CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Master's Thesis

# Tracking multiple VR users in a shared physical space

**Isabella Skořepová**
**Open Informatics, Computer Graphics**

May 2023
Supervisor: Ing. David Sedláček, Ph.D

# ZADÁNÍ DIPLOMOVÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

| | | | |
|---|---|---|---|
| Příjmení: | **Skořepová** | Jméno: **Isabella** | Osobní číslo: **466134** |
| Fakulta/ústav: | **Fakulta elektrotechnická** | | |
| Zadávající katedra/ústav: | **Katedra počítačové grafiky a interakce** | | |
| Studijní program: | **Otevřená informatika** | | |
| Specializace: | **Počítačová grafika** | | |

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Sledování více uživatelů VR světa ve sdíleném fyzickém prostoru**

Název diplomové práce anglicky:

**Tracking multiple VR users in a shared physical space**

Pokyny pro vypracování:

1) Proveďte rešerši technik a postupů umožňujících sledování více uživatelů virtuální reality (VR) ve sdíleném fyzickém prostředí. Pohyb každého uživatele bude řešen samostatným systémem (tj. sledování náhlavní soupravy, ručních ovladačů nebo prázdných rukou, případně dalších objektů).
2) Vyberte vhodné metody s ohledem na dostupné zařízení a náročnost nastavení systému (předpokládejte homogenní i heterogenní VR systémy, např. Oculus Quest 2, HTC Vive, Windows Mixed Reality).
3) Vybrané metody implementujte a porovnejte s referenčním stabilním systémem (např. optické sledování systémem Vicon / Optitrack či podobnými). Změřte zpoždění (latency) způsobené synchronizací. Zvolte způsob implementace vhodný pro následnou distribuci (například jako knihovna, plugin nebo rozšiřující balíček).
4) Navrhněte a implementujte demonstrační aplikaci využívající implementované metody.
5) Aplikaci otestuje s dostatečným počtem uživatelů (alespoň tři skupiny po čtyřech uživatelích). Při testování dbejte na bezpečnost uživatelů VR světa.

Seznam doporučené literatury:

1] Jason Jerald. 2015. The VR Book: Human-Centered Design for Virtual Reality. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA.
2] Joseph J. LaViola, Jr. et all. 3D User Interfaces: Theory and Practice, second edition. 2017. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.
3] Steve Aukstakalnis. Practical Augmented Reality: A Guide to the Technologies, Applications, and Human Factors for AR and VR (Usability). Addison Wesley 2017.

Jméno a pracoviště vedoucí(ho) diplomové práce:

**Ing. David Sedláček, Ph.D. katedra počítačové grafiky a interakce FEL**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **10.02.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

| | | |
|---|---|---|
| Ing. David Sedláček, Ph.D.<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |

# Acknowledgement / Declaration

I would like to express my gratitude to my supervisor, Ing. David Sedláček, Ph.D., for his guidance over the course of writing semester project and later of this thesis. I would also like to thank him and the Department of Computer Graphics and Interaction for giving me access to equipment I needed for the completion of this thesis.

I'd also like to thank all my friends for their support and understanding when I was finishing the thesis. And lastly I'd like to thank my plush shark Blåhaj for her emotional support.

**Figure 0.1.** @blahajcz on instagram

I declare that this thesis represents my work and that I have listed all the literature used in the bibliography.

Prague, 25 May 2022

Prohlašuji, že jsem předloženou práci vypracovala samostatně, a že jsem uvedla veškerou použitou literaturu.

V Praze, 25. května 2022

*Isabella Skořepová*

# Abstrakt / Abstract

Tato práce představuje návrh a implementaci komplexního systému pro vytváření víceuživatelských zážitků ve virtuální realitě (VR). Práce se primárně zaměřila na vývoj pluginu založeného na enginu OpenXR se zvláštním důrazem na jeho integraci se systémem Unity. Efektivita systému byla vyhodnocena prostřednictvím kvalitativního testování uživateli a kvantitativního testování bez uživatelů.

Jádro práce se zaměřuje na různé úvahy a rozhodnutí vykonaná při vývoji a na popis různých systémů, které spolu vzájemně interagují. Nejzajímavější částí systému je kalibrační algoritmus, ale mimo něj bylo třeba vyvinout a otestovat i řadu dalších podpůrných systémů. Implementace byla provedena pro engine Unity pomocí zásuvného modulu OpenXR v programovacím jazyku Rust. Výsledná aplikace podporuje samostatné VR headsety i Virtuální realitu pro PC.

This thesis presents the design and implementation of a comprehensive system to create multi-user virtual reality (VR) experiences. The primary focus of this work was developing an engine-agnostic OpenXR-based plugin, with a specific emphasis on integrating it with Unity. The effectiveness of the system was evaluated through qualitative user testing and quantitative testing without users.

The core of the thesis focusses on various considerations and choices taken while developing the thesis and on the description of various systems that interact with each other. The most interesting part of the system is the calibration algorithm, but a lot of other supporting systems were also developed and tested. Implementation is done for the Unity engine using the OpenXR plugin in the Rust programming language. The resulting application works across both standalone and PC-based headsets.

# Contents /

# Chapter 1
# Introduction

Imagine you have an idea of a perfect virtual reality (VR) experience. It could be a multiplayer VR arena shooter à la laser tag, an exhibition in an art gallery, or an escape room. You want it to be for multiple people in the same room - in a shared physical space.

How would you create such an experience? What problems do you need to address before you make your idea a reality?

Let us break it down into different parts: You have multiple users, in the same space, each wearing their own headset. Each headset has its own computer[1] and depending on the headset, possibly has its own tracking system with separate origins. The author of such VR experience has to make all of that talk to each other and align, on top of creating the VR experience itself.

You have to create a local network real-time multiplayer application, which depending on the application in question usually has an existing off-the-shelf solution. Second, you have to make sure that the coordinate systems of each headset align and that the application is compatible with the headsets you are targeting.

VR technology progressed considerably since the days of first head-mounted displays (HMDs)[1], but requirements for implementing good same-room multi-user experiences[2] is still not very well documented — this work aims to fill that gap.

**Figure 1.1.** Illustration generated using Midjourney AI.

[1] The headset is either attached to a computer in case of PC Virtual Reality, or the headset is its own computer in the case of standalone headsets.
[2] Users being in same physical space as opposed to connected online.

## 1.1 Goals

The primary goal of this thesis is to create a plugin for one game engine, which facilitates the creation of multiplayer VR experiences in the same room. The plugin will do everything from calibration to applying the calibration, and finally a basic device and object synchronisation. The goal is for it to provide everything you need to implement multiplayer in static environments.

The final goal is to evaluate the implementation with and without users. To do that, an application has to be implemented. It can be as simple as multiple users in the same room which may or may not correspond with the physical room they are in. The evaluation should consist of users doing a simple task (e.g. handing each other an object) and then evaluating how well they could execute it. The testing without users should evaluate the application on some key metrics – namely latency and precision. Both metrics should be measured as a comparison with the existing single-player solution.

## 1.2 Thesis Structure

This thesis will be divided into multiple chapters that describe the steps needed to implement such an application. First, the thesis will analyse requirements and constraints arising from the assignment of this thesis and the resources available to the writer of this thesis. This part of the thesis also explores existing solutions. Then in chapter 3, a game engine plugin will be designed from an architectural standpoint. Chapter 4 will contain notes on the actual implementation of the plugin. Then in chapter 5 a test application used to demonstrate the plugin will be designed and implemented and used to evaluate the key metrics. After the implementation is done, chapter 6 will test the implementation with users. Finally, the final chapter, chapter 7, contains conclusions and ideas for future improvement.

# Chapter 2
# Analysis

First, I want to describe related solutions, the problems they solve, and how they solve them. To that end, I first have to decide and describe the target devices, because it greatly influences which solutions could be chosen and helps me focus only on the relevant existing solutions. Afterwards, I would like to outline ways to solve the calibration problem. Then, existing related solutions in this space can be described.

After the related solutions are described in detail, I want to dive deeper into what exact problems need to be solved and what steps need to be taken to do so. Finally, before I can move onto designing the technical solution, I need to make a few more decisions on the exact problem this thesis will be solving.

## 2.1 Overview and problem statement

It is helpful to start by creating an overview of an end goal of the work before embarking on going deeper. The end goal of this thesis is to have a Virtual Reality (VR) application. This VR application should allow interaction of multiple people in the same (physical) room, all wearing VR headsets and being able to interact with each other – ideally in both virtual and physical form. The application should support at least four people at the same time.

The current generation of consumer VR headsets has settled into a similar feature set with regard to motion tracking. All popular currently sold consumer VR headsets support both rotation and position tracking required for room-scale experience. But most of them only support it for one headset and two controllers at the same time. Sometimes, they do have their own system for interaction between headsets, but there is no standard way to combine their tracking spaces into one tracking space shared between headsets from different manufacturers (this is what the assignment means by heterogeneous VR systems).

Therefore, the application should not only work on various headsets, but also provide a way to synchronise them and support using any combination of headsets in single session. All of this should be wrapped into a reasonably easy-to-use plugin for a chosen engine. This plugin should support at least four users, but there should not be any unnecessary limitation on the number of users it supports at the same time.

Apart from supporting various headsets with disparate tracking systems and most of the logic being contained in a plugin, what else should be in the resulting application? The application should contain some way for users to interact with each other. Some interaction is a natural result of them being in the same room — for instance, they can speak and listen to each other without the application having to implement voice chat. Then there is the obvious feature of being able to see each others' controllers and headsets. This is the core feature of the plugin and should work flawlessly, and as will be explained later, it is not as simple as it looks. Lastly, the application should have some virtual objects to interact with, which should be synchronised between headsets as users expect that what they see, the others could see as well.

### 2.1.1 Target devices

The first primary consideration when designing any VR experience is the choice of the target device. For this work, it is even more influential since the tracking system and operating system it runs on might limit some technology choices. The headset will affect most of the other decisions because different devices have different capabilities and tracking systems.

According to Jung[2] can be divided into multiple categories. Ranging from Smartphone VR to PC-tethered, or from Stationary, through Desktop scale, and Room scale to Mobility.

Of particular interest to this thesis will be devices with positional tracking, as stationary users cannot interact in real-world space. Another requirement will be controllers with positional tracking so that users can see each other's hands.

The application created as a part of this thesis will therefore target most devices with 6DOF tracking, meaning that they provide an application running on it with information about its position and rotation in the real world.

The most popular consumer VR headset[3] at the end of 2021 is Quest 2[4], making it a primary target. It features 6DOF tracking using four cameras, which it uses to reconstruct the surrounding world. Therefore, it is a good representation of an inside-out tracking system using SLAM (Simultaneous Localisation And Mapping).

The second most popular headset is the Valve Index and will be used as a representative of the Lighthouse-tracked family of headsets.

The third most used PC-tethered headset is Oculus Rift S. It uses a very similar tracking system to Quest 2 and was discontinued in June of 2021. For those reasons, it will not be used in this thesis.

Another family of headsets is Windows Mixed Reality[1]. They are inside-out tracked similar to Quest 2, but since they use a different implementation, they might have different characteristics. Due to their falling popularity since I started working on this thesis, I will treat them as optional – if time allows, I will try to run the app on one of them, but they will not be included in the full testing and I will not be spending significant effort on trying to run the app.

Lastly, there is PICO 4, which added OpenXR support[5] while developing this thesis and has similar specs to Quest 2. As it is a very similar headset to Quest 2, assuming that OpenXR support is good, everything that works on Quest 2 should work on PICO 4 with minimal adjustments. As such, I will try to get my application to run on the headset after everything else is done.

To conclude this section: I will test the app on representatives of two most popular ecosystems - Valve Index and Quest 2. Additionally, I will try the implementation on few more headsets and report on how well it works.

## 2.2 OpenVR Space Calibrator

The primary inspiration for many choices taken in this thesis was the OpenVR Space Calibrator [6] (SC) project. It is a plugin for SteamVR (OpenVR is the API that is used in SteamVR).

SteamVR has support for using devices from multiple tracking systems at the same time, but it is not very useful on its own, because the tracking systems of differing headsets are all but guaranteed to be misaligned. This is where SC comes in. On a

[1] Contrary to their name they have nothing to do with mixed reality and instead are virtual reality

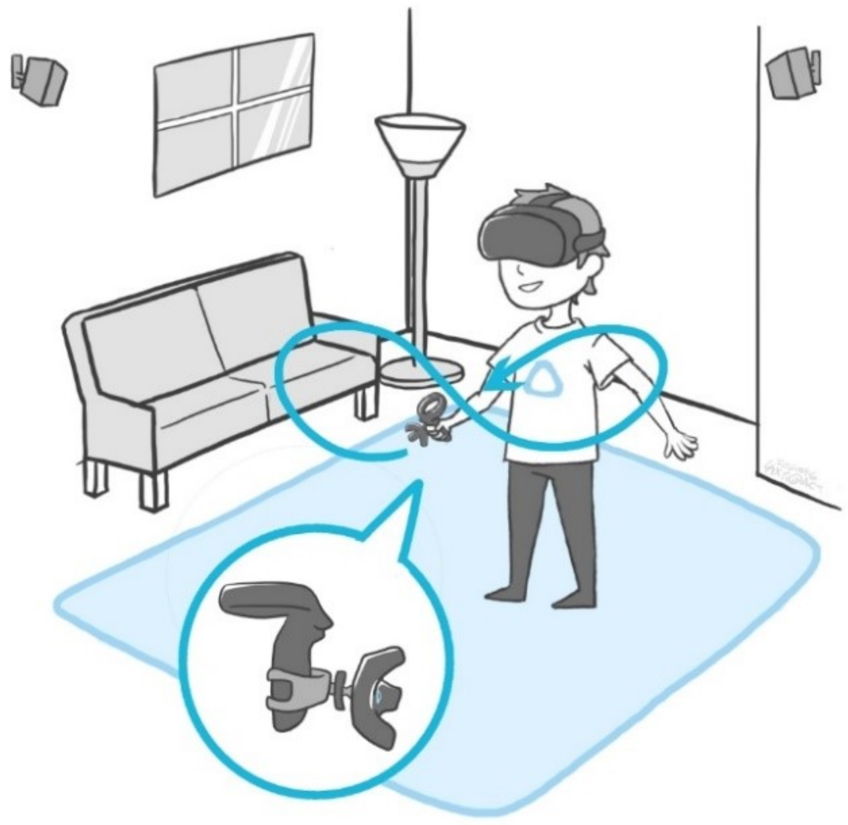


**Figure 2.1.** Illustration of how SC works. Taken from VIVE blog[7].

high level, what it does is it rewrites data coming from one tracking system to match data coming from another. This allows the application accesing the system to not have any special support for this situation and just pretend like all the devices are in one global tracking system.

SC consists of three parts: calibration algorithm, a driver for applying the calibration, and a GUI to give user control over this. This is basically a subset of what this thesis aims to achieve.

The driver part of SC hooks into OpenVR API and modifies Poses (orientation + position) passing through it so that the disparate tracking systems match up. It listens for commands from the calibration algorithm to change the calibration. How exactly it does so is not that important for the purposes of this thesis, as it depends on the API used and OpenVR does not work natively on Quest 2[2], which means that it cannot be used as a basis for this thesis.

The more complicated and unique part of SC is the calibration algorithm. From the user's perspective, the way it is used is as follows: Pick two tracked devices, one from each tracking system. Hold them together and trigger the start of calibration. Then start moving both controllers together in the horizontal number eight pattern to try and sample as many positions and orientations as possible — see figure 2.1. Once the calibration is done, it will be applied.

[2] You can run OpenVR-based applications on Quest 2, but only if you connect your headset to a PC running SteamVR. I want my plugin to support standalone mode.

The user can select one of three calibration presets: fast, slow, and very slow. The difference is that slower presets collect more data to hopefully make the calibration more precise. Users of standalone headsets connected via WiFi are encouraged to use slower presets and move controllers slower to make the calibration process more reliable. Under the hood, each preset corresponds to a different number of samples collected (see table 2.1). The time between sample collection is fixed at $t_s = 50\,\text{ms}$.

| Preset | Samples ($n_s$) | Min. time ($t_{min}$) |
|---|---|---|
| Fast | 100 | 10 s |
| Slow | 250 | 25 s |
| Very slow | 500 | 50 s |

**Table 2.1.** How long does it take to do calibration in SC

You might notice that the time listed in the table is double of what $t_{min} = t_s \cdot n_s$ might suggest. The reason for this is that the number of samples is collected twice. Once to calibrate rotation and once to calibrate position. This leads to $t_{min} = 2 \cdot t_s \cdot n_s$. Note that the time listed is a minimum time, because SC does trigger sample collection when the time since the last sample is greater than or equal to 50 ms and this function is tied to framerate. The estimate also does not include the time it takes to compute the calibration, but that should be negligible.

### 2.2.1 How does SC compute offsets

The SC project explains the basic concept behind the calibration in a file called `math.pdf` in the repository, but as it is the foundational idea around which this thesis is built, it deserves further explanation. The process consists of two separate steps — sample pair collection and computation. Both steps happen twice — once to calibrate orientation and once to calibrate position.

**Sample collection** occurs in regular intervals of at least 50 ms. When it decides that it is time to collect the sample, it just takes the last known pose of each device and stores it in a list. To decide if it has enough samples, it just uses hardcoded limit for number of samples. It offers three presets: fast (100 samples), slow (250 samples), and very slow (500 samples). Once enough samples are collected, it proceeds to the next phase.

**To compute the orientation** it uses Kabsch algorithm[8], which can recover the rotation matrix from the SVD decomposition. This algorithm takes paired positions and produces the best rotation in the sense of least-squares. After this computation is finished, the rotation is immediately applied and a second set of pairs of samples is collected, now with the rotation applied.

$$
\begin{aligned}
H &= P^T Q \\
H &= U \Sigma V^T \\
d &= \operatorname{sign}\left(\det\left(V U^T\right)\right) \\
R &= V \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & d \end{pmatrix} U^T \\
Q' &= RQ
\end{aligned}
$$

With the second set of points **it computes the offset**. This can be easily achieved as it is just a set of linear equations. When this offset is computed, it then applies it and the whole process is finished.

## 2.3 Approaches to calibration

There are multiple ways of approaching the calibration of the tracking spaces. This section will first categorise all available options.

The first option is to choose devices with a globally anchored tracking system, which means that all devices have the same origin of their respective tracking spaces. SteamVR's Lighthouse system falls into this category. The main drawback of this option is limited options for headsets - no option to use existing devices if they do not use the correct tracking system. It also disallows usage of headsets from multiple vendors as long as there is a shared reference point. However, the advantage of this option is simplicity.

The second option is to completely replace the tracking system with another system, such as OptiTrack for VR[9]. This option, once set up, is as convenient as the first option and is proven by existing commercial deployments such as Golem from DIVR Labs[10].

Neither of those options are applicable to this thesis, as the assignment implies a heterogeneous tracking system.

The third option is to assume that the difference between the origins of the tracking spaces does not change. This difference could be obtained by performing a predefined set of steps once at the beginning of each session. After this, the application relies on each devices' tracking system separately. Compared to previous options, this allows multiple tracking systems to be used at once. It also allows for the use of headsets with inside-out tracking, which are usually cheaper. A disadvantage of this option is that the calibration could drift over time (from the origin of the system moving), which would cause the calibration to become incorrect. Another advantage of this option is weight, as nothing extra needs to be attached to the system.

The fourth option is to determine the calibration continuously while the user is in VR. For example, having the HTC VIVE tracker attached to Quest 2, calibrating the Lighthouse-based headset to the Quest. Another option to do this would be to attach OptiTrack trackers to each headset. This option is different from replacing the tracking system with a different one by still using the integrated tracking for high-frequency data and the attached system only for low-frequency calibration updates.

### 2.3.1 Other alternatives

The categorisation introduced above assumes that you do not have deeper access to the tracking system. But if you are a vendor of SLAM-tracked system, you have one more option: map sharing. With access to map data, the headsets could compare the major detected features between themselves and perform this calibration with only the onboard tracking. This is how two existing solutions work [3]

The first is Vive Focus 3map sharing [7] that came out in November 2021. Map sharing seems to be a paid add-on to the headsets, and interested users are directed to contact sales. As such I couldn't get more information about it, but it is proof that it could be done with existing hardware.

[3] This is just a presumption of the author of the thesis, neither of them is open-source and they do not exactly go into great detail insofar of how they work.

The second is Meta's Shared Spatial Anchors[4]. Meta explains this in more detail than HTC and it works by uploading point tracking point cloud to Meta's servers. This could be seen as a disadvantage as it is a privacy concern and presumably requires the internet connection. It is unclear from documentation whether the upload/download from Meta's servers happens only once or if persistent internet connection is required.

Both of those solutions only work on headsets but from the same vendor, meaning that you can use them with Quest 2 and Quest Pro, but you cannot add Focus 3 to the same experience. This limits their use. The advantage of them is that they should be as reliable as the underlying tracking system and should never diverge.

