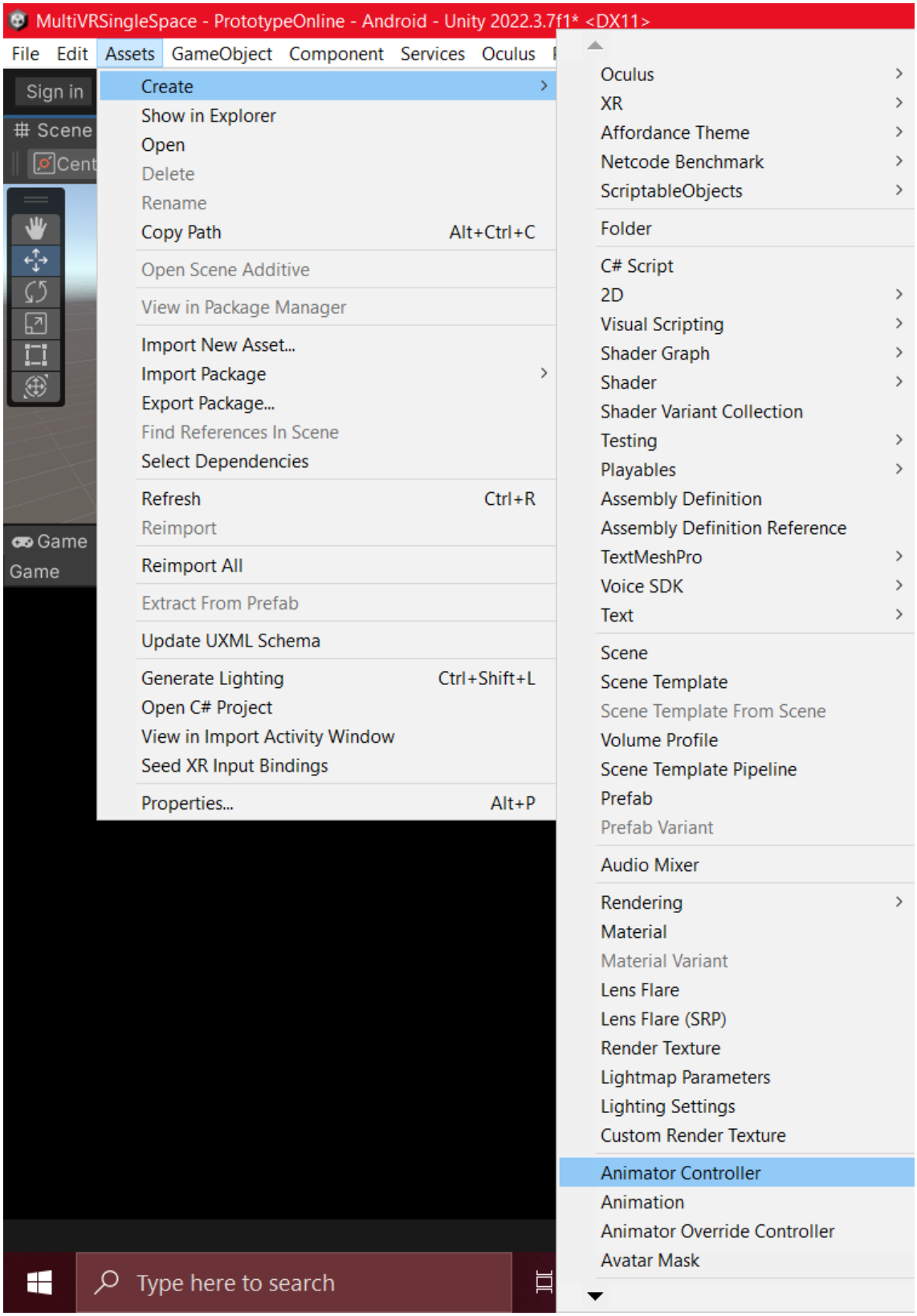
managed using a server-side method, while the display of other elements is handled client-side.

To register each answer on the server, the XR Simple Interaction component of the answer buttons first calls a public method on the local client, which then invokes a server RPC to add the submitted answer to the answer list.

