AARHUS UNIVERSITY SCHOOL OF ENGINEERING

BACHELOR PROJECT F2018

PROJECT NUMBER: 18043

A Comparison of Mobile- and Installed Virtual Reality with Focus on Object Manipulation

Authors
KASPER RIEDER
201310514 | AU506144
DANIEL VESTERGAARD JENSEN
201500152 | AU527964

Supervisors
KASPER LØVBORG JENSEN
HENRIK BITSCH KIRK

March 2, 2019

AARHUS UNIVERSITY

Abstract

Virtual reality is a booming technology with a huge potential for both the business and entertainment sector. Due to the cost and setup required for installed VR, it is interesting to make a comparison with the inexpensive and more accessible mobile VR in aspects considered important to the VR experience, such as object manipulation. To compare how accurate and efficient object manipulation is between the two platforms, a VR experiment system was implemented using Unity. The system was used to conduct an experiment which ran 21 users through the task of placing furniture into predefined positions in both mobile- and installed virtual reality. It was found that mobile VR was slightly outperformed by installed VR in accuracy and efficiency for this experiment. This small difference is interesting, as it could be an indication that mobile VR have potential use-cases where it can compete with IVR. The data was too inconclusive to make a definitive statement about the true objective difference between the platforms. A conducted user satisfaction survey found that a majority of users preferred the installed VR setup for object manipulation. The experiment system implementation resulted in a generalized toolbox of software components which can be used to further develop the experiment, or to aid development of new VR experiment systems.

Resumé

Virtual reality er en eksplosivt voksende teknologi med et kæmpe markeds potentiale i både forretnings- og underholdningssektoren. Grundet den høje omkostning og opsætning krævet af et installeret VR setup, er det interessant at foretage en sammenligning med mobil VR i aspekter som betrages vigtig for en god VR oplevelse, såsom object manipulation, da denne platform er både billigere og lettere tilgængelig. For at sammenligne præcision og effektivitet af manipulation af objekter i virtual reality imellem de to platforme, blev der implementeret et eksperiment system med brugen af Unity. Dette system blev brugt til at udføre et eksperiment som førte 21 brugere igennem en opgave i at placere møbler i forudbestemte positioner på både den mobile- og installerede platform. Eksperimentet viste at præcision var bedre på installeret VR sammenlignet med mobil VR. Dataen var for utilstrækkelig til at lave en general udtalelse om den sande forskel mellem de to platforme, dog indikerede det at forskellen i præcision og effektivitet muligvis ikke er så stor som man kunne tro. En brugerundersøgelse viste at størstedelen af brugerne foretrak den installerede VR platform til at lave objekt manipulation. Det implementerede eksperiment system resulterede i en generaliseret værktøjskasse af software komponenter som kan bruges til at videreudvikle eksperimentet, eller som kan bruges til at udvikle nye VR eksperimenter.

Preface

This document describes an investigation of virtual reality platforms with focus on object manipulation. The investigation is conducted with an implementation of an experiment system and execution of user surveys. From reading the sections 1 through 12 a complete understanding of the project can be achieved. For further details, read the appendices at the end of the document.

Besides these appendices, independent appendix files are provided in separate folders. These folders are:
ExperimentRecordings.zip, *Code.zip*, and *MiscellaneousFiles.zip*.

ExperimentRecordings.zip contains video recordings of users completing the object manipulation experiment. This folder is categorized into two subfolders, one for each platform: *Installed* and *Mobile*.

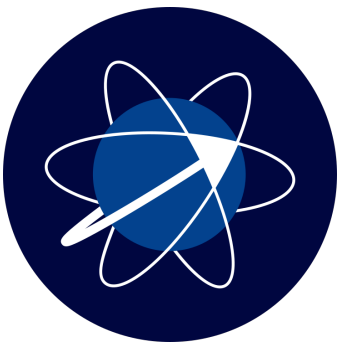
Code.zip contains the source code projects. Two subfolders are to be found: *TAPP* (Task-App: The Unity project containing our VR experiment applications) and *LiveExperimentDataApp* (The Android smartphone project for our experiment controller).

MiscellaneousFiles.zip contains an assortment of files used throughout the project, including executables for our VR experiments. Appendix I describes each file in detail.

All furniture models used in the experiments have been provided by Poly by Google [1], licensed under CC BY 2.0 [2].

This project is developed by two IKT students of Aarhus University School of Engineering in collaboration with Orbit Lab, and we would like to thank Orbit Lab and its members for loan of the VR equipment used in the project.

A special thanks to our supervisors Henrik Kirk and Kasper Jensen for providing valuable feedback and input.



Division of Labour

As this project was developed by 2 students using pair-programming, most development has been done together. For that reason, both individuals have shared knowledge of the implementation. However, the primary responsibility for two of the larger system components was split between the team. See table 1.

Name	System Component
Daniel Vestergaard Jensen	Ghost Object System
Kasper Rieder	IVR Remote Object Manipulation

Table 1: Divison of labour between some system components of the project.

Contents

1 Glossary

Gaze-Based Walking • A navigation method in VR, where the user is moved in the direction of his gaze

GoGo Manipulation • Object manipulation in installed VR using a hand-metaphor for the controllers to interact with objects.

Head-mounted display (HMD) • A display equipped in front of the eyes of the user, used to visualize a virtual space and track head movement.

Installed Virtual Reality (IVR) • A virtual reality system installed on a PC. This includes systems such as Oculus Rift and HTC VIVE.

Installed Virtual Reality Platform (IVR Platform) • The constellation of hardware making up all required hardware to experience IVR. As an example, this would include the HTC Vive and the personal computer used to execute the virtual reality environments.

Immersion • The objective level of sensory fidelity a VR system provides.

Minimum Viable Product (MVP) • A state of a product that only contains the bare minimum of functionality for the product to fulfil its purpose

Mobile Virtual Reality (MVR) • A virtual reality system for smartphone devices. This includes Samsung Gear VR and Google Daydream View.

Mobile Virtual Reality Platform (MVR Platform) • The constellation of hardware making up all required hardware to experience MVR. As an example, this would include the Samsung Gear VR and the smartphone used to execute the virtual reality environments.

Presence • The user's subjective psychological response to a VR system

Roomscale • A technology used by VR systems to track the user's movement in a one-to-one scale with the real world within a predefined area.

Virtual Reality (VR) • A computer-generated environment that simulates a realistic or lifelike experience. Essentially isolating the user from the real world and provides a virtual world for the user to interact with instead.

2 Introduction

In this section we will talk about Virtual Reality(VR), what it is and how it is used. We will take a look at the VR market and compare multiple VR systems' features to create a basis on which we will define a problem statement for further work.

2.1 Virtual Reality

Virtual Reality is not a new concept. It has been discussed and experimented with for decades. Devices attempting multi-sensual simulations go back as far the *Sensorama*, first shown in 1964. [3]

Figure 1: The Sensorama [4] one of the first immersive, multi-sonsory machines

Historically, the concept of virtual reality has been defined as a digitally created space which humans could enter by equipping sophisticated computer hardware. [5] In this digital space one can interact with virtual objects, people and environments.

The article *Virtual Reality A survival guide for the social scientist* (Fox, Arena, & Bailenson, 2009) [5] makes the point that the driving force behind the design and development of

VR comes from the need to provide a space for people to interact, without the constraints of the physical world.

Interaction with objects and people in a virtual environment is already achieved through the standard computer interface of a 2D screen, mouse, and keyboard. According to the article *Virtual Reality: Past, Present and Future* (Gobetti, & Scateni, 1998) [6] the standard computer interface is showing limitations when it comes to presenting and interacting with a 3D world, where tasks that should have been simple and intuitive are often difficult to complete. They see a potential in virtual reality as a way for humans to interact with virtual environments the same way they would with a real environment, thus making the human-machine interface more intuitive and effective as the user can reuse existing motor- and cognitive skills.

The book *Virtual Reality Technology* (Burdea, & Coiffet, 2003) [7] claims, that the aim of VR research is to provide a faster and more natural way of interacting with the computer, thus overcoming the communication bottleneck experienced with the mouse and keyboard.

Figure 2: Virtual Reality setup with gloves for interaction. [8]

In short, virtual reality is about interaction with a virtual environment, which according to the book *Developing Virtual Reality Applications: Foundations of Effective Design* (Craig, Sherman, & Will, 2009) [9], can be divided into three categories:

- Making Selections
- Performing Manipulations
- Navigation (Traveling)

According to the article *Virtual Environment Interaction Techniques* (Mine, 1995) [10], performing manipulations is one of the most important forms of interaction. Manipulations include changing the position, orientation and scale of an object.

This is further cooperated by articles such as *An Evaluation of Techniques for Grabbing and Manipulating Remote Objects in Immersive Virtual Environments* (Bowman, & Hodges, 1997) [11], which highlight that one of the defining features of VR, is the ability to manipulate objects interactively, rather than observing a passive environment.

This makes virtual reality an attractive technology for many industries as it can provide training and experience by putting the users in various simulated environments and situations.[6][5][12]

An obvious way to utilize the spacial understanding gained by using virtual reality, is the inspection of complex 3D models in a virtual environment. An example of this is Rolls Royce who have been using virtual reality to test and improve their designs. [6] VR testing can be done earlier in the development cycle than physical testing, making the process cheaper, and adds the benefit of multi-site collaboration on projects. [6]

Another benefit of VR is the relatively high "transfer-of-training" it provides. [13] By putting the user in simulated environments and scenarios tailored to maximize training, the users are able to gain experience with decision-making and interactions that are more easily transferred to the real world, compared to the standard mouse and keyboard computer interface. [6] In the medical field, virtual reality can be used to teach students about anatomy, surgeries, or make them experience the stress of a chaotic emergency room. [5] The military have been using virtual reality to give hands-on training in flight- and combat situations, which should train the decision-making process in a cheap and safe environment that would not be possible in the real world. [5]

Figure 3: The many usages of VR [14] [15] [16] [17]

To make virtual reality usage successful, both as a training tool and an entertainment experience, the VR application has to provide a high level of immersion to match the real world experience with a simulated one. [12] This is especially pertinent when it comes to training transfer, as it is argued that the training can only be as effective as the training system's sensory stimuli is realistic, as higher immersion can cause an increased sense of presence. [12]

To interact with an immersive virtual environment, a system of advanced hardware is required. In the article *Defining Virtual Reality: Dimensions determining telepresence* (Steuer, 1992) [18], Jonathan Steuer points out, that popular definitions of virtual reality have a focus on the hardware aspects of virtual reality.

> *"This [virtual reality] system usually includes a computer capable of real-time animation, controlled by a set of wired gloves and position tracker, and using a head-mounted stereoscopic display for visual output."*(Steuer, 1992)[18]

With the current generation of consumer-grade virtual reality systems, we have seen that this definition of a VR system is still relevant, as the latest technology makes use of stereoscopic head-mounted display (HMD), cameras that acts as positional trackers, and a set of controllers.

2.2 Current Generation of Virtual Reality Systems

We have surveyed the current market of virtual reality systems to better understand their differences, and to understand the current possibilities as a consumer. This analysis and comparison can be found in Appendix A. The comparison is based on VR brands holding the largest marketshare. This section will highlight our findings.

When comparing the various VR systems, what emerged was the fact that the greatest differences in advanced features were to be found between mobile virtual reality (MVR) systems and installed virtual reality (IVR) systems. MVR refers to VR headsets that are used in conjunction with a smartphone. IVR refers to VR systems used in conjunction with a PC. Significant feature differences across MVR and IVR systems were the amount of controllers included with the system, the amount of buttons on each individual controller, and whether or not the system supports roomscale and spatial control tracking.

Roomscale refers to the ability of a VR system to track a user's movement in the real world and map it in a one-to-one correspondance with the virtual environment. Spatial control tracking refers to the ability of a VR system to track hand-held controllers of a user. In the current generation of VR hardware, as described and shown in Appendix A, physical sensors installed on walls or other surfaces is required in order to provide this type of tracking.

Table 2 presents the findings of our virtual reality system comparison, in which the price of the VR systems is considered in relation to their platform and the advanced features

previously mentioned.

Brand	Price In USD	Platform	Amount of Controllers	Programmable Buttons on individual controllers	Roomscale	Spatial Control Tracking
Samsung Gear VR	$169.98	MVR	1	3	No	No
Google Daydream View	$99	MVR	1	3	No	No
Oculus Rift	$399	IVR	2	7	Yes	Yes
HTC Vive	$599	IVR	2	4	Yes	Yes

Table 2: Comparison of examined Virtual Reality systems.

A consequence of the fact that MVR systems has less advanced features also makes them more portable and cheaper to purchase relative to IVR systems. They are more portable because they are used with smartphones, without the need for complicated hardware setups such as PCs and tracking sensors, and they are cheaper because the VR hardware is simpler whilst also requiring a cheaper device on which to run the virtual environments on. When considering the final price of MVR and IVR, we attempted to estimate a realistic price were consumers to purchase a minimum compatible device. This is a device which fulfills the minimum recommended system requirements of the platform.

Table 3 presents these findings.

Brand	Standalone Price	Price of minimum compatible device	Combined Price
Samung Gear VR	$169.99	$236	$405.99
Google Daydream View	$99	$399.98	$498.98
Oculus Rift	$399	$749.99	$1148.99
HTC Vive	$599	$749.99	$1348.99

Table 3: Price differences taking minimally viable device price into account.

It can be seen that there is a considerable difference in the combined price of purchasing an MVR system compared to an IVR system.

As described in the analysis of Appendix A, one of the interesting things we found was that whilst the difference in advanced features between the MVR and IVR systems were big, MVR held the largest market share through Samsung's Gear VR.

2.3 Problem Statement

In section 2.1, 2.2 and Appendix A we have looked at what virtual reality is, and how it is used, and compared the VR hardware of the current generation.

We found that VR has a huge practical potential for businesses and society, but the powerful high-end systems of installed VR are expensive and requires an extensive

setup. On the other hand, mobile VR is more accessible, cheaper, and has a larger market share than installed VR, but is limited by the hardware, both in processing power and sensory input.

On this basis, it could be interesting make a comparison between mobile- and installed VR systems. Since smartphones are much less powerful than the computers required for an installed VR setup, we are assuming that the differences in visual and auditory quality are not yet comparable, but might be worth investigating when the smartphone devices become more powerful. We found that interaction was an important aspect of virtual reality, and especially the ability to manipulate objects is vital for many systems.

Therefore it could be interesting to compare manipulation of 3D objects in high-end installed VR, such as the HTC Vive, to mobile virtual reality, such as Google Daydream View.

If mobile VR can compete with installed VR when it comes to manipulation, it could be an attractive market for VR developers to look at, as barrier-to-entry for consumers is lower. This leads us to the following problem statement:

> ***What is the difference in accuracy and efficiency of object manipulation between installed- and mobile virtual reality systems, and what is the perceived user satisfaction between the two technologies?***

Manipulation refers to the interactions with 3D objects, such as selecting, moving and rotating them.

Efficiency refers to the time spend completing the given task, while accuracy refers to how accurately a user has manipulated the objects.

To investigate this, we want to make an experiment system, in which users complete tasks of object manipulation on both installed- and mobile VR systems. From this experiment system we want to gather objective data about efficiency and accuracy of object manipulation on the two platforms.

To measure user satisfaction for the two technologies, we will create questionaires that the users fill out as part of the experiment.

3 Experiment Design and Scope

In order to keep the object manipulation experiment simple and easy to understand, we wanted the tasks to be relatable and with a clearly defined goal. The simplest experiment idea, was to have users move 3D objects from position A to B, as this task can be applied in many themes, be it placing planets in the solar system, or making molecules out of atoms. As we wanted the task to be relatable to as many users as possible, we decided to use objects themed as furniture, as we believed that most people could relate to the task of placing furniture of varying size in a room. To be able to gather comparable data about accuracy, we wanted a system where all users had a similar experience, and for that reason we want the users to place existing furniture into predefined positions. These positions must be made clear through visual cues, and to do that we decided on the concept of *ghost furniture*, which is a transparent representation of the desired position and orientation. (see figure 4)

Figure 4: Illustration of ghost furniture, represented here as the black transparent furniture.

The goal is for the user to place matching furniture in this configuration as accurately as possible. Figure 5 shows an illustration of how it could look when furniture have been placed by a user.

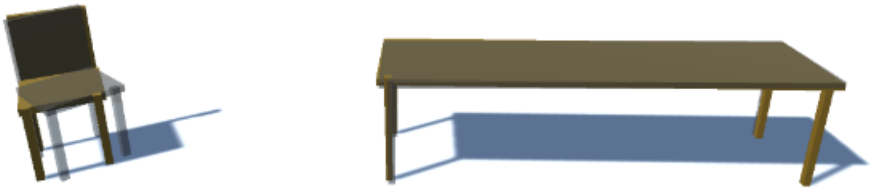


Figure 5: Illustration of furnitures placed in close resemblance to the ghost furniture configuration.

The objective measure of *efficiency* introduced in section 2.3 would be measured in terms of how long it takes users to place all the furniture into the predefined configuration.

The objective measure of *accuracy*, also introduced in section 2.3, would be measured in terms of how well centered the furniture is with its associated ghost furniture, and the alignment of their orientations. The greater the difference between the center alignment and orientation, the less accurate you would be. This concept is illustrated in figure 6, where imperfect accuracy is shown.

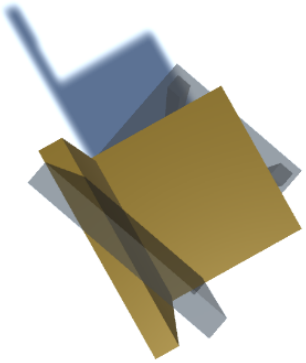


Figure 6: Illustration of the concept of accuracy.

Figure 7 illustrates perfect accuracy. The furniture is positioned in exactly the same spot as its associated ghost furniture, and their orientations match.

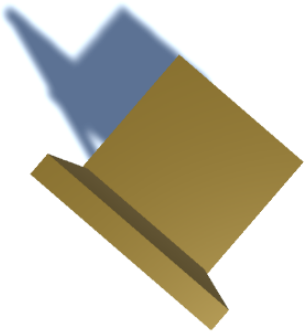


Figure 7: Illustration of perfect accuracy.

4 System Specification

To facilitate the comparison of object manipulation between installed- and mobile virtual reality, we want to create two VR applications targeting MVR and IVR respectively. These applications will facilitate the experiments containing tasks for the users to complete.

The only difference between the applications should be the control schemes and interfaces, while the tasks of the experiments are identical. The applications should save measurements such as accuracy and efficiency into a shared data persistence system.

We learned during development that an Experiment Controller was a practical addition to the system, to help manage experiments and to see what users were doing during ongoing experiments. The Experiment Controller is a device which can be used separately from the VR applications to interact with them, and receive live data from them.

A high level component diagram of the system is shown on figure 8

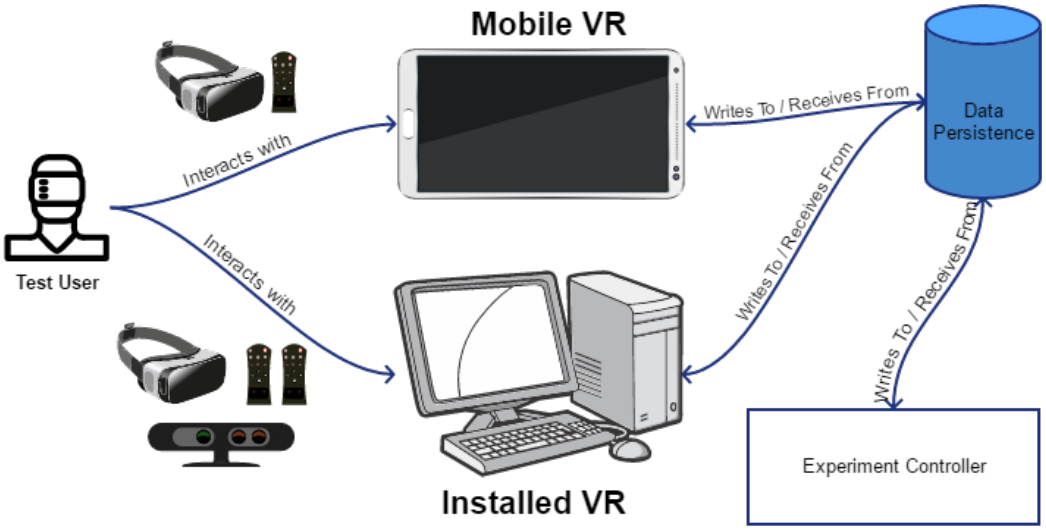


Figure 8: A high level component diagram of the system

From the system description as well as the high level component diagram of the system in figure 8, important required main components can thus be listed as:

- Two virtual reality applications, targeting MVR and IVR respectively, facilitating experiments containing the tasks which users should complete.
- A shared data persistence system, which will keep measurements such as accuracy and efficiency for both platforms.
- An Experiment Controller which can be used to interact with the VR applications, and receive important live data from them.

The following chapter will provide further analysis of requirements for the identified main components.

4.1 Virtual Reality Applications

As the virtual reality applications will host the object manipulation tasks which the user will then complete, there is a multitude of requirements for this component. Both applications must implement the exact same tasks, with the only difference being control schemes. Therefore they share system requirements. The applications must put the user in a virtual reality environment, where they can manipulate virtual furniture objects.

Furthermore, the applications will need to gather measurements, which then needs to be stored in a data persistence system.

These points have been summarized as functional requirements in table 4.

Requirement Number	Name	Description
#1	3D Rendition	The applications must render a three-dimensional virtual environment in which users can manipulate furniture.
#2	Asset Loading	The applications need to read a 3D model file format in order to load 3D furniture models used for the virtual environment.
#3	VR Input Handling	The applications need to accept and process input coming from MVR and IVR.
#4	Data Gathering	The applications need to gather data related to relevant measurements in order to determine the efficiency and accuracy of users.
#5	Accuracy Calculation	The applications need to calculate accuracy of furniture placements.
#6	Data Persistence	The applications need to write to shared data persistence in order to expose gathered data to third-parties.
#7	Two-Way Communication	The applications need to accept inbound commands from the Experiment Controller, and be able to deliver live experiment data to it.

Table 4: Functional Requirements of the applications

4.2 Data Persistence System

The key characteristics of the data persistence component of the system is that it has to be shared between the applications. Having a shared data persistence system is preferable in order to have all measurements and experiment data at the same location, in order to reduce fragmentation and other potential complexities with managing data in multiple places.

This point have been summarized as a functional requirement in table 5.

Requirement Number	Name	Description
#8	Shared Persistence	The data persistence system must be shared between various components of the system

Table 5: Functional Requirements of the database

4.3 Experiment Controller

In the development process of our project, we learned that it was preferable to have an Experiment Controller.

We faced several inconveniences during the execution of experiments. Firstly, it was difficult to follow along with users on the MVR platform, as the MVR platform works by having a smartphone placed within a head-mounted display (HMD). This makes it difficult to provide guidance. Secondly, we saw a huge potential in being able to restart the experiment remotely if something went wrong as user-based experiments are time-consuming. Being able to restart the experiment remotely is much faster than having to reboot the entire application.

Because of this, we decided to introduce the concept of an Experiment Controller. It should make it possible to monitor users on a VR platform where it is difficult to see what they are doing, and it should be possible to reset experiments, without having to restart the entire virtual reality application.

These functional requirements have been summarized in table 6.

Requirement Number	Name	Description
#9	Two-Way Communication	The Experiment Controller needs to be able to receive live data from virtual reality applications, and it needs to deliver data to them.
#10	Graphical User-Interface	The Experiment Controller needs a graphical user-interface so that visual data can be communicated (such as seeing what the user is doing), and interaction possibilities are also required for experiment resets.

Table 6: The functional requirements of the Experiment Controller.

4.4 Non-Functional Requirements

Besides the functional requirements, the system has non-functional requirements described in table 7. These requirements will be a deciding factor when choosing technologies, and designing the system software.

The Non-Functional requirements were chosen based on aspects which we thought would be important in order to have a usable experiment system.

Requirement number	Description
#1 Performance	Since the applications are targeting the VR platforms, performance of the system becomes important, as bad performance can have effects such as inducing nausea in the users. Performance is lower on a mobile phone and must accounted for in the implementation.
#2 Usability	The VR applications must be user-friendly in the sense that interacting with the virtual environment should be intuitive.
#3 Reusability	The VR applications should be developed with reusable software components in mind. This would make it possible to more easily conduct future research using parts of the VR application software.
#4 Extensibility	Adding functionality to the VR applications, such as new metrics to gather or new assets to manipulate, should be easy. Therefore the architecture and implementation should have a focus on extensibility.
#5 Robustness	The data gathered during the experiments are very important for further analysis, we need the system to have high robustness, to ensure no data is lost.

Table 7: Non-Functional requirements for the envisioned system

5 Development Process and Method

In this section we present the development process we have gone through to create the experiment system for this study.

5.1 Project Process

As a tool for our development process during this project, we were heavily inspired by the *Lean Startup* process [19]. The concept of *validated learning* through the *Build-Measure-Learn* feedback loop is one that we have decided to incorporate into our development process.

Figure 9 shows the overall concepts of the original Lean Startup Build-Measure-Learn feedback loop.

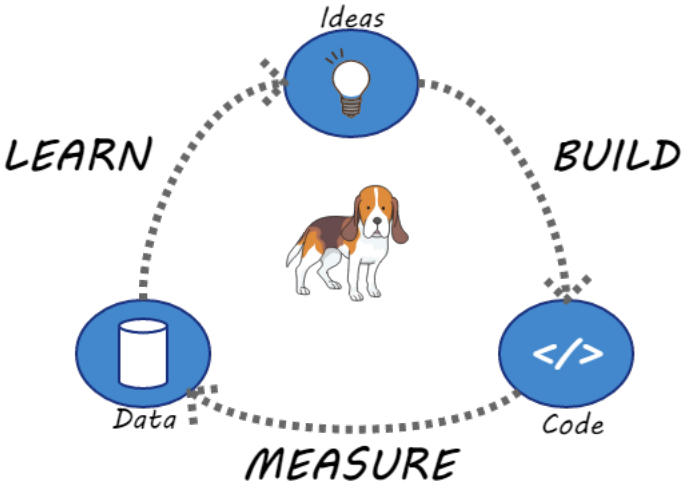


Figure 9: Overview of the Lean Startup methodology

The Build-Measure-Learn loop encourages quick prototyping and testing, so that validation of the product can be done early and often through measurable metrics. This can help test features against end-consumers, and through iterations arrive at a high-quality end product.

The Build phase of the loop is where obtained knowledge is used to produce a prototype or a minimum viable product (MVP).

The Measure phase, is where the prototype or MVP is tested with people, either internally or through release to a test-audience. During this phase data is gathered around the usage of the prototype/MVP to be used in the next phase.

The Learn phase is where the gathered data is analyzed to identify how to optimize the product, or to decide which features to implement, and how. This is done to validate whether or not you are getting closer to your goal, and as such is used for the next iteration, starting at the Build phase.

Our interest in these core principles are two-fold. Firstly, as we have no previous knowledge of developing VR applications, fast prototyping and internal testing allows us to determine how to implement basic VR functionality. Examples of how we have done experiments and prototypes can be seen in appendix D - The Navigation Experiment and appendix G - Virtual Reality Prototypes. Secondly, we want the VR user experience, and experiment setup, to not be affected by unsatisfactory software implementation.

In short, we want to apply validated learning to our development process to eliminate big uncertainties and bad implementations, and thereby optimize our final product.

5.2 Pair Programming

During the development of the product we used the technique known as pair programming [20].

The benefits of using pair programming was two-fold. Firstly, pair programming can increase the quality of the codebase as you have more skilled people to review and debate the written code instantly. Secondly, pair programming is a great way to share knowledge, as both participants will understand the functionality of the codebase.

Pair programming was used during the majority of the development process. It was mainly used when implementing non-trivial parts, such as the object manipulation or the cross-platform functionality.

5.3 Project Timetable

The project was limited to one semester. In order to manage the time properly, we created a timetable to plan out our development process. This timetable can be seen in table 8.

Week	Area of Focus
1 - 5	Analysis
3 - 5	Problem Statement
6 - 14	Report Writing
6 - 14	Documentation
6	VR Application Prototyping
7 - 9	Navigation Technique
10 - 12	Object Manipulation
13 - 14	Object Manipulation Experiment Application
15 - 16	Object Manipulation Experiment Execution
15 -18	Report Writing

Table 8: The timetable for the project.

The purpose of the analysis was to deepen our understanding of the academic literature surrounding Virtual Reality. This was to help us formulate a precise and relevant problem statement.

VR Application Prototyping was planned as a period in which we could get comfortable with basic development of Virtual Reality applications. It was used as part of our iterative Lean process in order to find and eliminate potentially big uncertainties of the technology before committing us to the future development phases.

The goal of the *Navigation Technique* phase was to develope and execute a navigation technique experiment. The purpose of this iteration was to eliminate uncertainties of navigation methods in virtual reality environments by involving user opinions, so that it would not affect our final object manipulation experiment.

Object Manipulation was planned as a phase in which we would implement the object manipulation methods to be used in the final object manipulation experiment.

Object Manipulation Experiment Application and *Object Manipulation Experiment Execution* were planned as being the actual development of the virtual reality applications supporting the object manipulation experiment, and then performing the actual execution of it with users.

6 System Design and Implementation

From the system specification, section 4, we described the primary components and the functional requirements of a system which would facilitate the experiment design of section 3.

In this section, we will describe the system architecture, and we will present the technical choices that was made to best fulfil the needs of this architecture.

6.1 General System Architecture

An overview of the system's architecture is presented as a deployment diagram in figure 10. This is the hardware identified in the component diagram of figure 8, in section 4 - System Specification, elaborated in more technical detail.

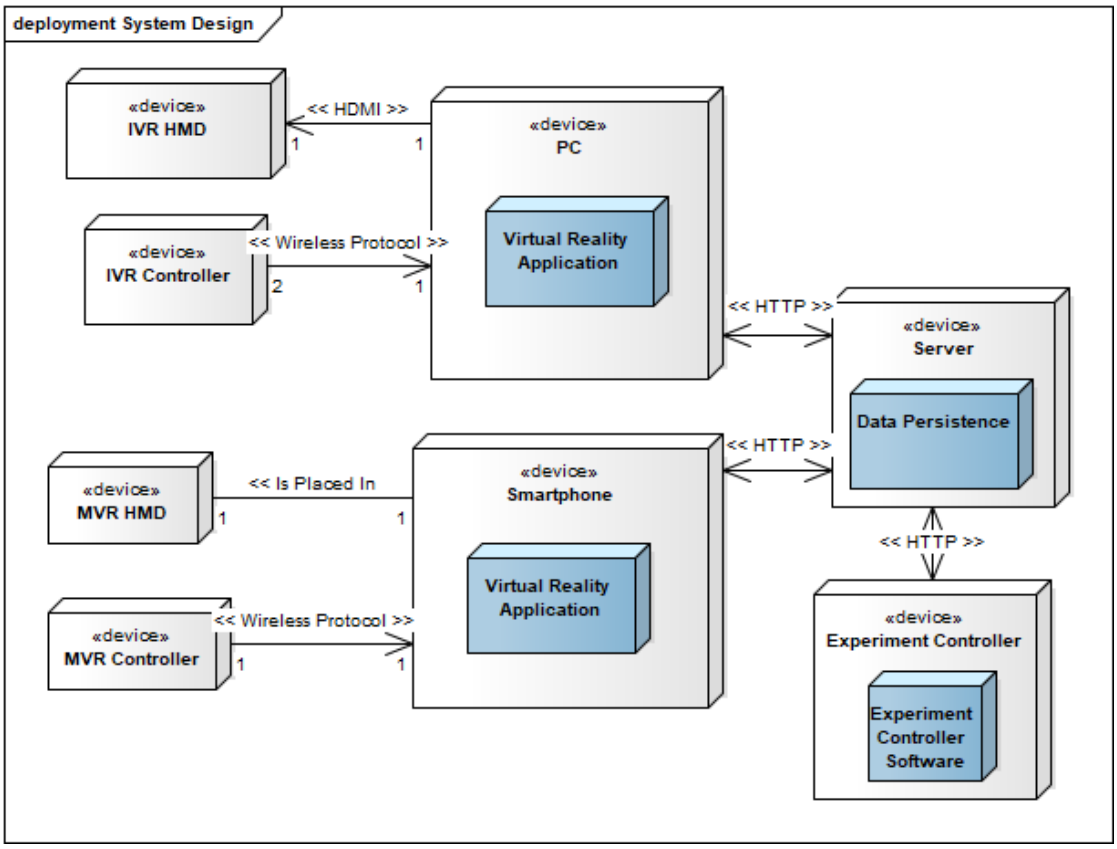


Figure 10: A deployment diagram of the system architecture.

From the architecture of figure 10, four primary areas of concern can be identified:

1. Virtual Reality Platforms.

2. Virtual Reality Applications.
3. Data Persistence.
4. An Experiment Controller device and Experiment Controller Software.

The MVR platform consists of a smartphone, a head-mounted display, and a single controller.

The IVR platform consists of a personal computer, a head-mounted display with two controllers.

A PC and a smartphone is required to execute the virtual reality applications. The PC and smartphone are also required to communicate with a data persistence server.

6.2 Technical Choices

These areas of concern, identified in the previous section, are vital for the realization of the product. This section will highlight the most important technical choices we have made for the product.

6.2.1 VR Platforms

As demonstrated in the deployment diagram of figure 10, section 6.1, an important area of concern is the VR platforms.

The following section will present the chosen platform for MVR and IVR, along with the reasoning for the choice.

MVR Platform
An important function of the MVR platform, is that it must have a controller included. The main candidates would then be Samsung's Gear VR [21] and Google's Daydream View [22] as they both feature a head-mounted display and a controller.

We have decided to use the Google Daydream View as it has a similar combined price as the Gear VR, while having the potential to hit a wider range of smartphone users as it isn't limited exclusively to Samsung phones.

Figure 11: Google's Daydream View headset [23]

For the MVR platform device, we chose to use a Google Pixel. The Google Pixel smartphone is Daydream-ready, meaning that it supports Google Daydream for VR applications.

IVR Platform

For the installed VR platform, we need the full range of interaction provided by two controllers and roomscale tracking. This makes the Oculus Rift and the HTC Vive the main candidates for this, as they both can provide the full installed VR experience.

We have decided to use HTC Vive, as it supports roomscale out of the box, whereas the Oculus Rift's out-of-the-box roomscale is only experimental, and would require a separate purchase of another Oculus Sensor to get the same room-scale experience as the HTC Vive. Also, Orbit Lab has a VR lab with a dedicated IVR platform, in which the HTC Vive is used. As the HTC Vive fulfils all our needs, it is the most optimal choice for our project. The PC used in the VR lab is an Alienware PC with the required hardware to execute the VR application. This PC was used during development of the project.

6.2.2 Game Engine

An important aspect to consider is the fact that this bachelor project spans a rather limited amount of time. Thus, it would be beneficial to use a game engine [24] to develop the virtual reality applications.

In table 4 of section 4 - System Specification, functional requirements were determined for the VR applications. Based on these, we ended up choosing Unity.

Unity is a 3D game engine in which you can edit the game world using an editor. See figure 12.

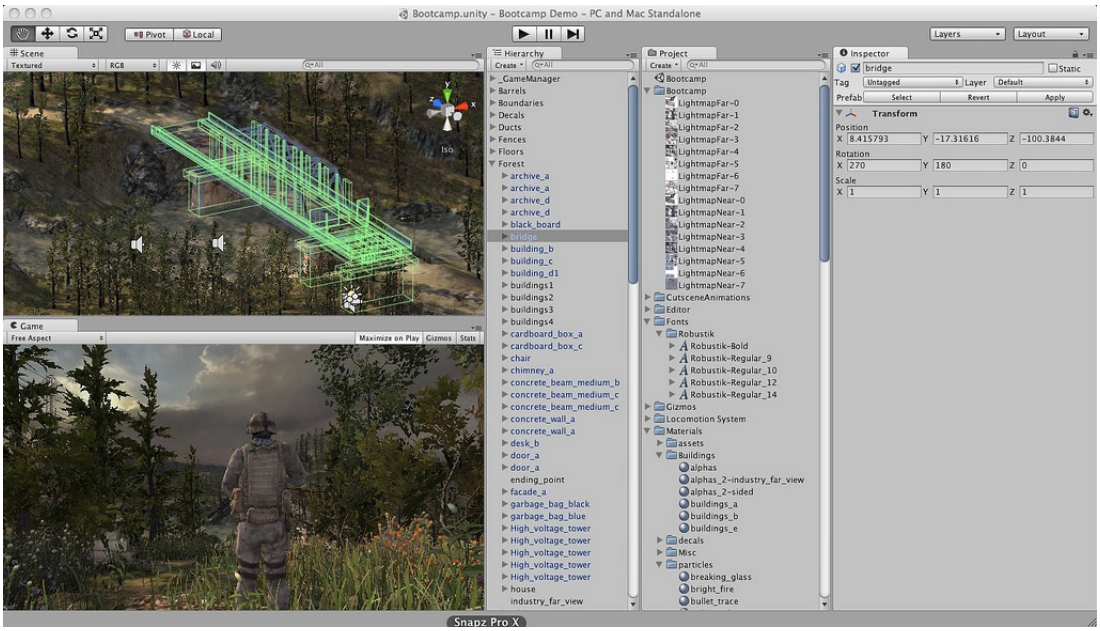


Figure 12: Unity's Editor (Screenshot by Ian Hughes) [25]

Because Unity can render 3D environments on multiple platforms, such as PC's and smartphones, functional requirement #1 - 3D Rendition is supported.

Unity has a powerful and easy-to-use asset workflow [26]. It supports typical asset types such as: image-, 3D model-, and audio files. Functional requirement #2 - Asset Loading is supported due to this, as well as non-functional requirement #4 - Extensibility.

Unity also supports VR inputs from many VR systems such as Google Daydream View and HTC Vive. As such, functional requirement #3 - VR Input Handling is fulfilled by Unity. Both OpenVR (HTC Vive) and Google officially supports Unity, by providing the game-engine with packages and sample projects, that makes VR development for these platforms more accessible.

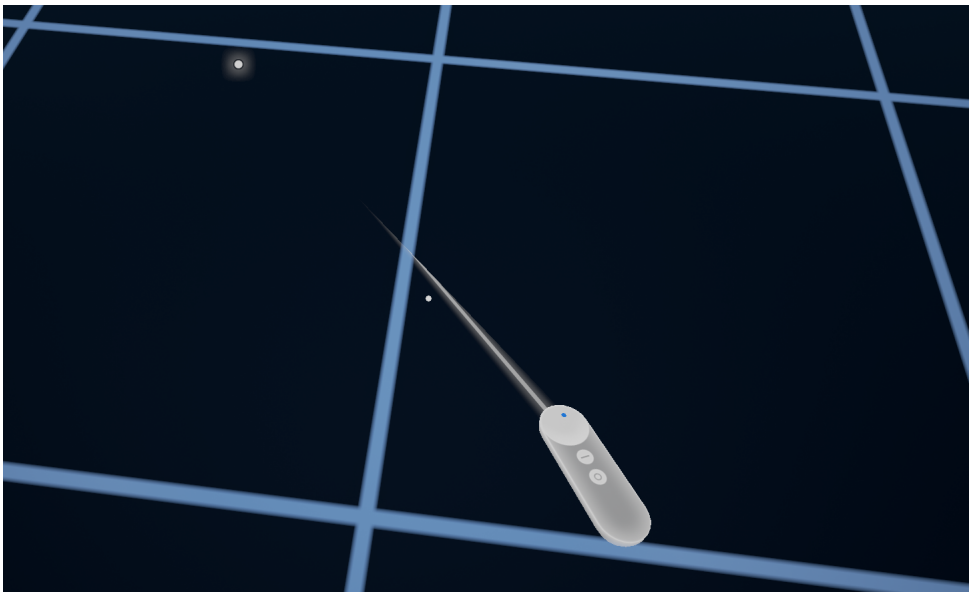


Figure 13: Google's official VR controller visualization as seen from a game [27]

An advantage of Unity is that programming can be done through C#, as Unity has support for the open-source implementation of the .NET framework Mono [28]. Thus development can be done in a general purpose programming language. Because of this, Unity supports our functional requirements #4 - Data Gathering and #5 - Accuracy Calculation.