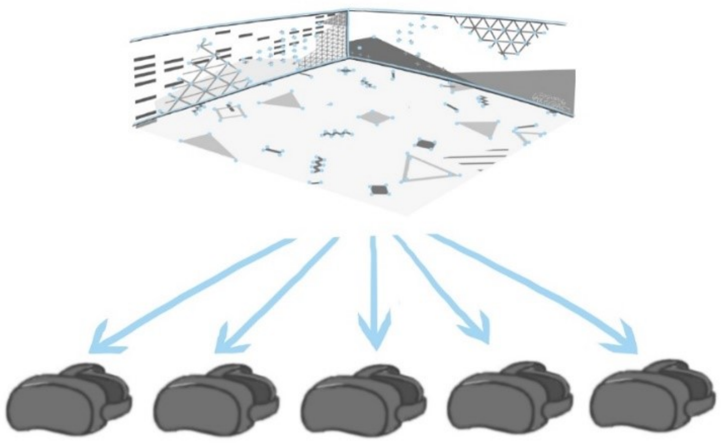


**Figure 2.2.** Map sharing as illustrated in Vive's announcement blog post.

As for the use of those solutions as a programmer, it is unclear how much work HTC's solution requires. Meta's solution is much more open, as the API is published as a set of OpenXR extensions and is listed directly in official OpenXR specifications.

### 2.3.2 Shared spatial anchors

The OpenXR specification has a set of optional extensions, some of which are vendor-specific. The nice thing about these extensions is that they are all specified within OpenXR, so at least the shape of the API is known. Some of these extensions allow developers to use what is called Shared Spatial Anchors.

The primary extension needed for Meta's solution is `XR_FB_spatial_entity_sharing`. First, you have to figure out a way to enable extensions on your engine of choice and enable this extension along with any additional extensions it builds upon. You also need to have your multiplayer already set up so that you can send information between headsets. Then you synchronise a list of `XrSpaceUserFB` handles that represent the connected users between headsets. When this is done, you call the `xrShareSpacesFB` function to share a spatial anchor and wait for the share event to complete.

[4] https://developer.oculus.com/blog/build-local-multiplayer-experiences-shared-spatial-anchors/

When all of that is done, the application can send an identifier of the spatial entity to the other headsets. The other headsets can then use this identifier to request the anchor and from there request the `XrSpace` it represents. This space can then be used as a reference when locating controllers or headsets.

All of this is more complicated than needed for a simple case of calibrating the spaces, but is also not as bad as it first sounds *if interacting with OpenXR directly*. If your engine does not have support for this, it could become difficult to add support. On the other hand, if it does, it presumably abstracts some of this complexity away.

### 2.3.3 Hand Tracking-Based Calibration

In Colocation for SLAM-Tracked VR Headsets with Hand Tracking[11] the authors describe another method that can be used to calibrate headsets. This method uses hand tracking as a shared anchor. It relies on each tracking system having its own hand tracking. Both headsets should see the same hand, and when the calibration is triggered, both headsets report the pose of the hand in their tracking space. This information is then used to compute the difference between both headsets tracking systems, and one of the headsets gets relocated to a new position.

Apart from the way they compute the calibration, the application of the offset is the same as in SC. The main drawback of this method is that it relies on hand-tracking, which is not available for Valve Index headset by default. Although there are add-on systems allowing hand tracking either by adding additional hardware like Ultraleap's Hand Tracker, or even by utilising the onboard camera like Monado's hand tracking[12].

## 2.4 Plugin requirements

To implement the plug-in, we have to decide which API and layer of the software stack to plug into. Many headsets have their own API - for example, VRAPI on Oculus/Meta headsets, OpenVR on headsets that use SteamVR (such as HTC VIVE). More recently, headsets started supporting a unified API called OpenXR which solves many portability problems, but being a newer API, it has some missing parts. But since older APIs are getting deprecated[13], I decided to use OpenXR as a baseline for my implementation, as it is supported on all target headsets.

On the other end of the software stack is the application/engine. Ideally I would like my work to have the potential to be usable in wide range of applications regardless of engine. To do this, I would like the base of the plugin to be engine-agnostic and work on the OpenXR layer of the stack. The work in the engine would be limited to creating a simple wrapper plugin. This engine-specific plugin should only load the generic plugin binary, implement functionality that is required but is not possible to implement only in the OpenXR layer (if needed), and expose to the application any interfaces that are not generally available in OpenXR (or interfaces that the engine lacks).

To limit the complexity of the implementation though, I will only focus on single engine. The engine I chose is Unity, because it is the engine with which I have the most experience. It is also very often used for VR content.

## 2.5 OpenXR description

OpenXR[14] is an open standard containing a set of APIs for access to Augmented Reality (AR) and Virtual Reality (VR) devices. It is implemented on all main consumer

VR headsets. It encompasses everything needed for VR including session lifecycle, rendering (it manages the render loop and composition, but defers the actual rendering to OpenXR or Vulkan and input methods (the action system). Of particular interest to us is the action system.

The action system is based on semantic paths to devices and action sets. Action set is a list of actions the application supports (e.g. teleport or select) and can be bound to one or more devices using interaction profiles. There are multiple well-known paths specified, but the runtime can add more, provided that it follows the rules outlined in the specification.

This gives us an opportunity to create engine and headset agnostic plugin, which will inject itself between the app and the runtime. OpenXR even provides facility for this with its concept of API Layers. The layers work by overriding OpenXR functions and when the function is called the layer can do anything and optionally call runtime's original function.

This layer can do anything that goes through OpenXR, within constraints. Namely it can collect headset's pose (position and rotation), collect list of controllers the app interacts with and their state. Their state includes pose and states of any buttons, joysticks, triggers, force sensors etc.

Theoretically it can also inject the list of remote controllers/headsets as additional local controllers, but this depends on how the application/engine is implemented. If it retrieves the list of devices from the runtime (there is nothing in base OpenXR to do this, but there might be some extensions for this), then it is possible provide the data using OpenXR and without modification the the app. But if the engine hard-codes a list of well-known paths then we are out of luck and we have to do some work in-engine. This needs to be taken into account when designing and implementing the plugin.

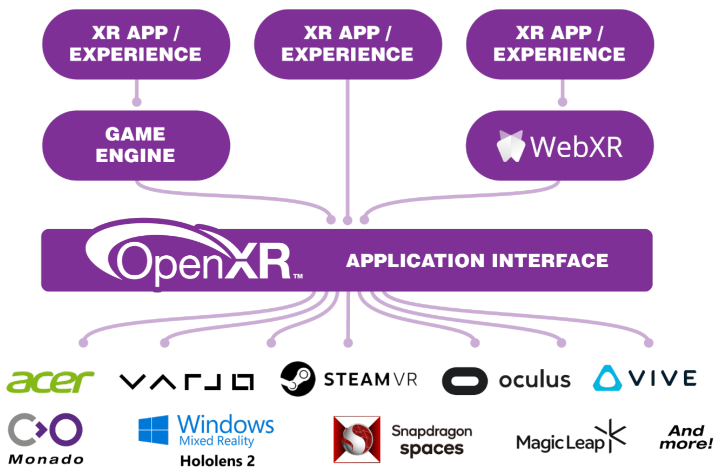


**Figure 2.3.** Illustration of where OpenXR slots in the stack of creating VR applications. Taken from `https://www.khronos.org/openxr/`.

One convenient thing for the implementation of the plugin is the concept Spaces. OpenXR goes to great lengths to *not* assume global coordinate system and instead every API that has to do anything with positions either takes or returns `XrSpace` handle. This allows us to introduce the concept of Server reference space.

Ideally, the app would be responsible to provide correct space handles to the runtime, but this would require changes to the engine. Instead, the layer could intercept any call taking a `XrSpace` handle and detect Stage handles and replace them with Server handles. Ideally, to be compliant with OpenXR specification the plugin should also emit correct events when changing the Server space position, but this might not be neccessary, depending on how Unity uses the reference space.

## 2.6 Networking

As each headset is its own self-contained system, there will be a need to network the headsets. It is enough for the application to only work on a local area network (LAN) as it is assumed that devices in the same physical space can connect to shared WiFi.

The chosen networking solution will have to be low latency as the actions of other users have to be propagated real-time to enable user to user interactions. Additionally, the solution will have to be resilient to momentary interruptions as consumer-grade WiFi can sometimes be slightly unrealiable and the solution should therefore handle momentary disconnect/reconnect gracefully. All chosen headsets support IP-based networking with TCP and UDP protocol and UDP should be used for real-time updates to make sure that they get delivered in timely manner.

## 2.7 Summary

As evidenced from previous sections, there are multiple moving parts required to make a collocated VR experience. The first is implementing the offline version of the experience – getting data from the headset and displaying the content, which is thankfully mostly handled by Unity.

The second part is to send the information between headsets. All targeted devices support IP-based networking, which makes it a natural choice. Therefore either peer-to-peer or server-based networked communication has to be implemented.

Last and perhaps the most essential part is calibrating the tracking spaces of various headsets. All headsets I encountered have tracking defined as one unit being one meter, which means the output of a calibration algorithm is translation and rotation[5].

It is important to note that before the calibration can be meaningfully implemented the previous two parts have be implemented first, because the position information has to be shared between devices.

[5] Some headsets have a slightly wrong scale of about 0.8% according to issue 23 on OpenVR Space Calibrator[6], which means that this assertion is not entirely true. However, it holds for the most popular units and no headsets in the current generation exhibit this issue, so it'll be assumed that the issue does not exist for the rest of this thesis.

# Chapter 3
# Design

This chapter will outline the main architectural decisions taken to implement the plugin. As described in section 2.7 several interlocking parts must be designed and subsequently implemented. This chapter will describe each of them in detail. Further, this chapter will describe a simple, possibly novel, calibration approach to be used as a stepping stone before implementing the full calibration.

## 3.1 Engine choice

As noted in section 2.5, some of the design depends on how the application/engine implements OpenXR integration. To limit the scope of the thesis, I had to choose a game engine to focus on. The options I explored were Unity, Unreal Engine, and Godot. The choice mostly depends on how well it supports OpenXR in current version as OpenXR support is somewhat new and only recently became the recommended option — in particular how it implements discovery of input devices.

My first choice was Unity[15] because I have some experience with creating VR experiences with it. Testing unity versions `2021.3.3f1` and `2022.2.12f1` with first iteration of my plugin revealed that it does use a hard-coded list of well-known paths. This is suboptimal as it means that I have to handle remote devices differently from local ones.

Then I tried Unreal Engine 5.2 preview 1[16] to test end to see if it does things differently. I did not explore it exhaustively, but the default template also uses a predefined list of paths, and a quick search through its source code revealed that it does not use the `xrEnumerateBoundSourcesForAction`[1] function, which would be necessary to allow multi-controller setups.

I wanted to also try the open source Godot engine, but when I tried it, Godot 4 was brand new and it did not have VR template ready.

Therefore, it seems that current versions of game engines do not support more than two controllers at a time (which is a regression compared at least to OpenVR plugin). This means that I have to add my own APIs to discover remote devices. As for the engine choice, since I am more experienced with Unity, I decided to focus on that game engine. But the bulk of the work should transfer well to other engines.

*Sidenote:* from reading OpenXR spec, I am not sure whether the case of having more than two controllers connected to a single system is supposed to be supported or not. There is the above-mentioned function to enumerate bound sources, but that potentially only results in paths the application already provided. Additionally, the `XR_HTCX_vive_tracker_interaction` extension defines an extra function to enumerate connected trackers.

[1] Note that text with `purpleish background` refers to a term defined in the OpenXR spec[14] and is clickable in the digital version. This is opposed to `grey background` which merely represents the verbatim value.

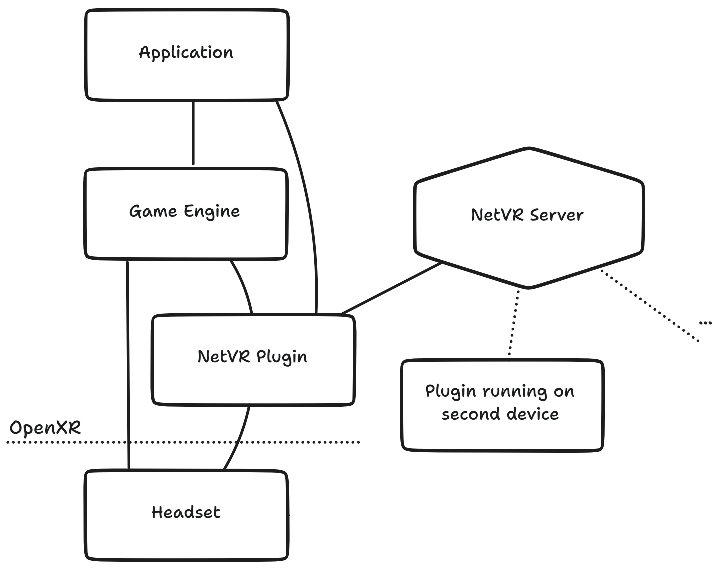


**Figure 3.1.** System component overview

## 3.2 System components

The whole system comprises multiple components that all communicate with each other and the outside world. Figure 3.1 outlines the main components of the system. I already have the game engine that dictates how the application is implemented.

It also informs how the main part of this thesis — the engine plugin — looks like. The plugin consists of two parts. One is implemented in C# and is basically a part of the application code. The other is compiled into native plugin and sits between Unity and OpenXR runtime, which allows it to intercept some OpenXR calls (some calls do not go through the plugin and go directly to the runtime). This simplifies the implementation of the C# "managed" part. Since not everything the plugin can do has existing OpenXR APIs, we need to add a few more APIs to the native plugin. The managed code then exposes those in a reasonable manner to the application directly (see section 4.3.1 for a more detailed discussion).

The language choice for the native plugin is limited as it needs to be able to compile into a dynamic library and interoperate with the managed code. The obvious choice here would be to implement it in C++, but since OpenXR is inherently multithreaded (even though Unity's implementation uses just a few threads), I would have to manually keep track of locks over various resources. This, while possible, is difficult to do correctly and even more difficult to verify and debug once it goes wrong. To that end, I decided to use Rust for my implementation, which mostly keeps track of various invariants for me. The implementation will not necessarily be an idiomatic Rust code as that requires more experience than I have, but it will hopefully provide some benefit.

## 3.3 Reading local device data

As I decided to use OpenXR, I can just use this universal API to read the information about controllers. There are two main approaches for doing this in context of the plugin's design — one is for the plugin to create its own `XrActionSet` and bind it to controllers and the other is to override relevant functions and use the same action sets that the engine uses. I decided to go with the latter approach as I already had to implement it to determine what functions the engine actually calls, and this allowed me to reuse that work.

The advantage of this approach is that it does not add extraneous actions for the runtime to deal with, and therefore some theoretical future action rebinding support will not break it as otherwise you might rebind only one of the duplicate actions. The disadvantage is that it relies on the action sets that the application uses, which might differ from application to application. This is not a problem for me as I target only one engine, but it might be if someone wanted to use my work with a different engine.

It does not, however, prevent me from changing the timing used for reading the data as I only store the handles to actions, not and do not interfere when reading the action data. Instead, the plugin has its own loop in which it reads the run-time data and sends them across the network.

## 3.4 Providing data to the application

Local headsets and controllers should be handled normally by using functions implemented by the engine, because they contain logic to compensate for the delay (as described in Chapter 18.7 of the VR Book[17]) built into them. This means that we only need to provide data about the remote headsets and controllers.

The ideal solution for this is to inject the data via standard OpenXR calls, but as noted in section 3.1, the most popular engines do not use the functions necessary for this. This means that I have to either modify the engine to use those functions, or add new functions which I will use from the game code directly.

It could theoretically be packaged as an OpenXR extension, but that complicates the work and is beyond the scope of this thesis. But there is nothing preventing packaging the API into an extension in the future. Therefore, I decided to make it a dynamic library which will be loaded by a script from the engine.

## 3.5 Communication between headsets

As noted previously, applications running on different devices need to communicate with each other in real time. To achieve this, a client-server architecture will be used. The disadvantage of this option is the potentially higher latency because there is an extra step in the flow of information between headsets. The advantages are greater scalability (since each device has to upload its information only once) and easier-to-understand information flow.

Since the plugin will be implemented in Rust, which has mature server library ecosystem, the server will be also implemented in Rust to lower the amount of context switching in development. It also helps in making sure that all messages are in the correct format, as the same code can be used for encoding and decoding the messages.

Three separate conceptual information channels will be used for various types of messages. First, we have the reliable time-insensitive channel, which is used for exchange of infrequently changing information. For this purpose, the TCP protocol or something similar is the ideal fit.

The second channel is used for real-time updates. Latency is critical here, but usefulness of each message diminishes quickly over time, since it is not interesting to know where the controller was a second ago if you know where it is now. UDP is a good mechanism for this.

The third channel is used during the calibration to exchange information needed for the given calibration method. Here, reliability becomes important at the expense of latency. There is nothing that really prevents it from merging with the first channel, but separation makes the design more clear and prevents blocking head-of-line across the channels. This channel would benefit from lower latency to make sure that we can update the calibration estimates in real time, but it is not very critical.

#### On latency

While I will use UDP for sending time-sensitive data which with low long-term relevance, it might be useful in the same cases to use the same protocol for both the data and configuration. This would imply using TCP or similar for both, which would increase latency. To that end, I want to debunk the misconception that this would result in unusable experience. My original proof of concept implementation, which had a server written in Javascript programming language and used Deno runtime, used WebSockets[18] and therefore TCP for all communication. I did not do the experiments with users, but from my personal experience it was fast enough for some uses. It was not fast enough to use remotely tracked controllers directly (where the latency was noticeable and somewhat unpleasant), but when other people were using the controllers, it was not noticeable.

#### Service Discovery and Multicast DNS

To find a location for the server, I originally wanted to use the mDNS and DNS-SD protocols. These are pair standard protocols which are based on sending multicast UDP packets to well-known port to discover services available on local network. The protocols are described in RFC 6762[19] and RFC 6763[20] for mDNS and DNS-SD, respectively. In the end, I decided against it as I could not find a working combination of libraries that worked across different operating systems. Surprisingly, I had more problems with resolving the address than with publishing it.

In the end I decided to just use IPv4 UDP multicast. When the client wants to connect, it periodically sends a probe packet which the server listens to and responds with its own packet. The client then connects to this server. This means that I have to choose a predefined port number and for that I chose `13161` as it is marked as `Unassigned` in IANA's port assignment database[2] and is most likely vacant.

## 3.6 Calibration methods

As explored in the analysis, there are many calibration methods, which could be implemented in this work. I decided to choose the most universal methods and developed a second, possibly novel method, which while less reliable is considerably easier to implement.

### 3.6.1 Complex calibration method

Main method to be used in this thesis is the method from OpenVR Space Calibrator. This method relies on a set of pairs of controller poses. User takes one controller from one device and one controller from other device, holds them together in such a way so that they slip as little as possible. Then they move the controllers in infinity sign. The application then records the poses and from that calculates both the offset between the

[2] https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

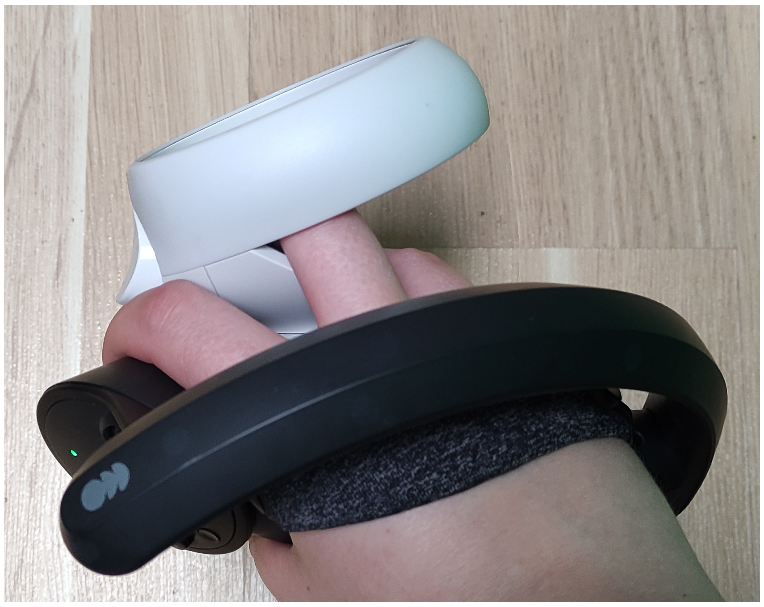


**Figure 3.2.** Example of how to hold controllers for the Space Calibrator method. In this example I'm holding Index controller and Quest 2 controller.

controllers and the offset between the tracking spaces. For more detailed explanation see section 2.2.