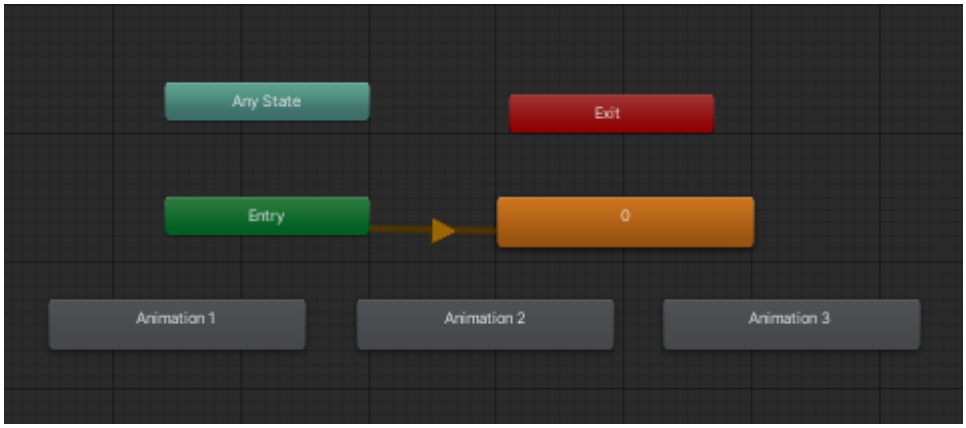
4.2 Adding synchronized animations

3D models and animations are not made by me. They are instead acquired from other source.

First thing to do after animations were imported, was to create Animation controller to handle the animations in Unity. This can be done by Assets > Create > Animation Controller, which is shown in Picture 10. Then all animations need to be added to the animation controller so that the entry node connects to an empty node, and each animation has its own node which connects to nowhere. This animation controller setup is shown in Picture 11. This way none of the animations begins to play when the Game Object is loaded, and the animations stop at the last frame and do not return to original state.



Picture 10. Dropdown menu for creating new Animation controller



Picture 11. Nodes inside the Animation Controller

Next step is to create script that can call the next state method in canvas' script when the animation finishes. This is done by creating public method in the script and then adding animation event to second last frame of the animation. The animation event cannot be added to the last frame as it will not trigger. The name of the animation finish method needs to be written in the animation event's function field, without the brackets at the end. These modified animations are used in the Animation Controller's nodes.

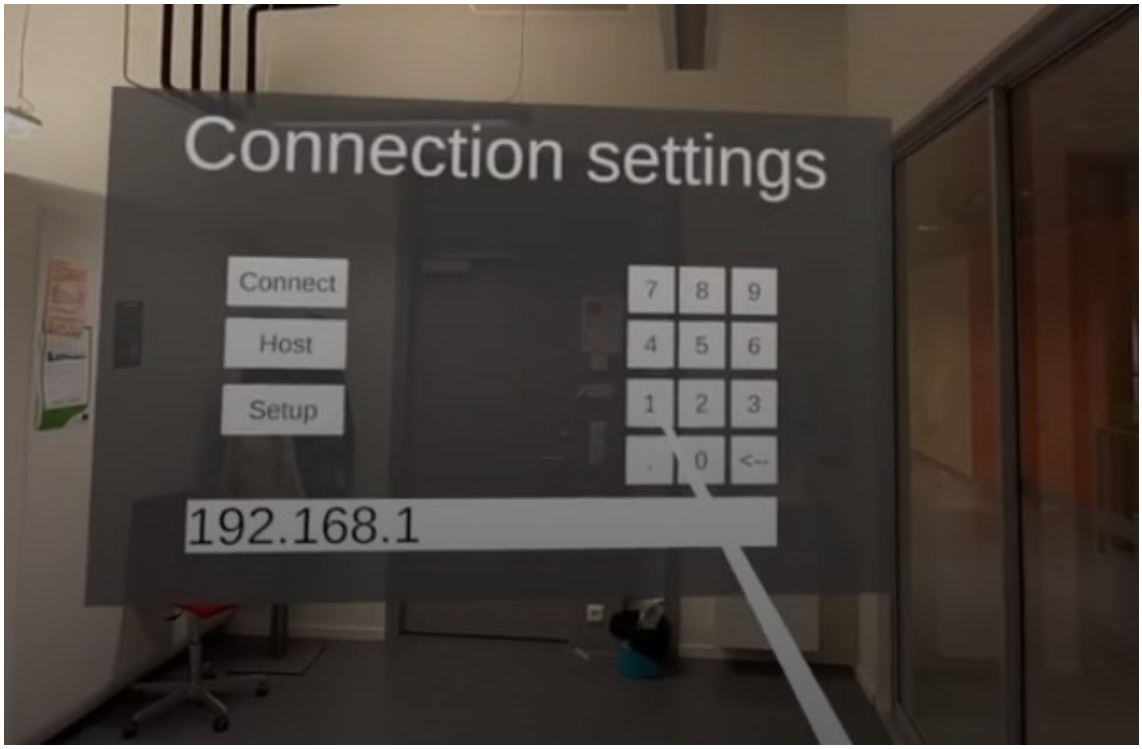
Then new components need to be added to the Game Object that contains all the 3D meshes. These components are "Animator", "Network Identity", "Network Transform (Reliable)", and "Network Animator". When all of these are added, the Animation Controller is added to the Animator component and the Animator is added to the Network Animator component.