As Unity uses C#, support exists for database writing either through the .NET framework or through HTTP-communication. Thus functional requirement #6 - Data Persistence is fulfilled.

We already have extensive experience working with C# as a programming language and as such time can be saved in the development phase of the project.

Overall, Unity is a great fit for implementing the envisioned system depicted and described in the system architecture of figure 10, page 25. The functional requirements are supported by the game-engine, which allows us to dedicate development time to the aspects of the product that is most relevant to the problem statement.

Throughout the technical descriptions and documentation of this project, we make use of several Unity-specific terms. See appendix B for descriptions of the most common terms.

6.2.3 Data Persistence

The system requires shared data persistence with good robustness to store the measurements gathered during the user experiments (functional requirement #8 - Shared

Persistence and non-functional requirement #5 - Robustness).

Keeping in mind the functional requirements previously mentioned, we have chosen Google's Firebase Real-Time Database[29]. Firebase fits in perfectly with our envisioned Client-Server architecture, but as the server hardware and software is hosted by Google, the implementation of the architecture becomes serverless [30]. Firebase is cross-platform friendly, as it works with both smartphones and desktop applications, and maintains an official integration to the Unity Game Engine. Two-Way communication is also possible by making use of Firebase's real-time functionality, in that changes made to the database will automatically be propagated to any clients listening in, which supports functional requirement #7 and #9 - Two-way communication.

Firebase is a NoSql database[31]. This is advantageous for our project, as there is no need to model complex relations between different entities. By eliminating the use of SQL databases, we won't have to manage schemas and migrations when updating the database. This fits very well with the non-functional requirement #4 - Extensibility. Since the data is "real-time" (events are being sent when new data has been saved to the database, allowing subscribers to react to it), this database would ensure any subscribers will have the data instantaneously.

6.2.4 Experiment Controller

To implement the experiment controller, we chose to write an application for Android smartphones. This was done because we wanted the application to be portable, and because Firebase integrates well with Android.

Developing an application for Android also allows for the possibility of a graphical user-interface, which fulfils functional requirement #10 - Graphical User-Interface.

6.3 System Architecture Implementation

The resulting implemented architecture, with the described technical choices, can be seen in figure 14.

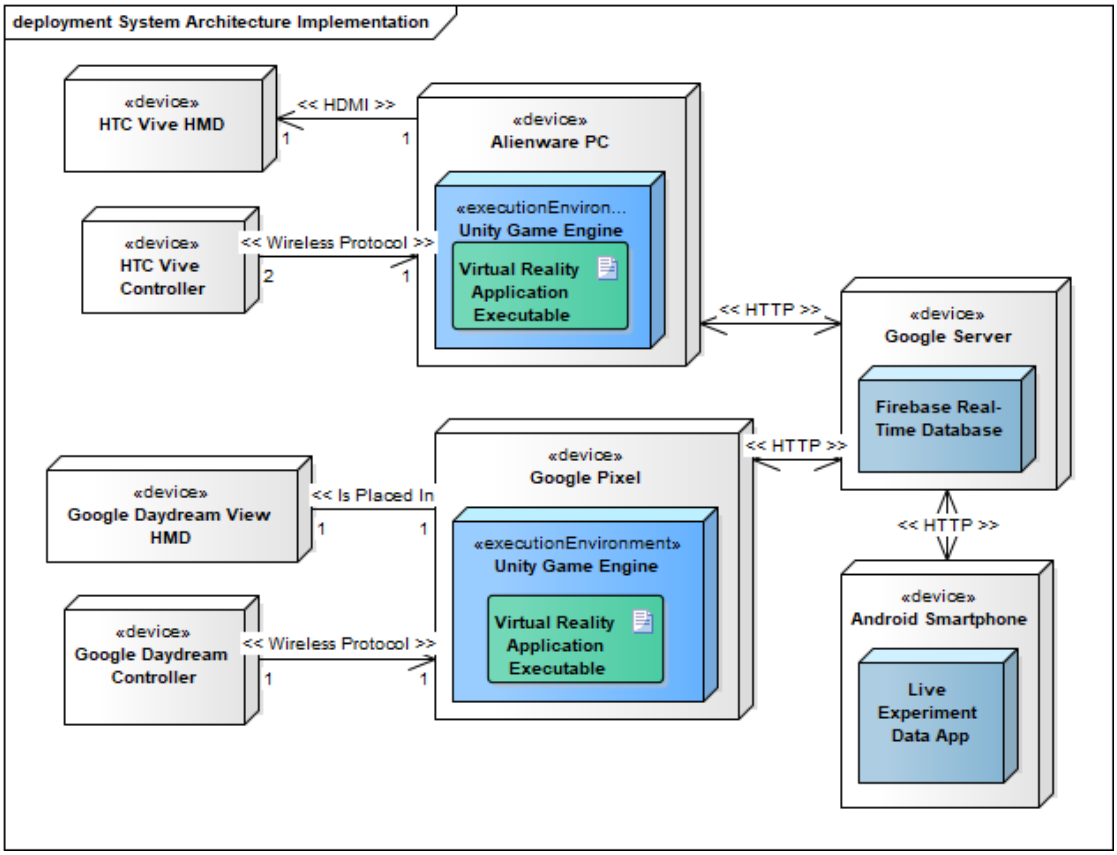


Figure 14: Deployment diagram filled with the technical choices.

6.4 Development Process Tools

During the development process we used tools to support it. This section will highlight some of the significant choices. Appendix H.5 - Process Tools describes other tools in more detail, such as unit testing and continuous integration which was used during in the project.

6.4.1 Version Control

To manage the sharing of source code between group members, we have decided to make use of Git and Github [32].

Git was chosen as a version control system in order to have complete history of files in the project, with an easy way to revert unwanted changes.

GitHub was chosen as the hosting solution for keeping our Git repository online because of their free student model, that allows us to create a free private repository.

As the development is done by a two-person team using pair programming, we have decided to not make use of Git's feature branches and instead commit directly to master.

6.4.2 Dependency Injection

When using Unity for cross platform development, we wanted a dependency injection framework to help us manage the platform specific instances of interfaces.

We decided to use Zenject [33], an open-source lightweight dependency injection framework specifically made for Unity. Since it's made for Unity, the injection works for both mobile and installed projects. The framework can inject both normal C# classes as well as Unity specific MonoBehaviour classes.

6.5 Implementation

In this section, we will highlight aspects of our implementation which played key roles in our solution. These implementations relate to the realization of the functional requirements described in section 4 - System Specification.

6.5.1 Event Logging

Event logging has been a vital aspect of our solution. As functional requirement #6 - Data Persistence of section 4 - System Specification, states, the virtual reality applications need to gather measurements.

In order to realize event logging, we implemented an Experiment Logger System as part of the Experiment Toolbox (See appendix F.7).

The Experiment Logger System is a generalized software component which makes use of the *Publish-Subscribe* Pattern [34]. It is realized as a class in our system by the name of *ExperimentLogger*. The sequence diagram of figure 15 gives an example of how it is applied within our Object Manipulation Experiment.

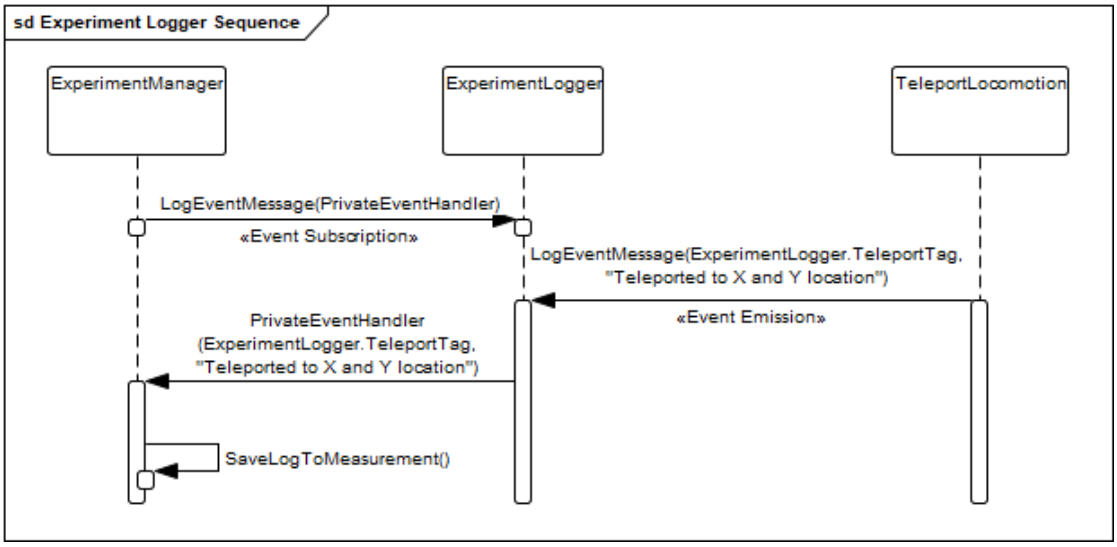


Figure 15: A typical use-case sequence of data logging. This is an example taken from the Object Manipulation Experiment.

In this example, we have an *ExperimentManager* script of our Object Manipulation Experiment, which subscribes to log events by calling the *ExperimentLogger*. A *Teleport-Locomotion* script emits log messages relating to every time that a user makes use of teleport navigation within the experiment. This log message is then published to the *ExperimentManager*, which can then do further processing before eventually saving the measurement to persistent storage.

We chose to implement this system by use of the Publish-Subscribe pattern in order to conform to our non-functional requirement #3 - Reusability. The log system should be a general component that can be used by multiple components of any experiment, without any of these components needing to know about each other. The loose coupling given by the Publish-Subscribe pattern lends well to this purpose.

Because of the practicality of this pattern, it has also been used extensively throughout other parts of our system, including the Ghost Object System - see appendix F.8, and the Waypoint System - see appendix D.4.

6.5.2 Data Persistence

In order to facilitate data persistence through use of the Firebase Real-Time database, we made use of the *Repository Pattern* [35] in conjunction with dependency injection. The implementation we made is the Firebase Measurement Repository System of the Experiment Toolbox, see appendix F.4.

Figure 16 shows the interface and classes making up the Firebase Measurement Repository System.

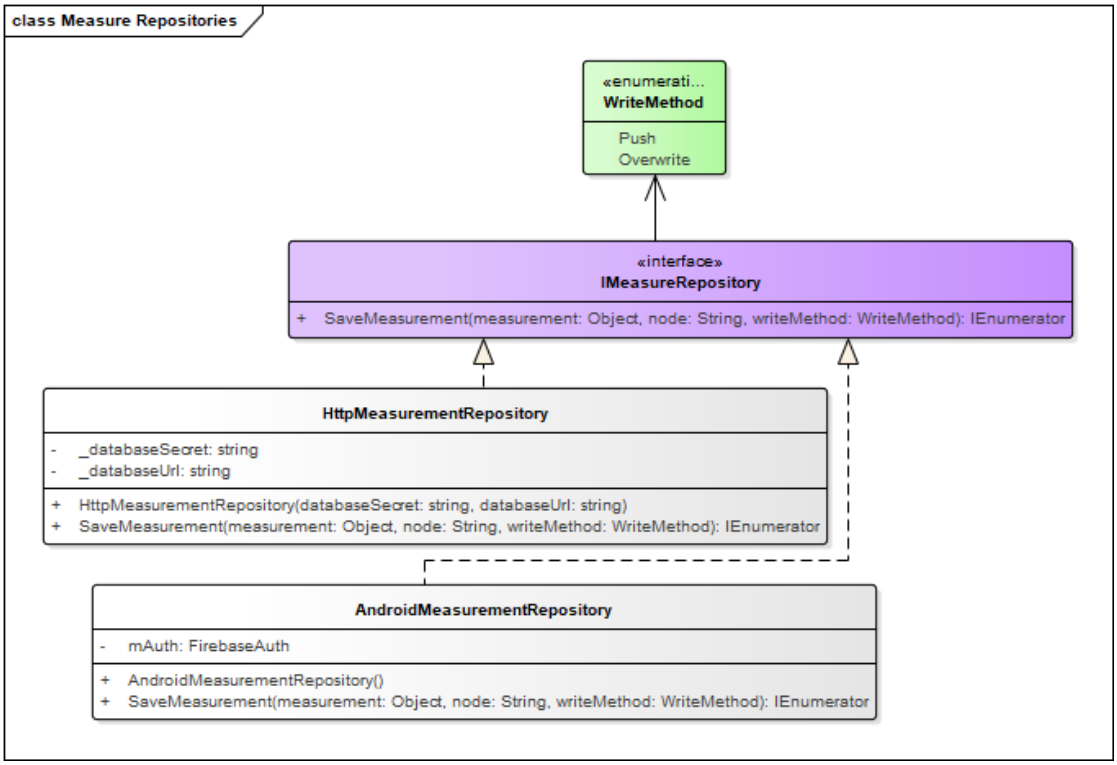


Figure 16: The class diagram for the Firebase Measurement Repository System.

We made use of interfaces in the measurement system. One thing we wanted to achieve by this was creating a unified contract across platforms for saving data. We make use of different implementations of the *IMeasureRepository* interface in conjunction with dependency injection to facilitate cross-platform development. This also supports non-functional requirement #4 - Extensability.

The use of the repository pattern ensured that specific implementation details about the persistence data platform remains hidden from clients.

6.5.3 Ghost Object System

The Ghost Object System uses the concept of ghost furnitures described in section 3 - Experiment Design and Scope. An image of the Ghost Object System in use in the Object Manipulation Experiment can be seen in figure 17.

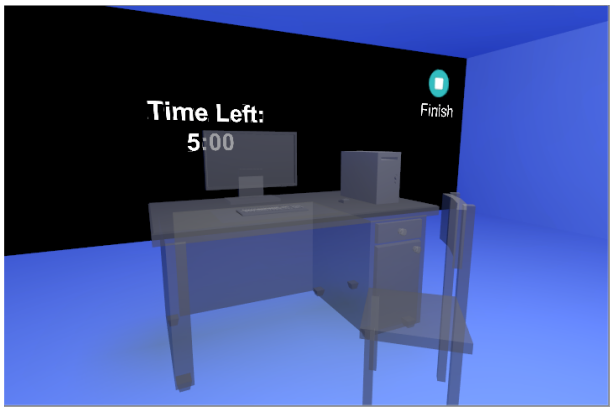


Figure 17: An example of Ghost Objects from the Object Manipulation experiment.

The Ghost Object System is part of the Experiment Toolbox (see appendix F.8), as we saw the potential of its general purpose. The Ghost Objects can represent any objects, which in our case is furniture. This ties in well with Non-Functional Requirement #3 - Reusability.

Through the unity editor it is possible to configure ghost objects according to specific behaviour, such as associating a specific object with it. This is shown in figure 18. The Ghost Object will then facilitate calculations of accuracy relative to this associated object.

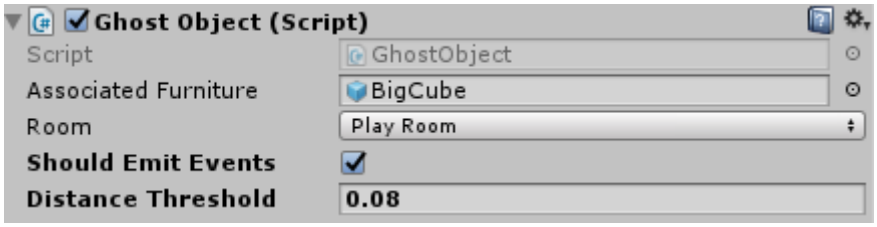


Figure 18: An example of Ghost Object settings from the Object Manipulation Experiment.

See appendix 6.5.3 - Ghost Object System for a more technical description and How-To guide of the Ghost Object System.

6.5.4 Experiment Controller and Two-Way Communication

The Experiment Controller is implemented as an Android smartphone app. A picture of the app with the 2D map representation of the Object Manipulation Experiment, and the Reset Scene button can be seen in figure 19.

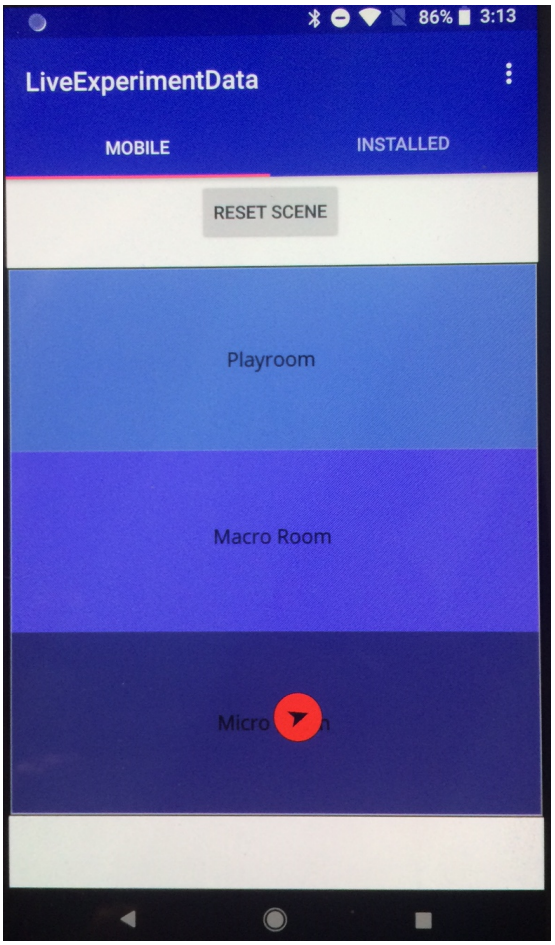


Figure 19: The Android smartphone application that is our experiment controller.

See appendix E.5 - Live Experiment Data Smartphone Application for a more technical description of the Experiment Controller software and appendix I.10 for a video demonstration of the app in use.

The virtual reality applications send live data about the user's current position and view direction to the experiment controller so that it can be displayed on a 2D map representation of the Object Manipulation Experiment. The experiment controller is also able to send a "Reset Experiment" event to the virtual reality applications in order to reset the Object Manipulation Experiment.

The flow of the two-way communication is presented in the sequence diagram figure 20.

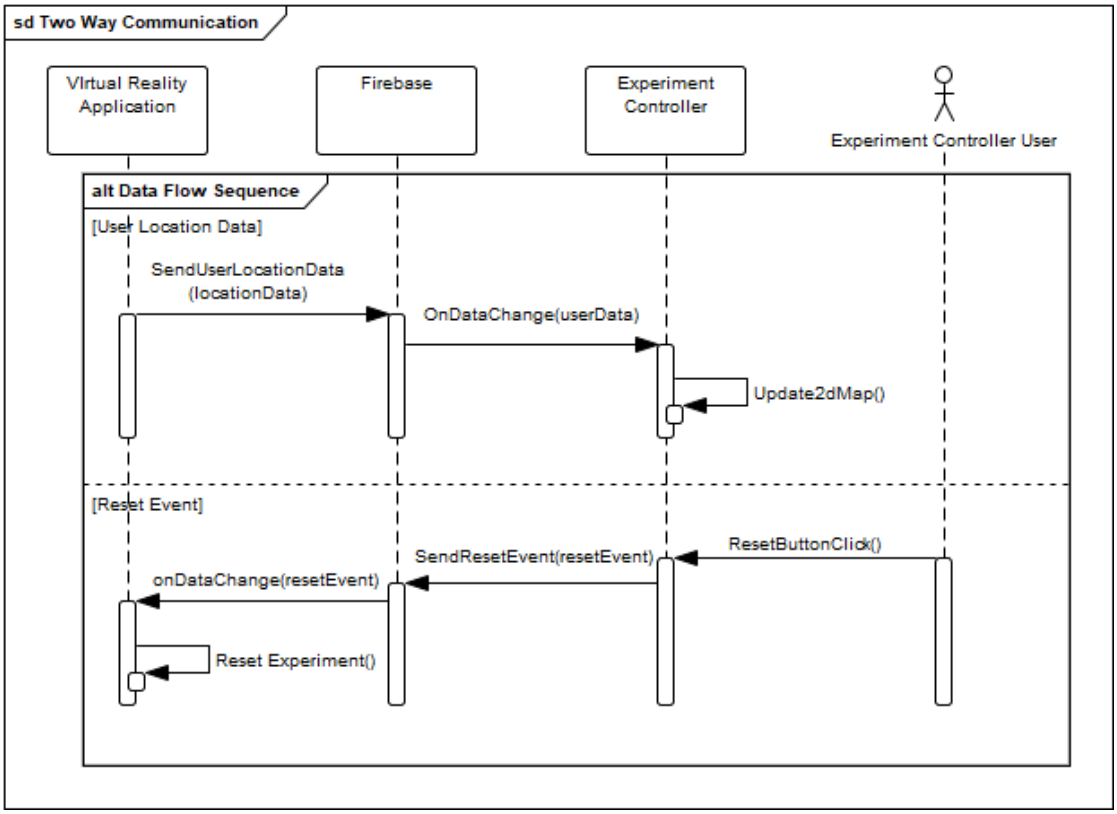


Figure 20: The ExperimentManager class diagram.

In order to facilitate the two-way communication for the virtual reality applications, we developed a generic software component for the Experiment Toolbox; the Firebase Event Listener System (see appendix F.2). We make use of the Service Callback design pattern [36]. Essentially, the system operates with the concept of *Node Listeners* and *Node Listener Callbacks*. The basic flow of a virtual reality application receiving an event can be seen in figure 21.

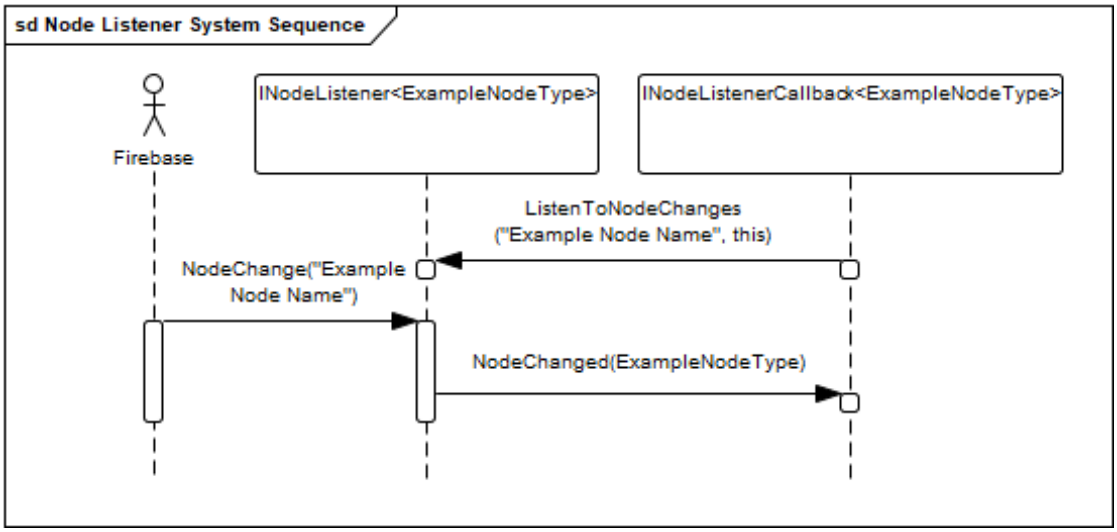


Figure 21: A basic example flow of the Firebase Event Listener System.

In the virtual reality application for our Object Manipulation Experiment, a script implements the *INodeListenerCallback* interface in order to receive callbacks whenever the specified node changes on Firebase. Using a realization of the *INodeListener* interface, the script specifies the name of the node to listen for changes, and passes itself as the callback. From this point on, any changes to the node will be received in its callback method - the *NodeChanged* method seen in the sequence diagram. This is how the virtual reality applications receive the Reset Experiment event.

6.5.5 The Server and Data Persistence

The Firebase Database stores JSON blobs of measurements in a one-to-one conversion from a C# readmodel class to a Firebase JSON blob as seen in figure 22.

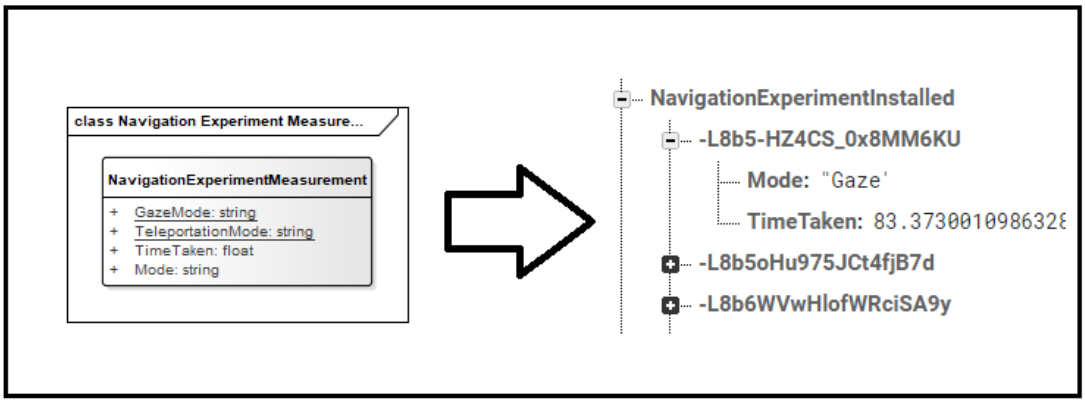


Figure 22: Illustration of C# class mapped to Firebase representation

The database consists of a number of nodes, each containing childnodes, which are the JSON representation of the data. In the example on figure 22 the data has been put under the "NavigationExperimentInstalled" node. To keep our measurements separated by platform and experiments, we have created multiple nodes on Firebase split by purpose.

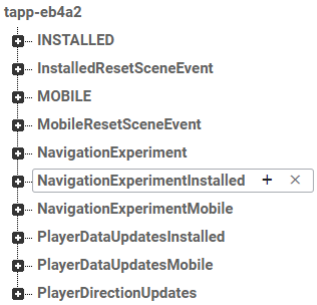


Figure 23: Illustration of the Firebase Nodes

Each entry pushed into the database will be placed under the designated node, and automatically given a unique ID.

This makes it easy to extend the server to support more platforms or experiments in the future, as you can create new nodes to store the experiment-specific data.

6.5.6 The VR Clients

Cross-platform development

An important decision of the structure of the Unity project, was how to handle the cross platform development for the experiment, as the virtual environments must be identical for the two platforms. The challenges of programming to two different platforms becomes apparent when trying to create an identical environment for both, as each platform has platform specific ways to implement VR functionality such as headtracking, controller- and interaction handling.

To solve this problem, we considered two options.

The first option was to have a single scene shared among the two platforms, and using an initializor script we could fill in the scene with the platform specific prefabs, depending on the target build platform.

The second option would be for each platform to have their own scene, with the scenes sharing a base "environment prefab". This way the environment on each platform would be identical, and the platform specific prefabs would only need to be set once.

We decided to go with option two, using a shared base "environment prefab" where platform specific scripts are added in the different scenes. This was done for two reasons;

Firstly, when trying to live debug the VR applications, we would have to change the initializor script to initialize the platform we wished to debug on. This quickly became cumbersome when switching between the platforms during development. Secondly, using an "environment prefab" shared among scenes makes it easy to create a new scene for a new platform in the future, as you'd only have to handle the VR input methods for the new platform for you to get the platform ready. This also helps us conform to the non-functional requirements #4 - Extensibility and #3 - Reusability.

An example is the environment from the navigation experiment. This room contains many objects, and a waypoint system, that should be identical on MVR and IVR. We turned the setup into a prefab that could be reused in the platform specific scenes. A screenshot of this prefab can be seen in figure 24.

Figure 24: The shared prefab of the navigation room.

In figure 24, *NavigationRoom* is the root object of our prefab. Within this object we nested everything else that was part of the shared virtual environment. All the furniture is placed below its own nested object, and the same goes for the waypoint system and lights.

7 Navigation in Virtual Reality

As mentioned in section 2.1 - Virtual Reality, navigation is an important aspect of VR. While not the focus of this investigation of object manipulation in VR, we find it important to improve the user experience by eliminating distracting factors that an unsatisfactory implementation of VR navigation could induce, and as such conform to our non-functional requirement #2 - Usability.

From looking at current virtual reality applications and games, we have seen that the most commonly used methods of navigation are *teleportation* and *gaze-based walking* [37][38][39].

Teleportation is where the user can use the controller to point to a destination in the 3D space, and by clicking a button instantly move to that location.

Gaze-based walking is where pressing the forward button will move the user in the direction of his gaze. While this method of 3D navigation may induce a higher degree of motion sickness than teleportation, a study by Daniel Brenners [40] has shown that this mode of navigation is preferred in MVR as users have reported to have more control and it felt more immersive (higher sense of presence). However, from our own anecdotal experiences we found that nausea was a problem when using this method, and therefore we decided to conduct our own experiment.

7.1 Navigation User Experiment

As we are using a lean-inspired process, we have created a user-experiment to decide which of the navigation modes to use in the final object-manipulation experiment. In this navigation-experiment, the users will have to navigate a 3D space by teleportation, and gaze-based locomotion, and give feedback on the preferred method through a questionaire (see appendix I.1).

We gathered 20 people to go through the navigation experiment; 10 for each platform, with half starting with the teleportation navigation mode, and the other half with gaze-based walking, to reduce any bias that might occur from trying a specific method first. This AB/BA split of navigation methods the test subjects went through is shown on figure 25

	1st Method	2nd Method
User 1	Teleportation	Gaze-Based Walking
User 2	Gaze-Based Walking	Teleportation

Figure 25: The split of the users trying teleportation and gaze-based walking

As we only had 20 people to do the experiment, we know that the data is not statistically significant, but we will use it as an indication of preferred navigation method nonetheless.

The full description, and a summation of the results can be found in appendix D.

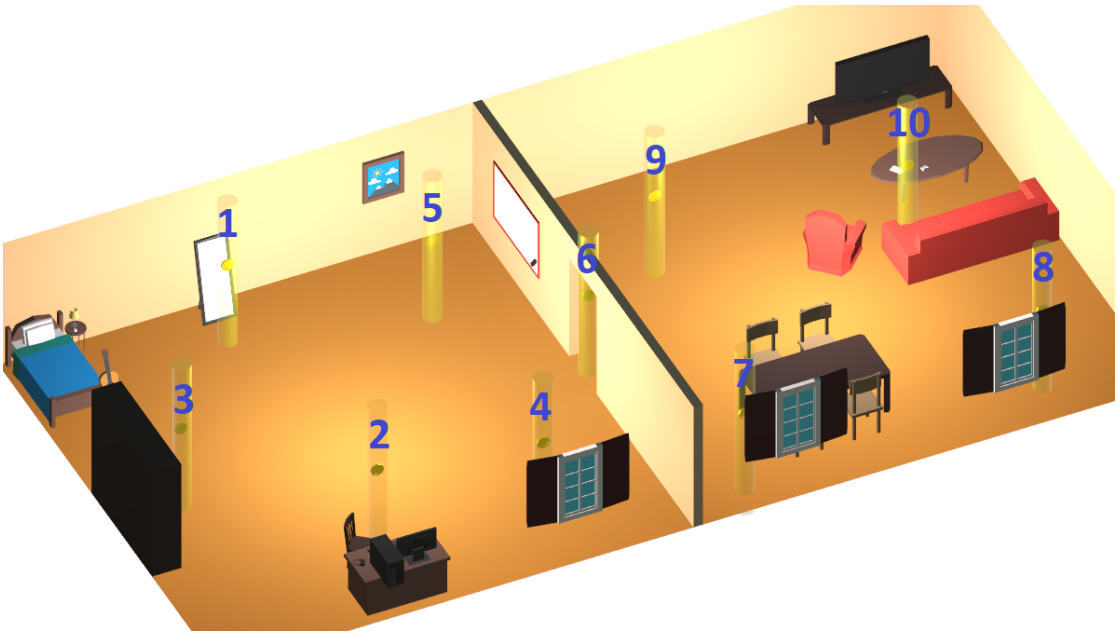


Figure 26: The navigation experiment room seen from above, with the order of the path drawn on top

7.2 Navigation Experiment Conclusion

This section will highlight the findings from the navigation experiment. The full results are described in appendix D.3

From the MVR experiment, we see that the users had a harder time orienting themselves, and felt somewhat more nauseous when using Gaze-Based walking compared to teleportation. People found it weird to be "walking" in VR while remaining stationary in the real world. When asked what navigation method they preferred, a small majority said teleportation as seen in figure 27.

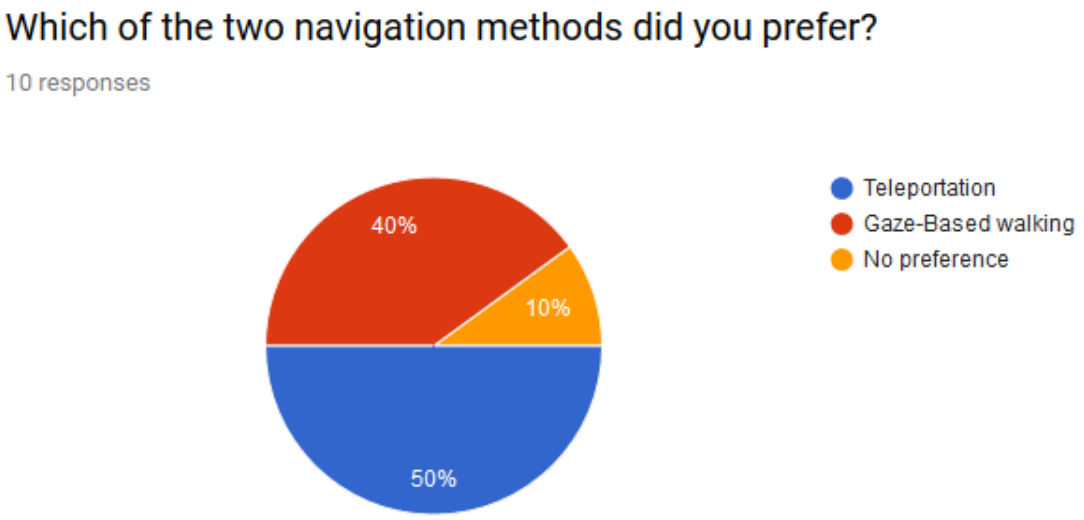


Figure 27: Navigation preference for Mobile VR

In the IVR experiment, the users experienced even more nausea when using Gaze-based walking than they did in MVR experiment, and they were having an overall harder time orienting themselves. Some users felt strong nausea and had to stop the experiment. As such there was a strong preference for teleportation, as seen in figure 28

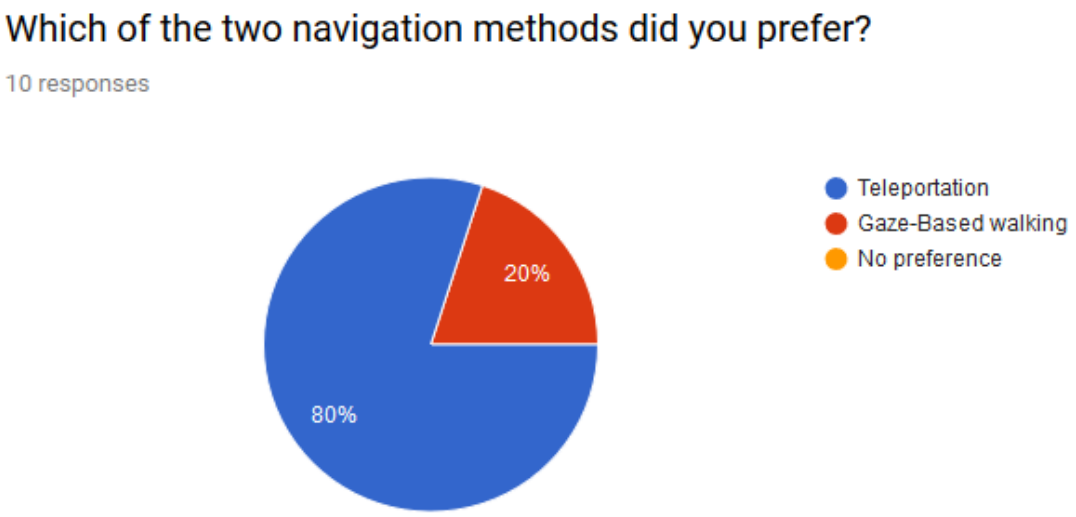


Figure 28: Navigation preference for Installed VR

Since the majority of IVR users, and a small majority of MVR users, preferred teleportation, we will use teleportation as the navigation method for the object-manipulation experiment.

8 Object Manipulation in Virtual Reality

To answer the question asked in the problem statement, section 2.3, we developed a system inspired by the task description in section 3, in which a user has to place furniture on top of ghost objects as accurately as possible, using the platform specific controllers.

The detailed experiment description and results can be found in appendix E - The Object Manipulation Experiment.

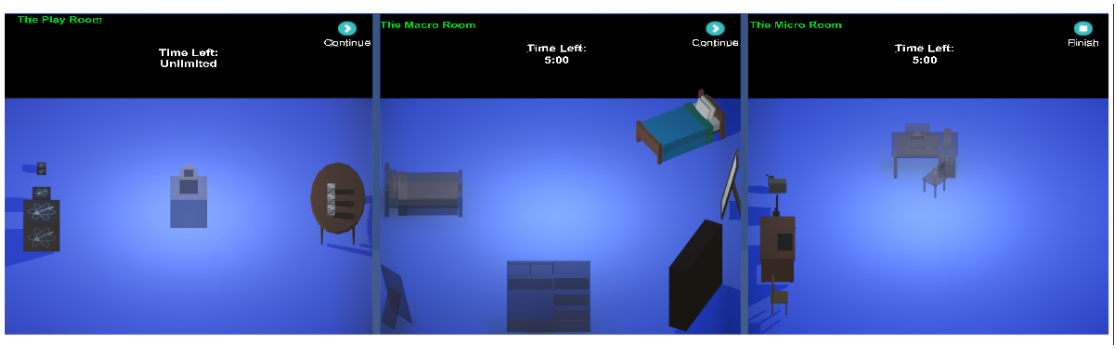


Figure 29: The Object Manipulation experiment room seen from an orthographic view.

8.1 The Experiment Rooms

The experiment had the users go through three rooms; the playroom, the macro room and the micro room.

Figure 30: Orthographic view of the playroom.

The playroom in figure 30, was intended for the users to get familiar with the controls for the platform, making sure they knew how to teleport and manipulate objects using the platform specific methods. The users had to prove that they understood the concept of the tasks and controls, by placing the cubes in the matching ghost object positions. Only when the cubes had been placed within a small margin of error, could the user continue.

When pressing the continue button in the playroom, the user was moved into the macro room (see figure 31).