The main reason for choosing this method as opposed to other methods available is its universality as it could be used with basically any headset which supports 6DOF tracking. As the calibration is only one subsystem of the whole, it should not prevent adding other calibration methods in the future.

I will have to make some slight tweaks and additions to the method to make it work reliably over network as the recommendation of SC is currently to move ”slowly when calibrating and using the Slow or Very Slow calibration modes”. This behaviour is caused by timing of the collected samples diverging.

The possible improvement here is to sample the device position at precisely specified time, instead of at ”last frame before calibration tick”. This makes sure that the collection happens in regular intervals, but more importantly makes sure that samples from both tracking systems represent the same moment in time. I am not versed enough in OpenVR to know if SC could've implemented this, but OpenXR does provide facilities which make this possible.

Further improvements would lie in tweaking various constants in the calibration, or determining them dynamically. Moreover, it should be possible to use the same set of samples for calibrating both rotation and position, possibly cutting the calibration time in half. Those improvements will not be implemented in this thesis as there are more important things to explore, but it would definitelly be interesting.

### 3.6.2 ToD method

To aid with the rest of the implementation without having to implement the complex calibration routine a simple one-time calibration method was developed. This method is simpler to understand and easier to debug as its result could be computed by hand and the result could be checked separately. This allows to check the application of calibration separately from its computation.

To calibrate the tracking spaces of two or more headsets the user puts each headset on top of each other (see figure 3.3). This roughly aligns positions and directions of the `VIEW` reference space. User then presses a button in the dashboard and the calibration is done. I call this method the Tower of Doom (ToD) method as the tower becomes increasingly unstable with each headset that is added to the stack.

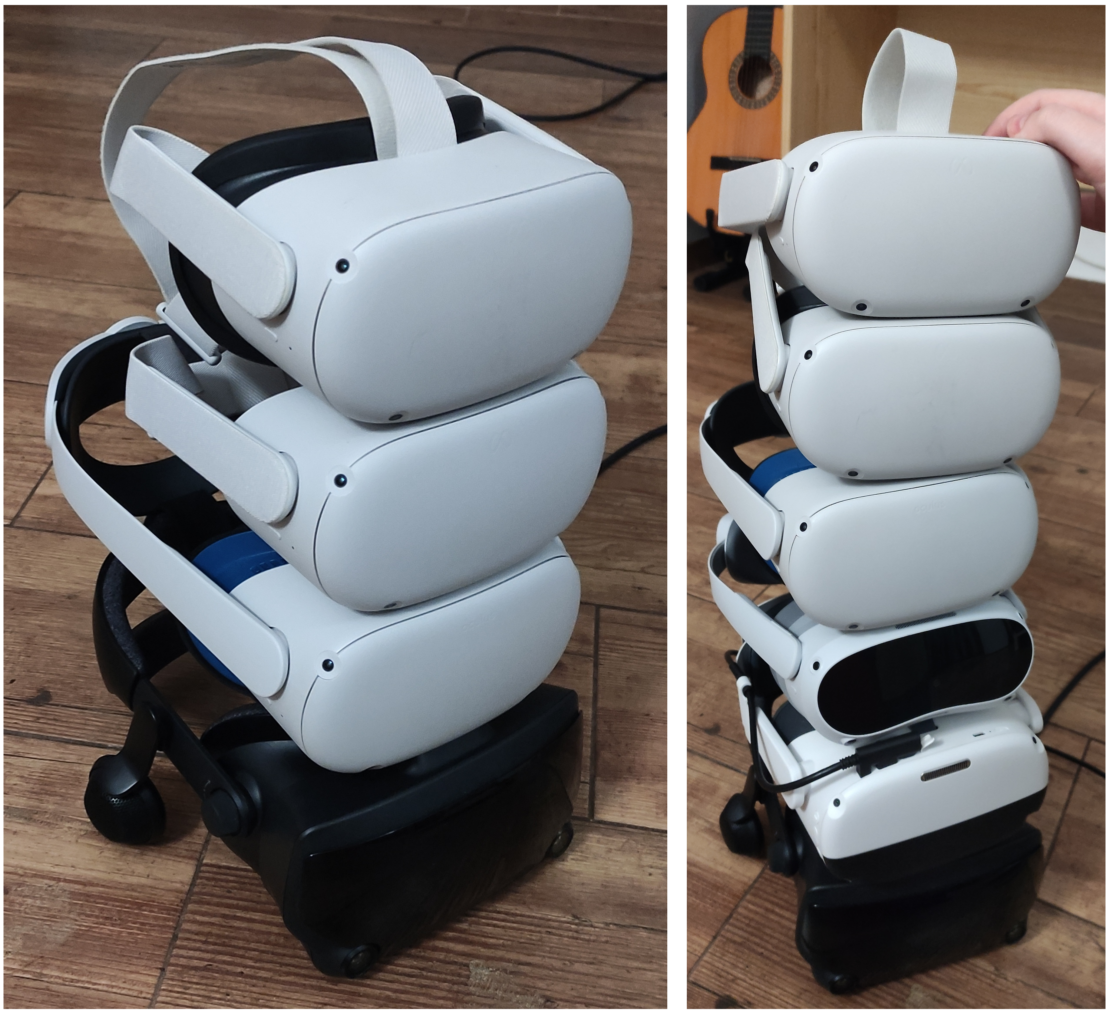


**Figure 3.3.** Stack of headsets used for tower of doom method. Demonstrates how the method works in practice and also shows that while possible to use, the method does not scale well beyond four headsets (at least for those particular models).

The server first fetches current position and rotation of all currently connected headsets. It consequently calculates transformation between the headsets while assuming that position of the ground and direction of the up vector is correct between each headset. The transformation is (assuming that y axis is up):

$$
\begin{aligned}
T_x &= \sin(r_y) \cdot z - \cos(r_y) \cdot x \\
T_y &= 0 \\
T_z &= \cos(r_y) \cdot z - \sin(r_y) \cdot x \\
R_x &= 0 \\
R_y &= -r_y \\
R_z &= 0
\end{aligned}
$$

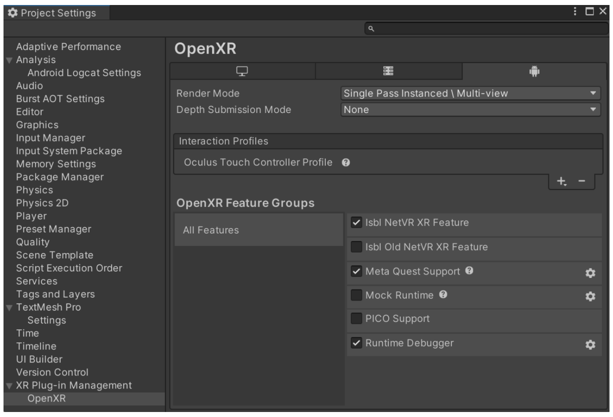
This puts the origin of the system of coordinates under the stack of headsets. And forward vector corresponds to the direction in which the headsets were looking.

The above assumptions stem from the fact that both tested headsets have calibration procedure which sets floor and therefore translation in the y axis should be zero and from the fact that the headsets have accelerometers which means that they can measure direction of the gravity. I call this method a tower of doom as while stacking two headsets on top of each other is not hard, stacking more headsets is feels a bit dangerous as at least Quest 2 headsets have a rounded exterior.

There is an alternative and even simpler formulation, which changes the rotation formulation to apply reversed values of all angles. This formulation requires the headsets to be level with ground and looking directly forward as to not skew the ground plane. This formulation is useful if rotations are represented as quaternions and euler angle decomposition is not readily available, or if said decomposition does not have the properties needed for the calibration to work properly.

## 3.7 The Unity plugin

Apart from dynamic library which handles interfacing with OpenXR and server communication, there also needs to be a C# wrapper which handles loading of the plugin and some light abstractions to make the usage more natural. The entrypoint of the plugin will use Unity's `OpenXRFeature`[3]. This provides us with exactly the hooks needed to instantiate the app and also allows the user to enable/disable the plugin at will by checking a checkbox in OpenXR settings (see figure 3.4).



OpenXR

Render Mode: Single Pass Instanced \ Multi-view
Depth Submission Mode: None

Interaction Profiles
Oculus Touch Controller Profile

OpenXR Feature Groups
All Features

- ✓ Isbl NetVR XR Feature
- ☐ Isbl Old NetVR XR Feature
- ✓ Meta Quest Support
- ☐ Mock Runtime
- ☐ PICO Support
- ✓ Runtime Debugger

**Figure 3.4.** Unity's OpenXR feature settings window

Beyond hooking into OpenXR lifecycle, the plugin will provide a few Unity components. One of the components will handle spawning and destroying remote devices and will allow the developer to set a prefab used for remote devices. Another component will be attached to each remote device and will allow querying information about it and will also provide custom Editor UI to surface this information in the editor. Those components together will allow the developer to handle the devices in any way they want, eg. by spawning correct model for each controller/headset.

Another pair of components will give developers convenient way to create objects synchronized between headsets. This will only provide basic position synchronization

[3] `https://docs.unity3d.com/Packages/com.unity.xr.openxr@1.7/manual/features.html`

and anything else will have to be implemented by the developer as there are other solutions for this problem.

The plugin will handle automatically converting to and from Unity's coordinate system to OpenXR coordinate system, so that all data in the native code is in OpenXR's system and all data (apart from calls to the native code) in C# code is in Unity's coordinate system and the probability of any mixup happening is minimized. Finally, the plugin will forward logs from the plugin to Unity's log system as by default stdout/stderr is not available from Unity's Editor.

All this functionality will be wrapped in UPM package (not to be confused with .unitypackage files), which makes sure that the package can be imported by placing the sources in correct folder and adding a reference to a `manifest.json` file. See section A.3 for instructions on how to use this package.

# Chapter 4
# Implementation

With the plugin designed and the basic architecture decided, it was time to start implementing the plugin. This chapter describes all major decisions and interesting discoveries that arose during this phase.

## 4.1 Language choice for the plugin

First, I had to decide what language to write the plug-in in. An option which would be convenient was to write it in C#. This, however, would prevent the plugin to be reused in other engines and would make it harder (but not impossible) to access raw OpenXR, which is required for requesting position of controllers in exact moment in time. Therefore, as noted in the design, I decided to write the plugin in a different language and compile it into a dynamically linked library (.dll on windows).

This leaves me with two possibilities: either C/C++ or Rust. I decided to go with Rust as OpenXR is inherently multithreaded API and, therefore, unless otherwise specified in the OpenXR spec, cannot rely on function calls being externally synchronised. It would therefore be somewhat difficult to make sure that every call is properly locked and there are no data races without borrow checker.

Additionally, another benefit is that Rust's async code is pretty not difficult to integrate with non-async code, which made the networking implementation very straightforward.

For the dashboard, I decided to go with the web-based dashboard which connects to the server via WebSockets. This was mostly due to my existing knowledge of web-based technologies and made it possible to create a decently usable and reasonably good looking dashboard without too much work.

## 4.2 Plugin subsystems

As outlined in the design, the implementation consists of a plugin which acts as a client and connects to a server. Furthermore, there is a dashboard, which gives the operator greater visibility into the current state of the system and control over calibration procedures. But the implementation could be divided further.

The first subsystem is a server connection system. This handles server discovery, connection, and reconnection upon failure.

The data model for device synchronisation is quiet simple: each device is owned by exactly one client, the one it is connected to. This client is responsible to send any data shape updates (e.g. if new controller is connected) to the server via reliable "configuration" channel and any data updates (e.g. controller pose) via unrealiable but fast channel. The server is then just responsible for batching and retransmission of these data. It also makes sure that it does not send data back to a client that sent it.

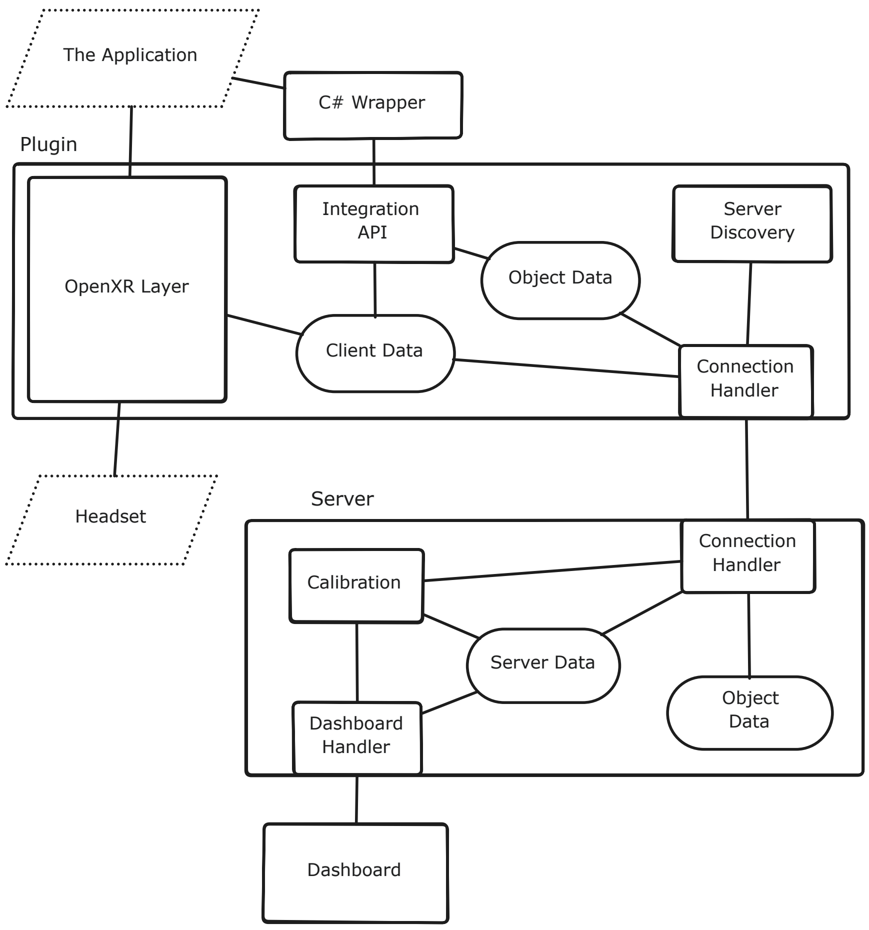


**Figure 4.1.** Diagram outlining various subsystems that make up the module.

When it comes to calibration, there are separate modules for this in both the server and the client. Dashboard sends a message that starts the calibration with some specified key parameters, and then the server takes over. The server then instructs the relevant clients to start sending samples.

When the calibration is finished, the server sends it to the client which changed position and this client applies it. When new data is sent from the client, it makes sure that it already has the calibration applied. This ensures that every part of the stack uses the same data and that the calibration is applied at source. The same coordinates on each headset represent the same point in space.

Finally, there is a subsystem for synchronising object poses. This is not the cleanest solution to this problem, but it was the most convenient way to implement this. Ideally, the app itself would synchronise its state in some way that is suitable for its usecase, maybe using the plugin as a transport layer as it already has a connection.

## 4.2.1 Server connection

To handle connection to the server I decided to use QUIC protocol[21] implemented by quinn rust crate[1]. I decided to go this somewhat unusual route because of the properties that were needed. I needed a protocol that would have connections, would have both TCP-like reliable and UDP-like unrealiable channels, and would support sending binary

[1] `https://github.com/quinn-rs/quinn`

data. And ideally, it should also have multiple separate channels and low overhead over raw UDP. I did not want to implement my own and after some searching found QUIC, which is used in HTTP3, which has all those features.

The server connection is established immediately upon starting the plugin. Since quinn requires an encrypted connection, but this application does not, I use self-signed certificates and accept any connection. If this implementation were done over the internet, this would have to change, so that we are not sending sensitive data without proper encryption. To find the server's IP address I use multicast packet on port `13161`, the port used for server and client communication is chosen automatically by the host operating system.

Once the connection is established, both the client and server start opening required one-way channels and accepting channels from the other side. To make sure that correct channels are opened, I use special message which is sent to the channel upon opening and verified on the other side. Since I want to send messages and QUIC only exposes streams (similar to TCP), I add a simple framing protocol which consists of first sending size directly followed by the data. All of this logic for opening the streams and framing messages is encapsulated in a struct so that other components can just send and receive messages. To further simplify the code, this layer also uses `serde` and `bincode` to serialise rust structs and enums.

**Possible improvements:** more robust server discovery mechanism — I could not get the server to work on MacOS. Second, some applications might need more control over the connection process. Additionally, it might be interesting to allow the application to use this existing connection to send its own messages.

### 4.2.2 Applying calibration

The calibration needs to be applied in two places. One of them is when sending data to the server, and the second is when Unity reads poses from OpenXR runtime. OpenXR allows us to construct spaces moved and rotated in relation to the reference space. This makes the case of applying the calibration when reading data for purposes of being sent to the server trivial - when new calibration arrives, construct a new XrSpace with the specified pose and use that for reading the poses going forward.

The other case is a bit more complicated as while OpenXR does allow *to construct* spaces with a specified pose, it does not allow *to modify* them. Instead, we have to detect when the app is constructing spaces that we want to be able to modify going forward and note their handles. Then hook every function that takes XrHandle and if it gets passed one of the spaces we want to modify, we replace it and pass our own space instead. Thankfully, in basic OpenXR there are only three functions that take XrSpace `xrLocateViews`, `xrLocateSpace`, and `xrEndFrame`.

The biggest trouble here is the `xrEndFrame` function, because the `XrFrameEndInfo` struct it takes can contain one of the 7 composition layer structs, each of which contains the space we are after. Ideally, I would copy the structure and apply my changes in the copy, but that is complicated here as each of those structures has different number of fields. As such, I decided to patch data that is passed from the application and then undo the patch before my function exits.

Overall, this makes sure that every part of the stack has consistent view of the world. There are alternative ways to implement this, but some of them require the translation of coordinates when transferring data between headsets. In this case, the application does not need any special logic to correctly display objects in the same position across headsets.

Note that this part is independent of how the actual calibration is computed and could be, for example, used with Shared Spatial Anchors as those also provide you with `XrSpace` handles.

**Possible improvement:** storing the calibration between application restarts. It is stored between connections to the server, but if the application itself is restarted, it has to be calibrated again.

### 4.2.3 Calibration subsystem

Before we can apply a calibration matrix, we first have to compute it. This is managed by the calibration subsystem. It consists of three parts: sample collection on the client, calibration orchestration on the server, and computation of the transformation matrix.

Once the operator specifies which controllers should be used for the calibration and presses the button to trigger it the calibration subsystem takes over. First, it sends two messages simultaneously to both clients involved in the calibration. This message has a dual purpose. The first is to inform them that they should start collecting and sending samples in specified intervals, and the second is a synchronisation pulse. Here, I assume that difference between the time it takes for the message to reach one client compared to the other is negligible. It is still better than relying on the system clock to do the synchronisation, as that is more likely to be wrong.

To collect the samples I use the `xrLocateSpace` function, which takes space and time. In case of controllers I use the action space and in case of headset I use the `VIEW` reference space. The second argument is more interesting as there I could either use the time of the collection happening or some time in the past. Here for the first sample, I use the current time, but for the subsequent samples, I use $t_0 + i * \Delta t$ where $t_0$ is the start time, $i$ is the current iteration, and $\Delta t$ is the specified time between samples (50 ms by default). This is not exactly the same as time now, because at least on Quest 2 the loop does not reliably wake up after the same delay.

Talk about delay in the loop. My first attempt consisted of collecting the sample and sleeping for $\Delta t$, which worked well on the desktop, but was very inconsistent on the quest. Instead, I switched to using `tokio::time::interval` for this, which resulted in a more consistent delay. It was still not consistent enough, so passing the correct time to `xrLocateSpace` is crucial. See figure 4.2 for a visual representation of the collected data.

After enough samples are collected, I feed them to a rust reimplementation of the algorithm described in section 2.2. For linear algebra I use a library called nalgebra[2], which allows me to port the SC's C++ implementation almost line by line. The whole calibration computation takes only a couple of milliseconds, which indicates that applying it continuously would be feasible.

When the computation finishes, the calibration subsystem sends a message to the target client (the one not used as a reference), and a new calibration is applied.

To support chaining multiple calibrations instead of having to use a single headset as a reference in all calibration (which is still recommended), I take care to differentiate between collecting samples from the reference and target systems. When collecting samples from the target system, the native Stage space is used as a base, but when collecting samples from the reference, the Server space is used instead. This means that the calibration takes into account the reference's calibration.