4.3 Creating Setup scene

The setup scene focuses on allowing the host user to position the question canvas and animation within the environment. These positions are then transferred to the online scene and synchronized across all users.

The first step was to create a new scene specifically for setup. This new scene is necessary to ensure correct orientation, guaranteeing that the reference Game Objects have accurate position and rotation values.

After creating the setup scene, a new button was added to the connection screen for access. The final connection screen is shown inside the application in Picture 12. This button also executes the same synchronization method used before connecting to or starting a server. The setup scene has its own VR rig, which is synchronized in the same manner as in the online scene.



Picture 12. Final connection screen inside the application

A new Game Object was created to transfer data from the setup scene to the online scene. This Game Object, created in the original scene, has a script that ensures it is a singleton (only one instance exists) and persists across scene loads. The singleton pattern is implemented by storing the script's class as an instance within the script and then, in the Awake method, checking if the instance is null. If the instance is not null, the current Game Object is destroyed. The Awake method is a default Unity method called before the Start method. The script also contains public variables to store the poses (position and rotation) of the animation and canvas.

Two Game Objects were added to the setup scene, allowing the host to position the canvas and animation in the desired locations for the online scene. The same Game Objects used as reference points for user position and rotation synchronization were reused for this purpose. A save button was then added to the scene. Upon activation, this button saves the poses of the reference points to the data Game Object and then loads the default starting screen.

Finally, a method for correctly positioning the canvas and animation Game Objects in the online scene was implemented. This was done similarly to the VR rig synchronization between the setup and online scenes, but instead of using the data object's transform values directly, the values stored in the public variables were used. For user convenience, the canvas in the online scene was configured to always face the local client using the "LookAt(<target>)" method, where the target is the local client's VR rig Game Object.

5 Conclusion

This thesis' objective was to develop a method for synchronizing the position and orientation of users when they connect to the same MR environment. This was done by using trilateration for finding and synchronizing the positions of users and then trigonometry for finding and synchronizing their orientation.

With the help of that synchronization script, a questionnaire application was created for VR and MR glasses where multiple users can answer questions and see animations based on the questions when at the same time teacher can explain those animations. This same script can be used to create more complex training simulations where teachers can observe students working in virtual environments.

Although the synchronization script could quite accurately calculate the positions and orientations of users, the accuracy can be improved. The accuracy was within centimeters, but in perfect scenario there should be no discrepancy between the position of user in virtual and physical environment as there is no inaccuracy of measurements in virtual environment. The main problem with the results was not the script itself but instead the placement of the reference objects. This can be improved by creating a better method for placing those reference objects.

In conclusion, all the targets of this thesis were achieved, but most of them can be improved further. Also the script created during this thesis can be used for other projects.

References

Doukhnitch, E., Salamah, M., & Ozen, E. (2008). An efficient approach for trilateration in 3D positioning. Computer Communications, 31(17), 4124–4129. doi:10.1016/j.comcom.2008.08.019

Meta. Meta Quest 3. https://www.meta.com/quest/quest-3/. Accessed 5.12.2024

Meta. “Passthrough.” Oculus Developer Center, https://developer.oculus.com/documentation/unity/unity-passthrough/. Accessed 9.12.2024

Milgram, Paul, and Fumio Kishino. “A Taxonomy of Mixed Reality Visual Displays.” IEICE Transactions on Information and Systems, vol. E77-D, no. 12, 1994, pp. 1321–28.

Mirror Networking. https://mirror-networking.com/. Accessed 10.12.2024

The Mono Project, https://www.mono-project.com/about/. Accessed 4.12.2024

Thomas, F., & Ros, L. (2005). Revisiting trilateration for robot localization. IEEE Transactions on Robotics, 21(1), 93–101. doi:10.1109/tro.2004.833793

“Game Objects.” Unity Manual, Unity Technologies, https://docs.unity3d.com/Manual/GameObjects.html. Accessed 4.12.2024

Unity Asset Store. https://assetstore.unity.com/. Accessed 4.12.2024

Unity Technologies. https://unity.com/. Accessed 4.12.2024

Unity Technologies. “Making objects grabbable.” Unity Manual, https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.4/manual/xr-grab-interactable.html. Accessed 9.12.2024

Unity Technologies. “Setting up XR Interaction.” Unity Manual, https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.4/manual/index.html. Accessed 9.12.2024

Unity Technologies. “XR Interaction Subsystems.” Unity Manual, https://docs.unity3d.com/Packages/com.unity.xr.management@4.0/manual/index.html. Accessed 9.12.20