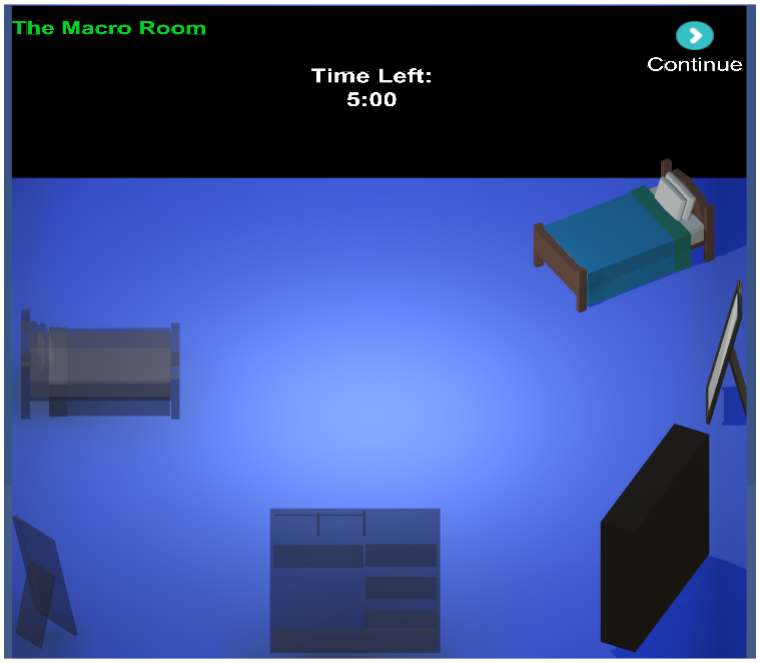


Figure 31: Orthographic view of the macro room.

In this room the user must, as accurately as possible, place the bed, closet, and mirror in the place of the matching ghost furniture. Unlike in the playroom, the continue button was always available to the users in the macro room, so that they could continue when they were satisfied with their placements.

To ensure that users wouldn't spend too much time perfecting the placement of an object, we implemented a time-boxing feature, where a timer was shown on the wall. When this timer ran out, the user could no longer interact with any of the objects in the room, except for the continue button.

When pressing the continue button in the macro room, the user was sent to the final room; the micro room (see figure 32)

Figure 32: Orthographic view of the micro room.

In the micro room, the users once again had to place furniture and other equipment in the allotted ghost object placements. Unlike in the macro room, these objects were small, and had to be stacked on top of each other, adding more complexity to the task. The time boxing feature was used again, to restrict the amount of time the users have to place the furniture.

Once the user was done placing the objects, they could end the experiment by pressing the finish button.

8.2 Object Manipulation on the Mobile Platform

To manipulate objects on the mobile platform, we made use of Google Daydream Element's standard implementation of object manipulation, as this demonstrates the best-practices [41]. The standard implementation makes it possible for the user to pick up, place, rotate and move objects remotely using the controller as a pointer.

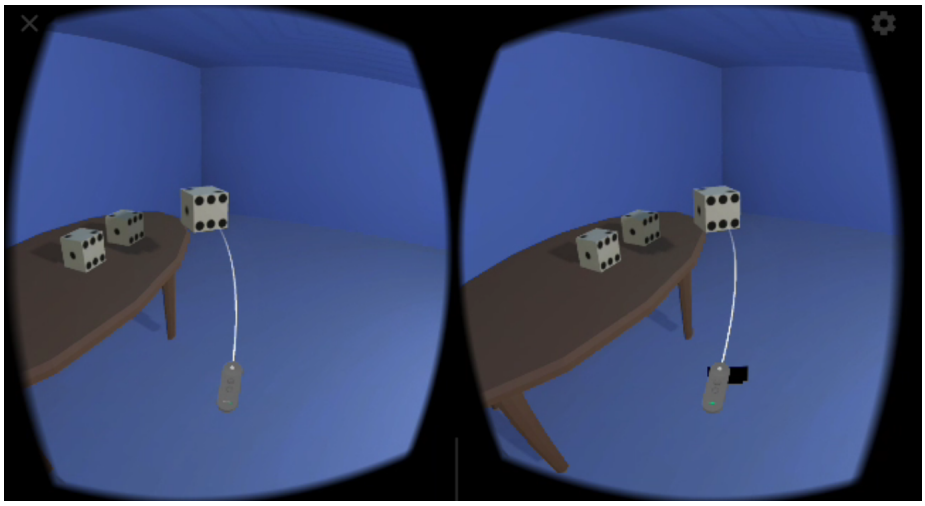


Figure 33: A user picking up an object using Google Daydream Element's standard implementation.

8.3 Object Manipulation on the Installed Platform

Object manipulation on the installed platform was implemented using SteamVR's own implementation of GoGo-manipulation, where the user can pick objects up into their hands, and have the physical movement translated into virtual reality movement. SteamVR's implementation was chosen since it is the standard implementation for hand-metaphor object manipulation, and it fit our non-functional requirements of #2 - Usability nicely.

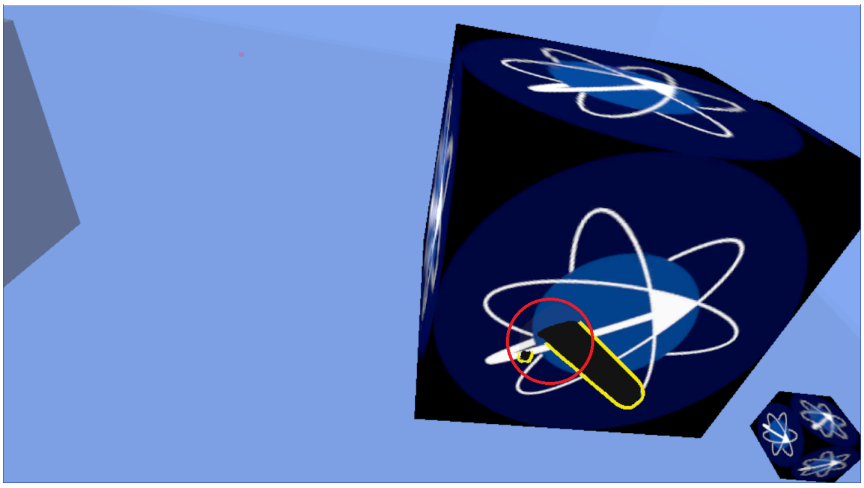


Figure 34: A user picking up an object using SteamVR's hand-metaphor

Besides the standard SteamVR object manipulation method, we also wanted the IVR users to be able to manipulate objects remotely. The only public implementation we found for this, was the Virtual Reality Toolkit (VRTK) [42] from the Unity Asset Store, but this did not have the move- and rotate features that we wanted. For that reason we decided to implement our own Remote Object Manipulation interaction method, where the user can pick up objects at a distance, rotate the objects, and swipe them towards/away from the user. For implementation details and a how-to guide, see appendix F.3 and appendix I.11 for a video demonstration.

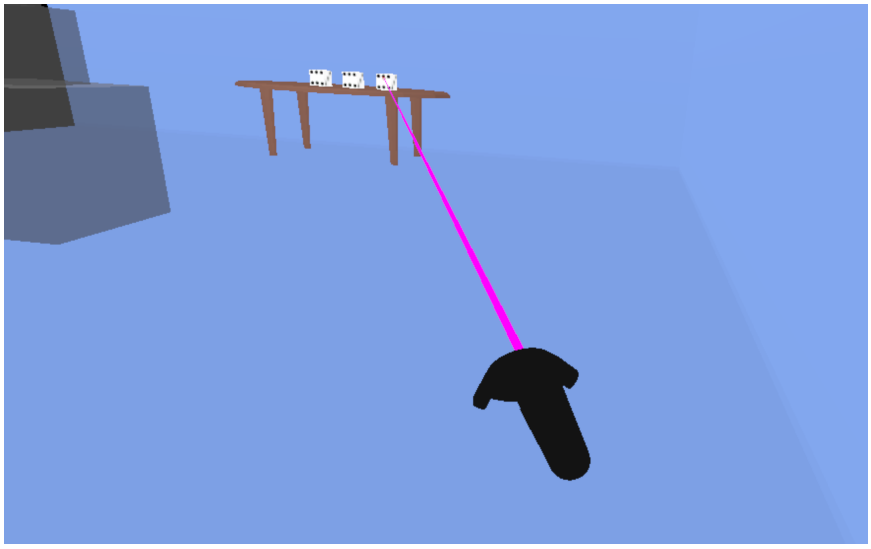


Figure 35: A user picking up a die using remote object manipulation on IVR

8.4 Experiment Data Gathering

Throughout the experiment we gather objective data about the accuracies and actions the user makes. For each furniture/ghost object pair, we get the distance between objects, as well as how aligned their orientation is in percentage. As we group the data by room, we can calculate the average accuracy for each room. Besides accuracy, we also measure how fast a user completes the macro and micro rooms. The full description of the gathered data can be found in appendix E.4

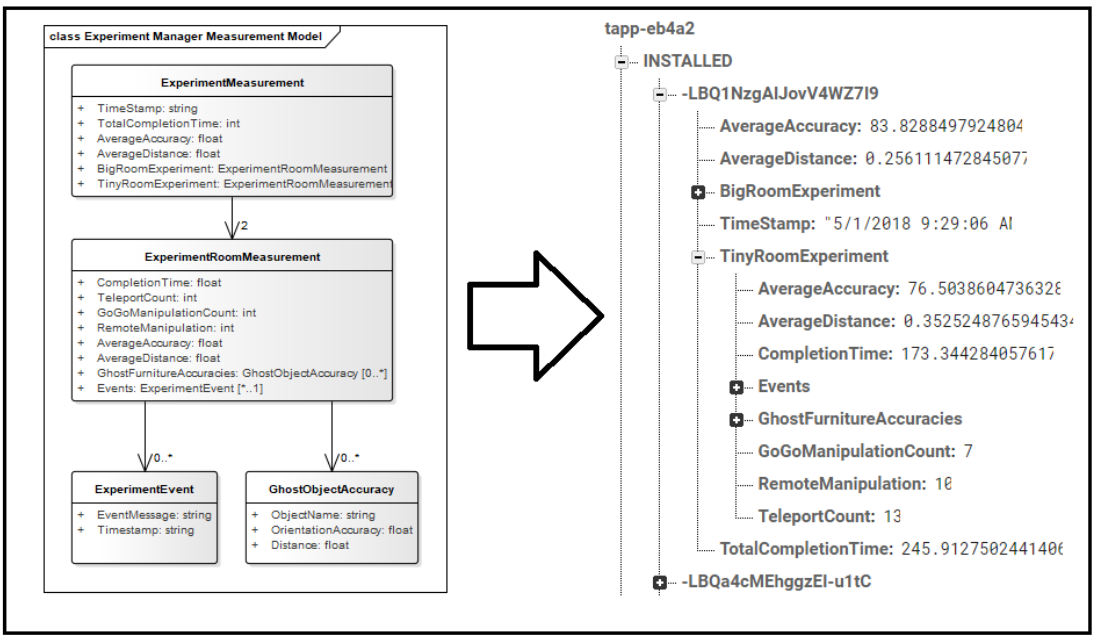


Figure 36: Overview over C# readmodels to firebase entry for the experiment

Besides the objective data, we also gather subjective data through a questionaire that the users fill out after trying the different platforms. This questionaire can be seen in appendix I.5

9 Results

This section will highlight the results gathered from the object manipulation experiment, which will be the basis for the discussion in section 10.

A complete description of the experiment results can be found in appendix E - The Object Manipulation Experiment.

9.1 Measured Results

As described in section 8.4, we have gathered objective, quantitative measurements about the completion time, distance- and orientation accuracy.

We noticed that the data had some extreme outliers, and from looking in the video recordings of the experiment (see appendix I.9), we could see that this was due to users accidentally knocking furniture away. These measurements are misrepresentative of the experiment, and will not be included in the results. See appendix I.8 and I.12 for a full overview of the data and calculations used for our analysis.

In table 9 the averages for the Macro room is presented, and from this we can see that, while IVR slightly outperforms MVR in orientation- and distance, MVR appears to do better in completion time, when placing large furniture.

Platform	Orientation Accuracy (%)	SD	Distance Accuracy (meters)	SD	Completion Time (seconds)	SD
IVR	98.2	2.6	0.117	0.2	126.3	29
MVR	97.6	2.1	0.121	0.1	97.5	37.6

Table 9: Averages for the Macro room, including standard deviations (SD).

With the null-hypothesis being that the averages are equal, we tried to see if the data was statistically significant. Using a 2-tailed student t-test, assuming unequal variance, we found that there was a statistically significant difference between the average completion times, as the p-value was 0.024. We found no significant difference in distance- and orientation accuracy. The calculation can be found in appendix I.8.

When placing tiny objects in the Micro room, IVR outperforms MVR in all measurements, as can be seen in table 10.

Platform	Orientation Accuracy (%)	SD	Distance Accuracy (meters)	SD	Completion Time (seconds)	SD
IVR	94.8	9.2	0.132	0.290	139.8	26.4
MVR	92.2	7.0	0.324	0.433	168.5	36.4

Table 10: Averages for the Micro room, including standard deviations (SD).

Using the null-hypothesis of the means being equal, we ran a two-tailed student t test, assuming unequal variance. Again we found that the averages for completion time was statistically significant, as the p-value was 0.018. For the micro room, there was no significant difference in distance- and orientation accuracy.

9.2 Questionaire Results

The users answered a questionaire, containing subjective likert-scale, and qualitative open questions, to help us asses the perceived user satisfaction of object manipulation on the two platforms. The questionaires can be found in appendix I.5.

Perceived User Satisfaction for IVR
On the installed platform 76.2% of the users answered that they agree or strongly agree with the statement:

> "*I found it easy to place **large** objects with the accuracy I wanted*"
> (Appendix E.3, page 142, figure 133)

80.9% of the users agreed or strongly agreed with the statement:

> "*I found it easy to place **tiny** objects with the accuracy I wanted*"
> (appendix E.3, page 143, figure 134)

When asked to comment on the IVR object manipulation, the users generally had two common topics; the roomscale and GoGo-manipulation made for a better experience as it was intuitive to pick up objects, and that remote object manipulation sensitivity for swiping and rotation felt "off".

Perceived User Satisfaction for MVR
On the mobile platform, only 28.3% of the users agreed or strongly agreed with the statement:

> "*I found it easy to place **large** objects with the accuracy I wanted*"
> (appendix E.3, page 139, figure 128)

14.3% of the users agreed or strongly agreed with the statement:

> "*I found it easy to place **tiny** objects with the accuracy I wanted*"
> (appendix E.3, page 140, figure 129)

When asked to comment on MVR, we found two common topics that got mentioned by multiple users. One being, that the sensitivity of the swipe and rotation on the controller felt mismatched - it was hard to get the desired accuracy when placing objects, and sometimes the object rotated when trying to swipe (and vice versa.). The second comment was, that people felt MVR was less intuitive since there was no roomscale tracking. People wanted to be able to make minor adjustments to their position, without having to drop what they were holding to do it.

A summation of the percieved easiness on placing large and tiny objects on the two platforms can be seen in table 11.

Platform	Large Objects	Tiny Objects
IVR	76.2%	80.9%
MVR	28.3%	14.3%

Table 11: Summation of the easiness of placing large and tiny objects

General Remarks
When asked which platform the users preferred, all but one (who had no preference) answered that they preferred HTC Vive (Installed VR), see figure 37.

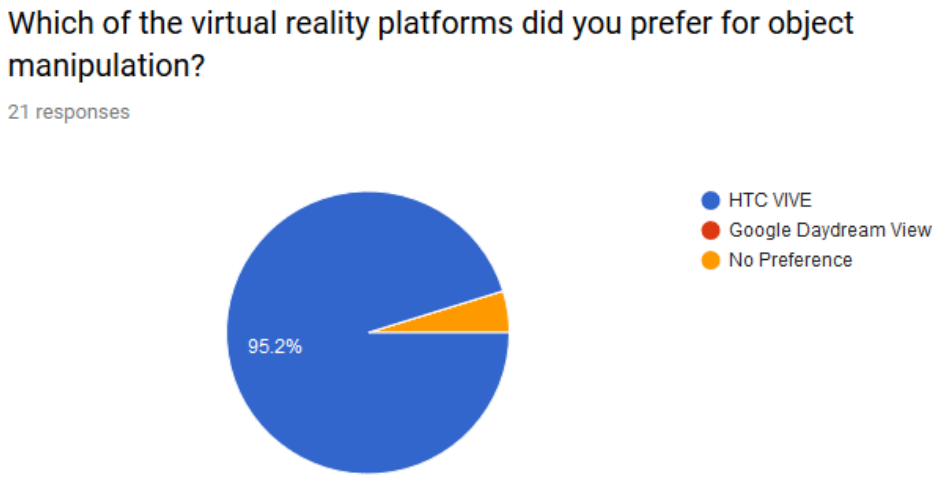


Figure 37: Distribution of preferred platform for object manipulation.

When asked about the greatest difference between the two platforms, some users gave the following answers.

> *"I prefer the HTC VIVE because it seems more intuitive more object manipulation that you can pick up objects with the controllers. On the other hand, I really like the wireless setup for the Google Daydream."*
>
> *"The VIVE felt more like an extension of myself, where the Daydream felt more like me trying to fight with the controller. It was to easy to do actions by a mistake on the daydream"*
>
> *"The sense of freedom i had because i could move around was a great thing in the HTC Vive platform. The controllers were also much nicer i the HTC Vive. I didn't have any problems with the focus when i tried the HTC Vive."*

10 Discussion

The results show that MVR and IVR are close in distance- and orientation accuracy, with the caveat that the experiment might suffer from implementation issues in both software and design. The results did also show, that MVR was faster than IVR when placing large furniture, while slower when placing tiny objects, and that there is a statistical significant difference between the means. Again, this data might be skewed due to issues with the experiment.

Some of the problems might stem from the relatively small sample size of the experiment. A larger sample size could possibly have given a more general depiction of the performance of the two platforms. Another aspect is the users' backgrounds. As most of the users come from a technical background, they have an above average experience with virtual environments and controls, and it could potentially skew the data.

Users completed the experiment in an A/B split, where one group started with IVR, and the other with MVR. This was done in an attempt to eliminate user bias that might come from trying a specific platform first. We did not map the user questionaire's to the MVR and IVR measurements, and as such we lost valuable information about user performance - something we know to fix in future iterations.

From our user feedback, and our experiment observations, we identified multiple areas of our implementation that could be improved with more iterations of our lean process.

The sensitivity of rotation and swiping using remote object manipulation as well as the predefined positions of ghost objects, could be improved with minor changes. Some users found the sensitivity unsatisfactory, and others had difficulty aligning furniture properly with certain ghost objects.

We also observed that the physics on furniture in the Macro Room had a very negative impact on some measurements, as users accidentally bumped furniture which they had already placed, thereby moving them, with little to no time left to get the placement right again.

We could also create a better tutorial in the Playroom, as we feel the conditioning was not properly structured. Many users would leave the playroom and forget the controls during the experiment, which we believe could be prevented if the playroom had a proper sequenced tutorial, guiding the users through object manipulation and navigation on the specific VR platform.

The completion time recorded in the measurements was not entirely representational as users had to manually stop the experiment for the measurement to be recorded. Some users spent time after having completed the experiment before realizing that they had to stop it by pressing a button.

As we learned in the analysis of our project, three important aspects of virtual reality are object manipulation, selection, and navigation. Through our navigation experiment, we

attempted to improve the quality of the object manipulation experiment by choosing the method of navigation which users would most prefer. In the object manipulation experiment we had no negative feedback on the navigation method. MVR users did however state that they needed a better way of performing micro-adjustments to their position, in lieu of roomscale.

The majority of users preferred the HTC Vive when asked about user satisfaction. While tweaks to our implementation could potentially alleviate MVR from some of the most common complaints, it cannot yet bring the much wanted roomscale functionality and freedom that the VR users prefer.

11 Conclusion

VR is a growing technology, but the steep price and extensive setup of an installed system makes it interesting to investigate the usefulness of the cheaper and more portable option of mobile VR systems. The purpose of the project was to compare object manipulation on mobile- and installed VR using the metrics of accuracy, efficiency and user satisfaction. To achieve this, we built an experimental system for conducting manipulation tasks in defined spaces, using the mobile VR platform Google Daydream View and installed VR platform HTC VIVE.

We gathered data from 21 users completing the experiment. The results indicated, that there was no statistically significant difference between the distance- and orientation accuracy for the two platforms across the micro- and macro rooms. There was a statistically significant difference in completion times for the two rooms, where MVR outperformed IVR when placing large objects in the macro room. These data could indicate, that the difference in performance between the platforms is smaller than first assumed, and that there are potential use-cases where mobile VR can be a viable alternative to installed VR. The experiment was adversely affected by different factors, such as software implementation, experimental design, and user bias, and as such the data cannot be used to make a definitive statement about mobile VR's performance in relation to object manipulation.

From the object manipulation questionaire regarding the user satisfaction of the two platforms, we found that users vastly preferred IVR, as users felt it easier to achieve the desired accuracy. Users commented that a much desired feature of the HTC Vive compared to the Google Daydream View was roomscale, as it gave a sense of freedom and felt natural and intuitive.

The project was developed using a lean-inspired development process to involve users early and often, and while we believe that it helped us create a better experiment, more time and iterations would have helped us ensure the quality of the experiment, and in extension, the experiment results.

Due to the inherently component-based design of Unity gameobjects/scripts, a by-product of the object manipulation experiment system, was the Experiment Toolbox. This toolbox consists of generic elements, such as the manipulation controls and ghost object system, that can be used to conduct new VR experiments, or further aid the investigation of object manipulation.

12 Future Work

If we had more time, we would have liked to take the object manipulation experiment through more iterations of our development process, in order to correct the lacking areas that we discovered during execution of our user experiment. It would also have been preferable to run the experiment with a larger sample size, which is a time consuming task.

The experiment dataset could also be expanded with a correlation ID for a user across measurements of different platforms. This would enable us to do more in-depth analysis and statistical calculations, which could prove useful. Another option would be to restructure the user experiment by separating users across both platforms to make the sets independent, in order to completely eliminate bias of the user's platform preference.

For the object manipulation experiment, we developed the proof-of-concept live experiment app. While it was useful being able to follow users on the 2D map, this app could be further expanded by, for example, having a true 3D live feed of the users vision.

Since the data gathered in our object manipulation experiment was less than conclusive, more experiments could be conducted with the use of our Experiment Toolbox, as it is still relevant identify areas where mobile VR is "good enough" to be preferred to IVR. The Experiment Toolbox could also be used to create new VR experiments in other areas, to help further the field of virtual reality.

References

[1] Poly by Google, Apr 2018. [Online; accessed 2. Apr. 2018] https://poly.google.com/user/4aEd8rQgKu2.

[2] Creative Commons — Attribution 2.0 Generic — CC BY 2.0, May 2018. [Online; accessed 28. May 2018]: https://creativecommons.org/licenses/by/2.0.

[3] Michael A. Gigante. Virtual reality: Definitions, history and applications. *Virtual Reality Systems*, June 2 1993.

[4] Minecraftpsyco. The sensorama machine. https://en.wikipedia.org/wiki/Sensorama#/media/File:Sensorama-morton-heilig-virtual-reality-headset.jpg.

[5] Jesse Fox, Dylan Arena, and Jeremy N. Bailenson. Virtual reality a survival guide for the social scientist, 2009.

[6] Enrico Gobetti and Riccardo Scateni. Virtual reality: Past, present and future. *Studies in health technology and informatics*, February 1998.

[7] Grigore C. Burdea and Philippe Coiffet. *Virtual Reality Technology, Volume 1*, volume 1, chapter 2. John Wiley & Sons, June 2003.

[8] File:ManusVR Glove 2016.png - Wikimedia Commons, May 2018. [Online; accessed 27. May 2018]: Manus VR - https://commons.wikimedia.org/wiki/File:ManusVR_Glove_2016.png - Under license https://creativecommons.org/licenses/by-sa/4.0/deed.en.

[9] Alan B. Craig, William R. Sherman, and Jeffrey D. Will. *Developing Virtual Reality Applications: Foundations of Effective Design*. Morgan Kaufmann, 1 edition, 2009. https://books.google.dk/books?hl=en&lr=&id=2P91gPYr5KkC&oi=fnd&pg=PP1&dq=virtual+reality+applications&ots=kmeWCQEV9h&sig=qWYLMxwt3LJwX38BIWZK7kQkqUY&redir_esc=y#v=onepage&q&f=false.

[10] Mark R. Mine. Virtual environment interaction techniques. 1995. http://lsc.univ-evry.fr/~davesne/ens/pub/virtual_environment_interaction_techniqu_129302.pdf.

[11] Doug A. Bowman and Larry F. Hodges. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. April 27 1997. https://dl.acm.org/citation.cfm?id=253301.

[12] D. A. Bowman and R. P. McMahan. Virtual reality: How much immersion is enough? *Computer*, 40(7):36–43, July 2007.

[13] F. D. Rose, E. A. Attree, B. M. Brooks, D. M. Parslow, and P. R. Penn. Training in virtual environments: transfer to real world tasks and equivalence to real task training. *Ergonomics*, 43(4):494–511, 2000. PMID: 10801083.

[14] Photos, May 2018. [Online; accessed 27. May 2018]: http://www.af.mil/News/Photos/igphoto/2001864957/mediaid/2440291.

[15] Reality check ESA384313 - Virtual reality - Wikipedia, May 2018. [Online; accessed 27. May 2018]: ESA - CC BY-SA 3.0 IGO - https://en.wikipedia.org/wiki/Virtual_reality#/media/File:Reality_check_ESA384313.jpg.

[16] Virtual Reality Demo, May 2018. [Online; accessed 27. May 2018]: NIH Image Gallery - https://www.flickr.com/photos/nihgov/28984638442 - Under license https://creativecommons.org/licenses/by-nc/2.0/.

[17] Samsung's Virtual Reality MWC 2016 Press Conference, May 2018. [Online; accessed 27. May 2018]: Maurizio Pesce - https://www.flickr.com/photos/pestoverde/26666393696 - Under license https://creativecommons.org/licenses/by/2.0/.

[18] Jonathan Steuer. Defining virtual reality: Dimensions determining telepresence. *Journal of Communication*, December 1992. http://onlinelibrary.wiley.com/doi/10.1111/j.1460-2466.1992.tb00812.x/full.

[19] The Lean Startup | Methodology, Oct 2017. [Online; accessed 9. Mar. 2018]: http://theleanstartup.com/principles.

[20] Pair Programming, Apr 2018. [Online; accessed 10. Apr. 2018]: http://www.extremeprogramming.org/rules/pair.html.

[21] Wikipedia. Samsung gear vr, 18 January 2018. https://en.wikipedia.org/wiki/Samsung_Gear_VR.

[22] Google. Smartphone vr, February 2018. https://vr.google.com/daydream/smartphonevr/.

[23] Maurizio Pesce. Daydream view vr headset made by google, October 4 2016.

[24] Game engine - Wikipedia, February 21 2018. [Online; accessed 22. February. 2018].

[25] Unity3d 3 editor, September 29 2010. [Online; accessed 27. Feb. 2018]: https://www.flickr.com/photos/epredator/5036240068. License for photo: https://creativecommons.org/licenses/by/2.0/.

[26] Unity - Manual: Importing Assets, Feb 2018. [Online; accessed 27. Feb. 2018].

[27] Unreal Motion Controller Support - Google VR, Dec 2017. [Online; accessed 27. Feb. 2018] : https://developers.google.com/vr/develop/unreal/arm-model.

[28] Home | Mono, Feb 2018. [Online; accessed 28. Feb. 2018] : http://www.mono-project.com.

[29] Firebase Realtime Database | Store and sync data in real time | Firebase, Feb 2018. [Online; accessed 27. Feb. 2018] https://firebase.google.com/products/realtime-database.

[30] Serverless Architectures, Apr 2018. [Online; accessed 16. May 2018]: https://martinfowler.com/articles/serverless.html-.

[31] Prof. Dr. Stefan Edlich. NOSQL Databases, May 2018. [Online; accessed 16. May 2018]: http://nosql-database.org.

[32] GitHub. Github. https://github.com/. Tilgået: 01-02-2018.

[33] Zenject, Mar 2018. [Online; accessed 26. Mar. 2018]: https://github.com/modesttree/Zenject.

[34] Publish–subscribe pattern - Wikipedia, May 2018. [Online; accessed 30. May 2018]: https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern.

[35] The Repository Pattern, May 2018. [Online; accessed 16. May 2018]: https://msdn.microsoft.com/en-us/library/ff649690.aspx.

[36] SOA Patterns, May 2018. [Online; accessed 16. May 2018]: http://soapatterns.org/design_patterns/service_callback.

[37] The Lab on Steam, May 2018. [Online; accessed 14. May 2018] https://store.steampowered.com/app/450390/The_Lab.

[38] COMPOUND on Steam, May 2018. [Online; accessed 14. May 2018] https://store.steampowered.com/app/615120/COMPOUND.

[39] Waltz of the Wizard on Steam, May 2018. [Online; accessed 14. May 2018] https://store.steampowered.com/app/436820/Waltz_of_the_Wizard.

[40] Daniel Brenners. Interaction design for mobile virtual reality, Aug 2016. [Online; accessed 22. Mar. 2018]: https://www.ischool.berkeley.edu/sites/default/files/projects/danielbrenners_mims_final_report.pdf.

[41] Google. Daydream elements | google vr | google developers. https://developers.google.com/vr/elements/overview#object_interaction. (Accessed on 05/01/2018).

[42] VRTK - Virtual Reality Toolkit - [VR Toolkit] - Asset Store, May 2018. [Online; accessed 28. May 2018]: https://assetstore.unity.com/packages/tools/vrtk-virtual-reality-toolkit-vr-toolkit-64131.

[43] Felix Richter. Who leads the virtual reality race?, September 7 2017. https://www.statista.com/chart/11006/vr-and-ar-headset-shipments/.

[44] IDC. Worldwide shipments of av/vr headsets maintain solid growth trajectory in the second quarter, according to idc, September 5 2017. https://www.idc.com/getdoc.jsp?containerId=prUS43021317.

[45] Gear VR, May 2018. [Online; accessed 14. May 2018]: https://www.flickr.com/photos/samsungtomorrow/32749196510. Used under the CC BY-NC-SA 2.0 License: https://creativecommons.org/licenses/by-nc-sa/2.0/.

[46] File:Samsung Unpacked 2017 Gear VR.jpg - Wikimedia Commons, May 2018. [Online; accessed 14. May 2018]: https://commons.wikimedia.org/wiki/File:Samsung_Unpacked_2017_Gear_VR.jpg. Used under the CC BY-NC-SA 2.0 License: https://creativecommons.org/licenses/by-nc-sa/2.0/.

[47] Samsung. Specs. http://www.samsung.com/global/galaxy/gear-vr/specs/.

[48] Samsung Galaxy S6, Black Sapphire 32GB (Verizon Wireless), May 2018. [Online; accessed 18. May 2018]: https://www.amazon.com/Samsung-Galaxy-S6-Sapphire-Wireless/dp/B00V7FXCZ2.

[49] Google Store, May 2018. [Online; accessed 21. May 2018]: https://store.google.com/us/product/google_daydream_view?hl=en-US.

[50] Google Store, May 2018. [Online; accessed 21. May 2018]: https://store.google.com/us/product/google_daydream_view_specs?hl=en-US.

[51] Sean Hollister. ZTE Axon 7 is now the cheapest Google Daydream VR-ready phone. *CNET*, Feb 2017.

[52] ZTE Axon 7, May 2018. [Online; accessed 21. May 2018]: https://www.zteusa.com/axon-7.

[53] Oculus. Oculus rift, February 12 2018. https://www.oculus.com/rift.

[54] Oculus-Rift-Touch-Controllers-Pair - File:Oculus-Rift-Touch-Controllers-Pair.jpg - Wikipedia, May 2018. [Online; accessed 14. May 2018]: https://en.wikipedia.org/wiki/File:Oculus-Rift-Touch-Controllers-Pair.jpg#/media/File:Oculus-Rift-Touch-Controllers-Pair.jpg. Public domain.

[55] File:Oculus-Rift-CV1-Headset-Front.jpg - Wikimedia Commons, Apr 2018. [Online; accessed 14. May 2018]: https://commons.wikimedia.org/wiki/File:Oculus-Rift-CV1-Headset-Front.jpg. Public domain.

[56] File:Oculus-Rift-CV1-Sensor-wStand.jpg - Wikipedia, May 2018. [Online; accessed 14. May 2018]: https://en.wikipedia.org/wiki/File:Oculus-Rift-CV1-Sensor-wStand.jpg. Public domain.

[57] Wikipedia. Oculus rift, January 28 2018. https://en.wikipedia.org/wiki/Oculus_Rift.

[58] Wikipedia. Room scale, February 6 2018. https://en.wikipedia.org/wiki/Room_scale.

[59] Oculus Ready PCs | Oculus, May 2018. [Online; accessed 18. May 2018]: https://www.oculus.com/oculus-ready-pcs/#pc-offers.

[60] CyberPowerPC - Gamer Ultra Desktop - AMD Ryzen 5 1400 - 8GB Memory - AMD Radeon RX 580 - 1TB Hard Drive - Black, May 2018. [Online; accessed 18. May 2018]: https://www.bestbuy.com/site/cyberpowerpc-gamer-ultra-desktop-amd-ryzen-5-1400-8gb-memory-amd-radeon-rx-580-1tb-ha 5833100.p?skuId=5833100.

[61] Vive. Vive vr system, February 12 2018. https://www.vive.com/us/product/vive-virtual-reality-system/.

[62] HTC Vive Now Up For Pre-Order, May 2018. [Online; accessed 14. May 2018]: https://www.flickr.com/photos/bagogames/25845851080. Used under the CC BY-NC-SA 2.0 License: https://creativecommons.org/licenses/by-nc-sa/2.0/.

[63] Unity - Manual: Scenes, Mar 2018. [Online; accessed 29. Mar. 2018]: https://docs.unity3d.com/Manual/CreatingScenes.html.

[64] Unity - Manual: GameObject, Mar 2018. [Online; accessed 29. Mar. 2018]: https://docs.unity3d.com/Manual/class-GameObject.html.

[65] Unity - Scripting API: MonoBehaviour, Mar 2018. [Online; accessed 29. Mar. 2018]: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html.

[66] Unity - Manual: Prefabs, Mar 2018. [Online; accessed 29. Mar. 2018]: https://docs.unity3d.com/Manual/Prefabs.html.

[67] Unity - Manual: The Inspector window, Mar 2018. [Online; accessed 29. Mar. 2018]: https://docs.unity3d.com/Manual/UsingTheInspector.html.

[68] Unity - Manual: The Game view, Mar 2018. [Online; accessed 29. Mar. 2018]: https://docs.unity3d.com/Manual/GameView.html.

[69] Unity - Manual: Asset Packages, Mar 2018. [Online; accessed 29. Mar. 2018]: https://docs.unity3d.com/Manual/AssetPackages.html.

[70] Unity - The Asset Store, Mar 2018. [Online; accessed 29. Mar. 2018]: https://unity3d.com/learn/tutorials/s/asset-store.

[71] Quickstart for Google VR SDK for Unity with Android | Google VR, Feb 2018. [Online; accessed 28. Mar. 2018]: https://developers.google.com/vr/develop/unity/get-started.

[72] Google VR SDK for Unity | Google VR, Dec 2017. [Online; accessed 28. Mar. 2018]: https://developers.google.com/vr/unity/reference.

[73] SteamVR Plugin - Asset Store, Mar 2018. [Online; accessed 28. Mar. 2018]: https://assetstore.unity.com/packages/templates/systems/steamvr-plugin-32647.

[74] ValveSoftware/openvr, Mar 2018. [Online; accessed 28. Mar. 2018]: https://github.com/ValveSoftware/openvr.

[75] blender.org - Home of the Blender project - Free and Open 3D Creation Software, Mar 2018. [Online; accessed 28. Mar. 2018]: https://www.blender.org.

[76] Unity - Manual: Unity Test Runner, Mar 2018. [Online; accessed 26. Mar. 2018]: https://docs.unity3d.com/Manual/testing-editortestsrunner.html.

[77] nunit, Mar 2018. [Online; accessed 26. Mar. 2018]: https://github.com/nunit/nunit.

[78] Unity - Services - Cloud Build, Mar 2018. [Online; accessed 26. Mar. 2018]: https://unity3d.com/unity/features/cloud-build.

[79] UV mapping - Wikipedia, Mar 2018. [Online; accessed 28. Mar. 2018]: https://en.wikipedia.org/wiki/UV_mapping.

[80] Unity - Manual: Input for OpenVR controllers, Mar 2018. [Online; accessed 27. Mar. 2018] https://docs.unity3d.com/Manual/OpenVRControllers.html.

[81] GvrControllerInput Class Reference | Google VR, Dec 2017. [Online; accessed 29. Mar. 2018] https://developers.google.com/vr/unity/reference/class/GvrControllerInput.

[82] SteamVR Plugin - Asset Store, Mar 2018. [Online; accessed 8. Mar. 2018]: https://assetstore.unity.com/packages/templates/systems/steamvr-plugin-32647.

[83] Unity - Unity Teams, Mar 2018. [Online; accessed 8. Mar. 2018]: https://unity3d.com/teams.

APPENDIX

Appendix A VR Technology Comparison

It is important to survey the current market of most popular Virtual Reality systems, in order to gain an understanding of the current possibilities for users. This insight can help highlight potentially interesting areas to investigate.

According to statistics from the *International Data Corporation (IDC)* [43] [44] the worldwide unit shipments of AR and VR headsets in 2017, categorized in brands, are as presented in figure 38.

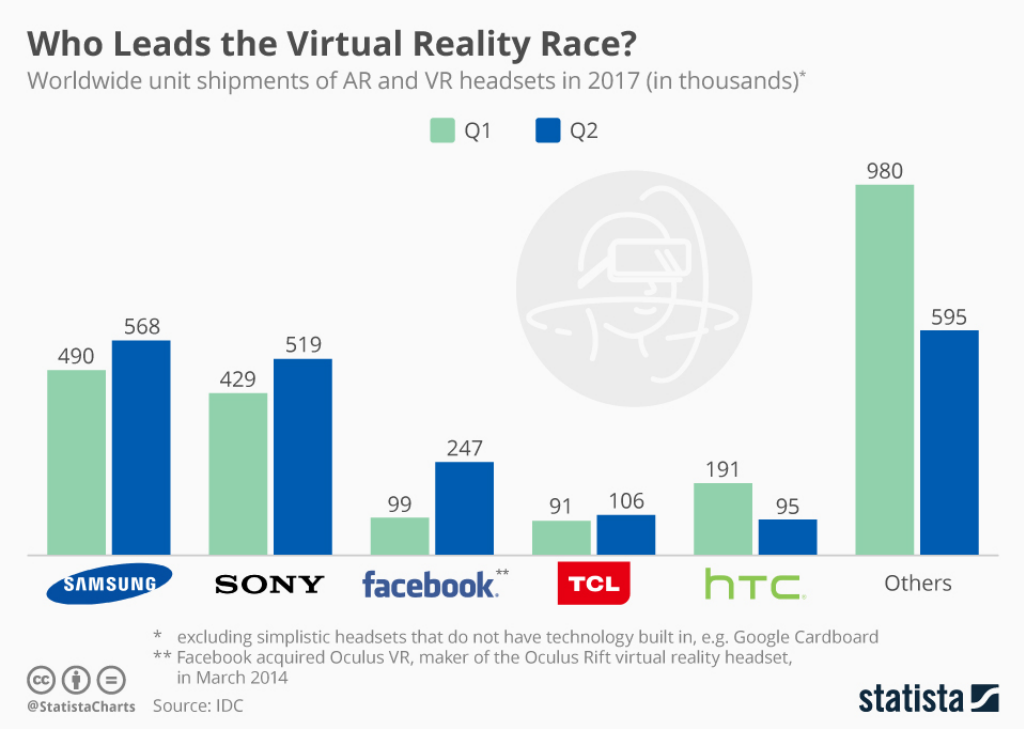


Figure 38: Worldwide Unit Shipments of AR and VR headsets in 2017 (in thousands). Source - IDC [43]

It should be noted that the brands of figure 38 refer to the Virtual Reality headsets mentioned in table 12.

Brand	Virtual Reality Headset
Samsumg	Gear VR
Sony	Playstation VR
Facebook	Oculus Rift
TCL	Alcatel VR
HTC	Vive

Table 12: Top 5 Virtual Reality headset brands.

These five companies have been picked as the top 5 VR and AR companies worldwide by IDC based on market share. [44]. This information is presented in table 13.