[2] https://nalgebra.org

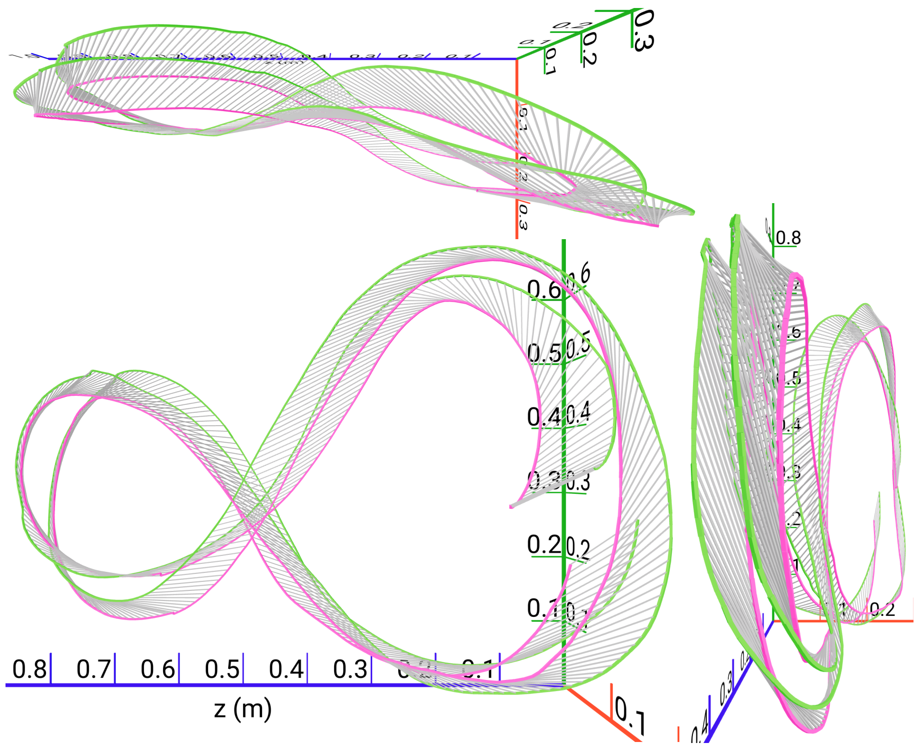


**Figure 4.2.** Diagram showing the data being input into the calibration algorithm. This 3D graph can be interactively viewed on the dashboard. The diagram shows the same data from different angles. It shows the recommended movement pattern for calibration. Coloured lines represent the origins of the controllers and grey lines represent the same moment in time. The grey lines should be of the same length.

**Possible improvements:** take into account the possible difference in time it takes the message to reach one client compared to the other. This is most evident when the server runs on the same computer as one of the clients.

Another improvement would be to support storing dependencies between calibrations – if A is calibrated to B and then C is calibrated to A, everything works. But if C were to be calibrated to A and then A was calibrated to B, it would break. This could potentially be resolved automatically and would probably need to be done in some more complicated cases of continuous re-calibration.

### 4.2.4 Collecting data about devices

To be able to synchronise anything between the headsets, I first have to collect the data. There are two basic approaches to doing this. One of them is for the plugin to act as a separate application — create its own actions and action sets and use those to collect the data. The second is to intercept calls from Unity and try to reuse the actions created by the application. I decided to go with the second route.

This has the advantage of being more clean insofar as to not adding extra actions and to make sure that software that remaps the actions does not desynchronise the remote and local set of actions. It is also more semantically correct as remote actions are semantically the same as local action. Another advantage is that, if implemented correctly, it automatically supports every interaction profile the application supports. The disadvantage is that it is more complicated.

The way it works is that I override every function which creates an action, action set, provides suggested bindings, etc. To override those correctly, I also have to override each parent handle to be able to know to which `XrSession` they relate. The good news is that all of those functions are only called at the start of the application and, therefore, are not performance critical.

Once I have the list of all the actions that the application uses, I send them to the server once as a configuration. This allows me to usually only send the data that actually change between frames and to also use integers instead of strings (index in an array) when referring to paths. The configuration contains a list of available actions, user paths, and interaction profiles.

**Possible improvement:** currently, I request data at current time. Instead, I could reduce apparent delay by estimating how long it takes to synchronise data and requesting poses for future frames.

### 4.2.5 Object synchronisation

To simplify one of the more common use cases (a use case exemplified by application implemented later in this thesis), I added one more auxiliary subsystem to the plugin. This subsystem allows synchronisation of poses of a fixed set of objects between headsets. It makes it possible to make more types of application work with just the plug-in and without adding custom networking code to the application.

The way it is implemented is that it mirrors the implementation of the device synchronisation algorithm. Each object is ”owned” by one headset at a time, and this headset is responsible for transmitting the object’s pose. The owner can give up ownership of the said object at any time, or any other client can steal ownership of the object. This model maps well to actions available to users — users can grab object by pressing trigger and release it by releasing trigger. They can also pass objects between each other by the receiving party pressing the trigger.

This setup also has the nice property that it hides the latency of the system, as the local position can be updated optimistically right away without notifying the server and the object therefore does not lag behind the controller used to move the object. The algorithm is not guaranteed to be *eventually consistent*, but will be consistent in common uses. When stronger guarantees are necessary, the developer using the plugin is encouraged to ignore this subsystem and use a different solution.

## 4.3 The Unity plugin

To make the native part of the plugin possible to use from an application, a C# code must be written. This code deals with communication with the native code and abstracts the API so that it feels more natural for use from Unity, while still preserving full control.

### 4.3.1 Rust and C# interop

The design of the plugin mandates that there are some new APIs apart from OpenXR that need to be implemented. Unfortunately, this also means that a calling convention must be established to pass data from Rust to C# and vice versa. Since it was not obvious from the start what exact data will be needed, I wanted a flexible way that would allow the passing of structured data in either direction.

One option for this would be to use JSON as that has implementation for all the languages I would need. Problem with JSON is that previous experimentation showed that it was not fast enough to be on the hot path, which my implementation would certainly be as ideally I would want to read data from the plugin each frame.

Another option would be to use the custom binary data format defined by C-like data structures, which is what OpenXR does. This is fine for OpenXR, as it is a standard and generally does not change except for a well-defined extension point. But I want it to be easier to change the protocol — without having to update multiple places in the codebase.

Enter serde-generate. Rust has a pretty nice ecosystem of serialise/deserialize libraries, which means that if you are in rust, you can just add some derive attributes to your struct and everything you need is handled for you. It is called serde[3] (serialize/deserialize) and I take great advantage of this in network communication – more about that in section 4.5.

Serde-generate[4] is a tool to inspect objects that are configured for serde (de)serialization. It then allows us to generate code in various languages to read and write the serialised data. So instead of having to manually write (de)serializers format and having to make sure that we keep them in sync in two languages, we just use the generated code.

APIs therefore just pass pointers to buffers around (plus size to prevent accidental buffer overruns), and only complexity with calling the API is making sure that we do not leak the memory.

#### Passing data from C# to Rust

As decided in the plugin design (chapter 3), with the exception of non-essential functions like logging, the plugin will always be called from the engine code. This greatly simplifies the transfer of data to the native code. Since we already have a serializer automatically generated, we can just serialise the data structure into `byte[]` and pass that along with its size, and everything just works.

#### Passing data from Rust to C#

Passing data in the opposite direction is slightly more complicated as it requires us to somehow allocate the buffer, then read it from it in C# and then deallocate it. We could solve the problem of deallocation by creating a buffer from managed C# code, but that has the problem of not knowing the buffer size ahead of time. Instead of adding a call to determine the size, which would in the general case require serialising the data twice, I decided to just allocate the buffer from Rust and pass the pointer to the managed code. The managed code then decodes the data and finally calls `netvr_cleanup` function which deallocates the buffer.

#### Data format

For the actual data representation, I chose bincode[5]. The choice was mostly made for me as one of two formats supported by serde-generate. The other one being BCS[6], which at the time of writing is a much less popular format, and its canonical representation features did not seem relevant to this project.

Bincode is a well-defined serialisation format with the only caveat being that it is not self-describing. This means that both the producer and the consumer have to agree on the exact structure specification. This is not a problem for this project, as I am not storing the data anywhere and can presume that every part of the system contains the same version of the specification. This is helped by the fact that there is only one place where the format for each data structure is defined and everything else is automatically derived from there.

[3] `https://serde.rs/`
[4] `https://github.com/zefchain/serde-reflection`
[5] `https://docs.rs/bincode/latest/bincode/`
[6] `https://github.com/diem/bcs`

#### Exported functions

As the plugin is implemented as a dynamic library, we need to define the list of functions that it exports. Some of those do not follow the bincode-based ABI outlined above, be it because of the need to pass functions, or due to them being implemented before the convention was established.

To give greater visibility into the Rust implementation, `netvr_set_logger` sets the function to be called when the Rust code wants to log something. This is in particular useful in Unity, as it allows us to redirect the output to the built-in console. This function is called first to allow for subsequent calls to be logged in.

The main function of the plugin is `netvr_hook_get_instance_proc_addr`. It takes a pointer to the `xrGetInstanceProcAddr` function and returns an alternative implementation which returns functions that wrap the underlying functions and perform the necessary actions. There is also `netvr_unhook` which reverts the effects of the hook function. This only has to be called in the event where you need to unload the dynamic library before `xrDestroyInstance` is called.

Finally, `netvr_get_fn` and `netvr_cleanup` are used as part of the bincode ABI. Get fn provides the caller with pointer to the bincode-bound function and `netvr_cleanup` does the neccessary cleanup of the output value.

## 4.4 Loading the plugin

Unity provides interfaces for loading and working with native libraries. It is based on `P/Invoke`[7] and works nicely out of the box. But it has one big disadvantage and that is that it does not allow reloading the native portion of the plugin. That by itself is not a problem for the built app, but it complicates development as I would have to restart Unity editor each time I changed the Rust code.

To workaround this limitation, I implemented a library loader. Based on the value of `UNITY_EDITOR_WIN` define it either uses normal p/invoke implementation using `DllImport` attributes, or it loads the library dynamically and uses Windows kernel32 API `LoadLibrary` to load the `.dll` file dynamically, which allows us to also unload it. Then uses the `GetProcAddress` function to get the pointer of the relevant functions.

Since Windows disallows modifying dynamic libraries that are loaded into a running process, I also copy the file to a temporary location before loading it to memory and then try to make sure to delete. There are some cases where the unload does not work properly (mostly in case of some errors), but it works well enough to be more useful than the raw p/invoke.

To have a singular place where to load and instantiate the plugin, I use unity's OpenXRFeature facilities, which allows me to register a feature which can then be turned on and off in Project Settings. Unity then makes sure that the plugin follows standard OpenXR lifecycle, which is exactly what we want. If the application wanted greater control over server connection, new APIs would have to be added, but the singleton could still be handled by this same system.

## 4.5 Network protocol

The first step to networking is establishing a server connection. To do that, we need to first find the location of the server. We could either hard-code a static IP address

[7] https://learn.microsoft.com/en-us/dotnet/standard/native-interop/pinvoke

which would be fine, but is undesirable as I potentially have to setup the testing on WiFi networks I don't have control over and therefore could not setup static DHCP leases. It would mean that I have to recompile and redeploy the application before testing. Another alternative would be to add GUI to set the IP address, but that again complicates the testing as some users in the test group will not be technical and would therefore have a hard time setting the IP address, which might mean that I have to connect each headset manually, then hand it to the users. Therefore, I decided to add network discovery.

# Chapter 5
# Test application

To demonstrate all approaches and concepts in previous sections, a simple application was implemented. This chapter contains everything about this application that was not already described in previous chapters. It follows a structure similar to the plugin design, just in a smaller scale as most of the functionality is contained in the plugin. It contains synchronised blocks, performs some critical logging and measurements, and has some useful debug functions. Additionally, this section describes the measurements and evaluation performed without users.

## 5.1 Design

Before I start implementing the application, I need to determine what I want out of it. The application has two basic goals. The first step is to act as a testbed during the development of the plugin. The second is to be used in user testing. The first goal manifested itself already in the previous chapter and means that I have to use the developed plugin. To better understand the second goal, I started by designing the experiment.

The user testing will be described in more detail in following chapter, so here I will only explain the list of features that will be needed for said testing, not how they are going to be used. There are three key features the application should support: showing realistic models of local and remote controllers, basic scene for users to be in and a few objects which can be interacted with. Additionally, it should have some additional features that are useful for development.

As most of the functionality is contained within the plugin, the application acts as a thin shell around it.

### 5.1.1 OpenXR and platform support

The application is created in Unity and uses the OpenXR plugin. As I do not need any platform-specific features and actually want to actively avoid any such features, I do not use platform-specific plugins. This makes some things slightly harder, but is worth it for the ease of porting and in general making sure that if I make something work on one platform (usually SteamVR as there I can just test in unity editor), it will most likely work on all other platforms.

This is not to say that there are *no* platform-specific things for each headset. The main difference between headsets is the model used for controllers (section 5.2), but there are other differences. On PC, the support is quiet simple as there I can just enable all supported controllers in the settings and everything works automatically — the loader loads correct OpenXR runtime, etc.

In standalone Android headsets, it is slightly more complicated as the OpenXR loader for Android does not yet exist[1]. This means that to support headsets from different

[1] This is according to PICO's documentation[22]. The OpenXR loader specification contains description of Android loader since version 1.0.20 released in October 2021[23], but it is unclear if it is actually

manufacturers, I need to build separate APKs with different settings, which is somewhat annoying. The biggest difference is in OpenXR settings — to build for Quest, you need to enable Meta Quest support and add Oculus Touch controller support, but you also need to disable PICO support and remove PICO controller profile. To build for PICO, you need to do the opposite.

If I were to be developing the app further, I would add a script to automate those changes, but since I added the PICO support late into the development, I just do this by hand. Also, the OpenXR Unity support on PICO is currently experimental. The above makes it impractical to be used in a full user testing, but I will do a reduced testing session if time allows.

## 5.2 Controller models

To be able to reasonably do the user testing that I want, the application has to display real models of controllers, and it has to display them in the correct location. This is harder than it looks as I could not find any of the shelf component which would display the model, but also allow me to display models of remote controllers. Therefore, I needed to implement this myself.

I downloaded the controllers from the Immersive Web project[24] (IW) because it was the only one, which contained the full set of controller models I was after in a convenient format and with a licence specified. This turned out to be a good choice as when I wanted to add support for PICO's headsets, the project was already updated and had those controllers as well. The project contains models in GLTF format. This is great, as there is a library that allowed me to load those models on demand – GLTFast[25]. It can be contrasted with SteamVR, which stores its models in a set of OBJ files with a bespoke JSON format, which describes where each model should be positioned.

One slight problem with IW's models is that they are made for WebXR, which in some cases uses slightly differing positions for the controllers, depending on the runtime used. To counteract this, I hard-coded offsets for the controllers of supported headsets. This means that support for new controllers cannot be added by just dropping new model into the build folder. It would be useful to extract this model loading logic to its own plugin and load the values from some kind of file, but that was beyond the scope of this thesis.

To know which model to load for the *local* controller, I consult two values: the controller name and its `characteristics`. This tells me what kind of controller it is and whether the left or right variant should be loaded. To know which model to load for remote controller, I used OpenXR equivalents of those values, which are interaction profile (eg. `/interaction_profiles/valve/index_controller`) and user path (eg. `/user/hand/left`).

As remote headsets are not directly interacted with by users, I use the same model for every headset model. This is also for practical reasons, as finding good models for each of the headsets I support turned out to be harder than finding good models for controllers. The plugin emits `generic_hmd` as a "path" for headsets, which is an invalid path in OpenXR, so it should never conflict with other devices. This should be changed if the plugin's API was ever packaged as a proper OpenXR extension.

The last consideration with models is that, ideally, various buttons on the controllers should reflect their real-world state. To be able to do this, the IW's models use standard

implemented in Quest, PICO, and Unity. This will hopefully be resolved in the future, which would mean being able to run the same apk on different headsets.

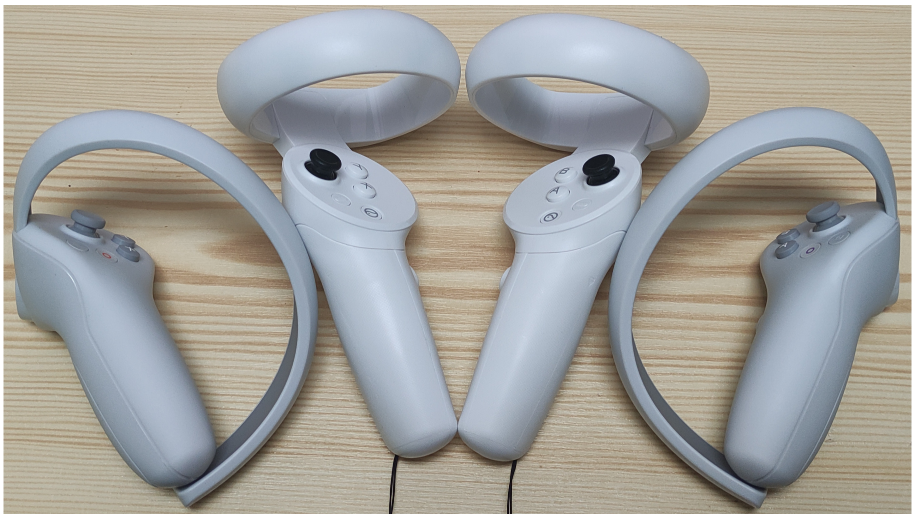


**Figure 5.1.** Photo of controllers used with PICO headsets. PICO 4 controllers are to the side and Pico Neo 3 controllers are in the middle.

names for various controls. The model loader implementation then has a lookup map between the IW's names and Unity's control names. It then wires up a component which updates the displayed objects transform according to the value of the control each frame.

One limitation of this implementation is that it relies on the fact that each physical controller corresponds to one Interaction Profile. This is true for some, but not all controllers. In the case of Vive vs Vive Pro it does not matter much as there the controllers only differ by colour. But in the case of PICO platform it is a bigger problem as there the controllers look nothing alike (see figure 5.1), but share the same button layout and therefore the application cannot meaningfully distinguish between them. From reading OpenXR spec it seems to be a limitation of the spec itself as controller models are provided only by extensions, which are not widely supported. As such, the application only shows Pico Neo 3 controllers, even when PICO 4 controllers are used. The same limitation applies to the headset models, but there it is less of an issue as users are not expected to directly interact with their headsets.

## 5.3 The scene

The scene contained within the app is very simple and utilitarian (see figure 5.2). User-visible objects are the floor, two cubes to the sides to avoid disorientation, and a table with coloured objects. All objects that do not support user interaction are white. The table is represented by a big white cylinder with height and diameter to support four adults who are standing around it and interact with objects placed on top of it.

The objects are four identical red cubes and eight spheres of varying sizes. All coloured objects have the same interaction logic: any user can grab any object at any time. The cubes are intended as a training exercise for the users so that we are testing the application and the synchronisation and not how well the users can use VR controllers. The spheres are of differing sizes and are laid in such a way that it is intuitive for the user to first grab the bigger one that is closer. Each user has their own colour of objects so that I can give instructions effectively.

The interaction logic is in more detail as follows: Each controller has an interaction zone represented by a white sphere floating in front of it. When user moves this sphere close to any object and presses trigger on their controller, they issue grab command to the plugin and the plugin starts sending updated position to other headsets. Similarly, when the user releases trigger button, the application informs the plugin that the user released the object. This proved to be an effective interaction model, maybe even too effective, as users never had any problem grabbing or passing the objects (see next chapter).

In addition to visible objects, the scene contains two other important objects, the local and remote device manager. Local device manager listens to events emitted by unity, informing the application that a controller was connected, and spawns the correct model for that model. The remote device manager periodically asks plugin for a list of objects and compares this list objects with devices in the scene. When it notices any differences, it adds or removes objects representing the device in question.

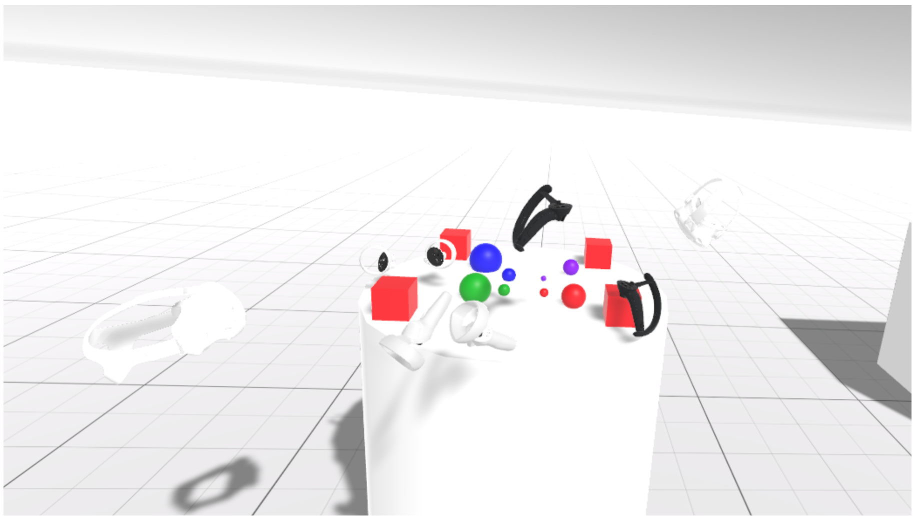


**Figure 5.2.** Screenshot from the application as a view of one of the users. This image was taken by laying headsets and controllers on available surfaces, so the positioning of headsets and controllers does not reflect the real intended usage. The objects are at their initial locations.