Company	2Q17 Shipment Volumes	2Q17 Market Share
1. Samsung	568.0	26.7%
2. Sony	519.4	24.4%
3. Facebook	246.9	11.6%
4. TCL	106.4	5.0%
5. HTC	94.5	4.4%
Others	594.8	27.9%
Total	2130.0	100.0%

Table 13: Worldwide Quarterly AR/VR Headset Tracker. Source: IDC, September 5, 2017 [44].

The following sections will analyse the technical specifications, and the system requirements, for some of the headsets which we found relevant for this project.

A.1 Samsung Gear VR

Samsung's Gear VR is a mobile Virtual Reality (MVR) headset. It is developed in collaboration with Oculus. By MVR headset, it is meant that the headset is to be used with smartphones. [21]

Whilst a relatively new addition [21], Gear VR supports wireless controller input with touchpad and back buttons. This can be seen in figure 39.

Figure 39: Samsung Gear VR with included controller. Photos by Samsung Newsroom [45] and Wikipedia. [46]

From here, it can be seen that the controller consists of a trigger button located at the back. Additionally, a trackpad can be found in front, with an additional count of 4 buttons.

The Gear VR headset has a Gyro sensor and an accelerometer [47].

Samsung Gear VR requires a Samsung Galaxy phone, with the newest model being Note8 and the earliest model being S6. [47]

Gear VR is currently set at a price of $129.99 USD.

Were you to use Samsung Gear VR with the minimum required smartphone, it would be the Samsung Galaxy S6. Looking at marketplaces such as Amazon, the Samsung Galaxy S6 has a starting price tag of $236 [48].

A.2 Google Daydream View

Google's Daydream View is, like the Samsung Gear VR, also an MVR headset. [49] Whilst Google did not have a spot for their MVR headset in the IDC top 5 worldwide

AR and VR headset companies, it is similar to the Samsung Gear VR in what is included. It is also a more recently released headset, and since it is from a large well-established company like Google, we thought it relevant to include in this comparison.

The Google Daydream View and its included controller can be seen in figure 40.

Figure 40: Google's Daydream View headset [23]

The wireless controller of the Google Daydream View consists of a total of 5 buttons. Of these, 2 of them are volume controls, and one is used as a home button. Google Daydream View uses the gyro sensor and an accelerator of the smartphone. [50]

The Google Daydream View requires a Daydream-ready phone. This includes phones such as: Google Pixel 2, Google Pixel, Samsung Galaxy S8, and more. [50]

It is suggested that the, or one of the, currently cheapest avaliable Daydream-ready phone is the ZTE Axon 7. [51] This smartphone currently stands at a price of $399.98. [52]

A.3 Oculus Rift

Facebook's Oculus Rift is an installed virtual reality (IVR) headset for the PC. [53]

When you purchase the headset, the following items will be found in the box: [53]

- The Headset
- A pair of touch controllers

- Two sensors

The hardware can be seen in figure 41.

Figure 41: The Oculus Rift headset with two touch controllers and a a sensor. [54] [55] [56]

The touch controllers go by the name Oculus Touch. Each controller consists of an analog stick, three buttons, and two triggers. [57]

The controllers are also fully tracked in 3D space relative to your position using two, or more, sensors. This means that they provide a realistic representation of your hands in physical space, relative to you. [57].

The sensor system is known as the *Constellation System* [57]. Besides being used to track the Oculus Touch held by the users, it is also used to support a system known generally in Virtual Reality literature as *Roomscale*.

Roomscale is called a design paradigm in Virtual Reality which allows a user to physically walk around within a defined play-area, with that movement traslating to movement within the virtual world. [58]

As already stated, Oculus Rift is a Virtual Reality headset for the desktop platform. Thus in order to use it, you will need a desktop PC. In table 14, the official minimum recommended specifications, highlighting the Graphics Card, CPU, and Memory is shown.

Graphics Card	NVIDIA GTX 1050Ti / AMD Radeon RX 470 or greater
Alternative Graphics Card	NVIDIA GTX 960 / AMD Radeon R9 or greater
CPU	Intel i3-6100 / AMD Ryzen 3 1200, FX4350 or greater
Memory	8GB+ RAM

Table 14: The official minimum recommended specifications for a desktop platform driving the Oculus Rift. [53]

From the official website of Oculus Rift, the current price for the headset, including two controllers and two sensors, is $399 USD.

Looking at PC's which contains the minimum required hardware, the official Oculus Rift website has their own listing of VR Ready PC's [59]. Of these, the cheapest on offer is a PC by the name of *Gamer Ultra*, which can be aquired for $749.99 [60]. This PC supports at least the minimum recommended specifications.

A.4 HTC Vive

In the same vein as the Oculus Rift, the HTC Vive is a Virtual Reality headset for the desktop platform. [61].

When you purchase the headset, the following items will be found in the box [61]:

- The Headset
- A pair of touch controllers
- Two sensors

The hardware can be seen in figure 42.

Figure 42: The HTC Vive Hardware. Photo by BagoGames. [62]

As can be seen, the hardware setup is essentially identical to the Oculus Rift covered in the previous section. In order to avoid unnecessary repetition for this comparison, nothing further will be said of the HTC Vive hardware setup.

Table 15 lists HTC Vive's official recommended minimum specifications.

Graphics	NVIDIA GTX 1060 / AMD Radeon RX 480, equivalent or greater
CPU	Intel i5-4590 / AMD FX 8350, equivalent or greater
Memory	4GB RAM or more

Table 15: The minimum recommended PC specifications for the HTC Vive. [61]

A notable difference from the Oculus Rift is the price. Currently, the HTC Vive is set at a price of $599 USD.

The VR ready PC officially listed by Oculus Rift, mentioned in the previous section on the Oculus Rift hardware analysis, also adheres to the minimum recommended specifications of the HTC Vive. Thus, a relatively cheaper PC for the HTC Vive would also be $749.99.

A.5 Conclusion

By examining aspects of the most popular Virtual Reality systems on the market today, it is possible to compare them in order to gain an overview of advantages and disadvantages between them.

In comparing the different systems, highlights have been chosen as for what we believe is most relevant in the comparison. The variables that have been chosen are: **Price**, **Platform**, **Amount of Controllers**, **Programmable Buttons on Individual Controllers**, **Roomscale**, **Spatial Control Tracking**.

The result of this comparison can be found in table 16.

Brand	Price In USD	Platform	Amount of Controllers	Programmable Buttons on individual controllers	Roomscale	Spatial Control Tracking
Samsung Gear VR	$169.98	MVR	1	3	No	No
Google Daydream View	$99	MVR	1	3	No	No
Oculus Rift	$399	IVR	2	7	Yes	Yes
HTC Vive	$599	IVR	2	4	Yes	Yes

Table 16: Comparison of examined Virtual Reality systems.

Furthermore, it is also interesting to consider the combined price of MVR and IVR with minimally viable devices for which to run the virtual environments. With the prices of minimally viable devices found in the previous analysis, a comparison can be seen in table 17.

Brand	Standalone Price	Price of minimum compatible device	Combined Price
Samung Gear VR	$169.99	$236	$405.99
Google Daydream View	$99	$399.98	$498.98
Oculus Rift	$399	$749.99	$1148.99
HTC Vive	$599	$749.99	$1348.99

Table 17: Price differences taking minimum viable device price into account.

From table 16 it is clear to see that MVR headsets, through Google Daydream View or the Samsung Gear VR, is the cheaper option compared to IVR headsets.

When also considering the price of the device which the virtual reality environments has to execute on, as seen in table 17, it can be still be seen that the MVR platform is realistically a cheaper route to go in order to experience virtual reality. The IVR platform has significant cost differences when it comes to both the headset and device required to use it.

While the MVR platform is the cheaper option, this also comes with some limitations compared to IVR. For one, the user only has a single controller with 3 programmable buttons. Compared with systems like the HTC Vive and Oculus Rift, who respectively boast a total of 8 and 14 programmable buttons across 2 controllers.

Additionally, MVR has no support for either Roomscale or Spatial Control Tracking without additional high-cost purchases, whereas the IVR systems has support for both

of these systems.

What this points to, in relation to the definition of immersion used within the context of this analysis, is that objectively, the level of sensory fidelity on MVR platforms is currently lower than that of IVR platforms. There is less controllers to work with, less buttons on the controllers, and no sensors to support Roomscale or Spatial Control Tracking.

The low sensory fidelity of mobile virtual reality means that there are less channels in which a user can interact with the virtual environment.

Despite the technical limitation of the MVR platform relative to IVR platforms such as the HTC Vive and Oculus Rift - presented in table 16 - the MVR platforms should not be ignored due to the fact that the mobile virtual reality market holds the majority share of virtual reality systems as presented in figure 38. The portability and less extensive hardware setup of MVR systems could serve a purposeful role within the virtual reality market.

Appendix B Unity Terminology

As a big part of our project have been developed using Unity, a lot of Unity-specific terminology appears throughout our documentation. Therefore, the following chapter will clarify some of the most essential concepts of Unity that have been used in our project in order to establish a common understanding of them.

B.1 Scenes

Scenes in Unity are the 3D space in which you create everything that will be in your game [63]. Essentially they represent levels. They are files in the Unity asset folder. You drag GameObjects into scenes.

B.2 GameObjects

A GameObject in Unity is a basic object used within scenes to represent anything imaginable. [64] Thus, GameObjects are used to represent objects such as trees, houses, furniture, characters, and so on.

GameObjects by themselves do not do anything. They are considered to be containers of scripts. This means that all custom behaviour that we have created during this project is programmed in scripts, which are then applied to GameObjects within scenes.

Figure 43 shows a game console controller from one of our scenes. To the left of the editor it is possible to view all GameObjects within the open scene. To the right, in the Inspector, is an overview of all scripts applied to this GameObject, in this case only a simply *Transform* script.

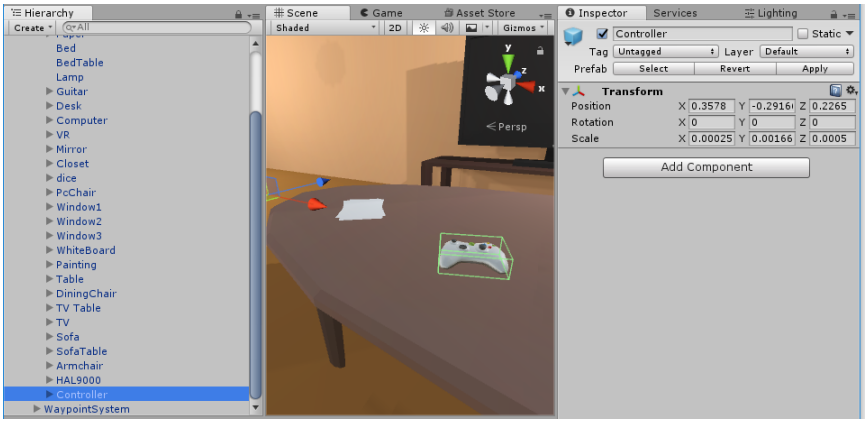


Figure 43: A game controller GameObject selected within one of our scenes.

B.3 MonoBehaviour

All scripts applied to GameObjects are essentially C# classes. These classes inherit from the base class *MonoBehaviour* [65]. Every script in Unity inherits from this class. MonoBehaviour is used by the Unity game engine to call predefined methods which developers can then fill out with custom logic. The predefined methods most typically used within our project has been: *Start*, *Awake*, and *Update*.

Figure 44 shows a screenshot of an empty script with these typical methods included.

```
using UnityEngine;

public class ExampleScript : MonoBehaviour {
    void Awake()
    {

    }

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

Figure 44: An example script. Here it is possible to see how the script is a class inheriting from MonoBehaviour.

B.4 Prefabs

Prefabs is a mechanism within Unity that makes it possible to easily re-use GameObjects. [66] Typically, you will end up configuring complex GameObjects used multiple times within one scene, or shared across many scenes. Were you to simply copy-paste GameObjects in order to achieve this, the problem would be that changes to one of these GameObjects would not be reflected across all of them. Thus, Prefabs can be equated to the concept of classes in object-oriented programming. You define one prefab representing a specific configuration of a GameObject, and this prefab can thus be instantiated many times. If you then change the Prefab, such as adding new scripts or changing the values of scripts in the inspector, these changes will be reflected across all GameObjects originating from this Prefab.

B.5 Inspector

The Inspector is a window of the Unity editor where it is possible to inspect the configuration of GameObjects within a scene [67]. Figure 45 shows a screenshot of the inspector after having selected a game controller GameObject within a scene. Here, it is possible to see what scripts that are applied to the GameObject - such as *RigidBody*, *Box Collider*, etc - as well as the values assigned to public attributes of those scripts.

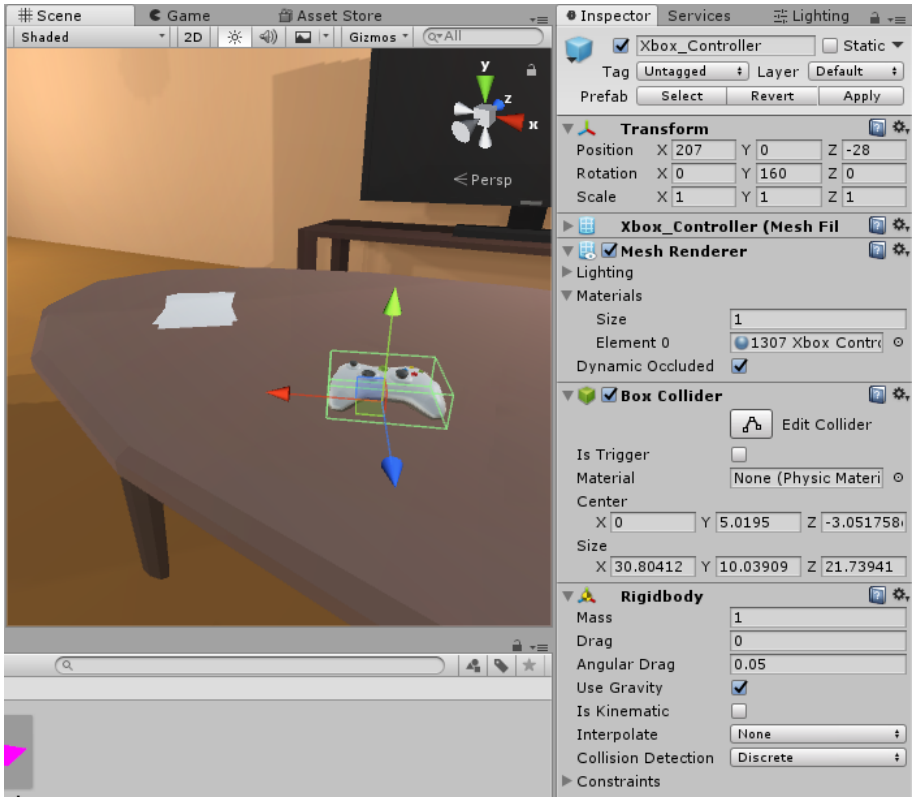


Figure 45: A screenshot of the Inspector, showing the configuration of scripts and script values from a selected GameObject from a scene.

B.6 Play Mode

The Play Mode is a feature within the Unity editor. It simply allows you to instantly play the scene you are currently working on [68]. This way, you avoid having to constantly do builds to executable files in order to try out your work.

B.7 Unity Packages

Unity packages - also sometimes referred to as Asset Packages - is a way to take Unity assets, such as sounds, 3D models, materials, scripts, and any other type of assets, and

package them into an easily redistributable format which can either be re-used internally between multiple projects or shared with others [69]. Through the Unity editor it is easy to import Unity packages.

B.8 Asset Store

The Asset Store is a built-in application of the Unity editor. It allows you to browse third-party packages which can add various assets to your projects [70]. This could be things such as scripts for achieving specific features, but could also be auditory or graphical assets such as sound effects, music or 3D models.

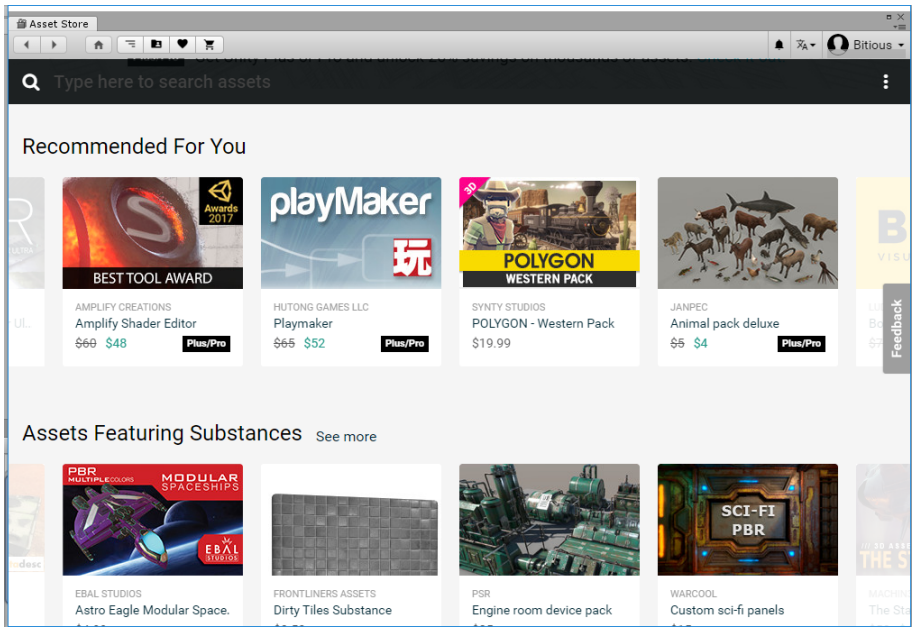


Figure 46: A screenshot of the Asset Store seen from the Unity editor.

Appendix C Virtual Reality Applications

This project consists of various virtual reality applications for IVR and MVR, ranging from the navigation experiments conducted during the project to the final experiments conducted at the end of the project.

All of these virtual reality applications were developed using Unity.

This section will describe how we used Unity to develop the applications, and will detail the technical solutions we implemented in order to create them.

C.1 Project Structure

All of the VR applications were developed within a single Unity project.

The reason for developing all of the VR applications within a single project was in order to ease the development workflow. All of the applications, whether IVR or MVR, ultimately had to share many of the same assets. For example, as it was important to create the same virtual environments on both platforms, all of the applications have to share the same 3D models for furniture and other environmental objects. Also, much of the codebase applies equally well across both platforms, thus much of the code would also have to be shared.

Because of these reasons, we decided that it was advantageous to contain all applications within the same project, in order to avoid having several projects duplicating a lot of assets. This could quickly turn into a maintenance nightmare when having to keep all shared assets in sync throughout all projects. Having everyting contained in a single project, sharing all assets, was seen as the optimal route.

One disadvantage of one single project, however, is that it quickly turns big. Thus it was important for us to keep a neat project structure, where it was easy to navigate and find what you were looking for. It was also important to handle cross-platform development in a simple manner, so as to not complicate the project structure unnecessarily.

We did this primarily by: Keeping a folder structure throughout the entire project that was as easy to navigate as possibly, and by doing cross-platform development by splitting up each experiment into two scenes. One scene for each platform.

This section will describe this folder structure and scene structure.

C.1.1 Folder Structure

The way that Unity deals with assets - assets being anything from 3D models, audio, third party plugins, textures, scripts and so forth - is by letting the user contain it all within a predefined folder by the name of *Assets*. Within this folder, it is up to the user to decide how to structure all created and imported assets.

Figure 47 shows our structure from the root of the asset folder.

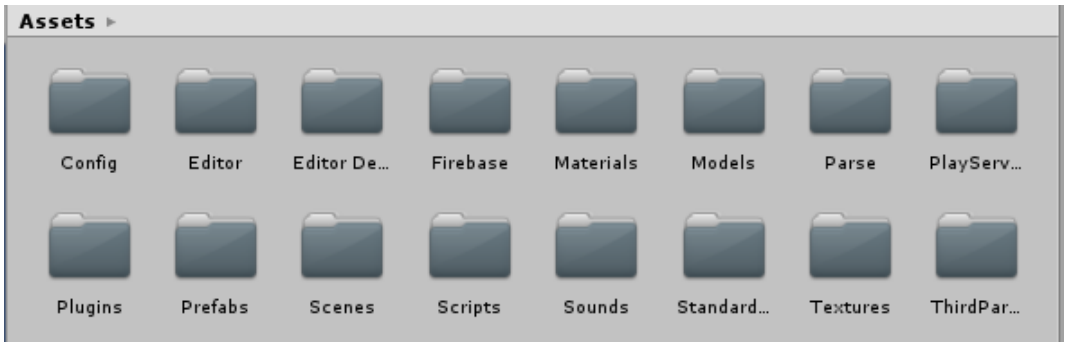


Figure 47: A screenshot of our project structure from the root of the Asset folder.

As can be seen in 47, we decided to split up the various assets into telling categories. This means that we have subfolders for assets such as *Textures*, *Scripts*, *Models* and so forth. This way, it was easy for us to navigate an otherwise big Unity project.

All of these subfolders are then further divided into telling categories, if it makes sense to do so. As an example, figure 48 shows the subfolder of our scripts.

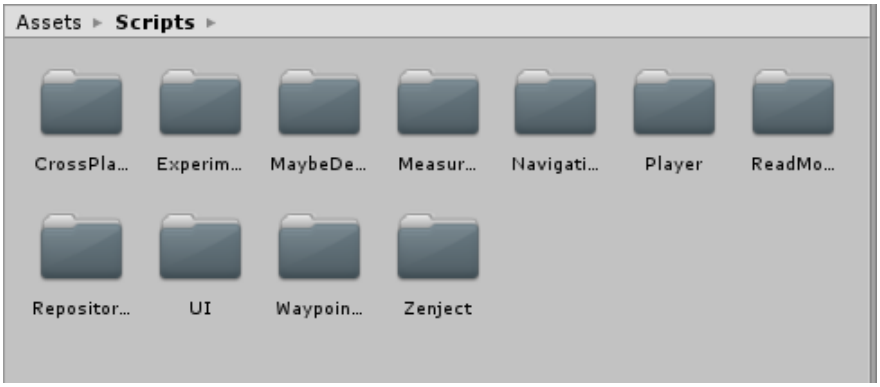


Figure 48: A screenshot of our project structure from the root of the Asset folder.

As can be seen in figure 48, scripts have then been categorized after functionality or domain.

C.2 Third Parties

C.2.1 Zenject

Zenject[33] is a dependency injection framework that we use for several aspects of our virtual reality applications.

Zenject is specifically designed and made for Unity.

We use Zenject primarily in order to ease the making of cross-platform friendly scripts, in order to make our software easier to extend and develop.

In order to make use of Zenject, we simply imported it as a third-party package from the Unity asset store. Figure 49 demonstrates Zenject as seen from the asset store within the Unity editor.

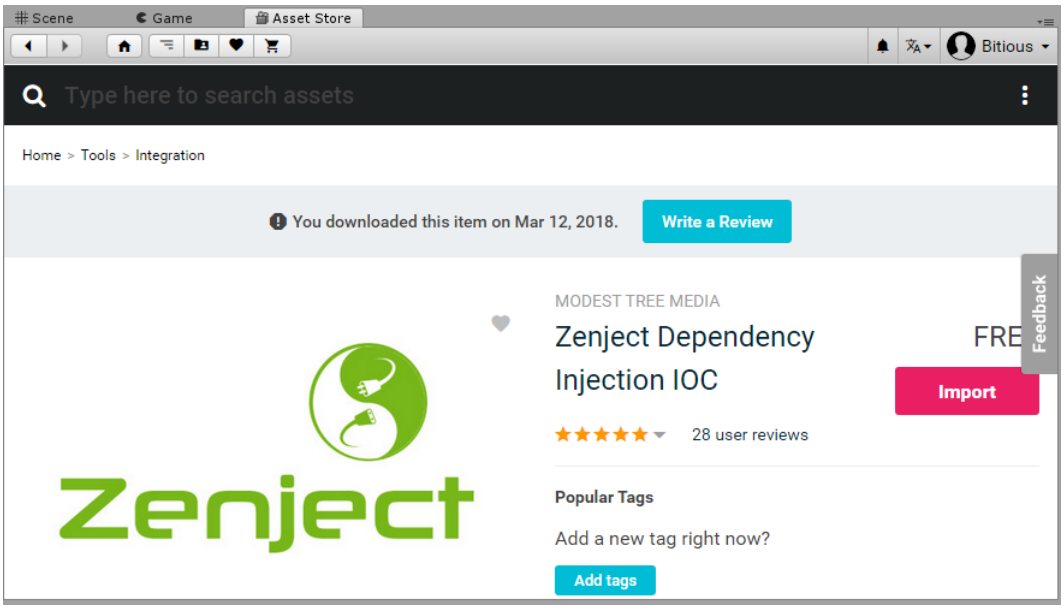


Figure 49: A screenshot of Zenject as seen from the asset store.

In order to get Zenject to work, we made use of two basic Zenject components:

- A *scene context* GameObject within any Unity scene that has scripts depending on Zenject. This GameObject is provided by the Zenject framework.
- A *MonoInstaller* script. This script configures the Inversion Of Control (IoC) container. It is here you decide which concrete implementations should be mapped to which interface type. You write this configuration yourself.

Thus, we implemented our own MonoInstaller script which configures the IoC container for our own needs.

Our MonoInstaller script goes by the name of *MainInstaller*, and figure 50 shows a basic class diagram of it.

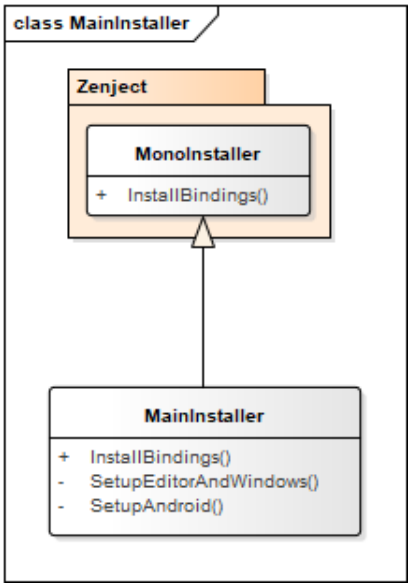


Figure 50: Our own implemenation *MainInstaller* of the *MonoInstaller* component from Zenject. MainInstaller configures our IoC container.

Figure 51 shows a flow diagram of the configuration process.

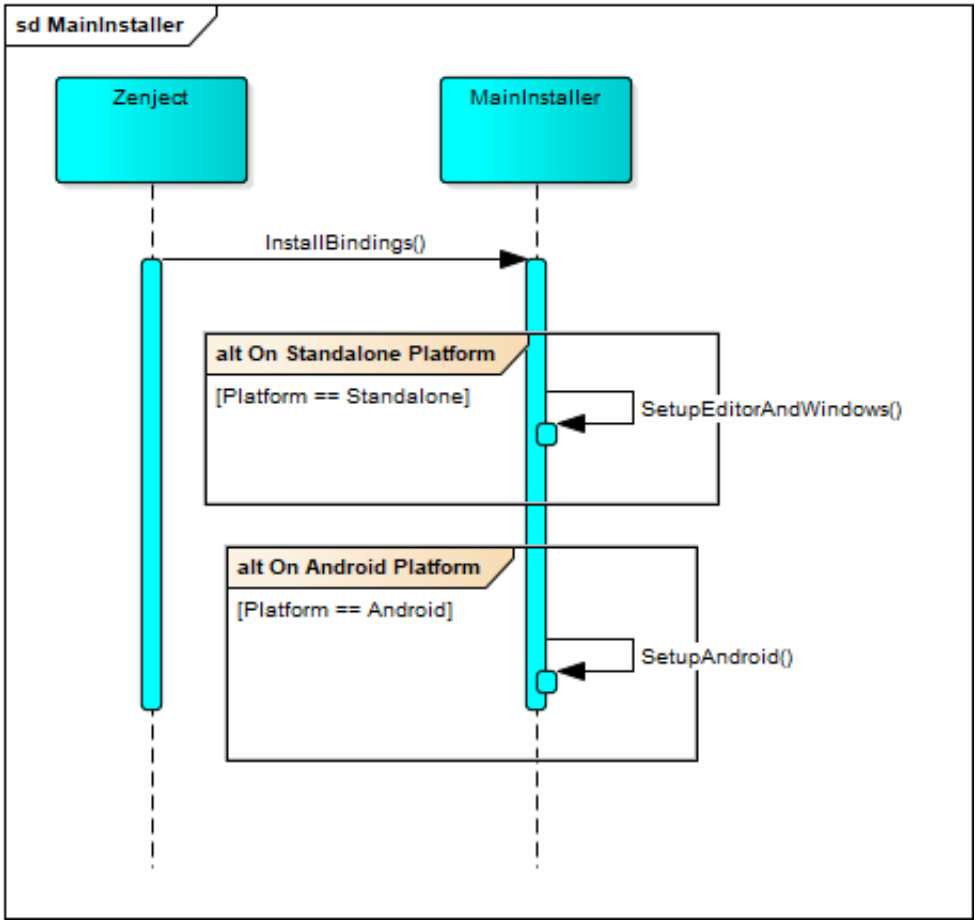


Figure 51: The sequence logic of *MainInstaller*.

Figure 51 thus shows how we use Zenject in order to control platform-specific implementations across platforms. In this diagram, the platform *Standalone* refers to IVR. Thus, if an application is run on a desktop computer, the IoC will be set up with implementations which works best for a desktop environment. Likewise, if the application is execute on an Android smartphone, the IoC will be set up with implementations which works best for a mobile environment.

C.2.2 Google VR SDK

In order to develop virtual reality applications for Google Daydream, we made use of the official Google VR SDK for Unity with Android [71].

The Google VR SDK for Unity is a Unity Package which is simply imported to a project through the editor.

The package contains various scripts, prefabs and example scenes relevant to virtual

reality with Google Daydream, which one can make use of and learn from. Furthermore, there is also online documentation available as a learning resource [72].

Prefabs and scripts used from the Google VR SDK will be mentioned in relevant sections throughout the documentation.

Instant Preview

The Google VR SDK comes bundled with prefabs that supports *Instant Preview*. Instant Preview allows you to use the Play Mode within the Unity Editor to instantly stream the game to a smartphone connected to the computer through a USB. Throughout development of the MVR application we made heavy use of this feature. Figure 52 shows a picture demonstrating Instant Preview.

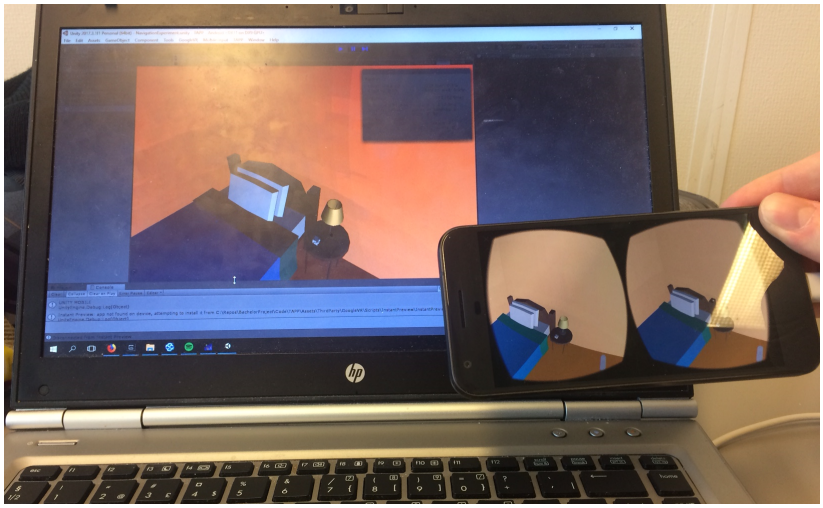


Figure 52: Instant Preview streaming the rendering of our virtual reality application from the Play Mode in the editor to the display of an Android smartphone. The smartphone is connected to the computer through USB.

In order to make use of Instant Preview, we simply needed to include the following prefabs from the Google VR SDK within the scene in question:

- *GvrEditorEmulator*
- *GvrInstantPreviewMain*

Instant Preview helped us tremendously throughout development, as building the APK to the actual device can be time consuming. Being able to instantly test features or bug fixes was really helpful.

C.2.3 SteamVR Plugin

In order to develop virtual reality applications for the HTC Vive, we made use of the *SteamVR Plugin* Unity Package [73], which is available from the Unity Asset Store.

Figure 53: The SteamVR Plugin package as seen from the Unity asset store.

SteamVR Plugin for Unity is a Unity package which wraps Valve's OpenVR SDK [74] so that it is useable from within Unity. It is not designed specifically for HTC Vive. Rather, it can be used to develop virtual reality applications with various different IVR hardware, such as HTC Vive or the Oculus Rift. The package abstracts the hardware, hiding it behind a common interface which can be used from within Unity scripts.

The package includes a host of scripts and prefabs which can be used for virtual reality applications. It also comes bundled with various demo scenes which you can use to learn the different aspects of the package, and how to use it with IVR hardware.

Prefabs and scrips used from the SteamVR Plugin package will be mentioned in relevant sections throughout the documentation.

Instant Preview

The SteamVR Plugin supports *Instant Preview*. This allows the game to be streamed to the VR headset straight from Play Mode in the Unity editor.

Just as for Instant Preview of the Google Daydream SDK, we relied heavily on the feature for IVR development. It was very quick and easy to test out new features and changes instantly.

There was nothing particular that had to be done in order to make use of Instant Preview with the SteamVR Plugin. As soon as the scene was configured with a SteamVR camera, Instant Preview could be used provided a headset is connected to the development computer.

Experiences with SteamVR Plugin

One peculiar thing we found with the SteamVR Plugin was the lack of documentation. The package itself comes included with a bare-bones *Quickstart* guide. This guide only contains short texts about simple concepts and a few of the prefabs included in the package. There is no API documentation, and many of the scripts and prefabs included within the package is completely undocumented.

Thus we spend quite a bit of time simply figuring out how to use the package properly to implement the features we needed, such as getting teleportation to work, or reading input from the controllers.

C.2.4 ProBuilder

ProBuilder is a Unity editor extension which enables support for doing 3D modelling and texturing from within the editor.

ProBuilder is a Unity package which can be downloaded and imported from the Unity Asset Store.

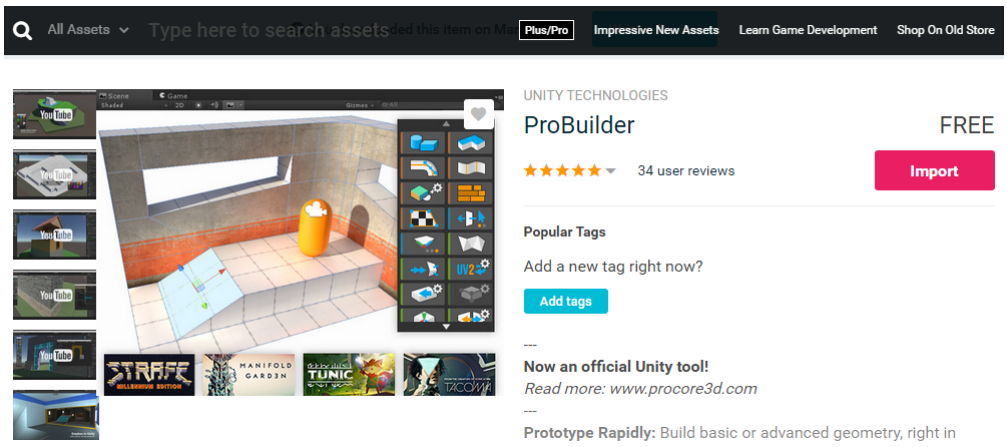


Figure 54: The ProBuilder editor extension as seen from the Unity Asset Store.

During development of the project we had the need to create basic virtual environments which experiments could be framed within. An example of this is the room used for the navigation experiment.

We could have also used external modelling software, such as *Blender* [75]. However, we shared no modelling experience within the team. Furthermore, the virtual environments which had to be created during the project were very simple. Therefore there was no need for any complicated external tools, when everything that we needed could be done within the environment which we already used in a greatly simplified fashion. The

ProBuilder editor extension was thus a great match for us, as it was simple to use, and we could integrate the modelling into our already existing Unity workflow.

When ProBuilder is installed in the Unity project, its features becomes available through the addition of a new toolbar option. This can be seen in the screenshot of Figure 55 which is taken from within our own project.

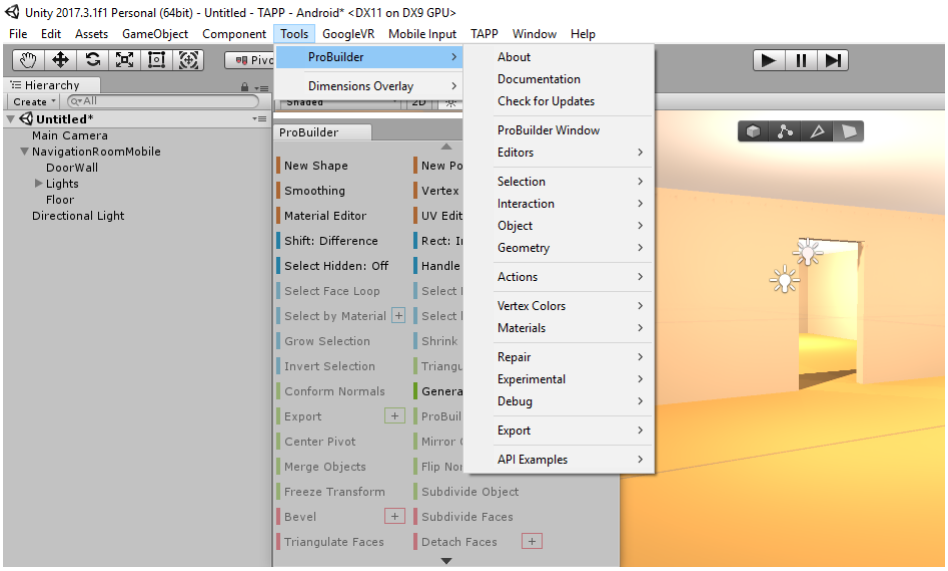


Figure 55: The new ProBuilder menu option after installing the ProBuilder package in our project.

C.2.5 3D Models

In the VR applications we needed 3D models for furniture, but creating them ourselves from scratch would take a lot of time, and since 3D modeling is not the focus of this project, we decided to make use of free models provided on the platform Poly, which provides user-created low-poly 3D models under a CC-BY license. As such all of our furniture and furnishings are models created by the user *Poly by Google* [1] under the CC-BY 2.0 license [2].

Poly is a platform that is accessed on the web, and figure 56 shows a screenshot of it. As can be seen, you can easily search 3D models by keywords.

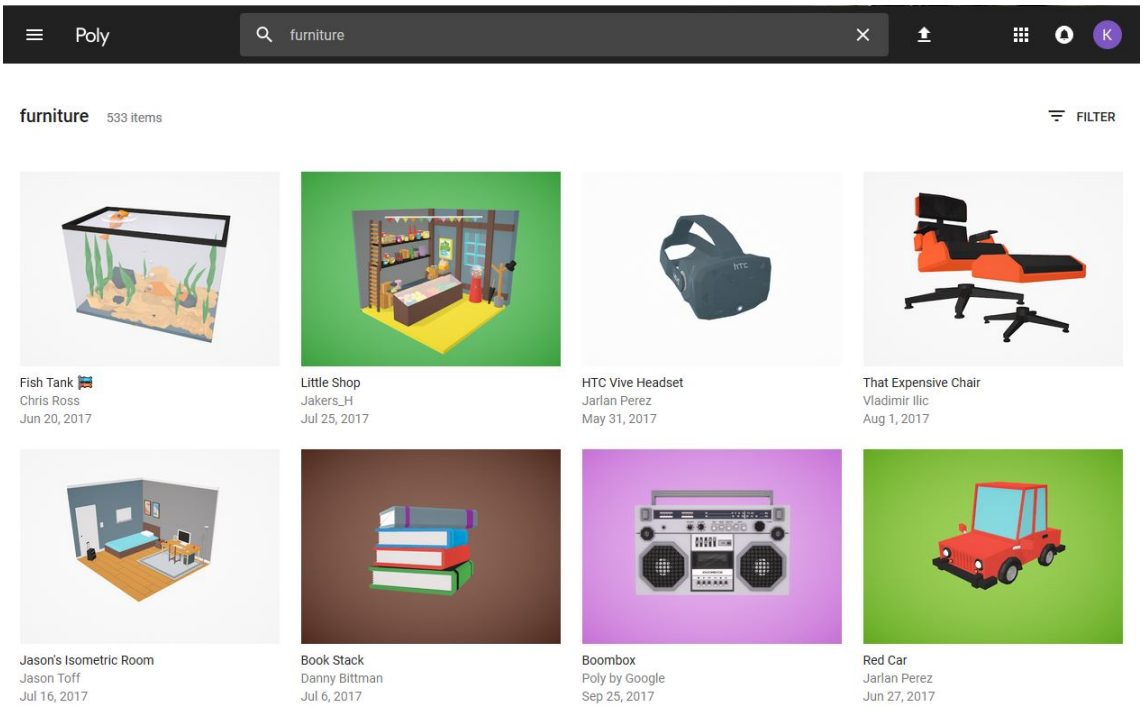


Figure 56: A screenshot of the web interface of Poly. A search for furniture has been completed, and the selection is shown.

C.3 Unit Testing

Unit Testing has been done of scripts within our Unity project where it makes sense to do so.

In order to unit test we made use of Unity's in-built unit testing tool from within the editor.

Figure 57 shows a screenshot of the in-built unit test tool in our project.

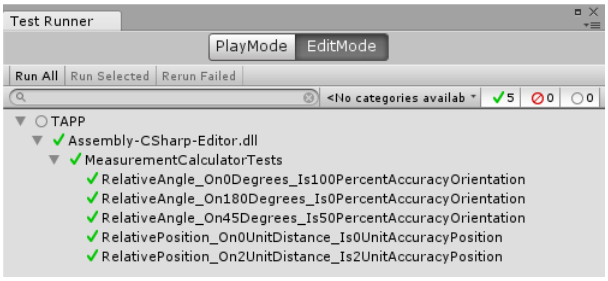


Figure 57: Unity's in-built test runner seen within the context of our project. Tests discovered within our project is shown within the window.

In order to implement unit tests with Unity, you have to place the scripts within a specific folder relative to the root of the *Asset* folder. The folder has to be named *Editor*, and should be placed in the root of the *Asset* folder [76]. This can be seen in the screen of figure 58.

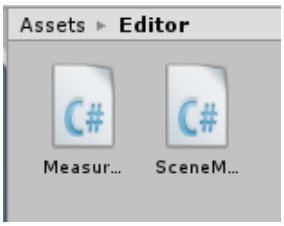


Figure 58: The *Editor* folder as seen from within our Unity project. Our unit test scripts is placed within this folder.

When unit test scripts are placed within this folder, Unity's test runner will automatically detect them and display them in the test runner window.

Unity has integrated the NUnit [77] test framework. Thus, unit tests are written using NUnit.

The unit tests of our Unity project is also used in associated with our continuous integration pipeline.

C.4 Continuous Integration

We have used continuous integration for our Unity project.

The primary purpose of continuous integration for us has been for monitoring the health of the VR applications during development. If a team member accidentally pushes changes that breaks the build across platforms, the continuous integration pipeline will detect it and notify us.

In order to make use of continuous integration, we used Unity's own cloud service, called *Cloud Build* [78].

The advantage of Cloud Build is that as it's Unity's own service, it is already well-integrated with Unity and thus easy to set up and use. It provides direct integration with the unity Editor, and also features a web-interface for detailed settings and histories.

On figure 59 is a screenshot of Cloud Build from within the editor, in the context of our project.

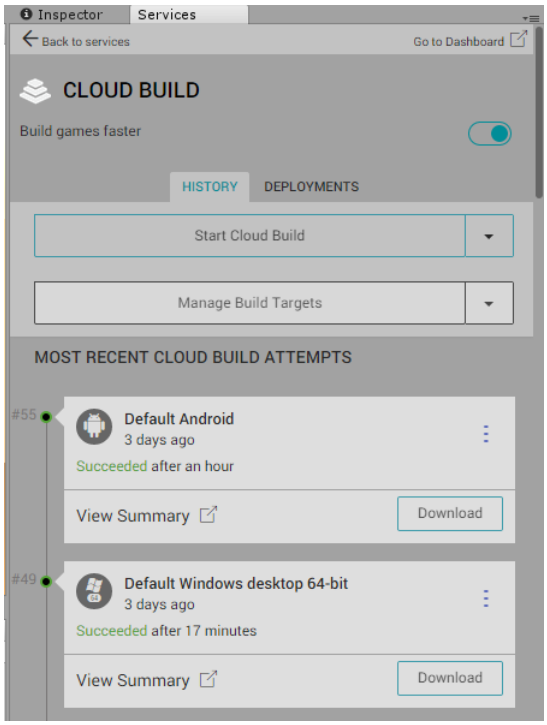


Figure 59: The Cloud Build service as seen from our project within the Unity editor.

Cloud Build from within Unity's editor gives a quick overview of the most recent builds performed, and whether they succeeded in building or not. In figure 59 it is possible to see that the most recent builds at the time was the Android platform target, and Windows Desktop platform target. It is also possible to see that they both succeeded in building.

On figure 60 is a screenshot of the detailed web-interface typically used in conjunction with the Unity editor.

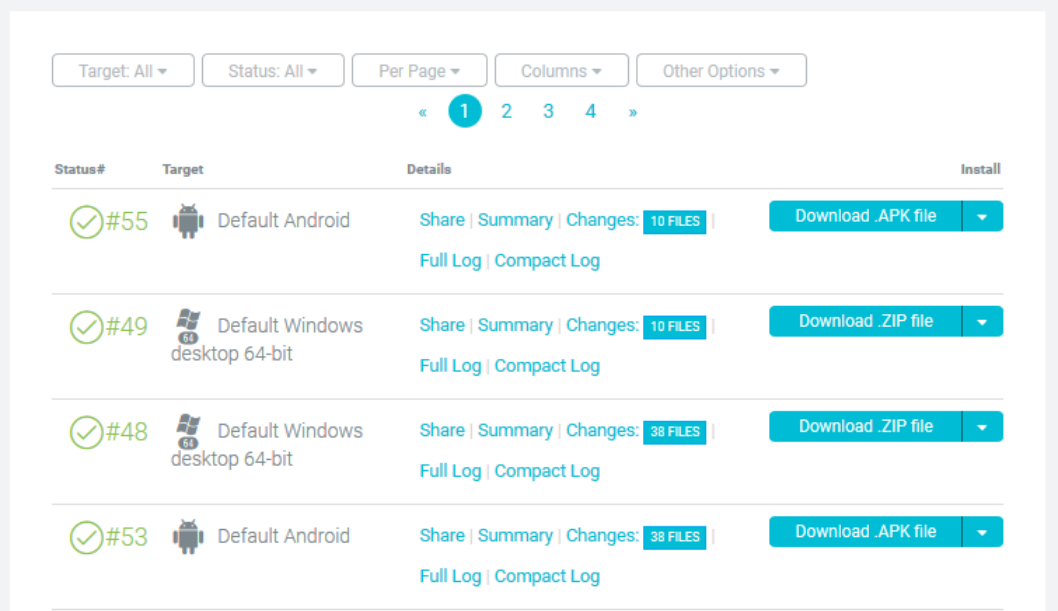


Figure 60: The Cloud Build service as seen from our project through the web-interface.

As can be seen in figure 60, the Cloud Build web-interface has a lot of features. Besides having a detailed build history of all builds ever completed, Cloud Build also generates change logs and full logs of the entire build process. Additionally, the generated executable file (whether it be an APK file for Android of EXE file for Windows) is also uploaded so as to be easy to download and share across devices.

Every time a commit is made on our GitHub repository, Cloud Build detects the changes of our Unity project and attempts to build the project and run any detected unit tests. Whether builds and tests succeed or fail, an email notification is sent with the results. An example of such an email notification can be seen in figure 61.

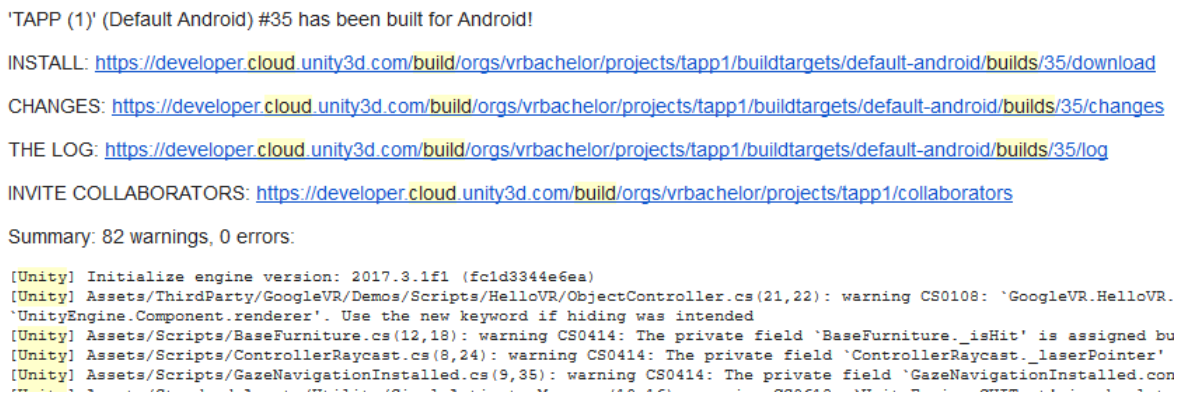


Figure 61: Screenshot of an email notification sent automatically from Cloud Build after a succeeded Android build.

C.5 Modelling of Virtual Environments

We used ProBuilder in order to model the virtual environments from within our experiments were framed. This includes the two-roomed environment used in the navigation experiment, as well as the object manipulation experiment.

During the project, we modelled two virtual environments. They are:

- The room used in the navigation experiment.
- The room used in the object manipulation experiment.

Figure 62 shows a screenshot of the virtual environment for the navigation experiment.

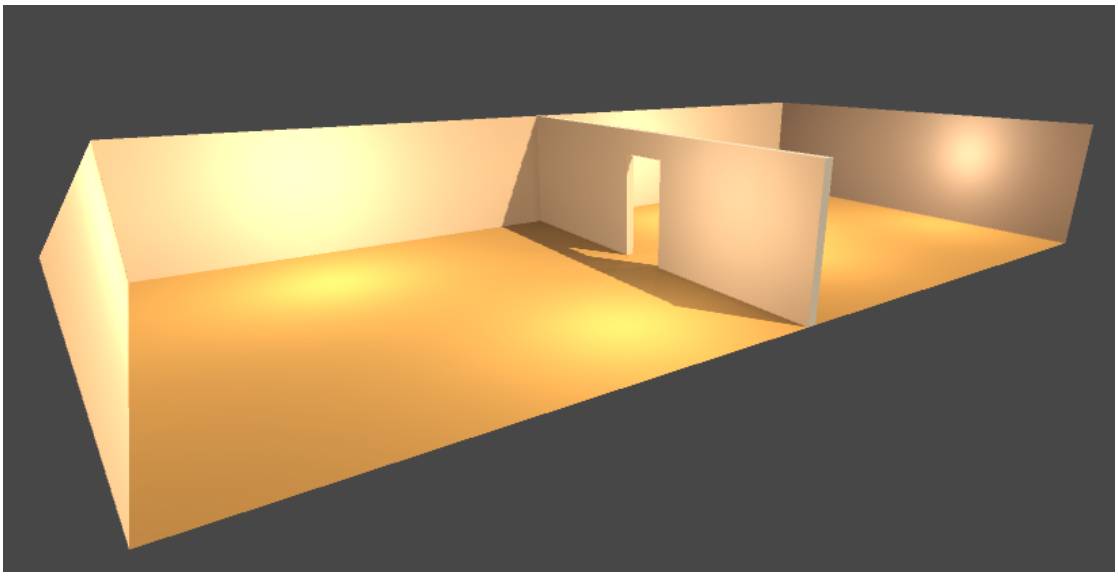


Figure 62: The virtual environments that we made using ProBuilder during the project.

Figure 63 is a screenshot from our project in which the ProBuilder window can be seen.

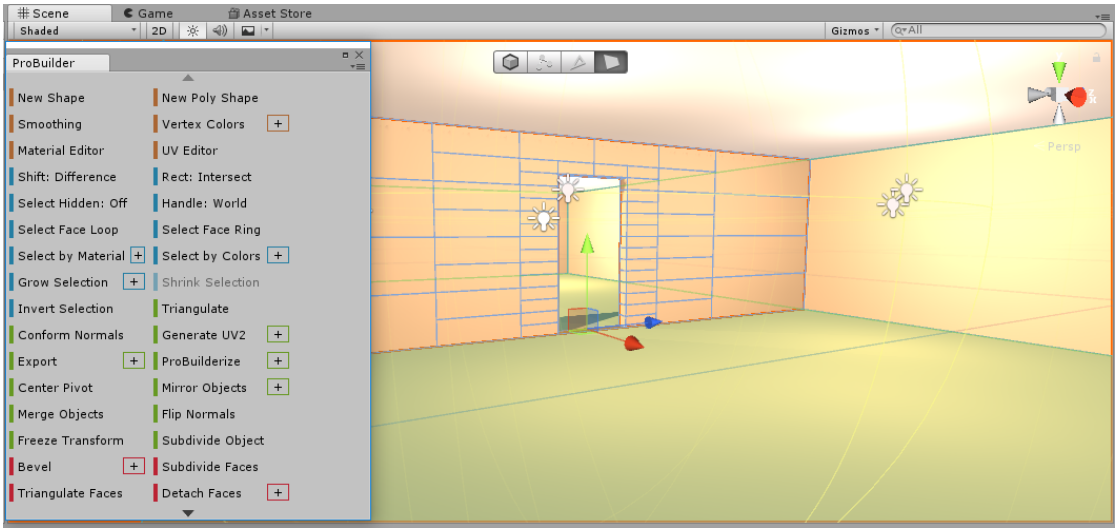


Figure 63: The ProBuilder window open in a scene of ours, with the room we modelled for the navigation experiment. Here, face selection is used to manipulate individual faces of the model (in this case the highlighted wall).

ProBuilder supports edit modes which you find in conventional 3D modelling software, such as: Object Selection, Vertex Selection, Edge Selection and Face Selection. For example, we made use of Face Selection in order to edit the wall in figure 63 so as to