## 5.4 Latency measurement

As the implementation does not employ any techniques for network delay, latency becomes an important property of the system[2]. To measure this delay due to synchronisation as prescribed in the assignment, a simple method was devised. The method ensures that what is measured is the *additionaldelay* due to synchronisation. This means that it should include everything, for example network delay, possible timing issues in the implementation, etc, but also that it does not include things like tracking delay of the system itself.

### Data collection

The way it works is that I added a script in which each frame logs pose (position + rotation) of each device in the scene. This script can be triggered by pressing a button (it listens to primary and secondary buttons on all controllers, which on all controllers

[2] See attached video named `delay-demonstration.mp4` for example of how this looks like. In the video I am holding two controllers together, and the white one is local and the black one is remote. The remote controller appears to be behind the local one

I own correspond to A, B, X and Y), and upon pressing any button again, it writes the log to disk for further analysis. This means that it can be used on desktop and on standalone headsets. The data collection then works as follows: start the app on two or more headsets, setup the scenario being measured, press the button, and do motions as prescribed by the testing scenario, press the button again. The file can then be extracted from the correct folder by using `adb` in the case of Android-based headsets, or by copying the file from the folder with the app.

After collecting the data, I can visualise the collected data on the dashboard (figure 5.3) or load the file into a script that processes the data and outputs relevant statistics. As I am collecting the data without users, the script assumes that I am measuring for the set of controllers that moved the most and, therefore, does not need to have a selection for which pair of controllers to use as opposed to calibration procedure.

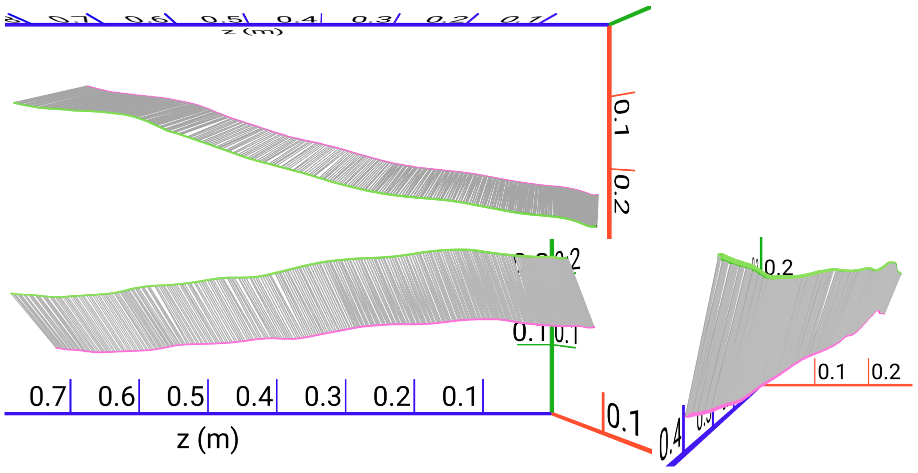


**Figure 5.3.** Screenshot of 3D visualisation of collected data available on the dashboard. Two controllers are moving in approximately a straight line. On the right is the view from the side, on the top left is the view from the top, and on the bottom left is the view from the perspective of the Quest 2 headset. Data are collected on the Index and the remote headset is Quest 2. The green line represents Quest 2 and the other line represents Index. Grey connections represent the same moment in time.

### Data collection limitations

This latency measurement method does not take into account that local samples are potentially in the future, as Unity does employ delay-hiding techniques. Local values are sampled from the position of game objects, which means that they are always sampled from the time of next frame, whereas the implementation collects samples at current time before sending them. This means that I am not comparing only the network latency, but this also adds to the measured time. However, I believe that this is the correct thing to measure, as it is the delay that the user ultimately experiences. This factor could be trivially counteracted in my current implementation, but I chose not to do so, since when I formulated this limitation I already conducted user testing and I wanted the measurements in this section to correspond with the same setup as was used in user testing.

### Procedure

The original testing procedure was as follows: The tester holds both controllers in one hand in such a way that they move in relation to each other as little as possible and moves them in a straight line horizontally with as little rotation as possible. The movement spans at least 40 cm and takes about 4 seconds. Note that those numbers are for the movement, not the measurement, as that includes some time before the movement starts and some time after it concludes. This method procedure was chosen because it can be analysed without the calibration being taken into account at all (but the calibration being done is beneficial for producing 3D visualisations).

To counteract the need for calibration, I only take distance travelled into account for analysis. This produces two curves in which the remote one is clearly separated from the local one. See figure ?? for a chart illustrating this.

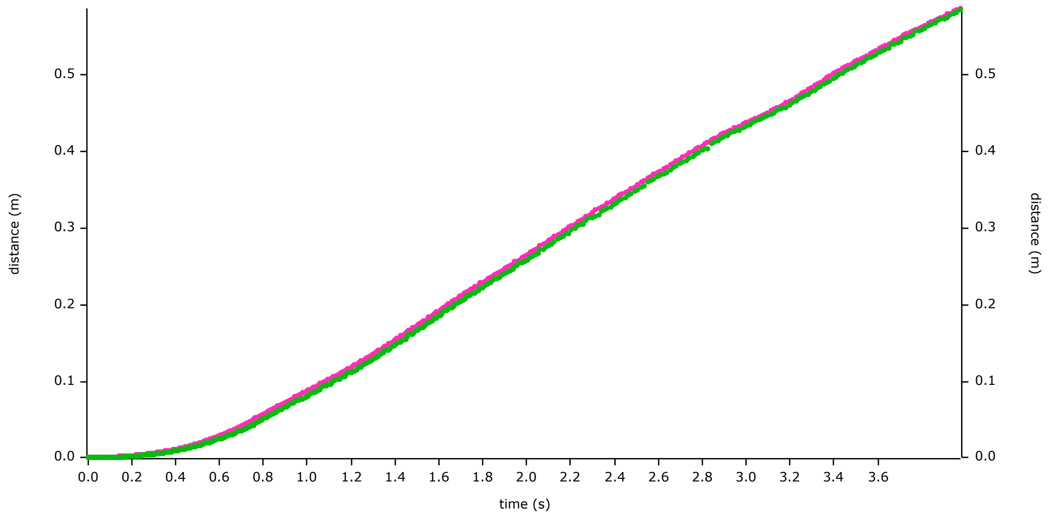


**Figure 5.4.** Chart depicting distance each controller travelled during the latency measurement process. The horizontal axis represents time, whereas the vertical axis represents distance. This chart shows the same data as in the previous figure. You can see the delay as a shift in the time domain.

The problem with this is that it does produce any significant results as any delay is obscured by differences in the tracking algorithms — cross-correlation yields zero which would indicate no delay, which is clearly not the case. Further experimentation and tweaking of the input parameters showed that differences in the tracking algorithms obscure the delay.

Instead, I decided to move the controllers forward and backward (still in a straight line). You can see the measured data on 5.5. To establish the delay, I measure the time between the peaks of the graphs. This yields between 3 and 5 frames, which corresponds to  55 ms.

## 5.5 Accuracy measurement

The second quality of the system I want to quantify is the accuracy of the whole system that was implemented. To do so, I would like to compare the system with a reference stable tracking system without drift. Both Quest's insight and Index's Lighthouse tracking system can be considered accurate for purposes of this thesis[26], but only Lighthouse is stable insofar that it does not move, e.g. across device restarts, sleep or even furniture move. As long as the base stations are in the same positions, the system does not move. Quest does not have such guarantees.

Since both systems provide good positional accuracy, it makes sense to use the same method as was used previously with tying two controllers together and measuring the

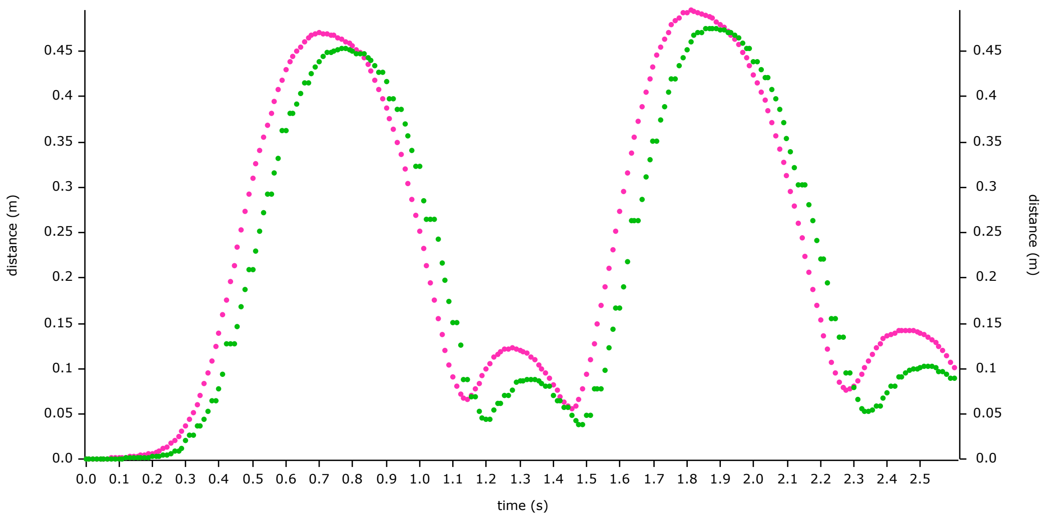


**Figure 5.5.** Modified delay measurement process. The chart shows the distance of controllers from the starting point in time as observed by one of the headsets.

difference in the distance between them. But, as with the previous two measurements, the details are very important. The two important factors here are: we have to take samples at the same moment in time as otherwise we would measure the effect of network delay and that was done in the previous section. The second important factor to consider is to measure the position in server space. The first part is identical as with the calibration algorithm, the second part is similar to the delay measurement.

With this in mind, data collection was implemented similar, but not identical, to the calibration process. The user can trigger this data collection from the dashboard and finish it on the dashboard. Advantage is that the same UI can be used for visualising the collected data, but extra care needs to be taken not to use these data for calibration, but instead use the actual calibration process that is intended to be used.

With the data collection processes implemented, it was time to collect data. For this, I wanted to see how data collection would work in two different scenarios. One scenario entails somewhat chaotic movement across the available space, and the second is as orderly and controlled as possible to give all the interacting systems the best chance. Both scenarios tried to cover widest possible range of motion I could in the space available, that would still work in that given scenario.

For the orderly scenario, I decided to mount both controllers on a stick and rotate them around a fixed middle point. Rotating office chair acted as an anchor. To make sure that the Quest controller was properly tracked, the controller was mounted in such a way that the tracking ring was visible from the chair and I was wearing the headset the whole time. The lighthouse-based index controller does not require this and, as such, does not require this special treatment. The stick used was telescopic, which allowed me to do measurements of varying diameters. The resulting measurement can be seen on figure 5.6.

It is worth noting that in the case of perfect tracking system and imperfect calibration a consistently changing gradient would be expected. On the other hand, even with perfect calibration, if one or both of the tracking systems were inaccurate, the colour would still change. If both the calibration and tracking were perfect, a solid colour corresponding to the real distance between controllers is expected. Another point worth

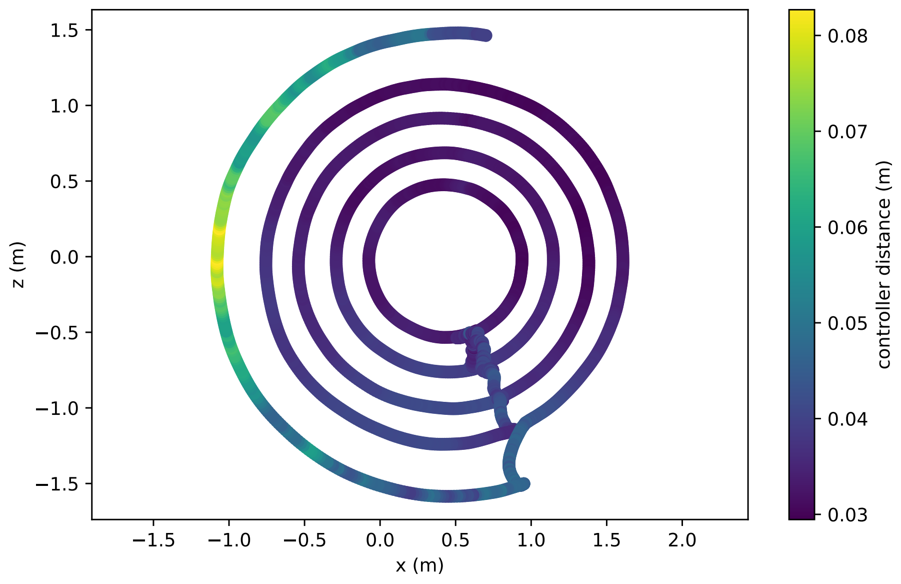


**Figure 5.6.** Positional accuracy measurement using the orderly scenario giving the tracking systems the best chance.

mentioning is that Lighthouse tracking is less accurate in the corners, and Quest is less accurate with increasing controller distance from the headset.

The second scenario is less controlled, more chaotic, and is also more realistic. Here, I walk through the whole room, back and forth. The result can be seen in figure 5.7. You can find additional images related to both scenarios in appendix B.

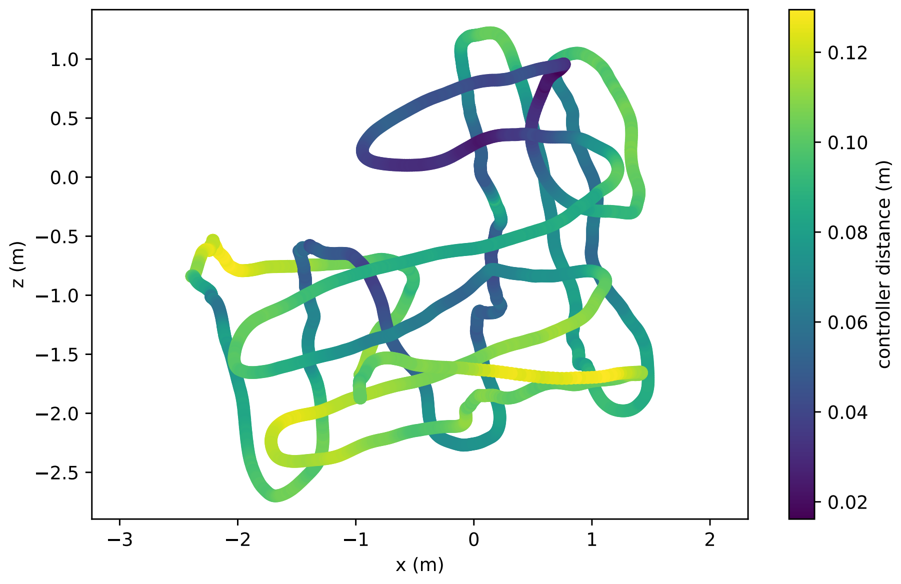


**Figure 5.7.** Results of positional accuracy measurements in the case of complete coverage of the tracked space using more chaotic movement patterns.

As you can see in the images, the baseline is very good, especially around the centre. For the circle scenario, a slight gradient can be observed towards the negative z axis. With the more realistic scenario, the picture changes, and the results are less consistent. You can see that the distance between the controllers sometimes changes dramatically. But it is worth noting that the same positions in the physical space do not have visibility to both lighthouse base stations, which means lowered accuracy (the space is L shaped).

Overall, the result is that the system is not perfect, but works pretty well and is definitely usable for some use cases.

## 5.6 Miscellaneous

The application also contains few other things, some of which are leftover from the prototype and some of which were useful during development of the plugin but are not really used anymore.

There is an alternative implementation of Unity's pose driver. This is useful as an easier to use abstraction over Unity's overcomplicated API for interacting with XR devices. Further, it has an editor drawer, which allows you to inspect the controller values in real time without having to open the dedicated inspector (which was broken in some versions of Unity, now it seems to work correctly).

Apart from those features, the application is intentionally kept minimal to show that the plugin indeed provides all necessary functionality to implement a multiplayer VR experience in a shared physical space. See Appendix A for instructions on how to build and use the application.

# Chapter 6
# User testing

To make sure that everything works as expected and that no important variable was overlooked, we need to test the application with real human people. The main priority of the experiment is the safety of the participants in case of any glitches. Another requirement is to take long enough for any drift.

## 6.1 Experiment design

The experiment is designed to test the application implementation in ways that are not possible with only one person and to find any unforseen problems in the design. I also want to try and see how much calibration errors affect the users.

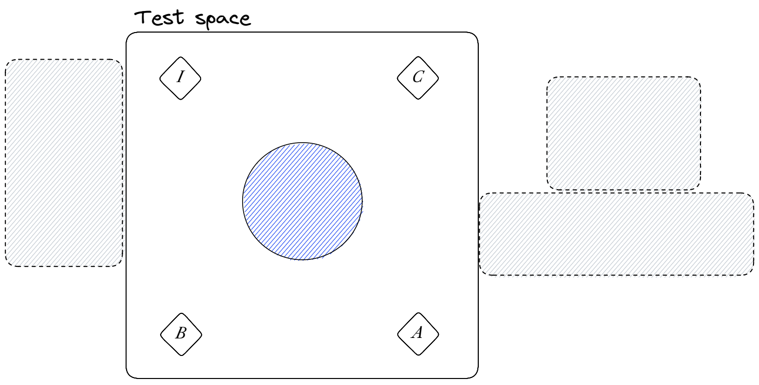


**Figure 6.1.** Simplified diagram of the test space. The big square shows the space in which the users will be present with diamonds representing their staring position. Dashed rectangles show closest obstacles around the space, and the blue circle represents a virtual table used for the second part of the experiment.

In figure 6.1 you can see the layout of the room in which I am working. I decided to go with square space for the users, and the size is limited by furniture present in the room (dashed rectangles to the sides represent said furniture).

There are a few things I want to examine with users mainly focused on things that cannot be tested without users[1] so as not to waste limited time. There are two main categories of problems with the implementation that could affect users: imprecise positioning and extra delay caused by synchronisation.

To test those, a few tasks were devised. I tried to focus on interactions between users, as that is what is special here. Interactions can range from interactions at distance, to short-distance interactions, to direct interactions.

[1] eg. delay due to synchronisation can be measured without users, but the effect of it on perception can't

## 6.2 Experiment script

For testing, I have three units of Meta Quest 2 and one Valve Index. As Index is wired, I have to keep in mind the cable when designing the experiment. The following script will talk about users using letter ABCI, where users A, B and C have Quest 2 and user I has Valve Index.

Before the experiment starts I will turn on the headsets, making sure to turn them on in different positions. After they are turned on, I will setup their respective boundaries (room setup for SteamVR and Guardian for Oculus) to make sure that users do not walk into a wall or furniture. Each headset will be prepared on the ground in place where the users should start along with *one* controller when testing begins.

The following are the exact steps which will be taken. In *cursive* are written verbal instructions to the participants. In normal text are notes which explain more details about each step and also contain information about what to look out for.

**Step 0.** *Welcome and thank you for agreeing to participate in the research...*

If the headsets are not on at this point, I will turn them on now.

Do a quick introduction to the research, what it is about, and make participants at ease. Once everyone arrives and is introduced, we can start with the experiment. Before the experiment starts, say that when something goes wrong in a way where they do not feel safe (e.g. think they are in danger of running into other people), they should report it immediately. If they have any other observations, they are encouraged to report it immediately, but they could also remember it for later reporting. Lastly, I will ask them if they agree to me recording the experiment so that I don't have to make as many notes in the moment.

**Step 1.** Entering VR

Now I assign each participant their headset and tell them to stand in their starting corner according to figure 6.1 and also tell them their letter. Then I demonstrate to them how to hold the controllers, where is the trigger button, and also where is the system button so that they do not press it by accident and I do not have to pause the experiment to help them get back to the test app.

When all explanations are finished, I help them put on the headsets and make sure that everyone has their headset dialled in so that they can see clearly.

**Step 2.** Calibration

After each participant enters the VR, I instruct them to open the app. They are instructed to open the app one by one, so that I can assign names to the headsets in the dashboard. For Valve Index I start the app myself, as there it is easy to do[2].

After the users started the app, I take the controller from a pair of users and perform the calibration procedure. I instruct the Quest 2 users to watch their controller to ensure a good calibration. They are also instructed to report when the calibration finishes. When all of this is done, I ask them if they see each other and if other headsets are moving to make sure that everything is still working.