make a door in the middle.

Another feature of ProBuilder that helped us greatly, as we were inexperienced in 3D modelling in general, was the easy-to-use texturing feature.

Usually it can be quite involved to texture 3D models, requiring knowledge of how to perform UV mapping [79]. ProBuilder made this easy for us, so that we could quickly create environments with colors. Texturing in ProBuilder is done through the *Material Editor*. Figure 64 shows how, when a wall is selected in our virtual environment, we are able to easily apply a specific material to that section of the model only.

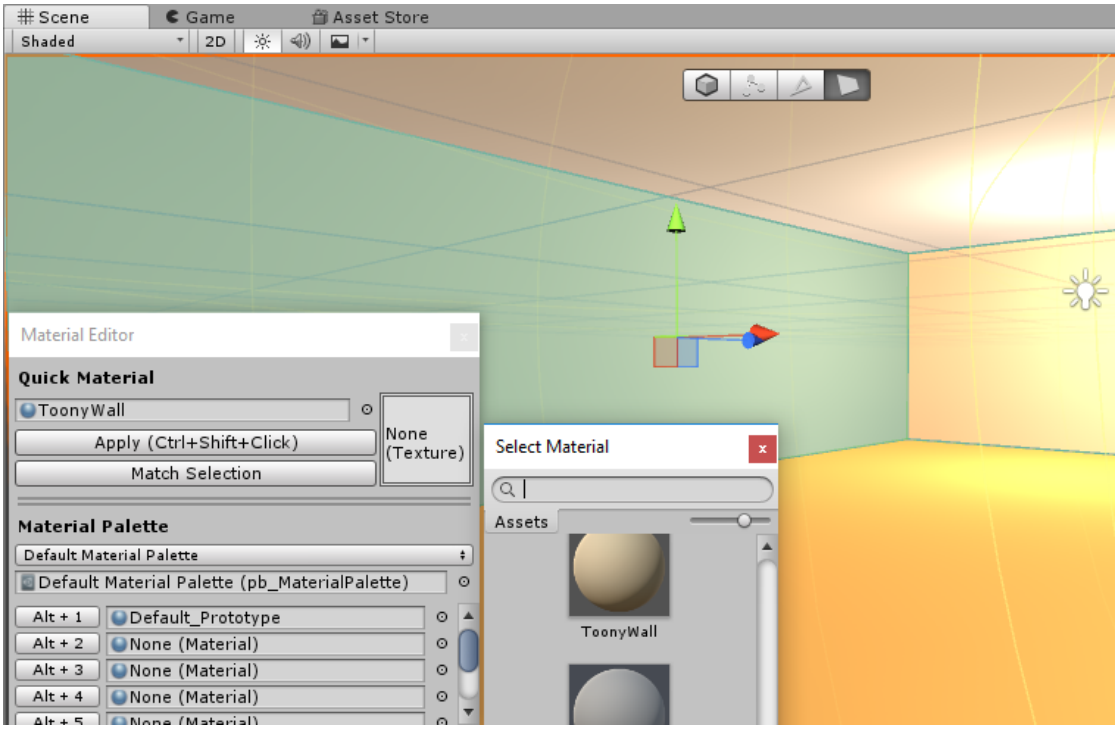


Figure 64: The Material Editor of ProBuilder. Here, we have selected a wall on the room of the virtual environment used for our navigation experiment. From here, we can easily apply new materials.

Using this feature meant that we did not need to have knowledge of UV mapping.

C.5.1 Reuse

It was important that the virtual environments we modelled could be re-used across multiple scenes.

Appendix D The Navigation Experiment

As part of our development process, we wanted to eliminate uncertainties that may arise when people are newcomers in virtual reality. Specifically we wanted to make navigation in a VR environment as good as possible, as navigation in VR often have the unfortunate side effect of inducing nausea or motion sickness in users.

From looking at current implementations navigation in VR applications, we have seen that the most common methods of navigation is teleportation and gaze-based walking. In the article *Interaction Design for Mobile Virtual Reality* by Daniel Brenners [40], it was discovered that users preferred gaze-based walking on the mobile platforms, but after trying that method of navigation ourselves and feeling extreme nausea, we decided to make our own user-test to see which navigation mode the users prefer in order to optimize the final VR experiment.

This section describes the implementation and execution of our navigation experiment.

D.1 Experiment Task Setup

In order to test the navigation modes we created an experiment in which the user have to navigate through multiple "waypoints" in a Virtual Reality Environment.

Figure 65: Screenshot from within the navigation experiment.

Using a menu (see figure 66), the user can choose the navigational mode of the experiment; either teleportation og gaze-based walking. After making this choice, the

user must navigate through the waypoint system, and when done with one navigation method, they should do the same for the next navigation mode.

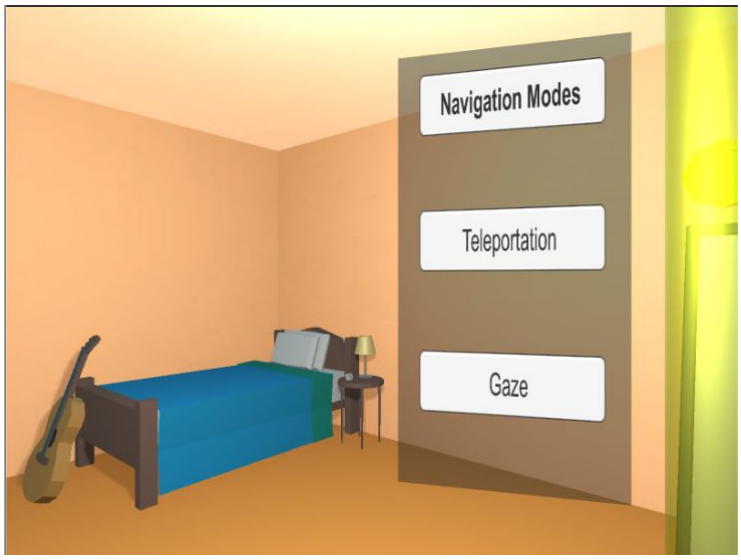


Figure 66: Screenshot of the menu visible in the navigation experiment room.

After a user finishes navigating the room, a measurement indicating the navigation mode and completion time is saved to firebase. The class diagram of the saved data can be seen in figure 67.

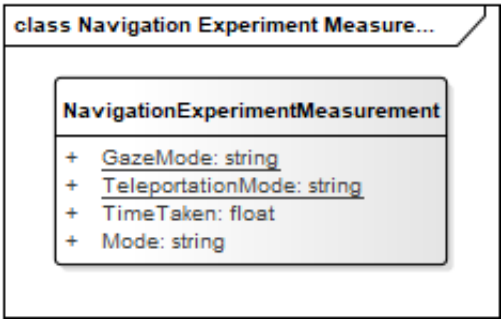


Figure 67: Classdiagram for the navigation experiment measurement

After completing the experiment, the user should answer a questionaire (See appendix I.1). The questions of the questionaire and the time measurement of the different navigation modes, should be enough to let us make a decision on which mode to use in the final implementation of the object-manipulation experiment.

D.2 Conducting the Experiment

We gathered people who wanted to participate in the experiment, and split them into two groups; one for each VR platform. Each group had to complete the navigation experiment for each of the two navigation modes. Half of the users on each platforms started with Gaze-based navigation, while the other half started with teleportation, to try and remove any bias stemming from trying a navigation mode first. The split is illustrated in figure 68.

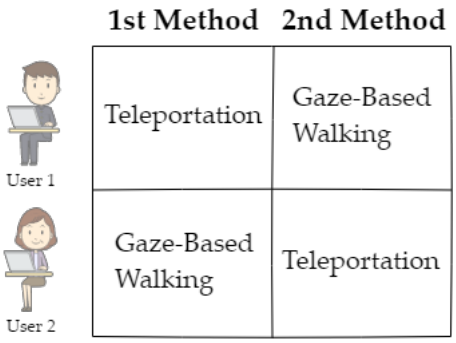


Figure 68: The split of the users trying teleportation and gaze-based walking

Figure 69: Photo of one of the test subjects trying the navigation experiment on the mobile platform.

The users were also told to try and complete the experiment so that we could gather some

data regarding the task completion time, but that it was okay to stop the experiment if they were feeling nauseous.

D.3 Results

We ran 20 people through the navigation experiment; 10 for each platform, and even though we know that this amount of people is not enough to be statistically significant, it was what we were able to gather within a reasonable time-frame, and we will use the data with all its imperfection, as it will still be an indication of user preferences for each platform.

The raw data can be found in the appendices I.2 and I.3

D.3.1 MVR Navigation

For the navigation experiment for the mobile platform, the users were distributed in the age range from 22 to 42 as seen in figure 70

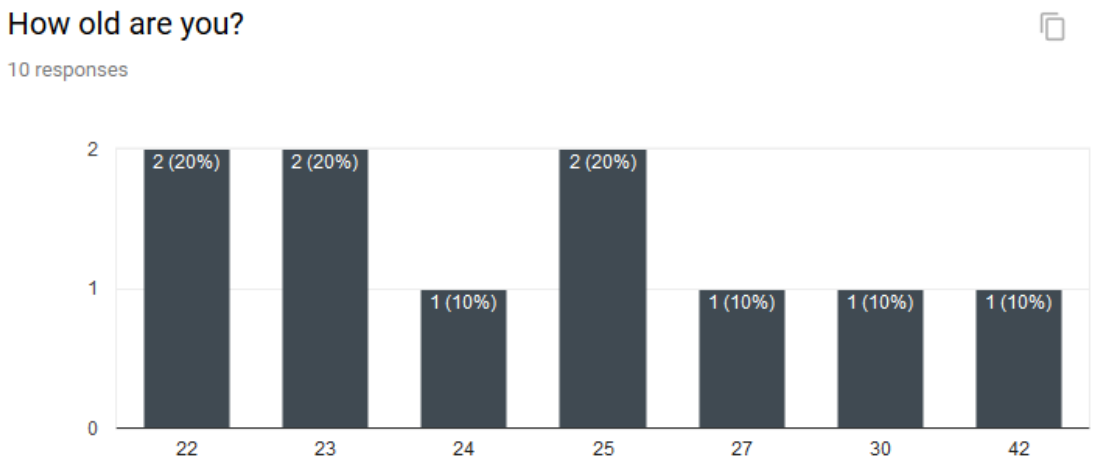


Figure 70: Distribution of age between MVR test subjects.

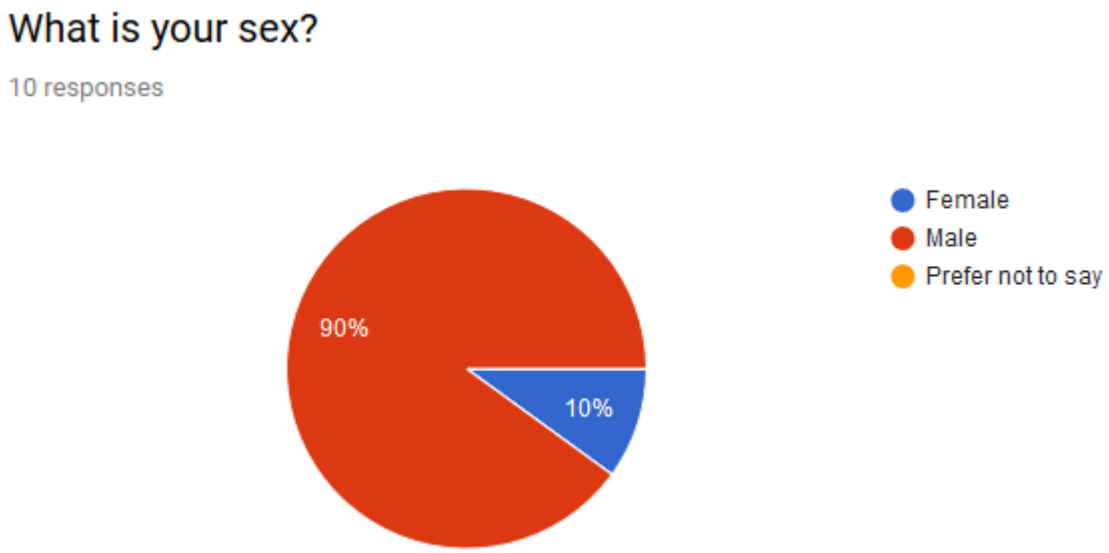


Figure 71: Distribution of gender between MVR test subjects.

Of the test subjects for this experiment, all of them were students, and their educational distribution can be seen on figure 72

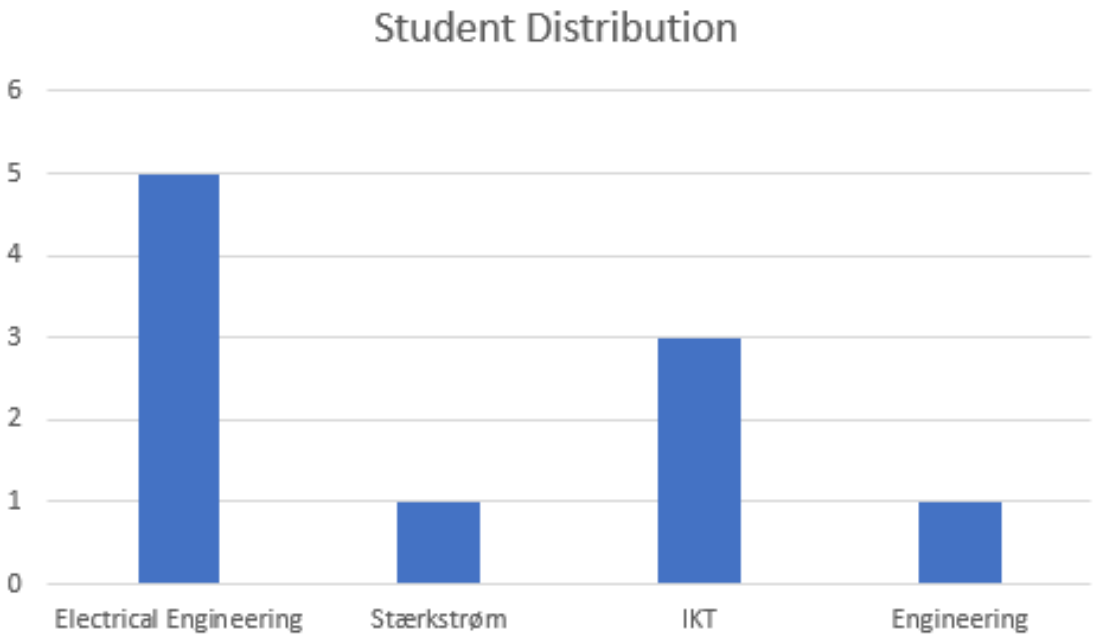


Figure 72: Distribution of education between MVR test subjects.

The test subjects most identified themselves as experienced with interacting with 3D environments, and 80% of the subjects had tried VR previously. Of those 80% most identified as "not that experienced" in VR.

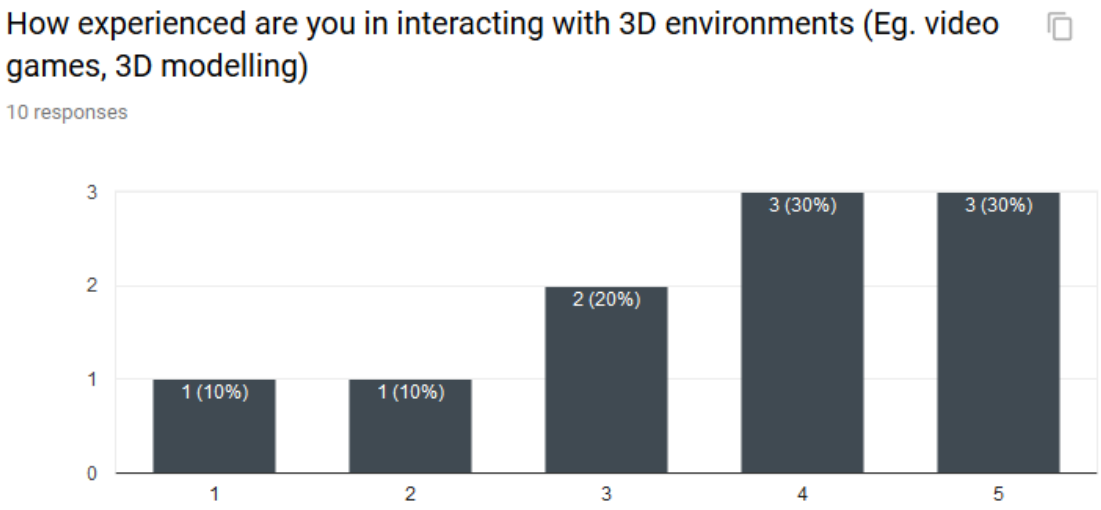


Figure 73: MVR ranking of experience with 3D environments with 1 meaning not that experienced, and 5 meaning very experienced.

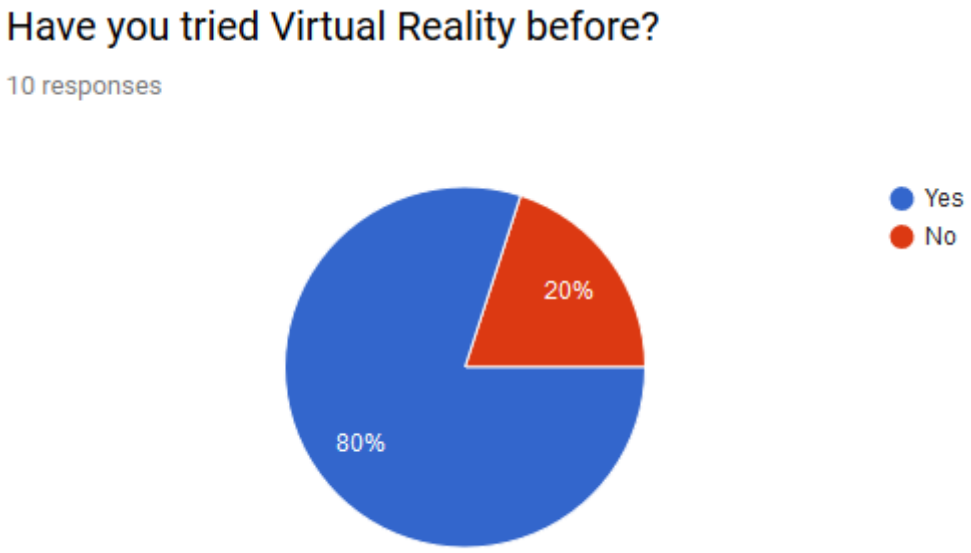


Figure 74: MVR distribution of previous experience with VR.

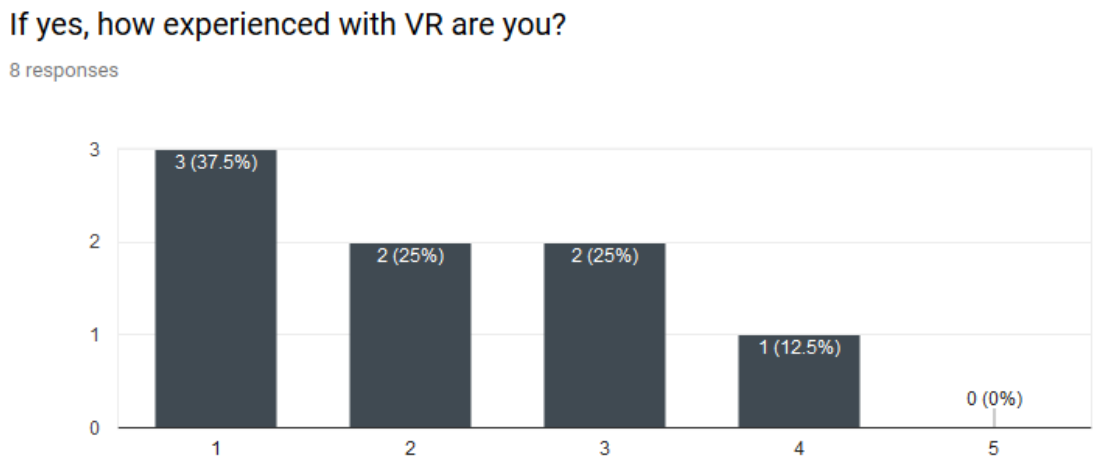


Figure 75: Ranking of experience with Virtual Reality with 1 meaning not that experienced, and 5 meaning very experienced.

When the test subjects tried teleportation, most users strongly agreed with the sentence "It was easy to orient myself and navigate the environment with teleportation" while most of the user disagreed with the statement "Teleporting made me feel nauseous or queasy"

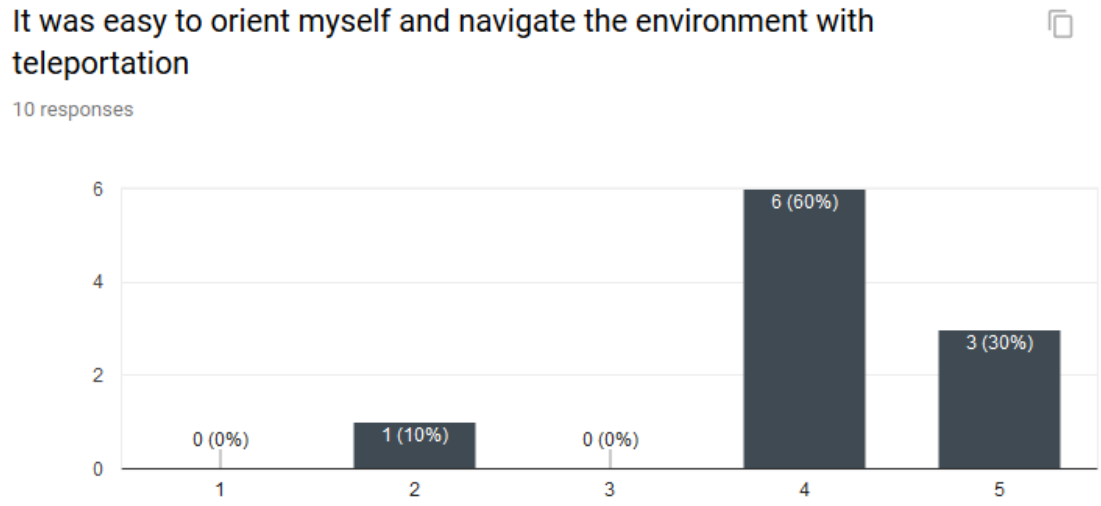


Figure 76: Statistics of agreeing or disagreeing with the statement "It was easy to orient myself and navigate the environment with teleportation" with 1 meaning strongly disagree, and 5 meaning strongly agree.

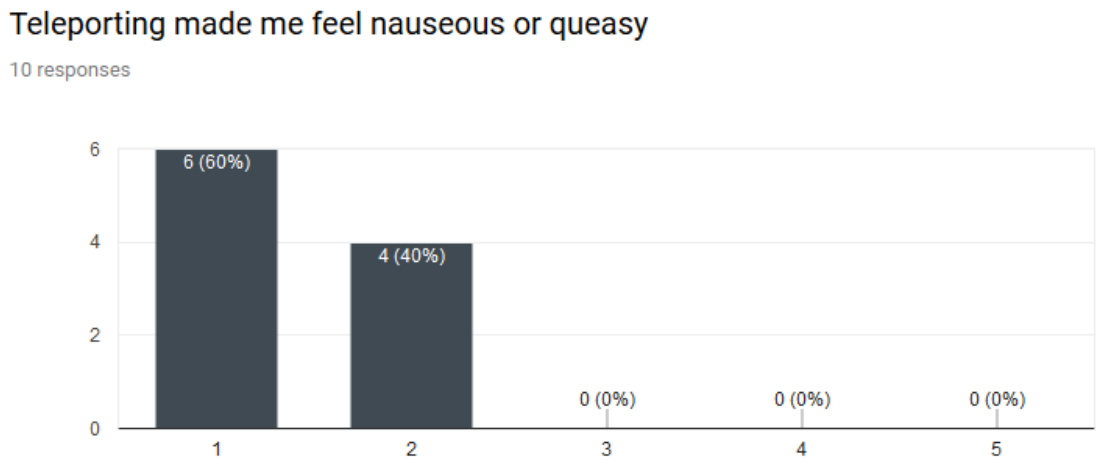


Figure 77: Statistics of agreeing or disagreeing with the statement "Teleporting made me feel nauseous or queasy" with 1 meaning strongly disagree, and 5 meaning strongly agree.

When asked about how they felt about Gaze-based walking, the users agreed with the statement “It was easy to orient myself and navigate the environment with gaze-based walking”, yet compared to teleportation, more people agreed with the statement “Gaze-based walking made me feel nauseous or queasy”.

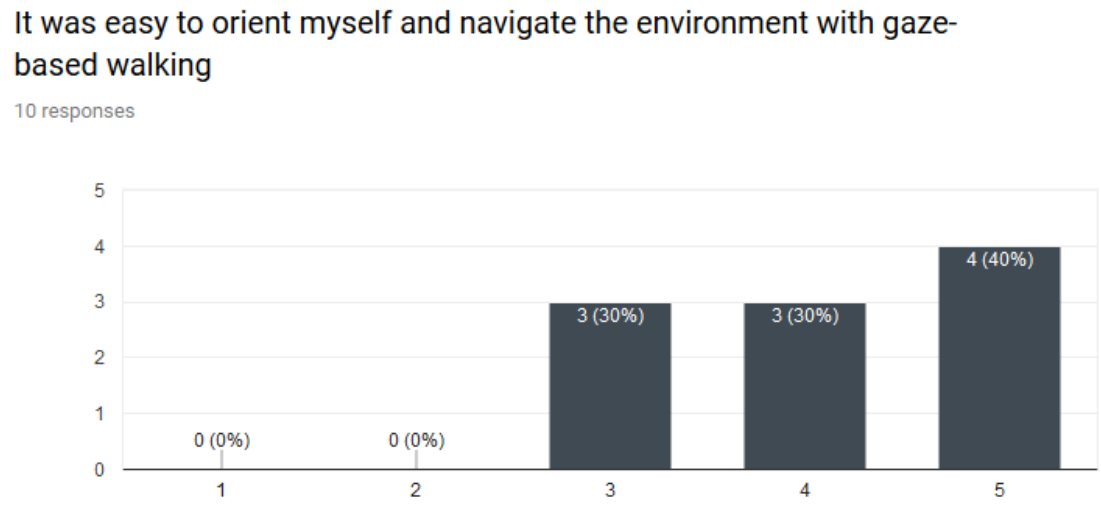


Figure 78: Statistics of agreeing or disagreeing with the statement "It was easy to orient myself and navigate the environment with Gaze-based walking" with 1 meaning strongly disagree, and 5 meaning strongly agree.

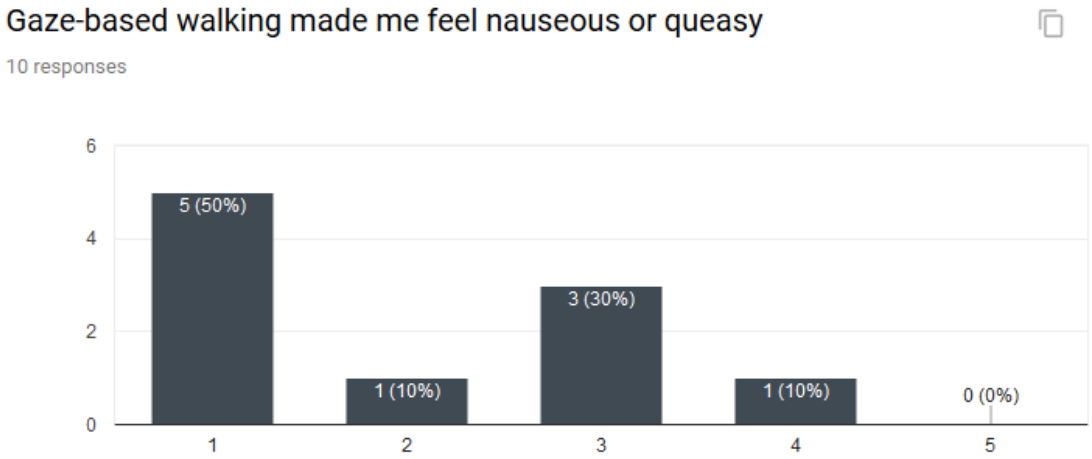


Figure 79: Statistics of agreeing or disagreeing with the statement "Gaze-based walking made me feel nauseous or queasy" with 1 meaning strongly disagree, and 5 meaning strongly agree.

As an objective measurement of the user's performance in the VR environment we have measured their task completion time (the time it took the users to go through the waypoints). The average speed using Teleportation was 81.7 seconds, while the average speed for Gaze-based walking was 110.2 seconds (See appendix I.4).

Finally, when we asked the users of their personal navigation mode preference, 50% of the users preferred teleportation, with 40% preferring Gaze-based walking, and the last 10% with no preference.

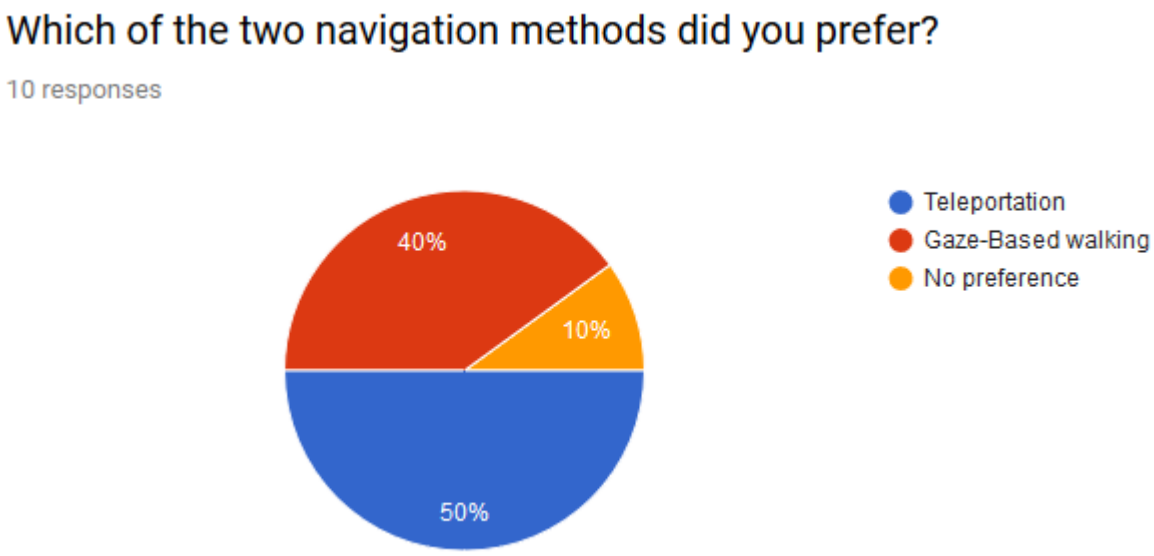


Figure 80: Navigation mode preference by users.

Users that preferred Gaze-based walking commented, that Gaze-based walking was more fluent, and made it easier to get around in the VR space because it felt like actual walking.

Users that preferred teleportation, commented that it was much faster than gaze-based walking, and that they did not like the feeling that they got from gaze-based walking.

D.3.2 IVR Navigation

For the navigation experiment for installed VR, the users were distributed in the age range from 22 to 29 as seen in figure 81

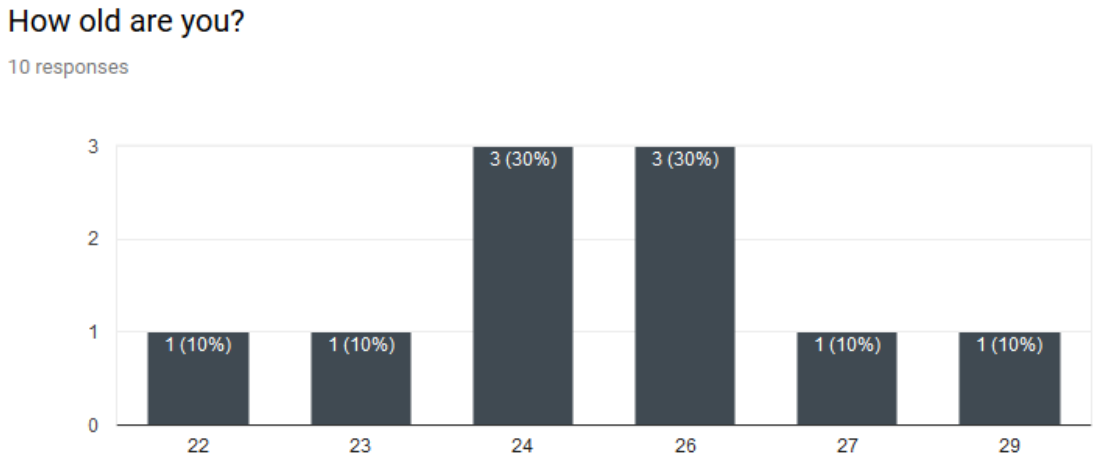


Figure 81: Distribution of age between IVR test subjects.

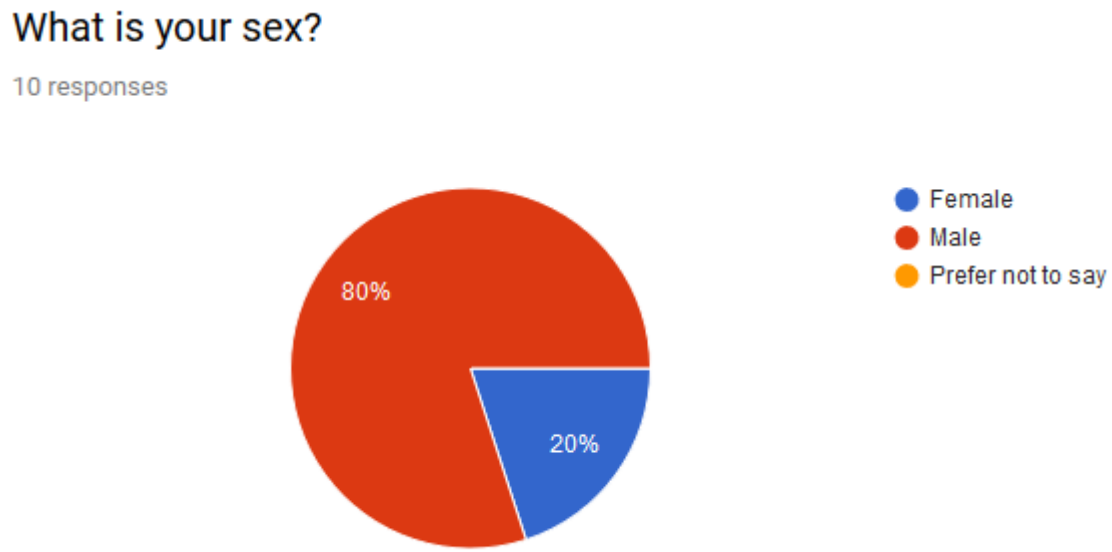


Figure 82: Distribution of gender between IVR test subjects.

Of the test subjects for this experiment, 80% of them were students, and their educational distribution can be seen on figure 83

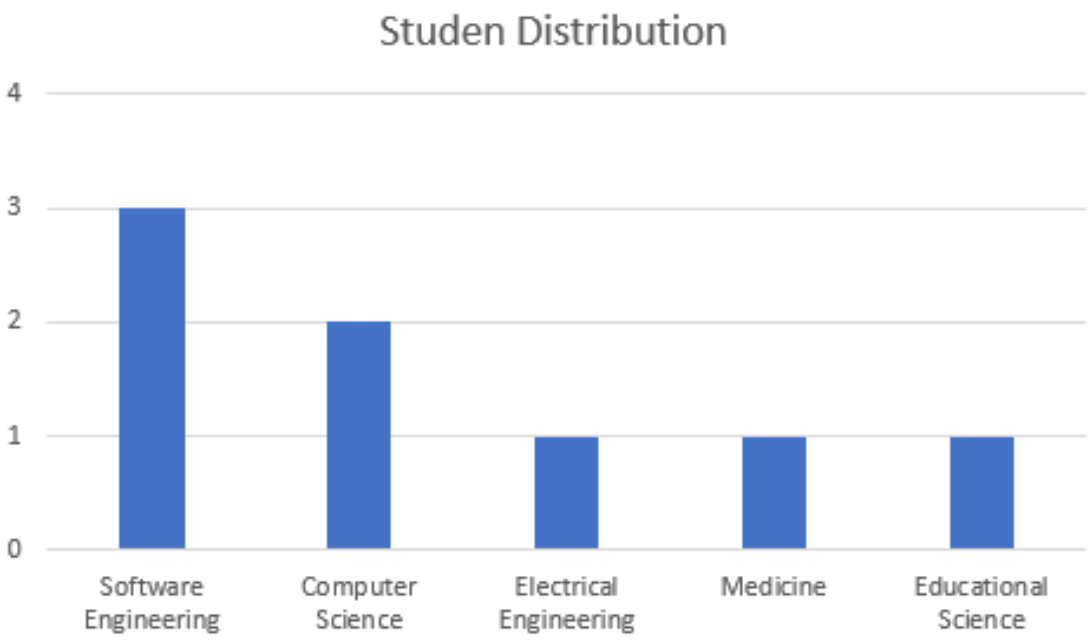


Figure 83: Distribution of education between IVR test subjects.

The test subjects were a mixed group of experience with interacting with 3D virtual environments, and 60% of the subjects had tried VR previously. Of those 60% most identified as "not that experienced" in VR.

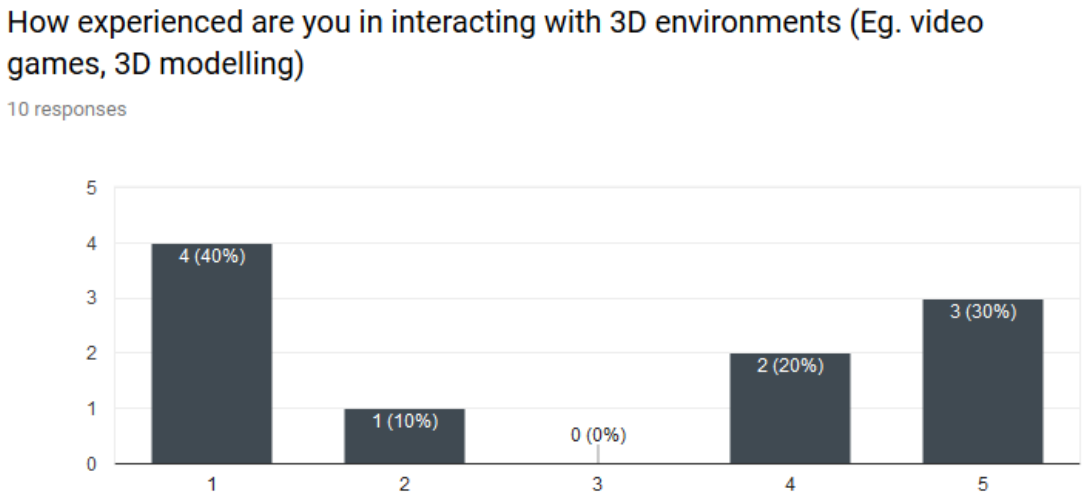


Figure 84: IVR ranking of experience with 3D environments with 1 meaning not that experienced, and 5 meaning very experienced.

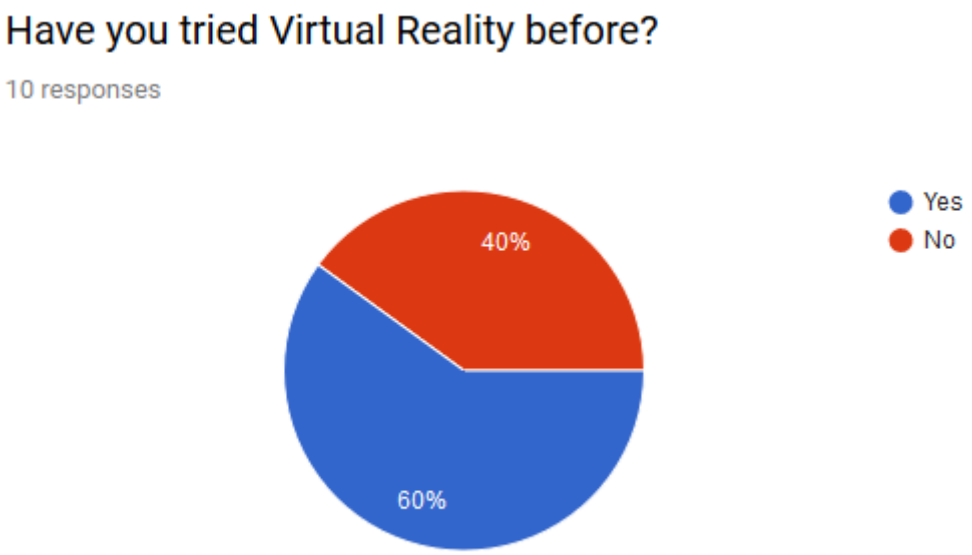


Figure 85: IVR distribution of previous experience with VR.

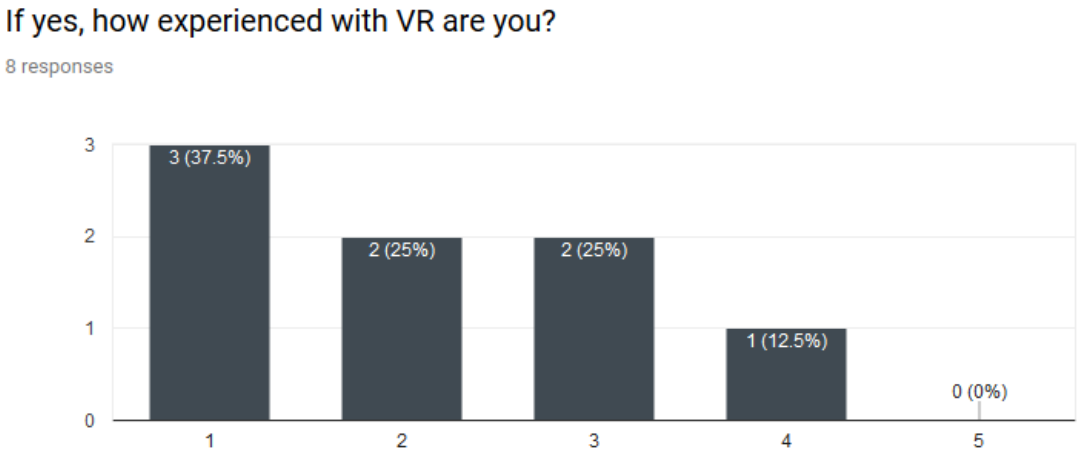


Figure 86: IVR ranking of experience with Virtual Reality with 1 meaning not that experienced, and 5 meaning very experienced.

When the test subjects tried teleportation, most users strongly agreed with the sentence "It was easy to orient myself and navigate the environment with teleportation" while most of the user disagreed with the statement "Teleporting made me feel nauseous or queasy"

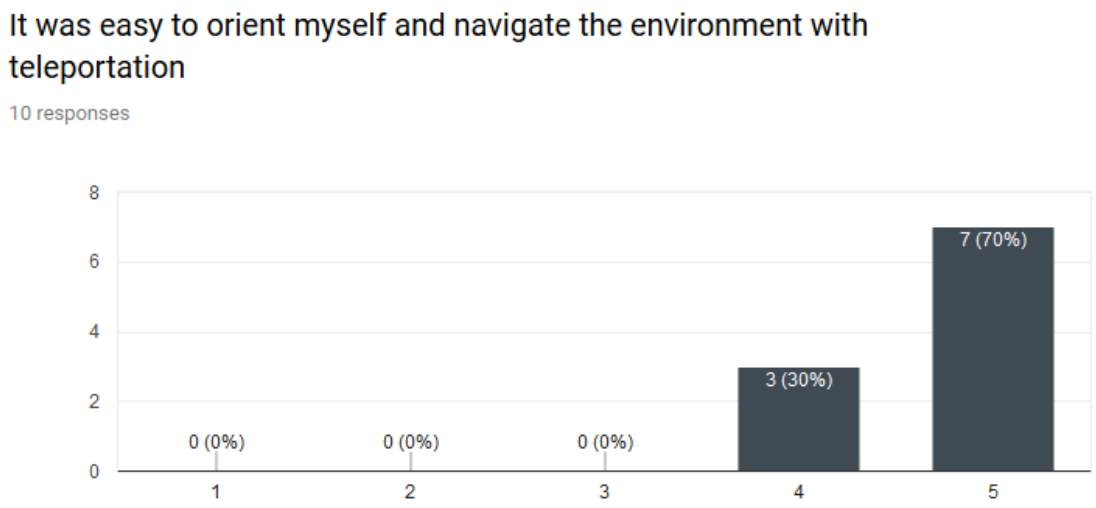


Figure 87: IVR statistics of agreeing or disagreeing with the statement "It was easy to orient myself and navigate the environment with teleportation" with 1 meaning strongly disagree, and 5 meaning strongly agree.

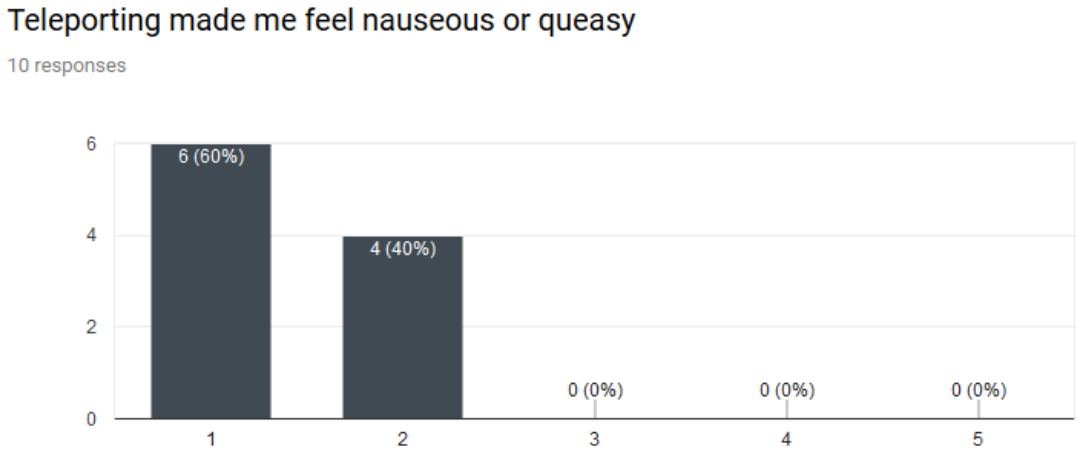


Figure 88: IVR statistics of agreeing or disagreeing with the statement "Teleporting made me feel nauseous or queasy" with 1 meaning strongly disagree, and 5 meaning strongly agree.

When asked about how they felt about Gaze-based walking, we're somewhat more split in agreeing it the statement "It was easy to orient myself and navigate the environment with gaze-based walking", and people strongly agreed with the statement "Gaze-based walking made me feel nauseous or queasy".

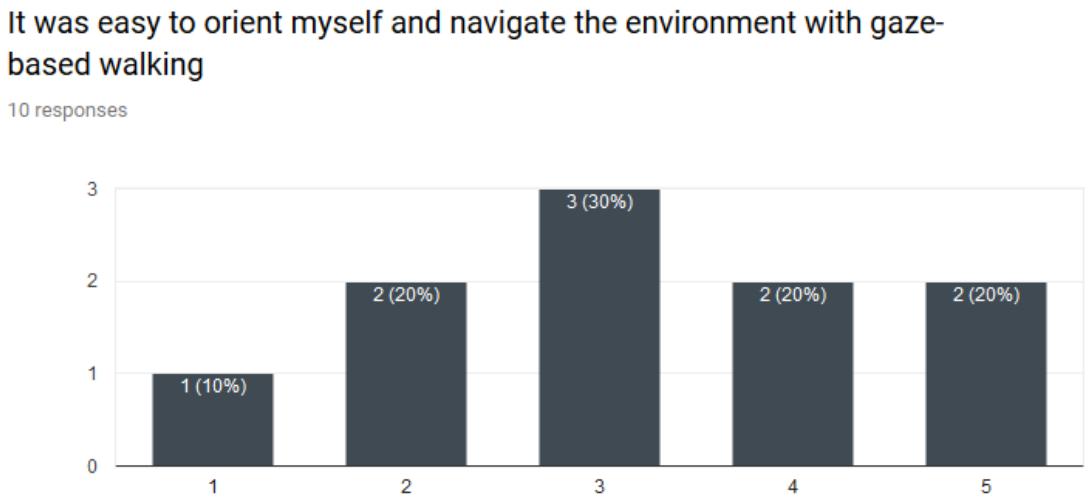


Figure 89: IVR statistics of agreeing or disagreeing with the statement "It was easy to orient myself and navigate the environment with Gaze-based walking" with 1 meaning strongly disagree, and 5 meaning strongly agree.

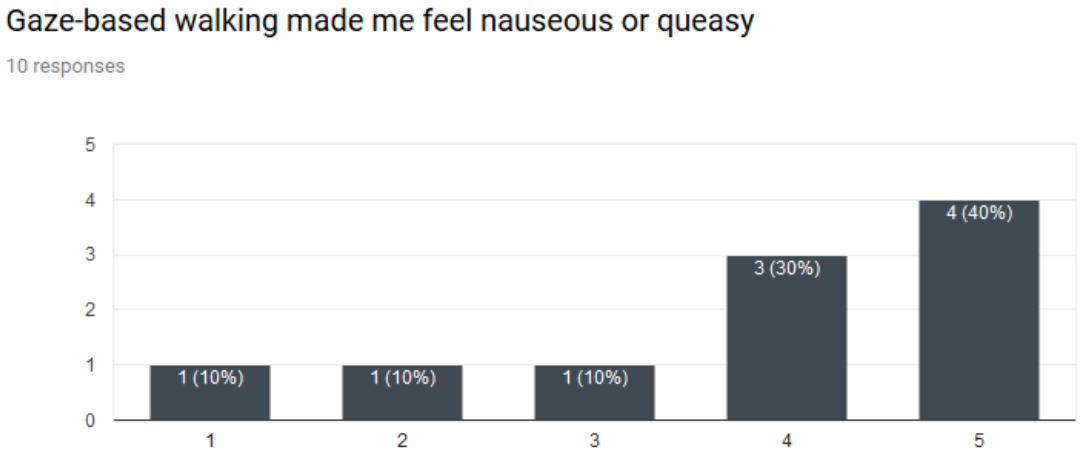


Figure 90: IVR statistics of agreeing or disagreeing with the statement "Gaze-based walking made me feel nauseous or queasy" with 1 meaning strongly disagree, and 5 meaning strongly agree.

As an objective measurement of the user's performance in the VR environment we have measured their task completion time (the time it took the users to go through the waypoints). The average speed using Teleportation was 55.54 seconds, while the average speed for Gaze-based walking was 62.70 seconds (See appendix I.4).

Finally, when we asked the users of their personal navigation mode preference, 80% of the users preferred teleportation, with 10% preferring Gaze-based walking.

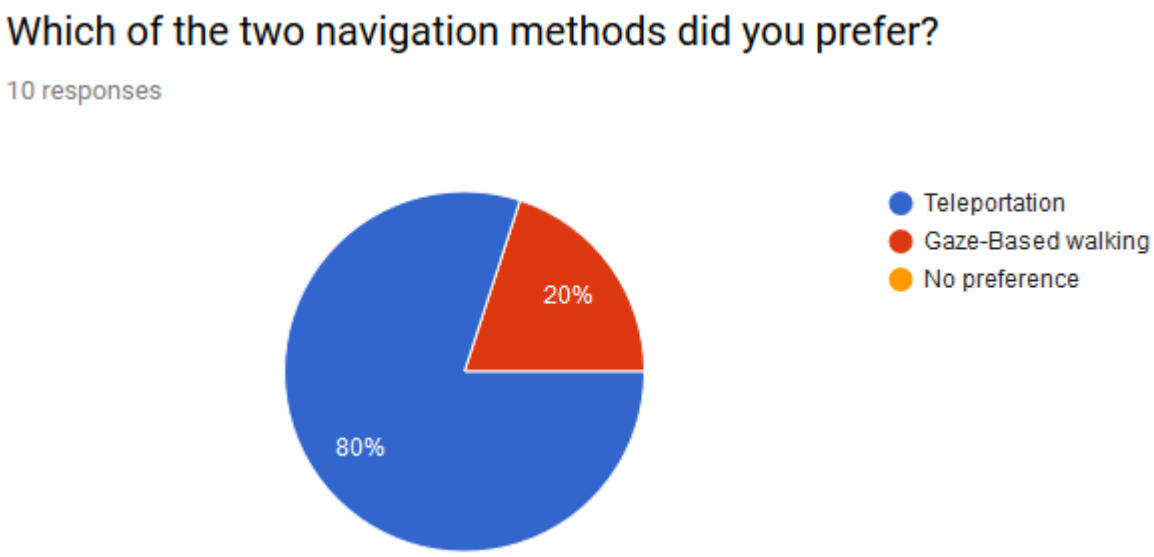


Figure 91: Navigation mode preference by IVR users.

Users that preferred Gaze-based walking commented, that Gaze-based walking felt more real, and authentic, but breaking near furniture did feel nauseating.

Users that preferred teleportation, commented that it the easiest and fastest of the two methods, and that they had more control of where they went. They also commented on, how gaze-based walking made them feel nauseous.

D.4 Waypoint System

In order to facilitate the navigation experiment, we had to implement a waypoint system.

In the navigation experiment, users were required to follow a predefined path through a virtual environment. This predefined path was implemented through the waypoint system.

Figure 92 shows a screenshot of the virtual environment used for the navigation experiment. Notice the yellow, transparent, pillars. These are the waypoints.

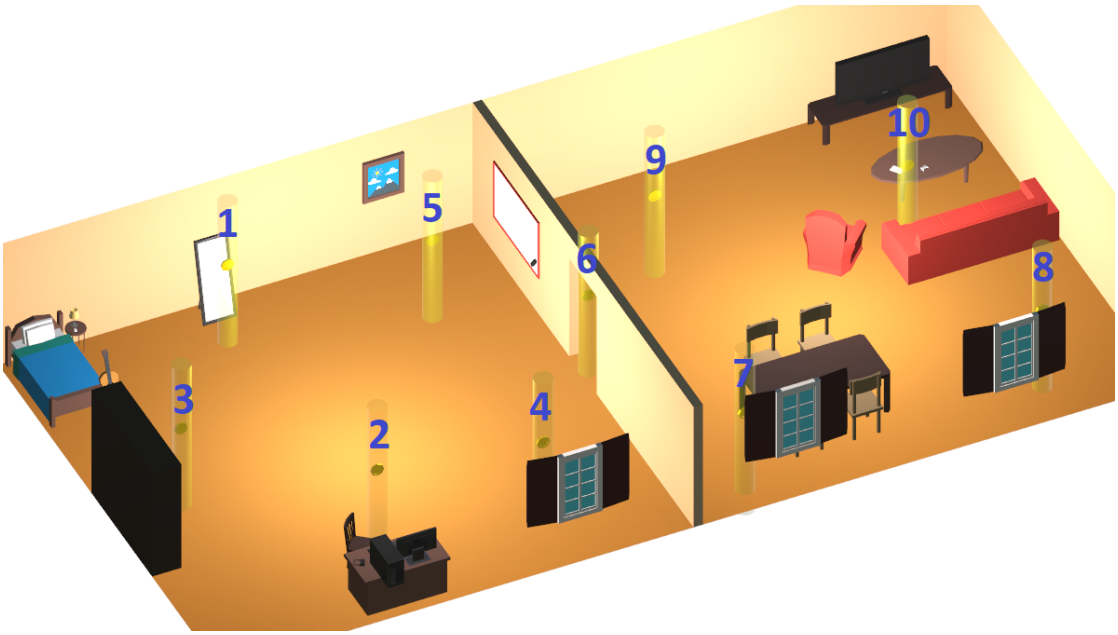


Figure 92: Screenshot of the virtual environment for the navigation experiment. The yellow, transparent, pillars are the waypoints describing a predefined route.

The navigation system consists of three scripts: *Waypoints*, *Waypoint Manager* and *CoinFlipBimbo*. These three scripts will now be described through practical examples.

An experiment consists of a collection of waypoints. All waypoints have their own unique *Order ID*. This Order ID goes from 0 and up. As an example, the navigation experiment presented in figure 92 consists of 8 waypoints. Thus, each waypoint will have assigned a unique Order ID from 0 to 7.

The Order ID is important. This is because only one waypoint will be presented at any one time during the experiment. Thus, at the very beginning of the experiment, the waypoint with an Order ID of 0 will be visible in the virtual environment. The user will then have to navigate into this waypoint. When this happens, the waypoint with an Order ID of 1 will be displayed, and the previous waypoint will be hidden. This process repeats until all waypoints within the experiment have been navigated to. It is the responsibility of the *Waypoint System* to provide this logic. In this way, it is possible to design a predefined route which a user must using a sequence of waypoints.

Figure 93 shows a waypoint up close.

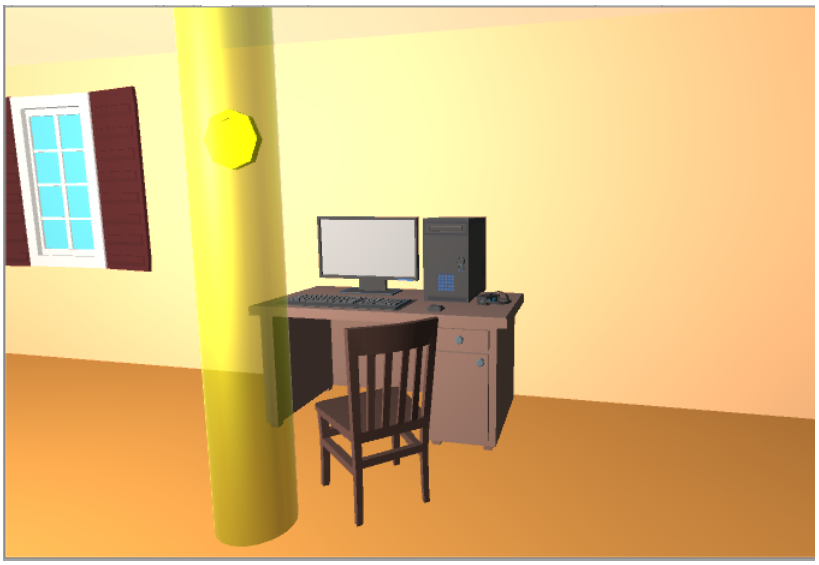


Figure 93: Screenshot of waypoint up close. An animated, spinning coin, is placed within the center of a waypoint.

In figure 93 it is possible to see the coin in the center of the waypoint. This is a spinning coin animated through the *CoinFlipRotater* component.

The three scripts can be seen in figure 94.

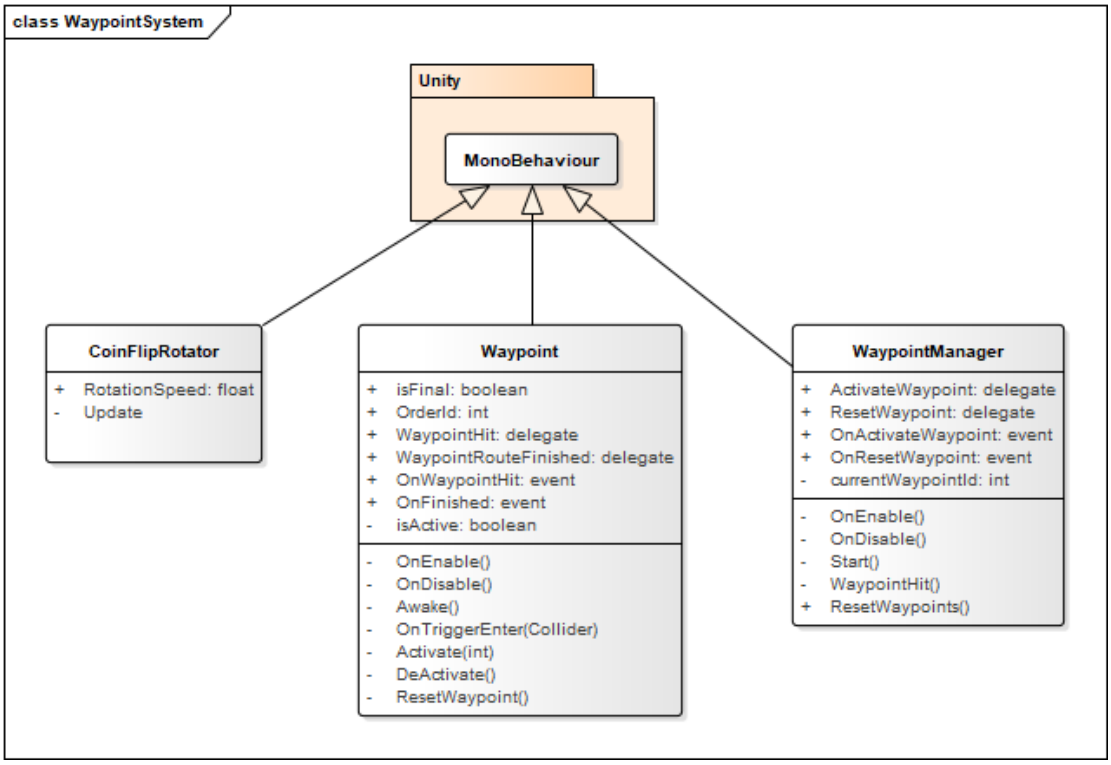


Figure 94: Class diagram of the three scripts composing the waypoint system.

The Waypoint System is based on events, in order for the *Waypoint* and *WaypointManager* to communicate important events. Furthermore, interested *outsiders* are able to listen in on these events as well, if it is important for the task that they have to perform.

These events are described in the following table.

Event	Emitter	Listener	Description
OnActivateWaypoint	WaypointManager	Waypoint	This event is emitted when a new waypoint has to make itself visible.
OnResetWaypoint	WaypointManager	Waypoint	This event is emitted when a client calls ResetWaypoints() on the WaypointManager. It is used by Waypoints in order to hide or show themselves in preparation for a user repeating the same route.
OnWaypointHit	Waypoint	WaypointManager	This event is emitted when a waypoint has been hit by the user. The WaypointManager uses it to determine what waypoint in the route should be activated next.
OnFinished	Waypoint	Interested Third-Parties	This event is emitted when the waypoint representing the final waypoint in a route is hit by a user. Other components within the project listen to this event in order to, for example, save experiment measurements when the route of the experiment is completed.

Table 18: Waypoint System Events

The *WaypointManager* has the responsibility of keeping the general overview of what Order Id is the current waypoint of the route.

Each *Waypoint* simply shows or hides themselves when they become the correct waypoint in the Order Id sequence. If a waypoint is hit and is the final waypoint of the route, it will also emit an *OnFinished* event.

The sequence diagram in figure 95 shows the general flow of the waypoint system, from when a user hits a waypoint.

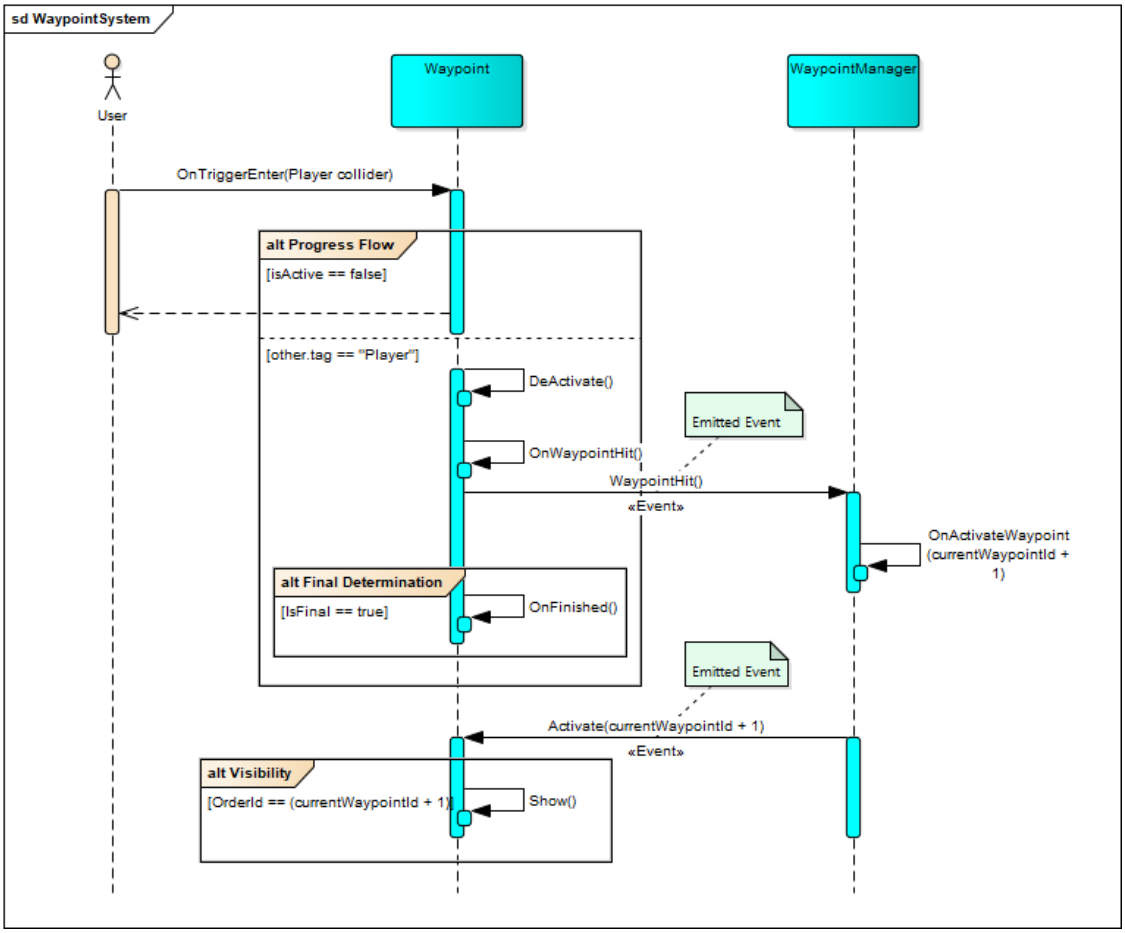


Figure 95: Sequence diagram of the flow of the navigation system, from the point of a user hitting a waypoint.

D.4.1 Waypoints

In order to make the creation of a route as easy as possible through the use of waypoints, the two attributes *IsFinal* and *OrderId* is exposed in the inspector. Figure 96 shows a screenshot of the scene for our navigation experiment, with a waypoint selected. Here, the attributes can be seen as visible and editable in the inspector, right side.

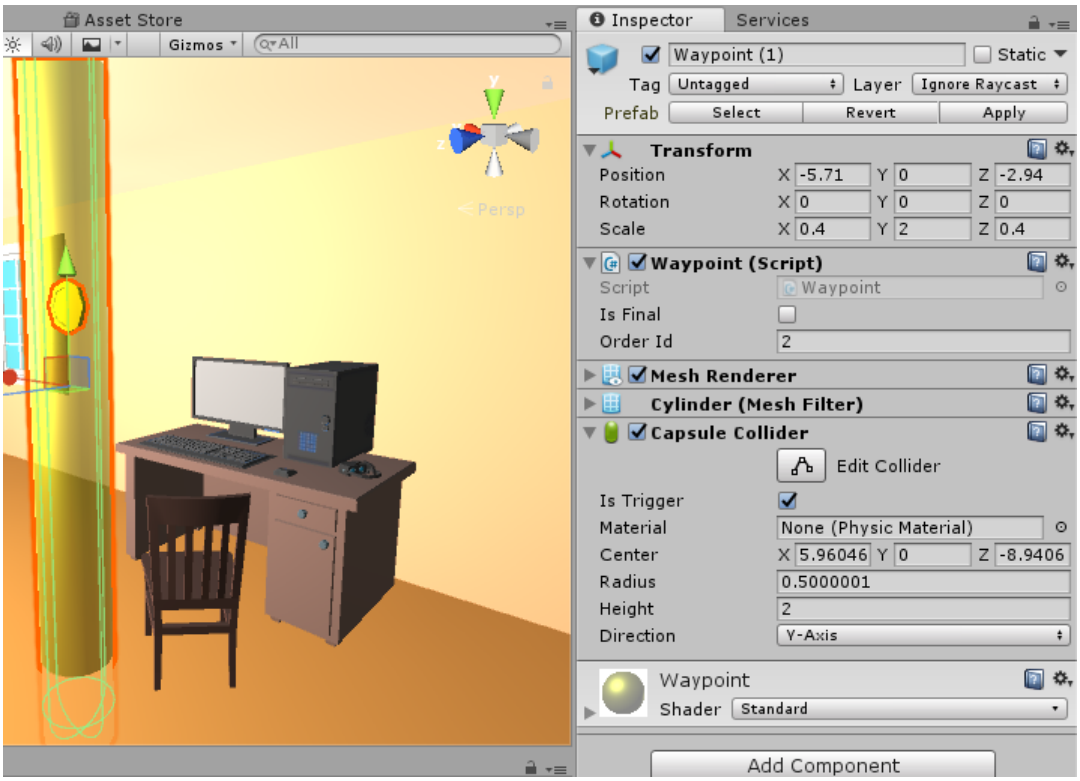


Figure 96: A waypoint selected in the navigation experiment scene. Its attributes *IsFinal* and *OrderId* can be seen in the inspector on the right.

These two attributes have been exposed through the inspector in order to make the waypoint system easily usable to create routes in any scenes. The *IsFinal* attribute can be checked to indicate that the waypoint is the last one in the route. This means that it will emit an *OnFinished* event when triggered (See table 18).

Furthermore, we made waypoints a prefab within the project. This means that it is an easily re-useable component that can be dragged into any scene without requiring manual setup.

D.4.2 CoinFlipRotator

This is a simple script which animates the coin at the center of the waypoint, by continuously rotating it.

The script exposes the *RotationSpeed* attribute to the inspector. This can be used to easily adjust the rotational speed of the animation. A screenshot of the CoinFlipRotator script from the inspector can be seen in figure 97.

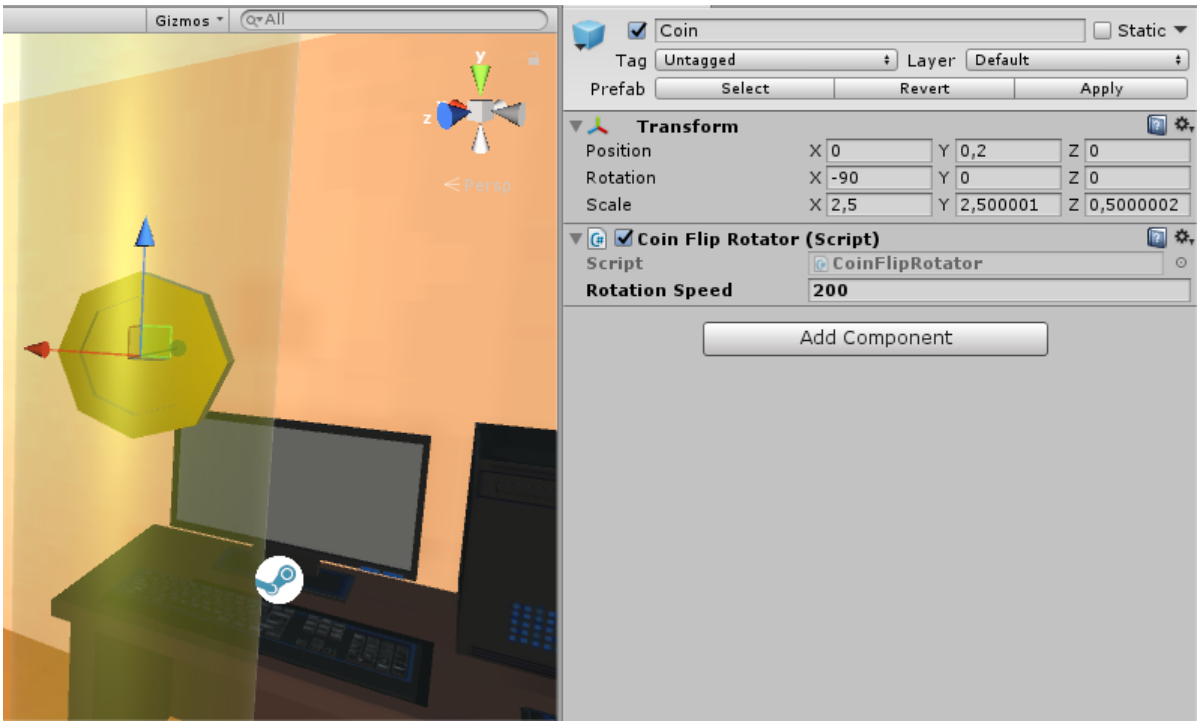


Figure 97: The CoinFlipRotator script as seen from the inspector. The *RotationSpeed* attribute is easily adjustable from here.

D.5 Navigation

Navigation is implemented differently depending on whether the platform is IVR and MVR. This is because navigation is heavily dependent on gathering input from the controllers, and these change depending on what platform you develop for.

The following two subsections will describe how we have implemented navigation on IVR and MVR respectively. During the project we implemented two modes of navigation: Gaze-based locomotion and Teleportation. These are the implemented modes that will be described here.

D.5.1 Installed Virtual Reality

In order to implement navigation for IVR, we made use of Unity's built-in input handler, as well as SteamVR SDK [73] for Unity.

Gaze-based locomotion

Gaze-based locomotion for IVR is handled through the script *GazeNavigationInstalled*. According to the SteamVR Unity documentation, the HTC VIVE controllers touch input from the trackpad is mapped to the Unity joystick ID's 2 and 4 [80]. Through the Unity

Input Manager we have mapped these ID's to the custom-defined axes *HtcViveLeftHand* and *HtcViveRightHand*. Figure 98 shows both of our custom-defined axes, with a detailed view of the *HtcViveRightHand* axis. Both axes are configured the same way.

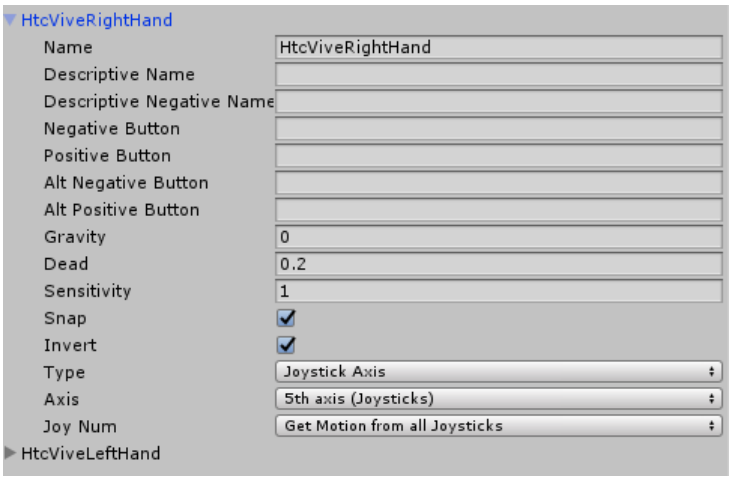


Figure 98: Screenshot from the Unity input manager

The script of *GazeNavigationInstalled* reads from the joystick input axes that we defined in the input manager, and applies that value to a forward movement vector. Since the script is applied to both controllers (referred to as *hands* in the code), it is important for the script to detect its relevant controller. If a script instance is applied to the left controller, it should only read the touchpad values from that controller. This is shown in the codesnippet of figure 99.

```
public void Update () {
    string axisName = hand.GuessCurrentHandType() == Hand.HandType.Right
        ? "HtcViveRightHand"
        : "HtcViveLeftHand";

    float axisValue = Input.GetAxis(axisName);

    Vector3 camForward = _mainCamera.transform.forward;
    camForward.y = 0;

    Vector3 forwardMove = camForward * axisValue * MovementSpeed * Time.deltaTime;

    _player.transform.Translate(forwardMove);
}
```

Figure 99: Codesnippet from the update method of the GazeNavigationInstalled script

Teleportation

Teleportation is implemented entirely using SteamVR's official prefabs from the Unity SteamVR SDK.

We felt that this was the most optimal way of doing it, as it provides all the features we need for teleportation and roomscale.

Figure 100 is a screenshot of one of our early prototypes using SteamVR's prefabs.

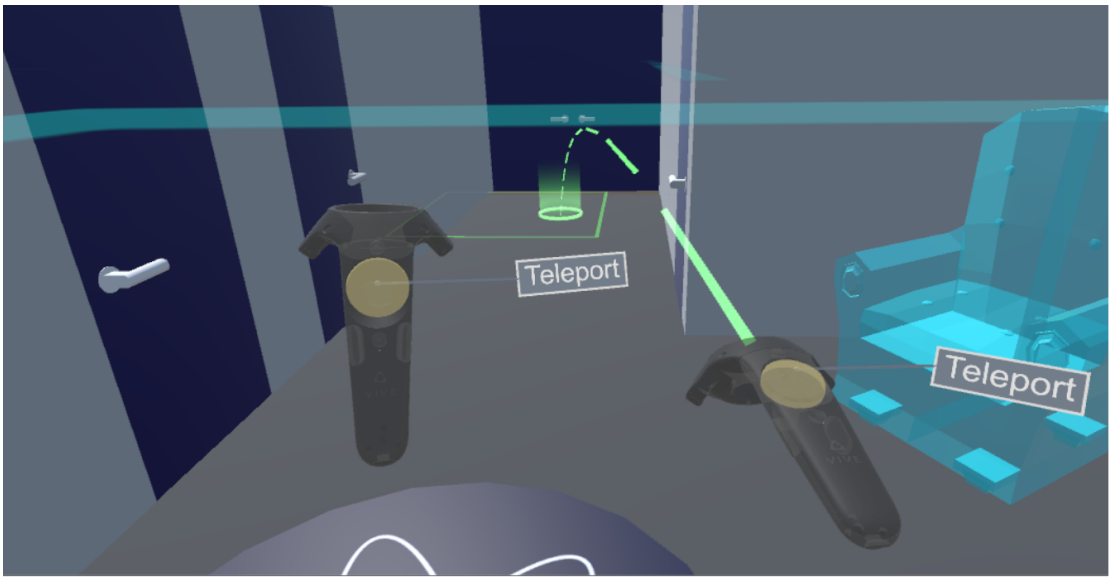


Figure 100: Screenshot demonstrating the SteamVR teleportation system in one of our prototypes.

In order to make teleportation work, you have to make use of two prefabs provided by SteamVR: *Teleporting* and *TeleportArea*. These two prefabs will now be described in detail.

Teleporting is a prefab which controls the general configuration for teleportation in your scene. Figure 101 demonstrates some of the possible settings.

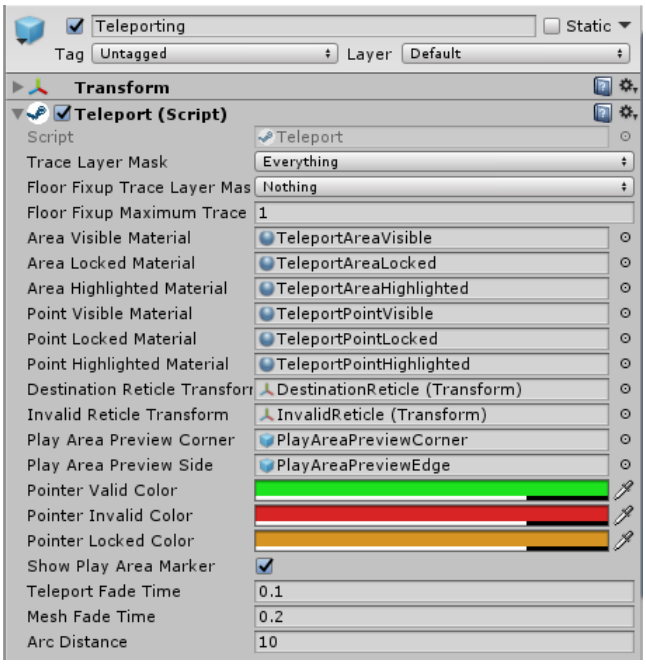


Figure 101: Settings from the Teleporting prefab within a scene of our project.

As can be seen, it is possible to configure things such as the materials used for different states of teleport areas within your scene. It is also possible to configure things such as the raycast arc, as seen on figure 100.

D.5.2 Mobile Virtual Reality

In MVR, gaze-based locomotion and teleportation is implemented using the *GVRControllerInput* class from the Google Daydream Unity SDK. [81]

Gaze-based locomotion
Gaze-based locomotion for mVR is implemented in the script *Gaze Navigation*. The script checks the *GVRControllerInput* class to see if the user is interacting with any controller input. In this script we specifically look for the touch-input on the trackpad. From the documentation [81] we see that, the controller trackpad is divided into a 1-by-1 coordinate system with (0,0) in the top left corner of the trackpad. Using this knowledge we can detect where the user is touching the trackpad, and use this to decide whether or not to move the user. This can be seen in the code snippet in figure 102.

```
public void Update ()
{
    if (!GvrControllerInput.IsTouching) return;

    var forwardMovement = _mainCamera.transform.TransformDirection(Vector3.forward);
    forwardMovement.y = 0;

    var touchPosition = GvrControllerInput.TouchPos;
    if (touchPosition.y < 0.25f)
        _player.transform.Translate(forwardMovement * Time.deltaTime, Space.World);
    else if (touchPosition.y > 0.75f)
        _player.transform.Translate(-forwardMovement * Time.deltaTime, Space.World);
}
```

Figure 102: Code snippet from the update method of the *Gaze Navigation* script.

We decided to create a "deadzone" in the middle of the trackpad where the user can rest his or her finger without moving. If the user touches the top of the trackpad, the position of the player will be moved forward in the direction the player is looking, and if the user touches the bottom of the trackpad, the player will be moved backwards relative to where the player is looking.

Teleportation

Teleportation for MVR is implemented in the script *Teleportation*. The script is placed on the "laser" prefab which is a part of the controller representation. We did this to be able to easily use the controllers transform as a starting point for the physics raycast (See figure 103)

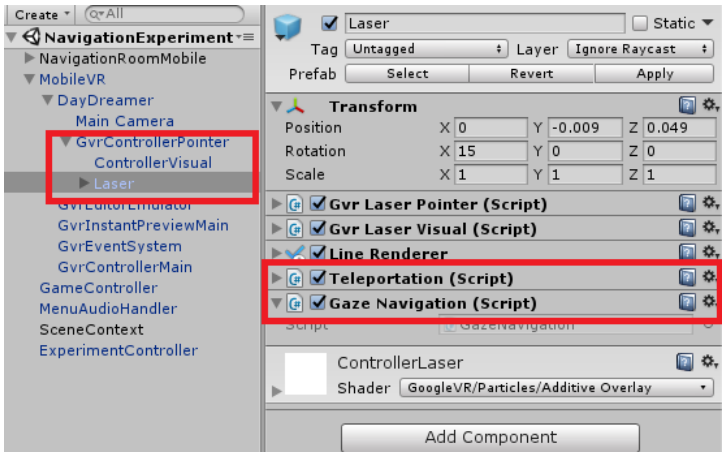


Figure 103: Placement of the navigation scripts in the controller prefab.

From the update method of the Teleportation script, we raycast 20 units in the forward direction of the transform. Using the RaycastHit class, we can see the gameobjects hit

by this raycast and get the hit coordinates in world space, which we then can use to move the player when pressing the teleport button. The button press is detected by using the *GVRControllerInput* class. Since we, for the final experiment, is going to have furniture objects in the scene, we want to make sure that the player cannot teleport on top of furniture objects. To achieve this, we simply check the tag of the RaycastHit object, and disallow teleportation if the hit object is teleportation. A code snippet of the the teleportation logic can be seen on figure 104.

```
public void Update () {
    var forwardDirection = transform.TransformDirection(Vector3.forward);

    RaycastHit raycastHit;
    if (!Physics.Raycast(transform.position, forwardDirection, out raycastHit, 20)) return;