Calibration will be performed in the following order: I will be used as the reference point for the server space as it should always have its origin in the centre of the test space. Then I will calibrate A to I and B to I – this is how I envision the system to be used under ideal circumstances. Every headset was calibrated to one central ground

[2] I *could* start the app even on Quests, but there I would have to setup adb over WiFi and I found that it is not worth it in this case

truth. Finally, I will calibrate C to B – this makes it have three hops of calibration to A and should therefore exhibit largest calibration error. The positioning and testing order, along with diagrams for the following steps, can be seen on figure 6.3.

**Step 3.** Waving

After everyone is ready and they see each other, the experiment can start. *Okay, everything seems to be working, we can start. Participant* I*, please move your controller up and say the word up at the same time, then move the controller down and say the word down at the same time. Repeat a few times.*

Then I proceed to call the participant I and ask everyone else in turn the following: *Participant* A*, did you see the participant* I *waving? Was there a noticeable delay? Did the place where the sound comes from match the direction from which you see the participant* I*?*

This step is designed to show that the position is approximately correct and that the delay is not extremely high. I repeat this until everyone waved. The order of the participants who wave is IACB.

**Step 4.** Passing controllers

*Form pairs,* I *with* C *and* A *with* B*, and move towards each other. Once you are close, stop.*

This creates a pairing of participants with two calibration steps away from each other.

*Now* I *will offer the controller to* C *and* A *will offer their controller to* B*.* C *and* B *will accept it.*

This step is designed to test whether the user can grab the controller even with imprecise positioning. I should observe and note how easily the group performs this task.

Q: *Have you had any trouble with this task?*

*Return the controller*

As in the previous step, notice any trouble and ask how well they are doing.

Then repeat the pass the controller test but in the opposite direction (B and C offer and I and A accept).

Q: *Are you right or left handed?*

**Step 5.** Touching controllers

*Now try touching your controllers with each other.*

This step reveals even minor divergences in the positions of the controllers.

Q: *Do the visual positions of the controllers match what you would expect? If not, try to estimate and describe the difference.*

**Step 6.** Different pairing

*Pair* A *with* C *and* B *with* I*.*

Then guide them through steps 4-5 again, A and B offer first. After it is done, return to your starting positions.

**Step 7.** Explaining the interactions

*In the middle of the room there is a table in the shape of a big cylinder with coloured objects on top. Stand around it.*

This sets up the stage for the final part of the experiment in which the users will try handing objects to each other. First, I have to teach them how to do it and make them comfortable with the control scheme (as I want to test the synchronisation and not VR interactions in general).

*Now everyone will grab the red cube in front of them. Use the trigger button to do so*

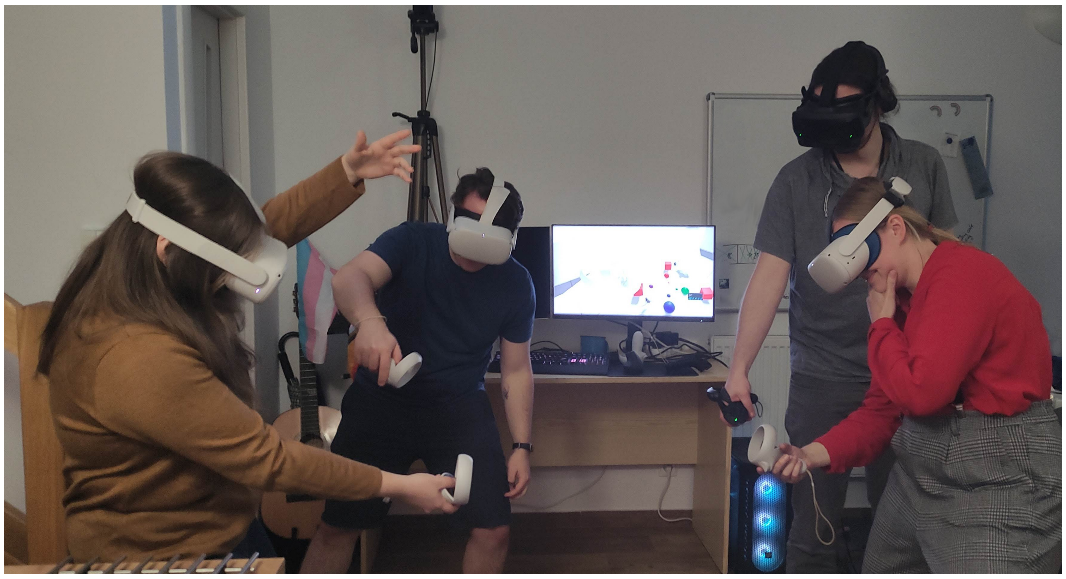


**Figure 6.2.** Photo taken during the third test session with participants standing around the virtual table.

This step is there to make them comfortable with the control scheme. If they have problems, help them, and maybe instruct them to try again. If something goes wrong, reset the positions of the spheres from the control panel.

**Step 8.** Passing objects
*Participant* I*, which should be closest to the blue ball, will take it and pass it to the person to their right (participant* B*).*
Once done, instruct them to pass it around the table until it gets back to the original person. Observe and note any problems. If you have problems with passing the object, let them try again.

**Step 9.** Instruct the participants to do the same with the smaller green ball, then the red ball, the purple ball, and with the smaller versions of each.

**Step 10.** Try touching the controllers again and report the error now. Is it different from when you last tried it?

**Step 11.** *Thank you, that is all in virtual reality. Please take the headsets off and remember that the virtual tables are not real; therefore, you cannot put controller or headsets on top of them.*
Experience shows that after some time spent in VR, people forget that some objects do not have real correspondence and will try to use virtual objects for support. This sometimes results in dropped controllers.

**Step 12.** Ask the participants some final questions and reward them for their time (figure 6.5).

If I feel like the following questions were not answered adequately during the test, I will now ask the following questions.
**Q** Did you notice at any point after the calibration was finished that the virtual representation of physical objects diverged from reality?
**Q** If yes, describe when and by how much.
**Q** How did the absence of tracked hands affect you?
**Q** Did you notice anything else?

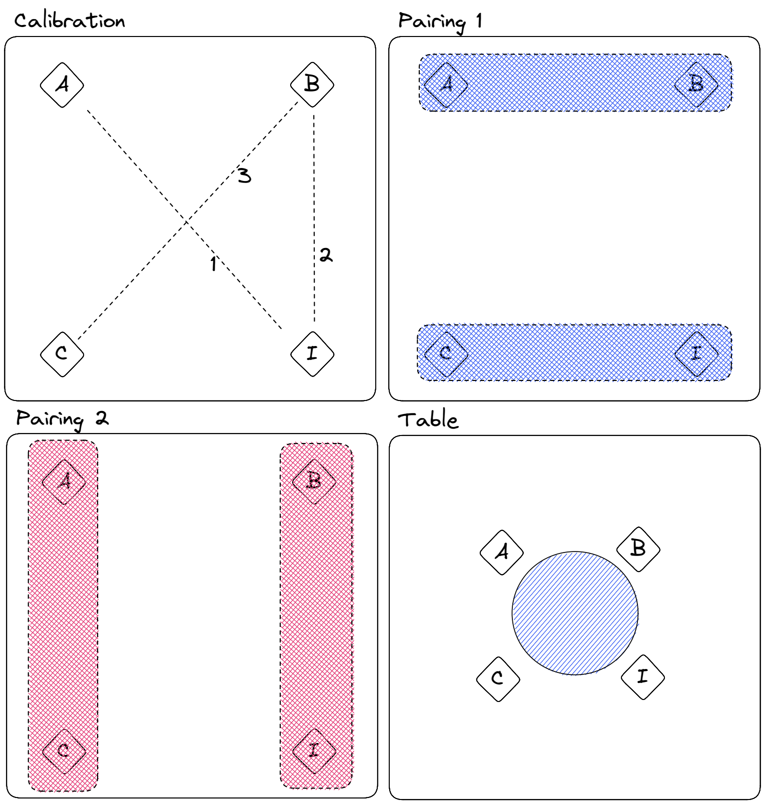


**Figure 6.3.** Diagram of four main parts of the testing sequence. First, the square shows the order of calibrations taking place, the second two squares show the pairing for pair activities, and the last square shows how the users are standing around the table.

## 6.3 First group

This test session was conducted using a test script very similar, but not exactly the same script, as you can see above. It was mostly the same, but with less detail, which turned out to not be optimal. I will call out the differences where relevant to the description.

**Setting up and starting** After explanations were finished, we proceeded to boot the Quest 2's (the Index was already up and running); this turned out to be a mistake as one of the units did not boot up correctly. Trying to debug the issue, I proceeded to forcefully turn off and back on. After that did not help, I decided to proceed with just three users instead of four. The headset did eventually turn on (before Step 7) and we redid the steps that the user missed. For clarity, I will reorder the events so that they match the steps outlined in script above.

Then, we started calibration. At first I made the mistake of not properly showing the controller to the user, which, of course, made the calibration imprecise. After that I rectified it. There was one more calibration that resulted in an error, and upon reviewing the video recording it seems like it was caused by sudden movement of one other participant during the calibration process. I want to examine this further in future testing sessions, probably as a bonus step after the main testing ends.

**Step 3, waving**. After completing the instructions, the wave went without major issues. Users did not report any perceptible delay.

**Step 4, passing controllers.** At this point my phone on which I was recording ran out of memory and I do not have this recorded so I cannot report more detailed info, but it went without major problem. The biggest issue here was with users not seeing their hands, only controllers, and so it always took them a bit of trial and error to grip the controller where the other user was not gripping it.

Another issue occurred for the Index controller where the users tried to use the tracking arch as a grip, which resulted in loss of tracking. Interestingly, this did not happen for the Quest touch controllers even though those are smaller and I would expect it to happen more there. See figure ?? for illustration. They held their own controllers in their right hand, and so they were gripping the received controller with their left hand. I did not ask if they were right or left handed.

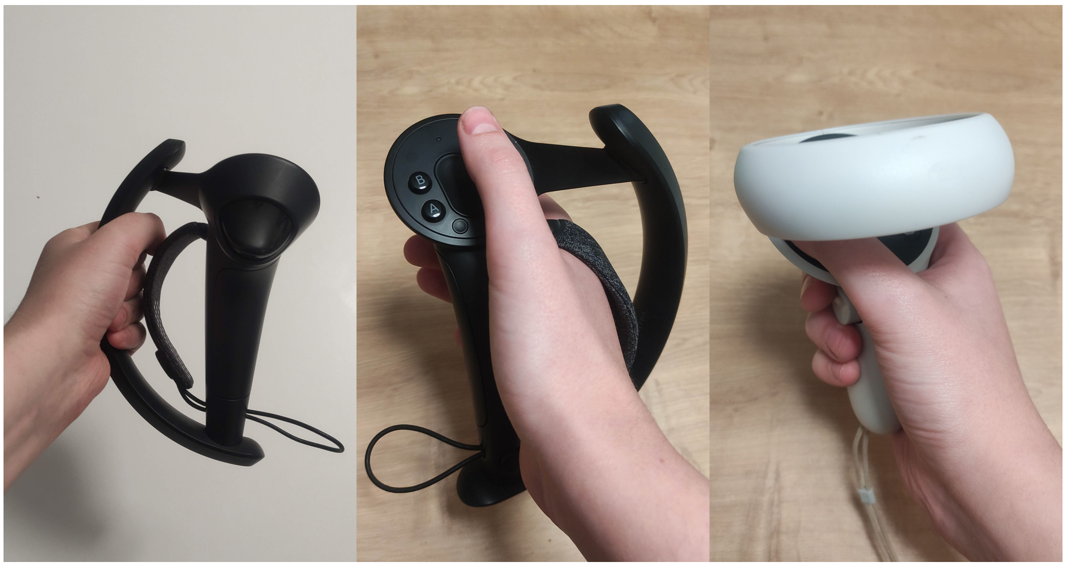


**Figure 6.4.** Controllers being held. On the white background, you can see how users tried to grip the Index controller when receiving it. On the brown background you can see how Quest and Index controller are supposed to be held.

**Step 5. Touch controllers**

With this step, they did not have major problems. They saw some misalignment, but since I do not have the recording I do not know what exact misalignment there was. However, it was not so severe that they collided unexpectedly though.

**Step 6. Repeat.** Repeating the previous steps did not reveal any major differences, and therefore serialising the calibration does not seem to cause major problems and it seems that even if the calibration was less precise, or only used SLAM tracked headsets were used, the system still works.

**Step 7. Explaining the interactions** Here we stumbled upon another roadblock insofar as the objects were not synchronised between headsets. This meant that we had to pause the testing while I investigated the source of the issue. It turned out to be caused by one of the headsets having an older build of the app, which caused it to not upload the list of the objects.

Once this was solved, we resumed the testing, but as the app currently does not store the calibration across restarts, we had to redo the calibration process. This time it went without problems.

After the second round of calibration was finished, the users blazed through the task of passing the objects without issue. They went through it with such ease that I decided to make it harder for future testing sessions by reducing the distance from which they could interact with the objects — the first version contained a grace distance, so even if the users were not exactly precise, they could still interact with the objects.

Step 10 and the questionnaire were not included in the script for this testing session.

**Further observations.** One last observation is that when everything was working, the users seemed to be enjoying it and outside of doing tasks they were interacting by doing various gestures. But while I was debugging the issues which resulted in them being alone in the space they reported being bored. At one point when one calibration went particularly wrong, one user reported discomfort.

One other issue that appeared while testing the app was that the internet was cut out for a bit. This shouldn't be a problem as everything is running locally, but it resulted in the server computer deciding to try to connect to a different WiFi access point, which is on the same LAN, but the broadcast packets used for server discovery do not reach it. This continued working until we had to restart the apps due to the synchronisation problem, after which the clients couldn't find the server until I manually changed the WiFi connection back. But it also shows that either the reconnection logic or QUIC connection migration works correctly.

There was also a moment when users reported glitches (other headsets stopping moving and then disappearing), which I attribute to above-mentioned WiFi issues. But they quickly resolved themselves. Additionally, those issues prompted users to stop what they were doing, which is a good sign for their safety.

**In conclusion.** When everything is done correctly, the application works. The calibration works correctly when fed reasonable data. The calibrations were either obviously wrong, or the error was imperceptible. There was no noticeable major drift in the procedure and the calibration is simple and fast enough to be used in groups of four.

Interesting observation is that the calibration algorithm could be improved by adding a post-step which would reject obviously-wrong calibration result based on whether the up resulting up vector is similar enough. This would prevent the mentioned discomfort the user experienced due to wrong calibration.

## 6.4 Second group

The second group's testing went much smoother. The group was less technically inclined, which made setup steps like opening the app less smooth, but once everyone was inside the experience the differences mostly disappeared. I unfortunately had problems with recording again — this time I was recording on my laptop which has enough space, but I did not check the microphone that was set in my recording software, and as a result lost all audio. But other than that, we had no major problems while testing.

**Entering VR and calibration.** For this test run, I had new and improved script and room diagrams, which made it much easier to know which headsets to calibrate to each other. Calibration went without major issues. I probably occluded vision from one of the headsets when calibrating C to B (as will become apparent easier), but the calibration process did not bring anything of note. Everything worked as expected on the first try.

I made a minor adjustment to the procedure by only asking them to enter VR when I wanted to calibrate their headset. This was to reduce the amount of time spent waiting, standing around, and doing nothing, which paradoxically did the opposite. Most of the time was spent on users entering the VR experience and could have been much smoother if I started the experience directly and then passed them the headset back.

The whole setup + calibration process took almost 12 minutes for three participants. I count the participant I separately, because there I started the application for them from the PC. From putting the headset on to starting the application took 2:51, 5:10 and 2:00, respectively. The outlier of more than five minutes was caused in part by the group chatting, which I did not want to interrupt, and in part by a mismatch in the controllers, which meant that we were trying to use controllers from other headsets. The calibrations took 30, 45 and 40 seconds, including setup on the dashboard, taking the controllers from participants, and returning them back. All of the measurements are approximate, as exact time for each task is hard to determine without audio.

| Participant | Time to enter the app | Calibration time |
|---|---|---|
| A | 2 min 51 s | 30 s |
| B | 5 min 10 s | 40 s |
| C | 2 min 00 s | 45 s |

**Waving.** In the waving step, participants did not report any perceptible delay.

**Passing controllers 1.** In the first pairing, participant C had noticeable trouble grabbing I's controller. Interestingly, participant I had less trouble receiving the controller of C. The other pair did not have any trouble with this task. When asked, no participant reported problems with taking the other controller, but when asked a more direct question of *was the controller where you expected it to be in space*, the participant C replied that it was not, but that they had no problem adapting to this discrepancy.

**Touching controllers.** Pairing A to B reported an error of approximately half a centimetre, while pairing C to I reported an error of a few centimetres, depending on the orientation of the controllers.

**Passing controllers 2.** This time everything went without bigger problem, but when grabbing the controller, it was visible that pairing A and C expected the controller to be in a different place. When asked, they did not report any problem, and since I did not notice the attempt to reach to empty air when watching them in person, only from recording, I did not push further.

**Touching controllers 2.** The results of this were similar to the first pairing – a very good match in pairing B to I and an error of a few centimetres in pairing A to C.

In conclusion to the first section, I would say that it is visible that the calibration works well when the input data are good (A, B and I), but even when the offset is noticeable under scrutiny, the participants did not have much trouble adjusting to it (this will become even more apparent later in the testing).

Lastly, I asked them if they would find it helpful to see their hands during the tasks, and they reported that while it was strange to not see their own hands, it did not make much difference. But for the task of passing the controller, it would have been helpful to see the other participants' hands to avoid unwanted touching other people's hands.

**Passing objects.** During this series of tasks, the participants did not have any major issues. In the end, the task was still too easy to show anything really meaningful. The only interesting observation is that this task was not affected even by a higher error in the calibration of the participant C.

**After completion.** After we finished, I informed the participants that the testing is over, but if they wanted, they could stay in VR and interact with the objects, since they seemed to be having fun. They tried various ways to try and break the application like sticking their head into the virtual table (it was the first time in VR for some of the participants). After some time, they started to make sculptures from the objects available.

After a bit of time I decided to give each participant the second controller but purposefully gave them the wrong one. Interestingly, they did not even notice at first, and only way they noticed was when they noticed a mark that is only visible on remote controllers. They did not have difficulty using the controllers, even with a wrong calibration (appendix B). The only point in time where they had trouble using the other controller was when all participants except B were looking at the ground because they were sticking their head into the virtual table. Meanwhile, the participant B, who was the tallest in the group, was busy collecting all objects as high as possible to keep them out of the reach of other participants. This was quiet surprising and shows that it is not a big problem to use different participant's SLAM-tracked controller in a setup where all participants are standing around a table.

When adding the other controllers, this was the only moment when the participants reported any kind of freeze in tracking. I suspect that this was caused by connecting the second Index controller (the quest controllers were already on, just not in participants' hands), as that requires transmitting more calibration data. If the plugin were to be deployed in any production capacity, this freeze should be examined.

**Conclusion.** In conclusion, the second testing session showed that for working with purely virtual objects, the setup works well enough, but for working with real tracked objects, hand tracking would be useful. Additionally, it showed that the calibration holds for at least 30 minutes, which is the time it took from the calibration being finished, until the first participant decided to exit VR.

**Figure 6.5.** A shiver of plush shark toys (blåhaj) given to the participants as a thank you for participation in the testing.

## 6.5 Third group

For the third group, the hope was that everything was well prepared and tested from the previous session. This would mean that the third group should show how the application would behave with an experienced operator.

#### Entering VR

This time, I decided to first let them enter the virtual reality world and then only perform the calibration. It took about 9 minutes for all participants to enter VR and start the application. It could have potentially been faster by about 3 minutes if I didn't need to rename the headsets in dashboard, which made it necessary for them to start the application one after another. Therefore, about 6 minutes for 4 participants represents the best-case scenario for the time to enter the VR application with one operator helping them.

This figure is important context for the time spent calibrating the headsets to each other, as this time is unavoidable, whereas the calibration time could potentially be improved with different methods.

#### Calibration

The calibration went very well with this group. Everything worked on first try and the calibration took about 140 seconds to fully complete. It could have been slightly faster if I were calibrating all Quest headsets to a central Index headset for example, which could have reduced the amount of time it took to find controllers in the dashboard, taking them from users and returning them back. But it means that in this case the calibration makes the time to enter the experience longer by about 33 %, which is considerable increase. It would be interesting to experiment with storing the calibration so that the headsets would only need to be re-calibrated if the offset gets too big.

**Waving.** Users did not report a perceptible delay for this kind of interaction.

**Passing controllers** Participants did not report trouble passing controllers, nor was there any problem or retry visible from observing them. ”The controller was exactly where I tried to grab it”. The second pairing went much the same.

**Touching controllers** Pairing of participants A and B reported an error of about 1 cm in one axis and none in the other two axes. The pairing of participants C and I reported slight overlap, and there due to the different shape of the controllers it was more difficult to determine more exact measurements.

With pairing C with A, the participants again reported an error of about 1 cm in one axis and none on the others. This makes sense, as any error between A and B should be passed to A and C due to the order in which they were calibrated. As the error was described as ”basically the same”, this would imply a very good calibration between B and C.

**Passing objects.** This time passing objects between participants went flawlessly. One participant reported a glitch at one point, but upon reviewing the video recording, it turns out that the supposed glitch was caused by one of the participants abruptly moving their controller.

Only new observation in this section was from one participant who noticed that when other participants were handling objects, the objects were not perfectly glued to the controller. This could be easily explained by the fact that the remote objects and controllers are synchronised separately and could therefore have slightly different delay and if one datagram happens to arrive on one frame and another datagram on another, there is visible delay. The participant who reported this was using Index headset, which was running on the same computer as the server (it is unclear if this makes a difference, but I wanted to note it in case it would).

**Free interactions.** Again, I left the participants to do what they wanted after the testing session ended. I gave them the wrong controllers again to see how they would use them and if they would notice the delay. They immediately noticed the marker that tells them that the controller is supposed to be used with different headset, so they knew that they were using nonlocal controllers.

This group reported latency when using remote controllers and trouble using the left controller to interact with objects. The trouble with interacting with objects turned out to be mostly caused by missing sphere, which indicates where is the interaction zone of the controller (as I did not expect to be testing this use case when designing the application). Interestingly, once they stopped examining the latency directly, two of the participants ended up only using the local controllers, and the other two used both controllers interchangeably when the group decided to build a snowman.

This means that if the latency could be avoided by using local controllers, it should be avoided. If the design of the application requires changing which controllers the user interacts with, or, for example, using remote VIVE trackers, it would be useful to implement more latency-hiding techniques. One such technique would be to estimate the latency and request position in the future from the runtime instead of current position when collecting samples to be sent over the network.

## 6.6 Testing conclusion

The testing showed that the current application works and that the current implementation is suitable for many use cases. However, if the use-case for the plugin requires users directly using objects tracked by remote system, the implementation would have to be improved. The implementation is good enough for taking and passing along the remotely tracked objects, but is not good enough for using those objects for doing fine-grained interactions.

The testing also showed that it is inherently more fun to do things as a group and even though the environment itself was not very engaging, the participants ended up having for at least part of the testing. Some participants even expressed excitement about the possible uses of VR technology. The biggest stated obstacle here is that the headsets are not comfortable enough for prolonged use.

**Figure 6.6.** One of the participants mentioned that it would be very cool to use this for a DnD campaign.

# Chapter 7
# Conclusion and future work

During this project, a complete system for creating multi-user VR experiences was designed and implemented. Most of this work was concentrated on creating an engine-agnostic OpenXR-based plugin, with special interest in Unity. Both qualitative user testing and quantitative testing without users was conducted and demonstrated that the system works well enough for user-to-user interactions.

User testing did not demonstrate any major issues with the implementation and showed the willingness of users to ignore larger discrepancies should they arise. Quantitative testing showed latency of 50 ms under stress and a positional error of between 3 and 7 cm, depending on the quality of the base tracking..

Apart from the core synchronisation and calibration systems, other supporting subsystems were developed, including dashboard, visualisation, logging, and analysis utilities. All the systems were integrated into one coherent application which demonstrates the capabilities mentioned above and was used for the testing. This work was also presented at the 25th Bilateral Student Workshop CTU Prague and HTW Dresden.

As this work had to implement most of those systems from scratch, there is a lot left unimplemented or unexplored. An area of interest would be general improvements in implementation, which could potentially improve latency. When there are improvements in the latency, it should also be possible to modify the system in such a way that the latency is hidden by using position prediction.

The calibration itself has multiple possible areas for exploration. An interesting approach would be to explore how to determine how much calibration is enough. Alternatively, if a lighthouse-based tracker device was attached to SLAM-tracked headsets, the calibration could be done continuously. Another possible area of exploration would be to better measure the parameters of the system, in more contexts, with different combinations of headsets, etc.

Lastly, a significant part of the implementation deals with integrating with OpenXR. This part could be reused for creating a platform-agnostic logging layer, which would be very useful but is not currently openly available (there is one for Meta's headsets, but not a platform-agnostic one). Rudimentary support for this was even implemented along with the rest of the software.

# References

[1] SUTHERLAND, Ivan E.. A head-mounted three dimensional display. *Fall Joint Computer Conference*. ACM Press, December, 1968, pp. 757–764. Available from DOI 10.1145/1476589.1476686.

[2] JUNG, Shinyong (Shawn), and Jiyong JEONG. A Classification of Virtual Reality Technology: Suitability of Different VR Devices and Methods for Research in Tourism and Events. In: *Augmented Reality and Virtual Reality*. Springer International Publishing, 2020. pp. 323–332. Available from DOI 10.1007/978-3-030-37869-1_26.

[3] VALVE. *Steam Hardware & Software Survey: December 2021*. [cit. 2022-01-14]. Available from `https://store.steampowered.com/hwsurvey/`.

[4] *Oculus Quest 2*. [cit. 2022-01-03]. Available from `https://www.oculus.com/quest-2/?locale=en_GB`.

[5] [cit. 2023-03-28]. Available from `https://www.picoxr.com/magic/eco/runtime/release/63eda4b428f1c3044230f7d3`.

[6] LI, Justin. *OpenVR Space Calibrator*. [cit. 2022-01-03]. Available from `https://github.com/pushrax/OpenVR-SpaceCalibrator`.

[7] YE, Shen. *Introducing New Features for VIVE Focus 3*. [cit. 2021-11-13]. Available from `https://blog.vive.com/us/2021/11/11/introducing-new-features-vive-focus-3/`.

[8] KABSCH, W.. A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A*. Sep, 1976, Vol. 32, No. 5, pp. 922–923. Available from DOI 10.1107/S0567739476001873. Available from `https://doi.org/10.1107/S0567739476001873`.

[9] *OptiTrack for VR*. [cit. 2022-01-03]. Available from `https://optitrack.com/applications/virtual-reality/`.

[10] LABS, DIVR. *Golem*. [cit. 2022-01-03]. Available from `https://www.divrlabs.com/cs/golem-vr`.

[11] REIMER, Dennis, Iana PODKOSOVA, Daniel SCHERZER, and Hannes KAUFMANN. Colocation for SLAM-Tracked VR Headsets with Hand Tracking. *Computers*. 2021, Vol. 10, No. 5. ISSN 2073-431X. Available from DOI 10.3390/computers10050058. Available from `https://www.mdpi.com/2073-431X/10/5/58`.

[12] *Monado's hand tracking: hand-waving our way towards a first attempt*. [cit. 2023-05-08]. Available from `https://www.collabora.com/news-and-blog/blog/2022/05/31/monado-hand-tracking-hand-waving-our-way-towards-a-first-attempt/`.

[13] OCULUS VR. *Oculus All In on OpenXR: Deprecates Proprietary APIs*. Available from `https://developer.oculus.com/blog/oculus-all-in-on-openxr-deprecates-proprietary-apis/`.

[14] *OpenXR homepage.* [cit. 2023-04-04]. Available from `https://www.khronos.org/openxr/`.

[15] Unity Technologies. *Unity Real-Time Development Platform.* [cit. 2021-11-16]. Available from `https://unity.com/`.

[16] *Unreal Engine.* [cit. 2023-04-04]. Available from `https://www.unrealengine.com/en-US`.

[17] Jerald, Jason. *The VR Book: Human-Centered Design for Virtual Reality.* Association for Computing Machinery and Morgan & Claypool, 2015.

[18] Fette, I., and A. Melnikov. *The WebSocket Protocol* [Internet Requests for Comments]. Available from `http://www.rfc-editor.org/rfc/rfc6455.txt`.
`http://www.rfc-editor.org/rfc/rfc6455.txt`.

[19] Cheshire, S., and M. Krochmal. *Multicast DNS* [Internet Requests for Comments]. Available from `http://www.rfc-editor.org/rfc/rfc6762.txt`.
`http://www.rfc-editor.org/rfc/rfc6762.txt`.

[20] Cheshire, S., and M. Krochmal. *DNS-Based Service Discovery* [Internet Requests for Comments]. Available from `http://www.rfc-editor.org/rfc/rfc6763.txt`.
`http://www.rfc-editor.org/rfc/rfc6763.txt`.

[21] Iyengar, J., and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport* [Internet Requests for Comments].

[22] *Quickstart - PICO Developer*. [cit. 2023-05-11]. Available from `https://developer-global.pico-interactive.com/document/native/openxr-sdk-quickstart/`.

[23] *OpenXR Specification Changelog.* [cit. 2023-05-11]. Available from `https://github.com/KhronosGroup/OpenXR-Docs/blob/main/CHANGELOG.Docs.md`.

[24]

[25] Atteneder, Andreas. *glTFast.* [cit. 2021-12-29]. Available from `https://github.com/atteneder/glTFast`.

[26] Canaviri, Lara Kuhlmann de, Katharina Meiszl, Vana Hussein, Pegah Abbassi, Seyedeh Delaram Mirraziroudsari, Laurin Hake, Tobias Potthast, Fabian Ratert, Tessa Schulten, Marc Silberbach, Yannik Warnecke, Daniel Wiswede, Witold Schiprowski, Daniel Hess, Raphael Brüngel, and Christoph M. Friedrich. Static and Dynamic Accuracy and Occlusion Robustness of SteamVR Tracking 2.0 in Multi-Base Station Setups. *Sensors.* 2023, Vol. 23, No. 2. ISSN 1424-8220. Available from DOI 10.3390/s23020725. Available from `https://www.mdpi.com/1424-8220/23/2/725`.

# Appendix A
# Manual

This chapter will contain various instructions on how to build the app, the plugin, the dashboard etc. Further it will also contain instructions on how to use the app and the dashboard.

## A.1 Compiling the source code

The whole system consists of multiple components, which are interdependent on each other. To help with this, when a someone wants to modify just one section without modifying the others I included the necessary build artifacts of each part with the source code. However, it still is possible to build everything from source and this section will explain building each part in the order you'd have to do if you were building everything from source. It should be possible to skip any subsection here to change one part of the app, but it might be necessary to rebuild dependent section to include files in resulting build.

### A.1.1 Dashboard

Dashboard is web-based and implemented in react. As such it needs node.js to be built and everything else is fetched from `npm`. I used node version `v18.16.0` but it should also work with node 20, which is the current latest version. Lastly, you should enable corepack by running `corepack enable`, which depending on your operating system and the manner in which you installed node might require administrator privileges. corepack is distributed along with node in official packages.

Once node is installed and corepack enabled, you can build the dashboard. Open terminal and navigate to `netvr-dashboard` directory. There run `yarn` to install dependencies and run `yarn build` to build the dashboard. This is everything you need to do build the dashboard.

Calibration visualisation relies on wasm build of either netvr-cpp or a part netvr-rust being present in correct directory to be able to apply the calibration to the data. See README.md in `netvr-cpp` or `netvr-dashboard/src/wasm2` respectively for instructions on how to build that. Instructions for building the rust version are also included below.

If you want to develop the dashboard run `yarn dev` to run the dev server, which listens on `localhost:3000` and automatically reloads the browser on any changes.

### A.1.2 Rust part

The rust code contains most of the implementation of the plugin and the server. As such it is probably the most important part of this project. To install all dependencies on non-windows system you can use direnv[1] in combination with nix[2]. This means that

[1] https://github.com/nix-community/nix-direnv
[2] https://nixos.org

you can just run `direnv allow` and all dependencies will be available. If you do not have nix or are on windows, you will need to install the dependencies yourself.

The main dependency is rust and to install it, I recommend using `rustup`[3]. With rustup installed you should be able to just run `cargo` commands described later in the `netvr-rust` directory and it will automatically download the required versions of rust toolchain based on contents of `rust-toolchain.toml`. However, if that does not work you might have to install the required versions yourself by running `rustup toolchain install 1.69.0`. This is the latest rust version at the time of writing.

Make sure that your rust installation is located in path without spaces as otherwise it breaks the android NDK as that is required for the android build. The best way to achieve that on windows is to set the following env variables system-wide: `CARGO_HOME` to `C:\Stuff\Rust\cargo` and `RUSTUP_HOME` to `C:\Stuff\Rust\rustup`.

Apart from rust toolchain you will also need a working C/C++ compiler, assembler and linker as some dependencies of the code have parts of them written in those languages. For windows, if you don't have a working compiler already, I recommend using Build Tools for Visual Studio 2022, which is a minimal install without the need to install whole visual studio. If you already have visual studio installed, but without the needed components, you can run "Visual Studio Installer" program and add the required components there (just the C/C++ compiler and assembler is required). If you are not sure if you have what you need, you can just try building the rust portion and if it fails on linker errors, or an error indicating that it failed to build .s, .c or .cpp file then you do not have working compiler or cargo could not find it. I also recommend installing the compiler in default location as otherwise it might make it more difficult for the build script to find.

If you are building on linux, then you probably have a compiler installed already. If you are on MacOS, then it should be enough to run `xcode-select --install` to install required portion of XCode.

To build for android, you need the android NDK. As you will probably want to have unity installed anyway, the best way to achieve this is to install NDK along with it. But if you have android NDK installed anyway, you can also use that. See cargo-ndk's documentation for instructions on how it finds NDK[4].

#### Build instructions

All the following commands are expected to be executed from terminal in the `netvr-rust` directory.

To install dependencies and build the whole project for current platform you can just run `cargo b` to create a development build, or `cargo b --production` to create a production build. The development build is enough for running the app from unity editor on windows, as there is a code which automatically find the .dll and loads it dynamically. This also builds the server and various extra utilities.

If you want to produce **windows** build of the native plugin and copy it to the correct directory you'll need cargo-post. Run `cargo install cargo-post` to install it. Then you can run `cargo post b --package netvr_plugin --release`.

For **android** version you need to have `cargo-ndk` which can be installed by running `cargo install cargo-ndk`. Once that is installed, you can run #1 (see below) to build the .so file followed by #2 to copy it to correct directory. You might also have to

[3] https://rustup.rs/
[4] https://github.com/bbqsrc/cargo-ndk

run `rustup target add aarch64-linux-android` to install the android target rust component if it does get setup automatically from `rust-toolchain.toml`.

```
#1:
cargo ndk -t arm64-v8a -p 32 b --package netvr_plugin --release \
    --target aarch64-linux-android
#2:
cp target/aarch64-linux-android/release/libnetvr_plugin.so \
    ../netvr-unity/Packages/cz.isbl.netvr/Runtime/Plugins/Android/arm64-\
    v8a/libnetvr_plugin.so
    # ^ note that there is space between .so and .., but no space
    # between arm64- and v8a. The command unfortunately does not fit
    # nicely on a page.
```

To run the server or a utility directly from source code folder, run `cargo run --bin executable_name`, eg. `cargo run --bin netvr_server` for server. You can also append `--release` to run the release version of the program.

To compile the wasm version of calibration algorithm you need wasm-pack and the relevant rust component. See wasm-pack's documentation[5] on how to install that and the component can be installed using `rustup target add wasm32-unknown-unknown` command. Then you can run `wasm-pack build netvr_calibrate --release` to produce the build. You will need to copy files from `netvr-rust/netvr_calibrate/pkg` to `netvr-dashboard/src/wasm2` to use them. You might also be able to just replace wasm file in build artifacts of the dashboard, but that is not guaranteed to work.

### A.1.3 Unity

Building unity portion of the codebase should be no different from building any other unity application. As usual, you can determine unity version used by looking at `netvr-unity/ProjectSettings/ProjectVersion.txt` and it is Unity `2022.2.12f1`. You can install this version by using Unity hub by clicking on a Unity hub link from the webpage for this release[6].

After opening the `netvr-unity` directory in Unity editor, you should be able to run the application directly in play mode. To build the application for windows, open File > Build Settings..., click on Windows, Mac, Linux and click on Build in bottom right corner. If there is a Switch platform button instead, click on that first and after unity completes the switch, you can click on the Build button.

To build for android, you do much of the same. In Build Settings window click on Android, if android is not currently active click on Switch and finally click on Build.

There is one more specific change you might have to do in Android. The project should be setup for the Meta's Quest platform. If you want to build for PICO, you have to change a few settings. In build settings, click on Player Settings... in bottom right. Then at the lower left, under XR Plug-in Management click on OpenXR and switch to the android tab. There remove Oculus Touch Controller Profile from the list and add PICO Touch Controller Profile instead. Finally, uncheck Meta Quest Support and check PICO Support instead. To switch back to Quest, you have to make the same changes but in reverse. Unfortunately in the current version of plugins, support for Meta's platform and ByteDance's platform is mutually exclusive even though they both use OpenXR.

[5] https://rustwasm.github.io/wasm-pack/installer/
[6] https://unity.com/releases/editor/whats-new/2022.2.12

## A.2 Short manual

To run the whole system you need to have two things running - server and client. To start the client, there is nothing unusual — either double click `netvr.exe` on windows, or run the app from Unknown sources on after installing it on Android. On windows, the system might ask you to give the application permission to access the network. If it does, grant it.

To install the app on Android-based headsets, you'll need to put the headset into developer mode. Putting Meta's headsets into developer mode requires having a developer account, while PICO headsets can be switched to the developer mode directly from settings menu by clicking the Software Version in About section seven times. Once your headsets allows side-loading APKs you can use your favorite tool to install the app. You can use `adb` directly from terminal, or use Meta Quest Developer Hub for Quest headsets, or upload the build directly from Unity by using the Build And Run button. On the PICO headset you can also copy the APK to the device and install it from file explorer.

To run the server double click netvr_server.exe and it should open a terminal window. It might ask you to give the app access to the local network, grant it as otherwise the clients will not be able to connect.

The client and server must be on the same local area network with working UDP broadcast. If this is the case and firewall is not blocking the network access, the clients should automatically find the server. The implementation currently only uses IPv4 (it does not have IPv6 support in server discovery) and UDP and uses well-known port 13161 for server discovery. It uses automatically assigned ports (`:0`) for client-server communication and those are printed on the console. The server has to be able to listen on `0.0.0.0:13161`. The server is very lightweight and should run on almost any hardware, but also stands in basically every critical path of the application.

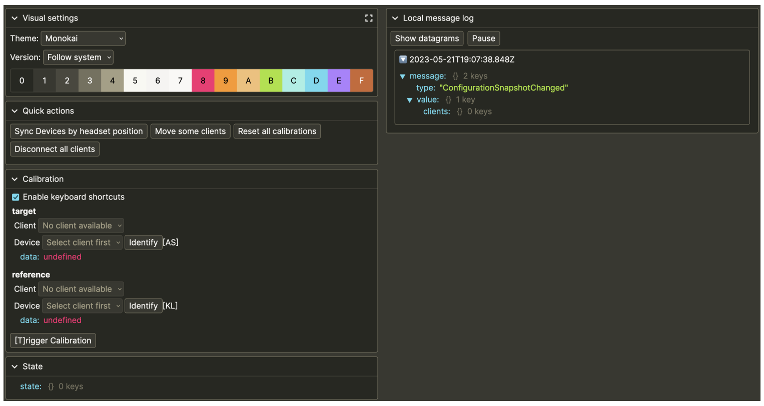
To connect to the dashboard, open `http://server-ip:13161` in your browser of choice. Any modern browser should work fine (sorry, no IE11 support here). Alternatively, you can also use `http://localhost:13161` if you want to access the dashboard from the same computer the server is running on. The server has to have TCP port 13161 open as web browsers connect via TCP instead of UDP. It uses HTTP and WebSocket protocols for connection.

### A.2.1 Dashboard

Dashboard contains most information needed to control the plugin and is also very useful for debugging what is wrong if something does not work. On figure A.1 you can see the view you will see when you open the dashboard for the first time.

The dashboard consists of five panels, which can be indivitually collapsed or expanded. On the left side are four control/status panels and on the right is a message log panel. Some panels contain structured data and there you can expend parts of the messages by clicking on the header or triangle, which denotes the status of the subobject. The dashboard tries to strike a good balance in choice of what to show by default.

The message log by default contains most messages the dashboard received from the server via WebSocket. It contains all configuration updates and some latest state update as otherwise it would have to show thousands of messages very quickly. The log can be paused by pressing Pause/Resume button to be able to focus on a message without it disappearing. You can also hide most frequent state messages and only show configuration messages by pressing Show/Hide datagrams.



Theme: Monokai
Version: Follow system

☑ Enable keyboard shortcuts

**target**
Client: No client available
Device: Select client first [AS]
data: undefined

**reference**
Client: No client available
Device: Select client first [KL]
data: undefined

state: {} 0 keys

```
2023-05-21T19:07:38.848Z
▼ message: {} 2 keys
    type: "ConfigurationSnapshotChanged"
  ▼ value: {} 1 key
      clients: {} 0 keys
```

**Figure A.1.** Basic view of the dashboard without any client connected and with all panels open.

On the top left is visual settings panel, which you can use to choose a theme used for the dashboard from a range of options. It shows the colors that will be used and you can choose from Light and Dark variants, or choose for it to follow you system settings. By default it follows system settings. This panel also contains a button which can be used to enter and exit full screen mode in the top right corner.

Second panel on the left is Quick Actions panel. This panel contains all actions which can be executed without inputting any data. The first button ”Sync Devices by headset position” triggers the simple calibration method described in section 3.6.2. The ”Move some clients” button chooses one client and applies translation to it to debug the application of calibration matrix without involving any calibration algorithm. Then there is ”Reset all calibrations” which instructs all clients to set Server space to be equal to Stage space. Lastly, the ”Disconnect all clients” button can be used to test the reconnect logic and it forcibly closes all client connections to the server.

The calibration panel is the most important panel of the dashboard as it is the main entrypoint for the calibration. Here you'll choose target client (the headset which will be moved in virtual space) and the reference client (the one that will not be moved). You also need to choose a controller from each. Controllers can be switch by using the shortcuts denoted in square brackets. Once client and controller are chosen, this panel also shows the data about the controller which can be used to identify if you are holding a correct controller (eg. move the controller above your head a observe if the z position changes accordingly). There is also a identify button, which is supposed to vibrate the controller, but it was not re-implemented upon moving from prototype version to the final version.

Once you have your chosen devices, then you can click on Trigger calibration, or (and I recommend this, as it is easier to do with non-dominant hand) by clicking the T button on your keyboard. The panel also contains Enable keyboard shortcuts checkbox, which allows you to disable above-mentioned shortcuts in case you need to write something.

Then there is the state panel, which contains the current system state. Click on the triangles to expand or close individual sections of the state tree.

Lastly there is one panel for each connected client. Those panels show basic information about the clients and allow you to rename them. You'll want to disable keyboard shortcuts on the Calibration panel before doing this.

### A.2.2 The VR application

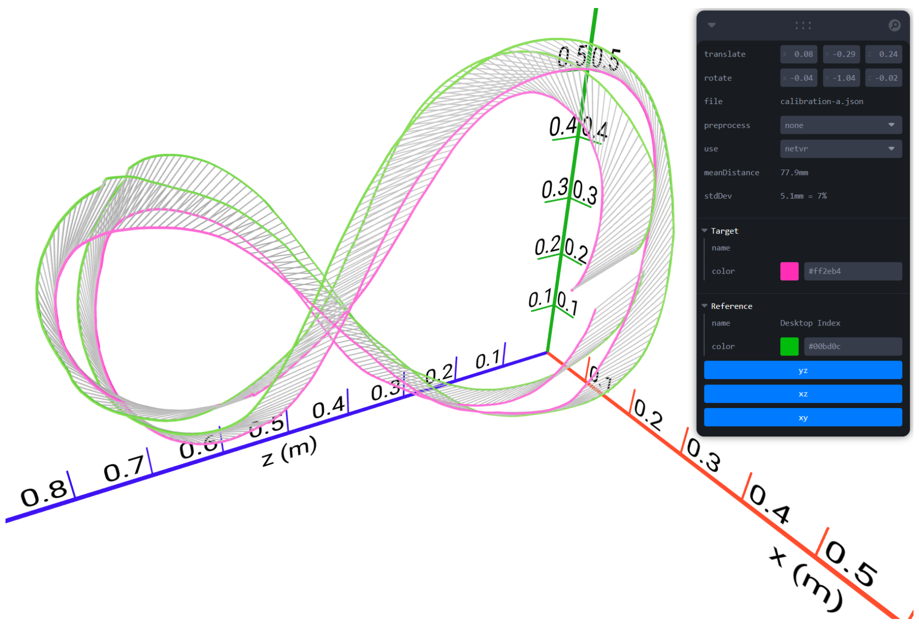
The application is very simple — there is no teleport, or menu. You can move through space by moving yourself. You should be able to see your controllers and also see state of their various buttons (how much is the button pressed, etc.). Once other clients are connected to the server you should also be able to see them immediately, but before the calibration is complete you'll likely see them in the wrong place. Once the calibration is complete, the virtual position should correspond with the real position of each device.

There are two actions which the user of the application can perform: interact with synchronized objects and record controller positions. To interact with synchronized objects, move the sphere that floats above your controller close to them and press and hold your trigger button. While you are holding your trigger button, the object should move with your controller. When you release the trigger button, the object should stay in its place. If someone else is holding a object, you should be able to take it from them in the same manner.

The other action that you can do is to record device movement. This can be done by pressing any face button (A,B,X,Y), doing the thing that should be recorded and pressing any of those buttons again. The log is then saved to the internal memory of the headset and can be extracted either by going to the directory with the app on windows, or by using similar methods you used to upload APK on android. Alternatively, the clients also upload this file to the server and it could be found in the `uploads` directory.

### A.2.3 Visualisations

When you do calibration, the server automatically creates .json file with a timestamp in its name in the same directory the server is running from. This json file can be visualized in the browser. Open your browser at `http://server-ip:13161/sample-viz` and you should be greeted with a 3D view and a control panel (it shows spinning cube briefly while it's loading but this should disappear, if not, check if the server is running and accessible). Drag and drop the calibration file to visualize it. See figure A.2 for an example of how it looks.



```
translate        X 0.08   Y -0.29   Z 0.24
rotate           X -0.04  Y -1.04   Z -0.02
file             calibration-a.json
preprocess       none
use              netvr
meanDistance     77.9mm
stdDev           5.1mm = 7%
Target
  name
  color          #ff2eb4
Reference
  name           Desktop Index
  color          #00bd0c
```

**Figure A.2.** Example view of interactive calibration visualiser on the dashboard.

It should then show the collected data in configuration. Use your mouse to change the viewport. You can also try changing some options in the top right, but keep in mind that those were mostly experiments useful during the development process and some of them might not work anymore. The defaults correspond to the actual calibration used. You can change the colors to going to the dashboard and changing the theme there.

Section calibration is for the data recorded by pressing controller buttons. Similarly to calibration visualisation, open web browser at `http://server-ip:13161/delay-viz` and drag and drop the file. This time it should end in .txt. Again the color settings are shared with dashboard.

Next visualisation is a trace of how unity calls the OpenXR methods, which the plugin overrides. This is currently disabled as it can produce multi-gigabyte files. The resulting trace files can be opened in Perfetto UI[7].

I attached some sample input files for all of these visualisations if you do not want to try creating some.

### A.2.4 Analyse program

You can also reproduce analyses reported above by running the analysis program. It is written in rust and you can run it from `netvr-rust` directory by executing `cargo run --package analyse --release path-to-data.txt`. It will output some statistics right into the terminal and create some charts and other files next to the input file. Those resulting files will have the same name but different extension.

If you do not have a working rust build environment, I also attached a built version of this program and you can run that as `analyse.exe path-to-data.txt`.

## A.3 Using the Unity plugin

It would hardly be a plugin, if you could not use it in new application. To do so you have to import it and then add some key objects to your scene. I'll assume that you already have a VR project and that it uses OpenXR as it's VR plugin. If not, change this first and then come back to this manual as OpenXR is a hard requirement for the plugin.

To import the plugin, copy `cz.isbl.netvr` directory from `netvr-unity/Packages` to the `Packages` directory in your Unity project. Afterwards, open `manifest.json` in the same directory and add `"cz.isbl.netvr": "file:./cz.isbl.netvr"` to the `"dependencies"` object. This object is customarily sorted alphabetically, but it is not neccessary. At this point, unity should automatically load the plugin.

To setup the plugin, you need to enable it in OpenXR Settings. Click Edit > Project Settings... (fifth from the bottom on my version of Unity). Then click OpenXR under XR Plug-in Management. Select a tab corresponding to your platform of choice and check the Isbl NetVR XR Feature. Repeat this for all platforms you wish to support. Once you enable the feature, the plugin will be automatically loaded when the game starts.

If you want the plugin to do anything, you also have to add it to your scene. Create Empty game object and add the Isbl Remote Device Manager script to it. Then set your prefab which you wish to use for remote devices. At this point, the plugin should work. If you want to have some information about the remote devices add the Isbl Remote Device script to the root of the prefab.

[7] `https://ui.perfetto.dev/`

If you also wish to use the plugin for simple object synchronization, then also add another game object, this time with Net Objects Manager script. Then, add your synchronized objects as children of this manager object. Each child should have Net Object script attached to it. Keep in mind that the plugin does not support any changes in the list of objects during the whole run of the *server* as it uses positions the first client connected uploads as a starting point. You can move those objects in the scene hierarchy after the initial setup is done as the manager only queries its children once in `Start()`.

To be able to move the objects you have to set `GrabbedBy` property to an instance of a controller when user grabs the object. Similarly, you should set this property to null when the user releases the object. Finally you should call `RequestMove` method periodically to set new positions and rotations of the object. See Grabbable.cs for example of how to achieve this.

# Appendix B
# Additional images

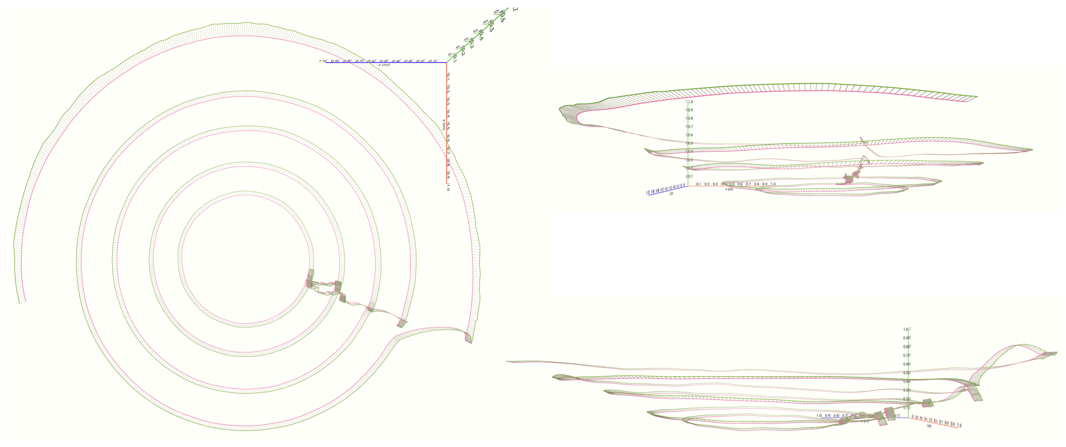
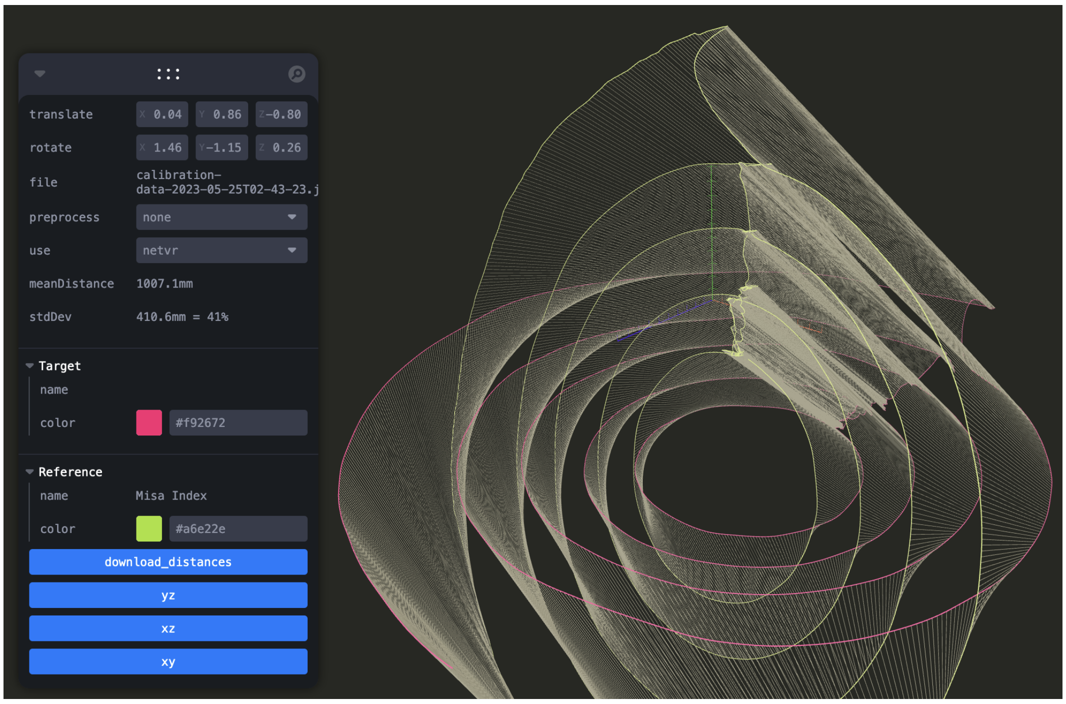


**Figure B.3.** 3D visualisation of the circular data collection session.



```
translate     X 0.04   Y 0.86   Z -0.80
rotate        X 1.46   Y -1.15  Z 0.26
file          calibration-
              data-2023-05-25T02-43-23.j
preprocess    none
use           netvr
meanDistance  1007.1mm
stdDev        410.6mm = 41%

Target
  name
  color       #f92672

Reference
  name        Misa Index
  color       #a6e22e
```

**Figure B.4.** The visualisations support dark mode, too. Also depicted: you can visualise situation if calibration produced a different result.

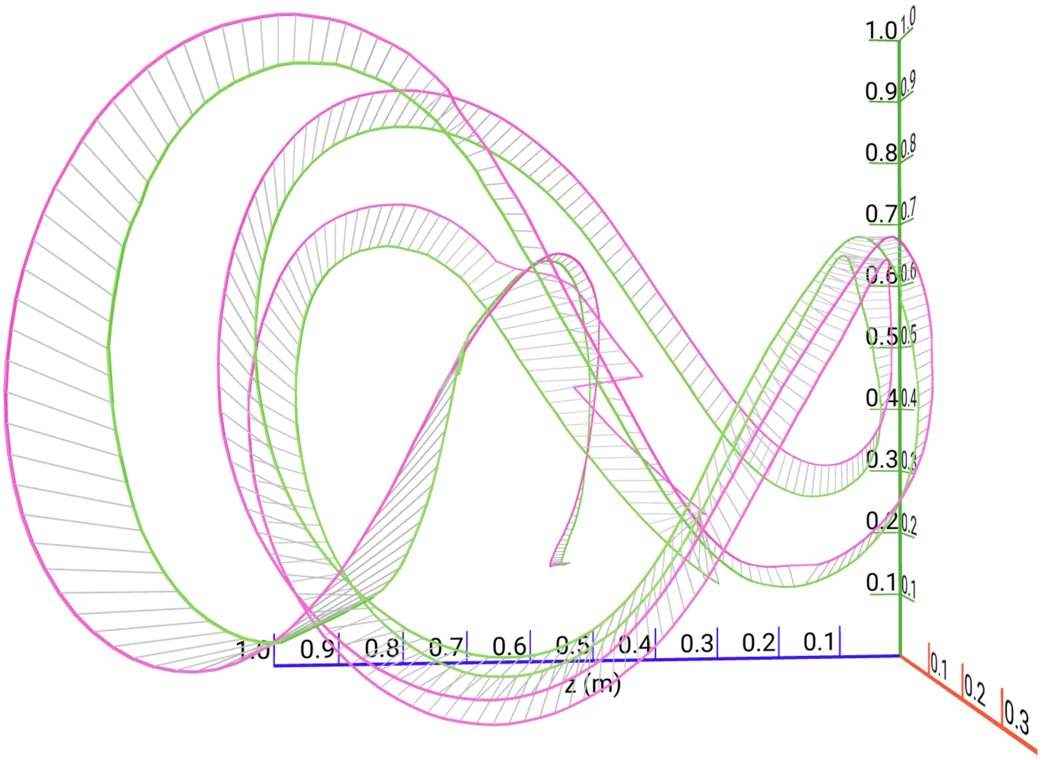


**Figure B.5.** When controller's tracking ring is occluded, the tracking becomes less precise. If this happens during calibration, the resulting calibration quality is affected. This image shows what it might look like. This image is from the second testing session B:C, where participants did report bad calibration of participant C.

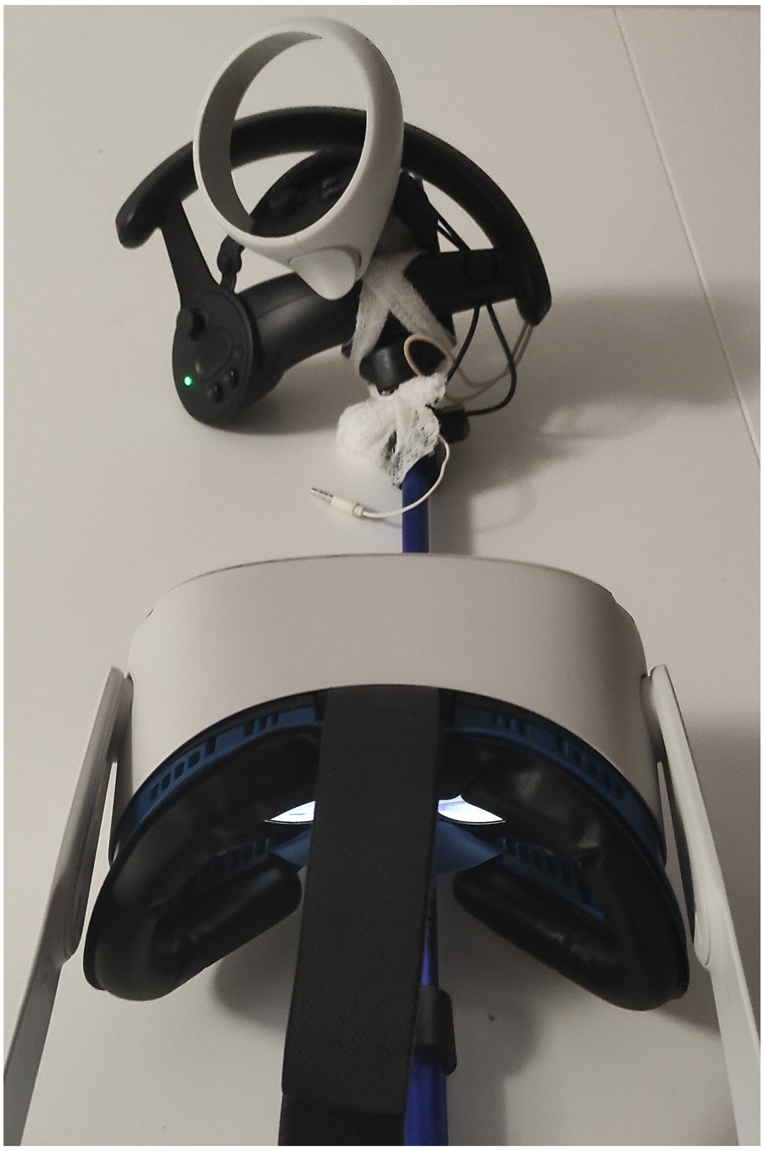


**Figure B.6.** Apparatus used for positional testing. Both controllers are placed in a clamp at the end of a telescopic selfie stick. This puts pressure on them and helps prevent movement. The clamp is padded, so slipping is unlikely. To make sure that the controllers do not pop out of the hold, the Index controller is tied in place and Index Controller's strap is used to hold the Quest controller in place. The Quest headset was on my head while performing the measurements, in the picture it is roughly at the position relative to the controller, where it was at the closest point during the testing.

# Appendix C
# Glossary

The following are definitions of terms that appear in this thesis. Some of the definitions are not exact or complete, as that would require more than one sentence, but the definitions contained here should be enough as a quick reference to help with understanding the previous text.

| | |
|---|---|
| DNS-SD | DNS Service Discovery. See section 3.5 and [20]. |
| LAN | Local Area Network. In this case, it refers to a part of wider network in which devices can discover each other and communicate directly. |
| mDNS | Multicast DNS. System for resolving DNS name without central DNS server. See section 3.5 and [19]. |
| OpenVR | OpenXR's predecessor implemented by SteamVR. |
| OpenXR | a set of APIs for access to virtual reality devices. |
| Pose | Combined position and orientation. |
| Rust | a programming language |
| SC | OpenVR Space Calibrator. See section 2.2 and [6]. |
| Server Space | Additional reference space defined in this work. Refers to the shared space between headsets. |
| Stage Space | The default OpenXR space used for room-scale VR |
| SteamVR | Software package used for VR on the desktop developed by Valve. It implements both OpenVR and OpenXR APIs. |
| VR | Virtual Reality |
| VRAPI | Legacy API used by older Oculus headsets. Replaced by OpenXR. |
| Vulkan | Modern 3D graphics API |
| 6DOF | 6 degrees of freedom. In the context of this thesis, it refers to the ability of a tracking system to track both the orientation and the position of a device. |