    _hittingFurniture = raycastHit.transform.CompareTag("Furniture");
    if (GvrControllerInput.ClickButtonDown)
    {
        if (!_hittingFurniture)
        {
            Debug.Log("Hit : " + raycastHit.collider.name);
            if (raycastHit.collider.CompareTag("Floor"))
            {
                _player.transform.position = new Vector3(
                    raycastHit.point.x,
                    _player.transform.position.y,
                    raycastHit.point.z);
            }
        }
    }
}
```

Figure 104: Code snippet of the update method in the *Teleportation* script.

D.6 Measurement Repositories

The measurement repositories of the navigation experiments are used to save the navigation experiment measurements to Firebase.

Figure 105 shows a class diagram of the classes comprising the measurement repositories.

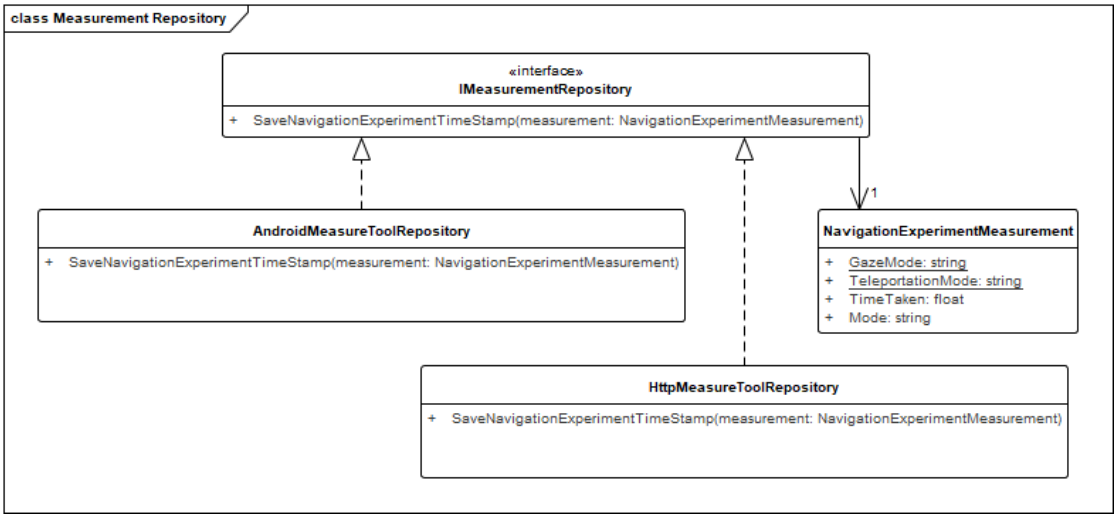


Figure 105: The classes and interfaces comrpising the navigation measurement repositories.

The interface *IMeasurementRepository* represents the possible operations of any concrete implementation of a navigation measurement repository. It contains the method *SaveNavigationExperimentTimeStamp*, which takes an instance of the model class *NavigationExperimentMeasurement*.

Relevant data for the navigation experiment was how long a user took to complete the navigation experiment, and which mode of navigation was used. Thus, the *NavigationExperimentMeasurement* model class contains the properties *TimeTaken* and *Mode* exactly for this purpose.

The properties *GazeMode* and *TeleportationMode* of the *NavigationExperimentMeasurement* are static constant strings which can be used by clients when setting the *Mode* property. It is simply pre-defined strings which can be used in order to avoid inconsistent spelling of the navigation modes used.

The classes *AndroidMeasureToolRepository* and *HttpMeasureToolRepository* are concrete implementations of the *IMeasurementRepository* interface. These are platform-specific implementations which can be used on Android or PC respectively.

See section D.7 for technical details of how these repositories were used practically in the navigation experiment.

Also notice that during the final object manipulation experiment, a generalized repository system for saving measurements was conceived and implemented. This is a separate system, the Firebase Measurement Repository System, documented in section F.4.

D.7 NavigationExperimentMeasurer

The *NavigationExperimentMeasurer* is a script in our system which has the responsibility of saving time measurements from the navigation experiment onto the database once the pre-defined route in the experiment is completed by a user.

Figure 106 presents the script in a class diagram.

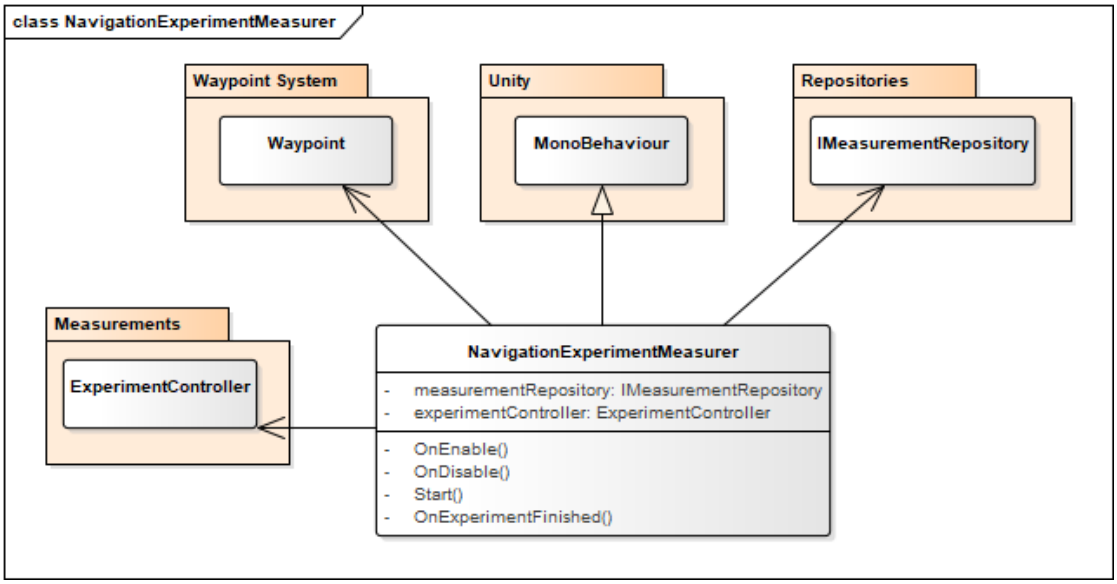


Figure 106: Class diagram of the NavigationExperimentMeasurer script.

The context of the methods of *NavigationExperimentMeasuer* is best understood through the flow of its lifetime. Thus, figure 107 presents the primary sequence for the lifetime of this script. It utilizes the event system of the Waypoint System described in section D.4 in order to listen in on the *OnFinished* - described in table 18 of section D.4 - event that is triggered once a user completes the pre-defined route. When this event is triggered, the time in seconds which it took to complete the route is fetched from the *Experiment-Controller* of the scene, and afterwards saved to the database through the repositories presented in section D.6.

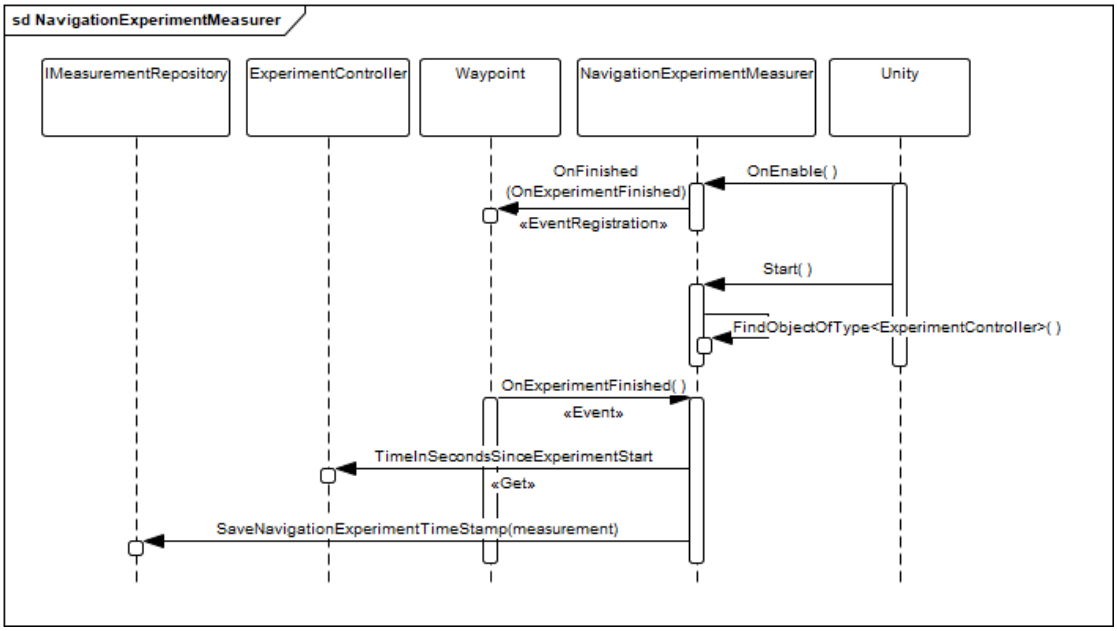


Figure 107: Class diagram of the NavigationExperimentMeasurer script.

D.7.1 Cross-Platform Considerations

NavigationExperimentMeasurer works cross-platform through the use of Zenject.

The attribute *measurementRepository*, seen in the class diagram of figure 106 is determined and injected by Zenject at runtime depending on the platform which the application is running on. In this way, *NavigationExperimentMeasurer* is easily reuseable, and can easily be used to extend to new platforms.

D.8 NavigationPlayerController

The *NavigationPlayerController* is a script in our system which has the responsibility of keeping a shared state of the current navigational mode of the player during the navigation experiment. Additionally, it also configures anything that has to do with the navigation.

Figure 108 shows a class diagram of the script.

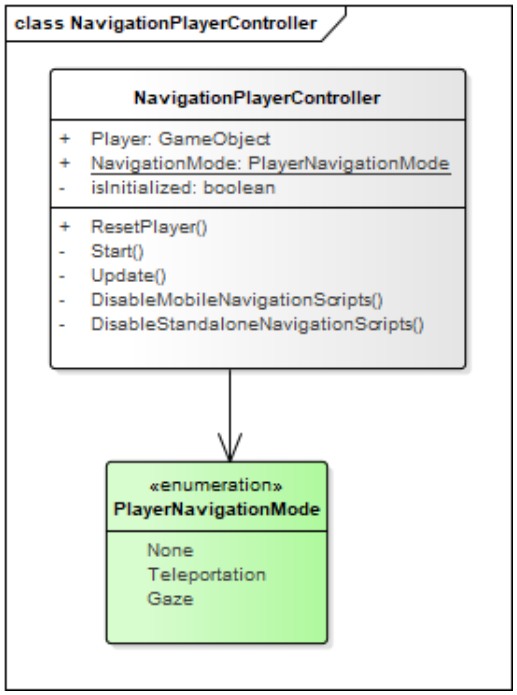


Figure 108: Class diagram of the NavigationPlayerController script.

The script has two primary purposes.

It disables all navigation during the first launch of the navigation experiment. This is to ensure that users are forced to choose a navigation mode themselves.

It is also used by other scripts of the navigation experiment to reset the player to a starting position. This is done through the publically exposed method *ResetPlayer*.

The initialization flow of the script can be seen in figure 109.

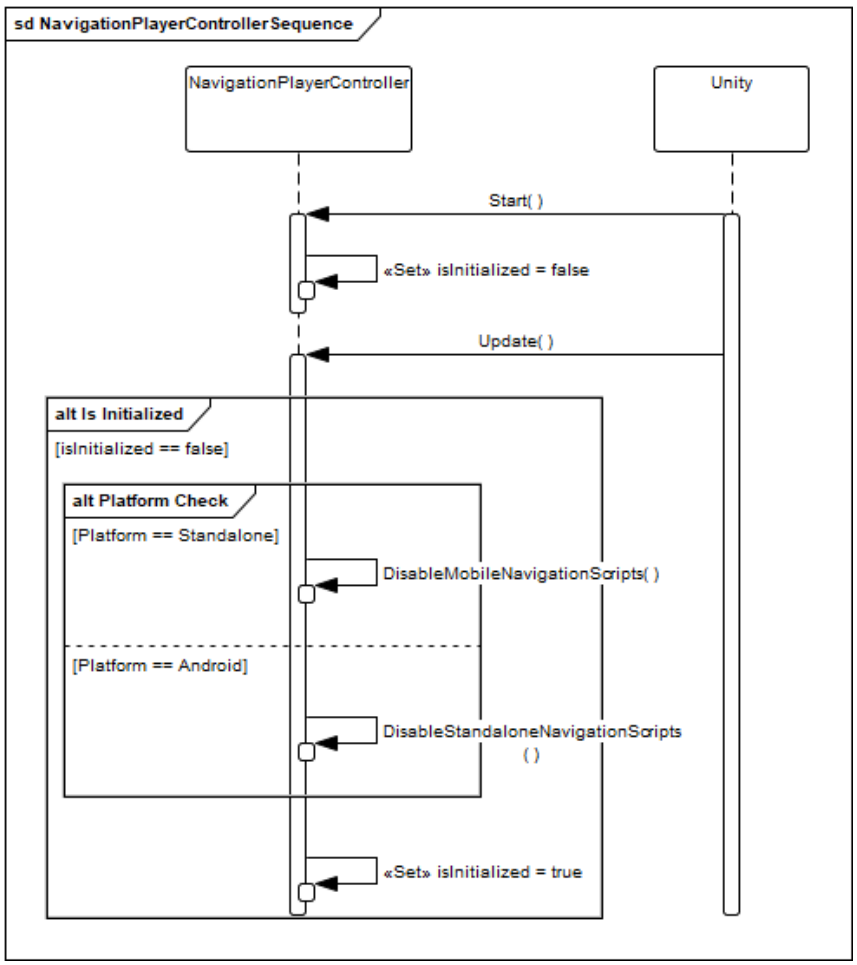


Figure 109: Sequence diagram of the NavigationPlayerController script.

D.9 Navigation Experiment UI

As the navigation experiment consisted of users having to try out two different modes of navigation, it was important that we had an easy way to switch between the two different modes during runtime of the navigation experiment on both platforms.

In order to achieve this, we made an interactable 3D menu that can be brought up at any point during runtime. The final result can be seen in figure 110.

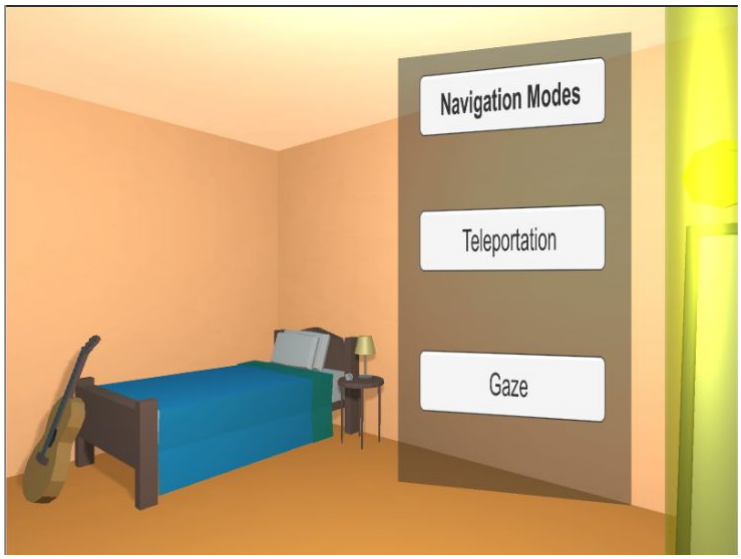


Figure 110: Screenshot of the menu visible in the navigation experiment room.

As can be seen in figure 110, the menu is a 3D panel that is placed within the virtual environment. By interacting with the buttons, the navigation mode can easily be changed.

The menu is also animated, in the sense that it will gradually rotate towards the player, so as to always be facing in the correct direction. This means that if you make the menu visible within the world and start to move around, the menu will continually look towards you.

The 3D menu is the same on both IVR and MVR platforms. Therefore, we made the 3D menu as a prefab which could be shared amongst the two platform scenes. However, as we learned in the UI prototypes for both the installed and mobile platform - see section G.3 - each platform required different platform-specific components on the UI GameObjects comprising the menu. As such, both platforms use a shared menu prefab as a starting point, and then each platform scene will add the necessary platform-specific components.

D.9.1 Scripts

The menu system for the navigation experiment is comprised of multiple scripts. These scripts can be seen in the class diagram of figure 111.

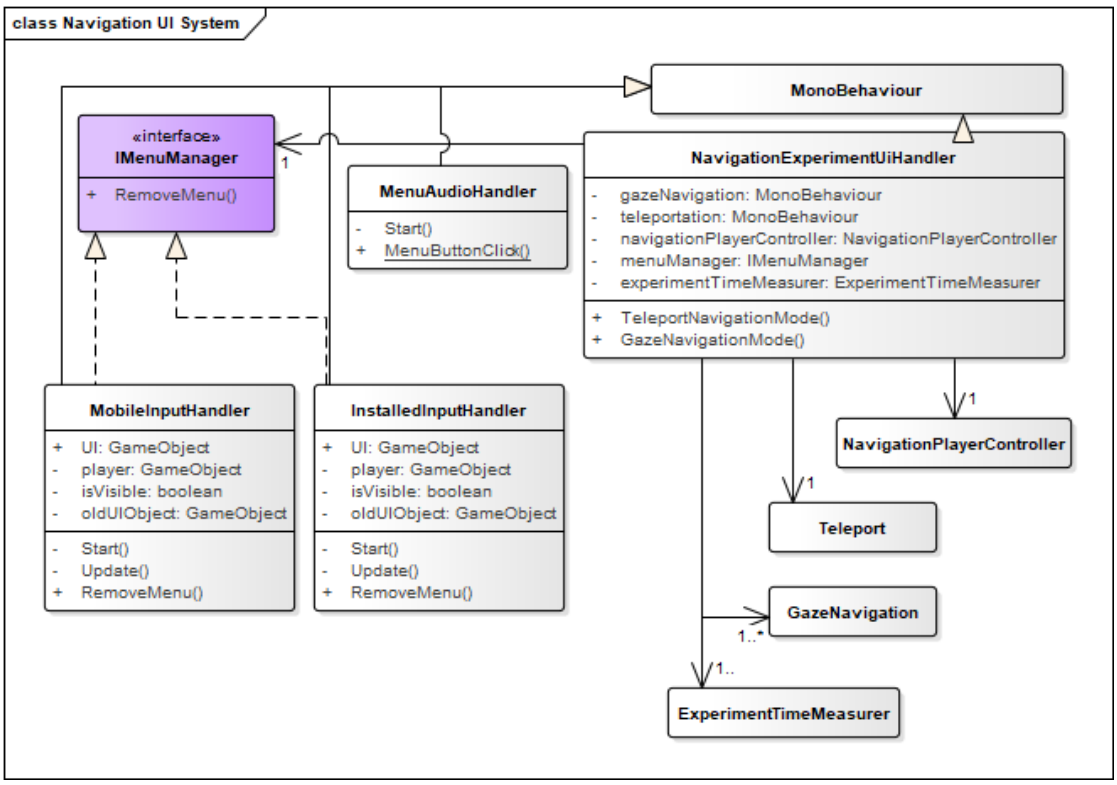


Figure 111: A class diagram of the classes comprising the menu system of the navigation experiment.

From the class diagram of figure 111, there are important concepts that make up the menu system. They are: *Menu Managers*, *MenuAudioHandler*, and the *UI Handler*.

The *NavigationExperimentUiHandler* is the UI handler. This script contains two public methods, *TeleportNavigationMode* and *GazeNavigationMode*. These methods are responsible for toggling the navigation mode which the player will use to navigate the virtual environment. These methods are hooked up to click events on buttons in the UI. This script is platform agnostic.

The menu managers handles the menu GameObject of the virtual world. These scripts have the responsibility of reading the input of the platform-specific controller, and from that hide or display the menu in the virtual environment. Additionally, these menu managers can be used by the *NavigationExperimentUiHandler* to remove the menu from the virtual world.

The *MenuAudioHandler* is used to play a button click sound when a button is clicked.

The general flow of these scripts, beginning from user input, can be seen in the sequence diagram of figure 112.

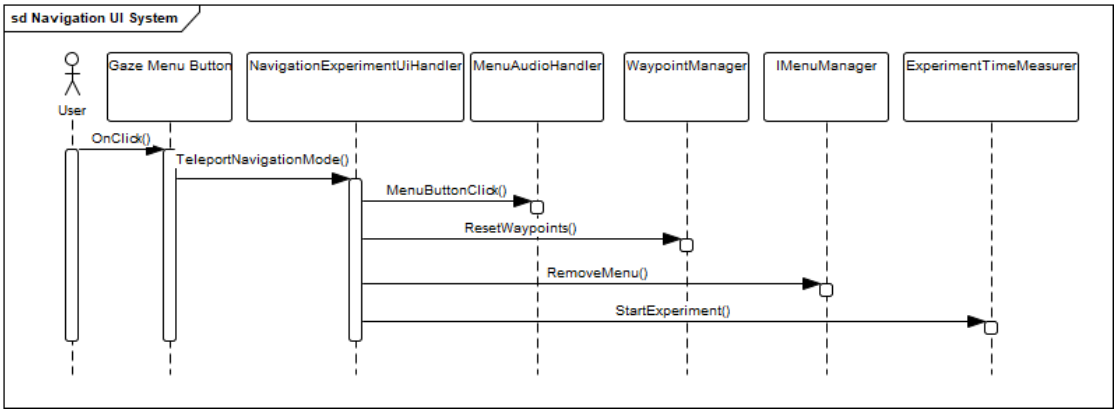


Figure 112: A sequence of the typical flow of the menu system from the point of a user clicking a button on the navigation experiment menu. This example is from the point of reference of the Gaze menu button being clicked.

LookAtPlayer Script

The menu system also consists of a script called *LookAtPlayer*. The sole responsibility of this script is simply to constantly rotate the menu to face the player.

D.9.2 Prefab

The shared prefab of the menu, shared amongst both platform scenes, is essentially a hierarchy of GameObjects. A screenshot of the structure of the menu can be seen in figure 113.

Figure 113: Screenshot of the structure of the shared navigation menu.

These GameObjects make up all the visual makeup of the menu shown in figure 110, without any of the platform specific scripts attached.

Here, the responsibility of each GameObject in the hierarchy will be explained:

- *SharedNavigationUI*: This is the GameObject which acts as a parent for all other objects making up the menu. It contains the *LookAtPlayer* script.

- *UIHandler*: This GameObject contains the *NavigationExperimentUiHandler* script. All menu button clicks are routed to the methods of this script.
- *Background*: This is simply the GameObject for the visual dark-transparent background seen on the menu.
- *Experiment Selection*: This GameObject is a UI canvas. In Unity, it is required that all UI buttons be a parent of a canvas.
- *TeleportationButton*: This is the GameObject making up the button on the menu that allows the user to select *Teleportation*.
- *GazeButton*: This is the GameObject making up the button on the menu that allows the user to select *Gaze*.
- *Title*: This is the GameObject making up the text title "Navigation Modes" seen at the top of the menu.

From this shared prefab there exists two platform-specific menu prefabs. The reason for this is that whilst they share the same general structure, each platform requires certain platform-specific scripts to be attached to the button GameObjects in order to work with respectively SteamVR on the installed platform and Google Daydream on the mobile platform.

Appendix E The Object Manipulation Experiment

To answer the question

> ***What is the difference in accuracy and efficiency of object manipulation between installed- and mobile virtual reality systems, and what is the perceived user satisfaction between the two technologies?***

we had to create an experiment, from which we could gather both objective data about the object manipulation as well as subjective data about user satisfaction.

E.1 Experiment Setup

The experiment we came up with was a simple task of placing furniture in predefined positions as accurately as possible, using the platform specific VR equipment. The experiment was split into three phases as can be seen in figure 114.

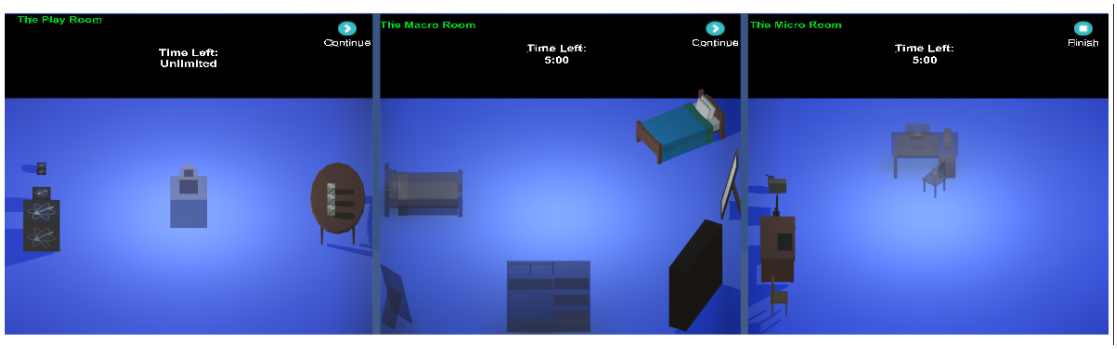


Figure 114: The Object Manipulation experiment rooms seen from an orthographic view.

The first phase of the experiment placed the user in the "Play room" (figure 115) which is used to condition the user to the platform specific controls. In the playroom the user have to place the three cubes with orbitlab logos on them in the positions indicated by the ghost objects. As soon as the cubes have been placed accurately, a continue button appears, allowing the user to proceed to the next phase of the experiment.

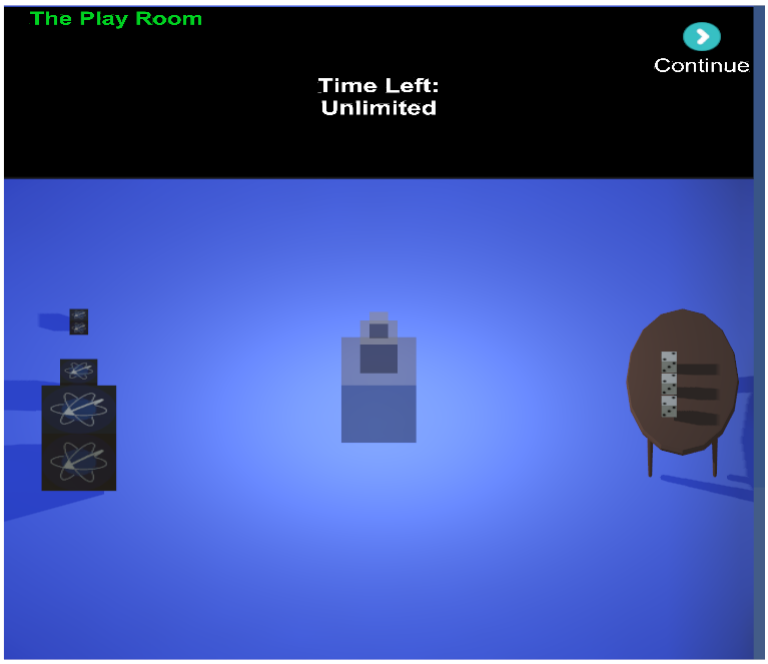


Figure 115: Orthographic view of the playroom.

The second phase of the experiment, is the "Macro room" (see figure 116). In this room, the user will have to handle large objects, as they are tasked to place a bed, a closet, and a mirror on top of the ghost objects. To ensure that users do not spend too much time trying to perfect the placement of a single furniture, we implemented a time-boxing feature that, after a predefined time, would disable the ability to interact with objects in the room. In this room the user had 2 minutes and 30 seconds to complete the task. When the user was either satisfied with the placement of the furniture, or when the timer had run out, he could proceed to the final phase, by pressing the continue button.

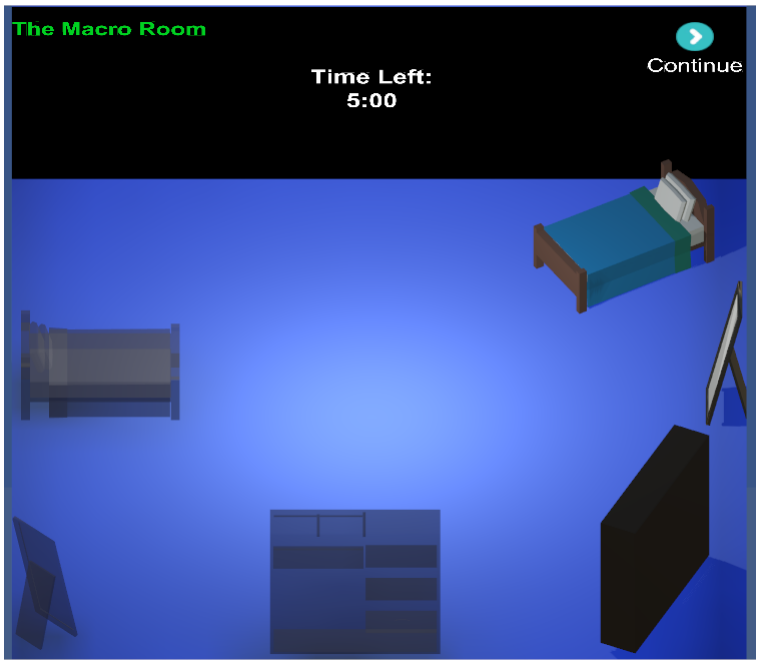


Figure 116: Orthographic view of the macro room.

The final phase is the "Micro room" (see figure 117). The task in this room is also about placing furniture in predefined positions, but the objects are smaller and has to be placed on top of each other. As in the macro room, the time-boxing feature is used again to limit the time spent fiddling with accuracies.

Figure 117: Orthographic view of the micro room.

E.2 Conducting the Experiment

We gathered people to participate in the experiment, and in order to remove bias that might stem from trying one platform before the other, we did an AB/BA split of the users as seen in figure 118.

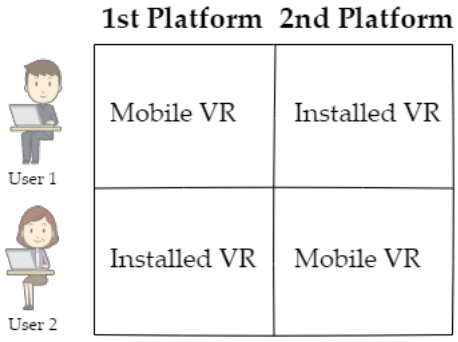


Figure 118: AB/BA split of the participants of the object manipulation experiment

Before the users started the experiment, we had them fill out the background information

in the questionaire (see appendix I.5). After completing the tasks on a platform, we had them fill the questions for that platform before continuing the experiment on the next platform. This was done to limit the effect of experiencing one platform before another.

E.3 Experiment Results

We ran 22 people through the object manipulation experiment. 2 of those people did not finish the MVR part of the experiment; one because the person got nauseous, and the other gave up because the person was left handed, which was not supported. We have decided to discard the left-handed person, as that person had a wildly different experience than the rest. We have kept the data from the person who got nauseous, as this is a big part of the user experience.

We noticed that the data had some extreme outliers, and from looking in the video recordings of the experiment (see appendix I.9), we could see that this was due to users accidentally knocking furniture away. These measurements are misrepresentative of the experiment, and will not be included in the results. See appendix I.8 and I.12 for a full overview of the calculations used for our analysis.

As we only have a data pool of 22 people, we know that our data set is not statistically significant, but we will still use it as an indication of user preference for object manipulation on installed- and mobile VR.

E.3.1 Questionaire Results

This section will present the results from the questionaire, and the raw data can be found in appendix I.6.

The users in the object manipulation experiment was distributed in the age range from 22 to 34, as seen in figure 119. 76.2% of the users were male, and 23.8% was female, as seen in figure 120.

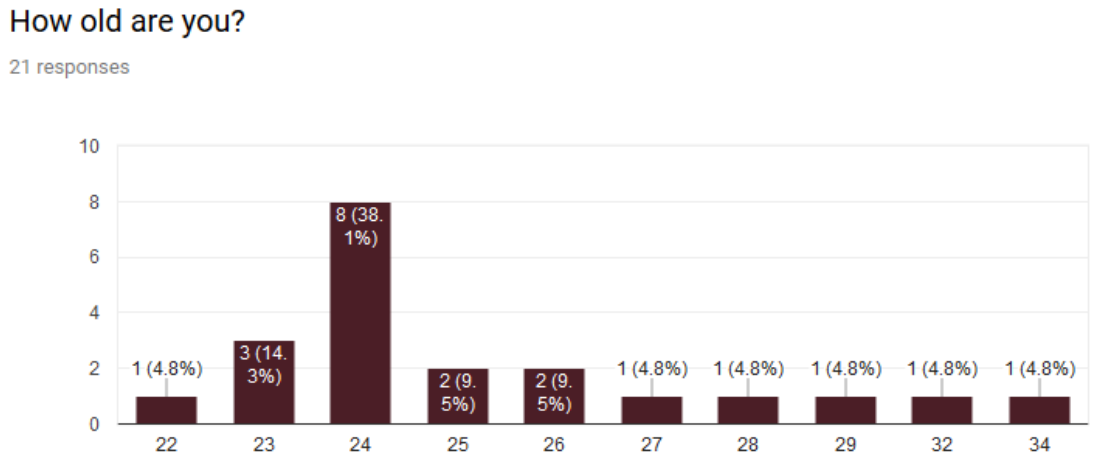


Figure 119: Distribution of age of the subjects in the object manipulation experiment

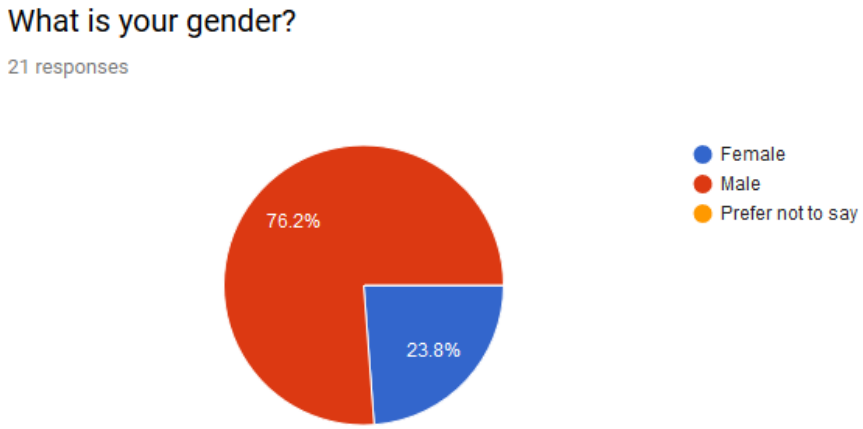


Figure 120: Distribution of gender of the subjects in the object manipulation experiment

90.5% of the subjects were students, most of them from IT-Engineering, while a few were from healthcare technology engineering, and a single from electrical engineering as seen in figure 122.

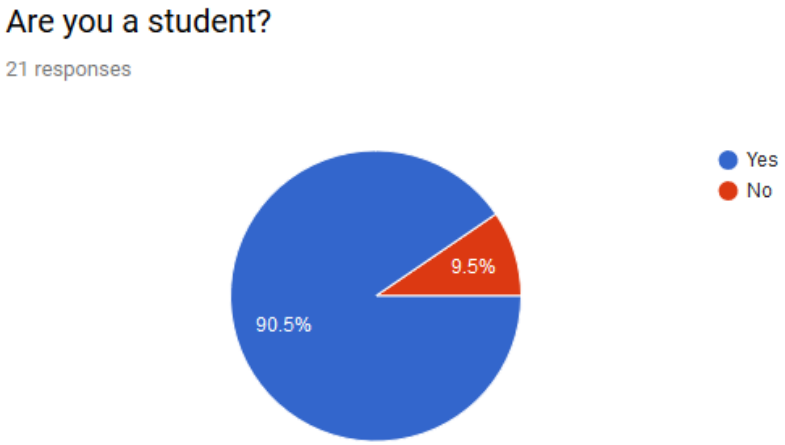


Figure 121: Percentage of students among the subjects in the object manipulation experiment

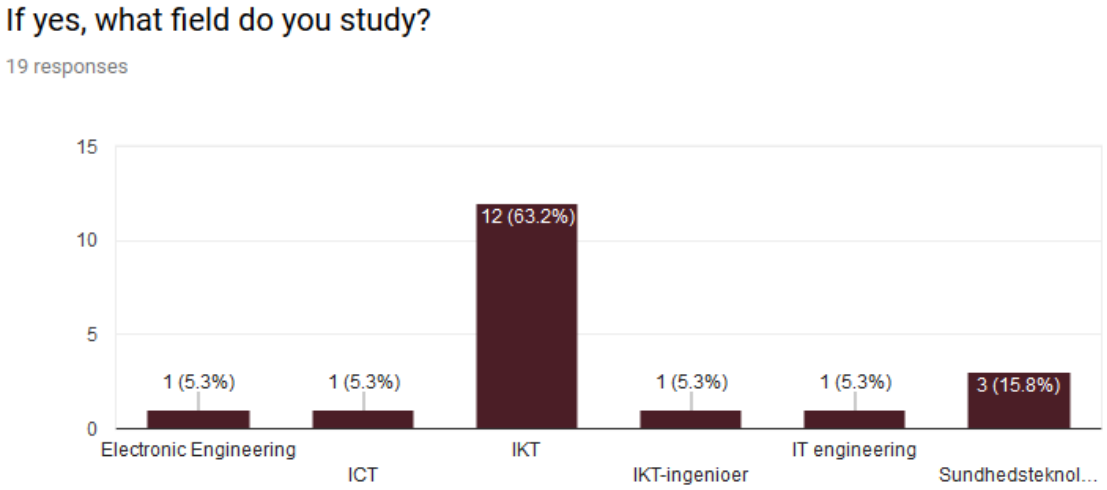


Figure 122: Distribution of students among fields

A majority of the subjects rated themselves at extremely experienced in interacting with 3D environments (figure 123). Of the subjects, 81% had tried VR before participating in the experiment, but of those 81% most identified as not very experienced as seen in figure 124 and 125.

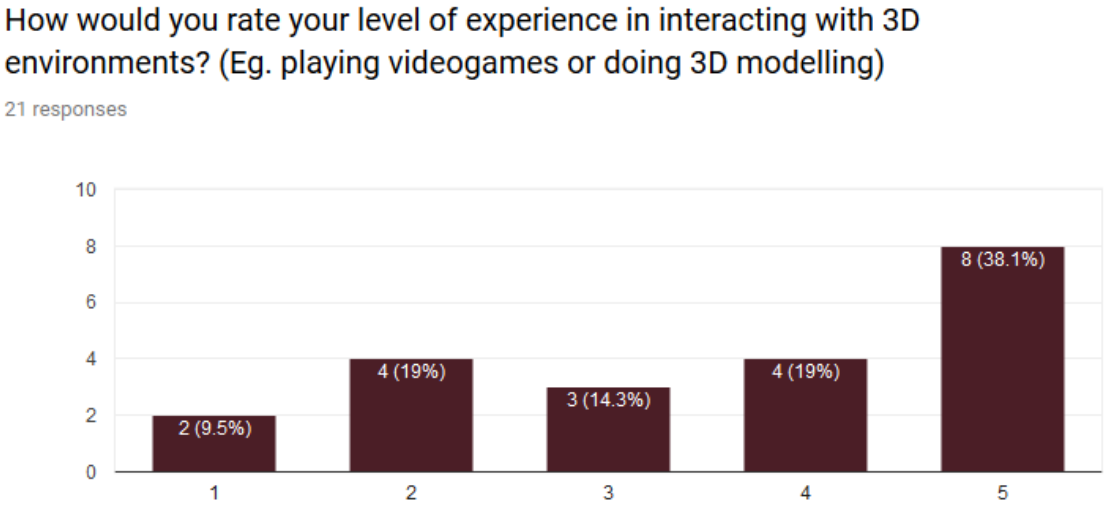


Figure 123: The users' self rating of experience with 3D environments. 1 being Not At All Experienced and 5 being Extremely Experienced

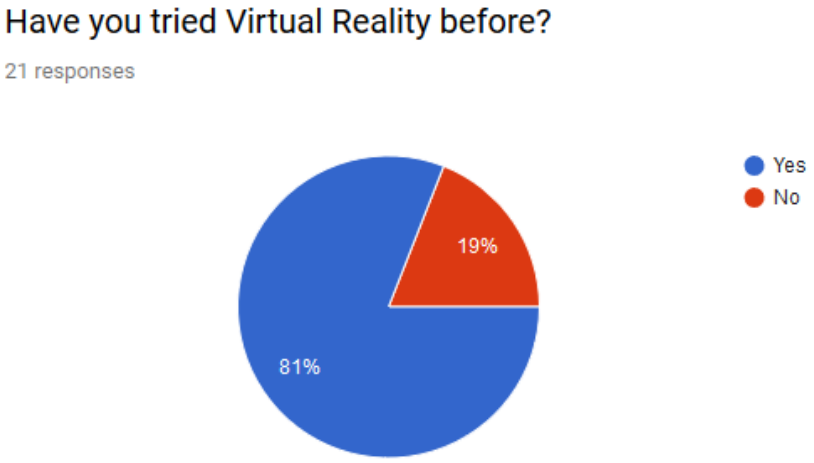


Figure 124: Percentage of subjects who have tried VR before

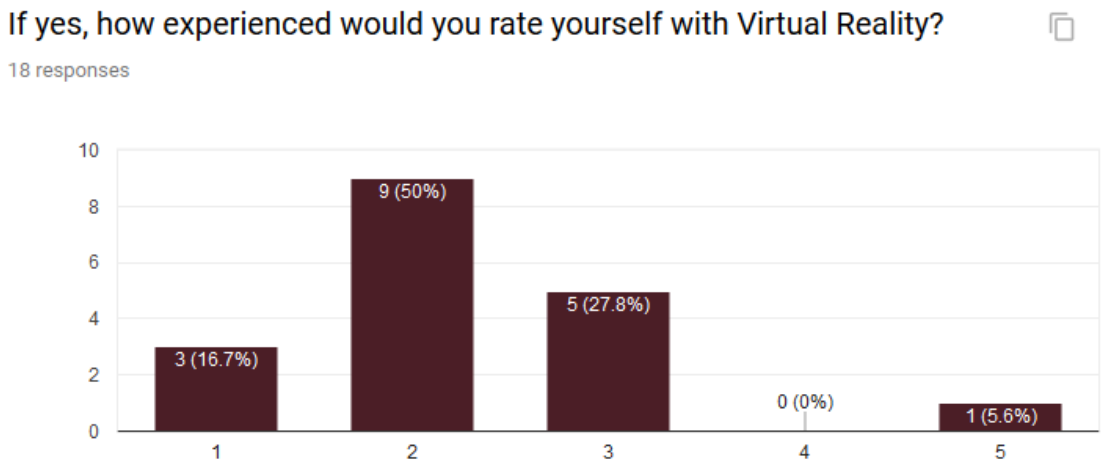


Figure 125: The users' self rating of experience with VR. 1 being Not At All Experienced and 5 being Extremely Experienced

When using MVR, people somewhat agreed that it was easy to pick up objects, both near and far away (figures 126 and 127).

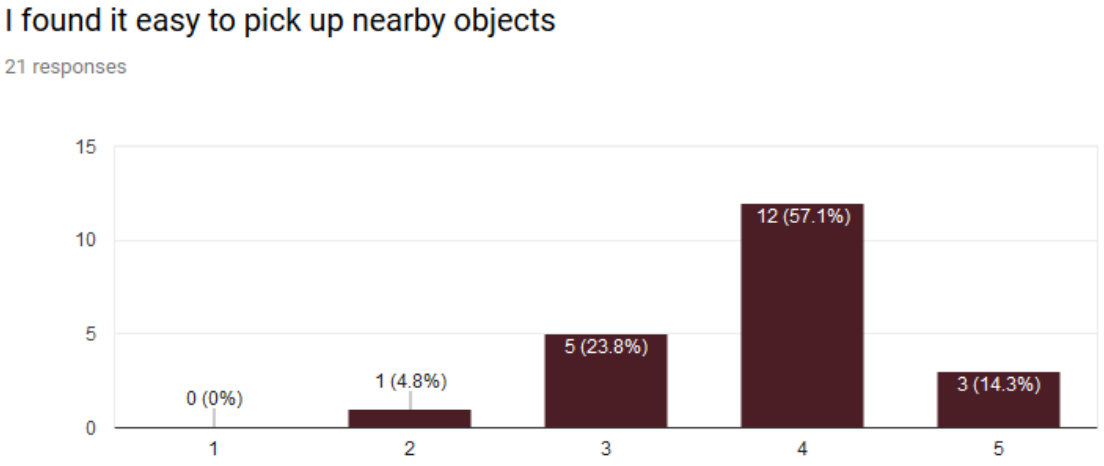


Figure 126: How users agree with the statement "I found it easy to pick up nearby objects". 1 being I Strongly Disagree and 5 being I Strongly Agree

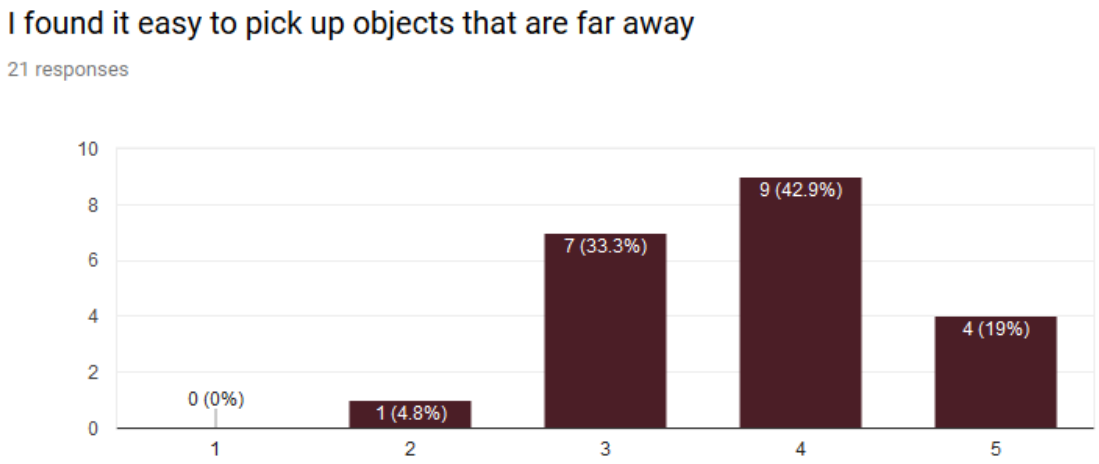


Figure 127: How users agree with the statement "I found it easy to pick up objects that are far away". 1 being I Strongly Disagree and 5 being I Strongly Agree

The MVR users were mostly neutral when asked if they felt it was easy to place large objects as seen in figure 128. Most disagreed with the statement, that it was easy to place tiny objects with the accuracy they wanted, as seen in figure 129

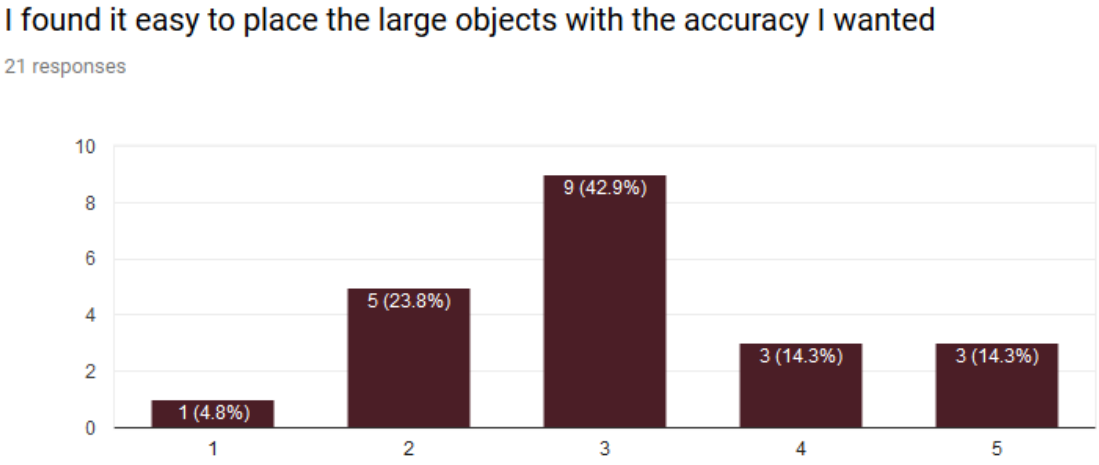


Figure 128: How users agree with the statement "I found it easy to place the large objects with the accuracy I wanted". 1 being I Strongly Disagree and 5 being I Strongly Agree

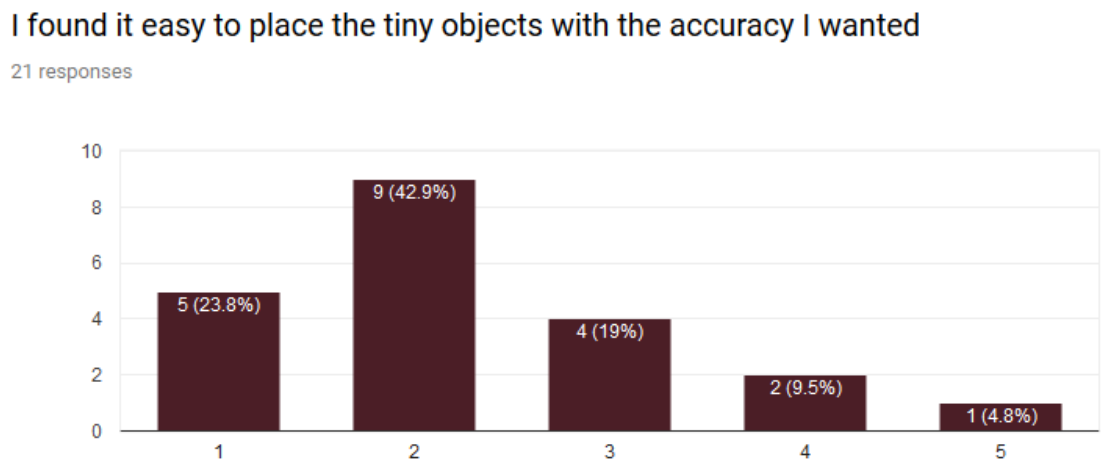


Figure 129: How users agree with the statement "I found it easy to place the tiny objects with the accuracy I wanted". 1 being I Strongly Disagree and 5 being I Strongly Agree

The MVR subjects were very split when asked if they agreed that it was easy to place objects on top of each other, as seen in figure 130.

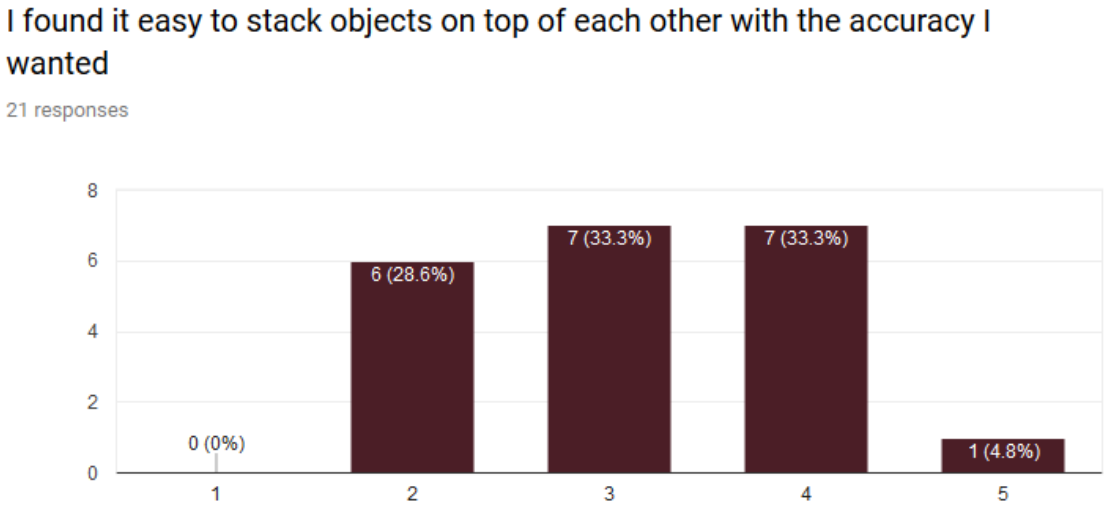


Figure 130: How users agree with the statement "I found it easy to place objects on top of each other with the accuracy I wanted". 1 being I Strongly Disagree and 5 being I Strongly Agree

When asked to comment on object manipulation in MVR, the users had two common comments.

1. There was a mismatch between the sensitivity when rotating and swiping objects

closer/away, and sometimes they would rotate when they wanted to move it, and vice versa, making hard to achieve the desired accuracy.

2. They felt it less intuitive, as they could not move around physically.

This was highlighted in the following quotes from the questionaire:

> *"Really difficult to grab small items (like the mouse). Mismatch between rotation speed and moving (sliding thumb) speed. Difficult to hit properly when sliding thumb to move objects."* - User on MVR object manipulation.

> *"It was less intuitive because I could not move around. But moving objects around with the laser pointer was seemingly faster. One downside with the laser pointer here was that it was very hard to pick up small objects far away, like the mouse and the keyboard, from the other side of the room (the starting position)."* - User on MVR object manipulation.

When using IVR, people mostly agreed that it was easy to pick up objects, both near and far away (figures 131 and 132).

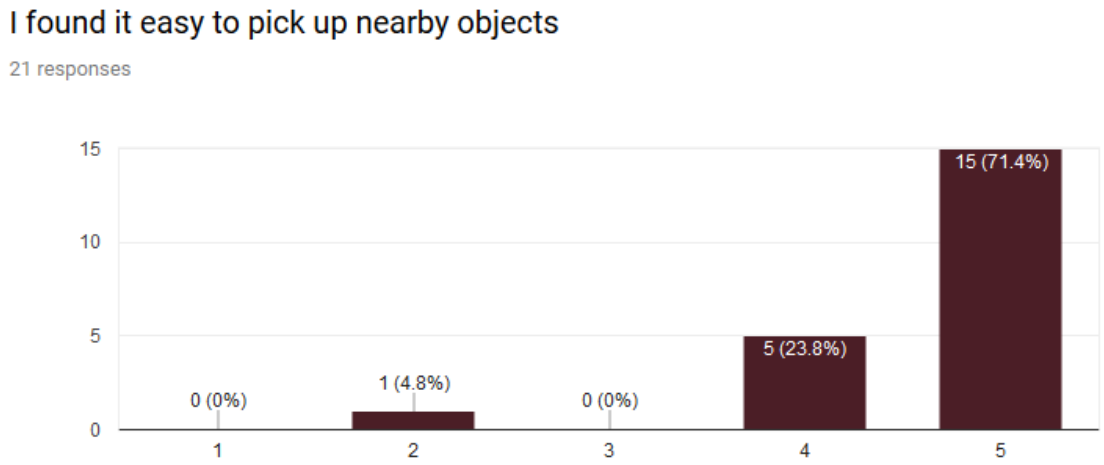


Figure 131: How users agree with the statement "I found it easy to pick up nearby objects". 1 being I Strongly Disagree and 5 being I Strongly Agree

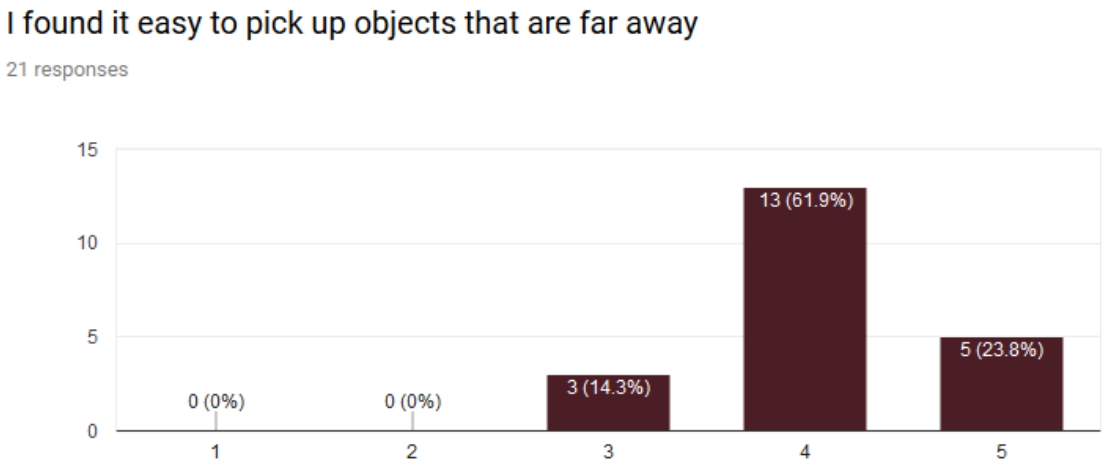


Figure 132: How users agree with the statement "I found it easy to pick up objects that are far away". 1 being I Strongly Disagree and 5 being I Strongly Agree

The IVR users generally agreed that it was easy to place large and tiny objects with the accuracy that they wanted, as seen in figures 133 and 134

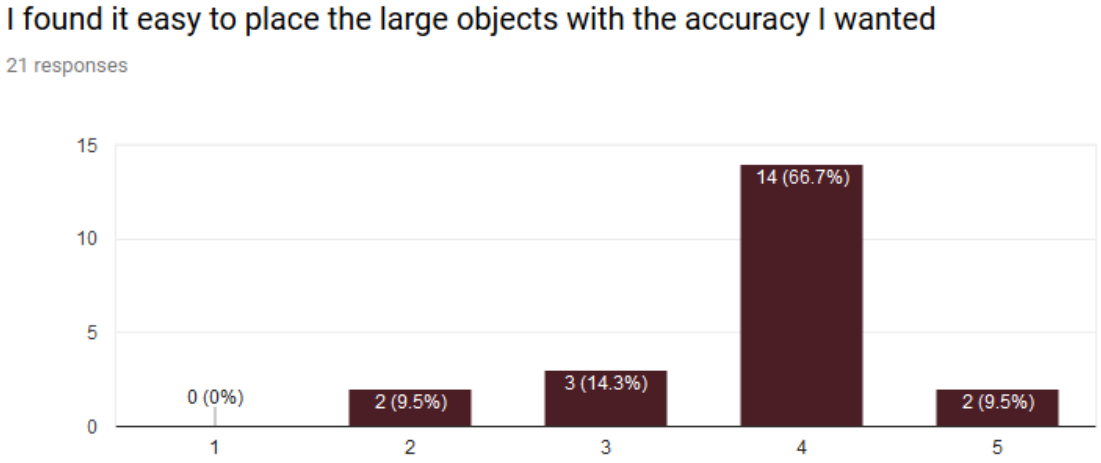


Figure 133: How users agree with the statement "I found it easy to place the large objects with the accuracy I wanted". 1 being I Strongly Disagree and 5 being I Strongly Agree

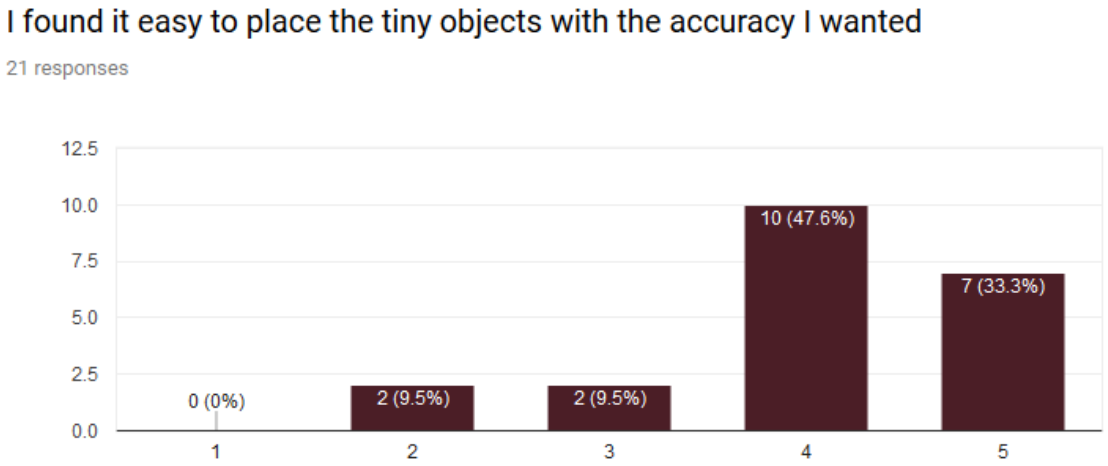


Figure 134: How users agree with the statement "I found it easy to place the tiny objects with the accuracy I wanted". 1 being I Strongly Disagree and 5 being I Strongly Agree

The IVR users also mostly agreed that they found it easy to place objects on top of each other with the wanted accuracy, as seen in figure 135.

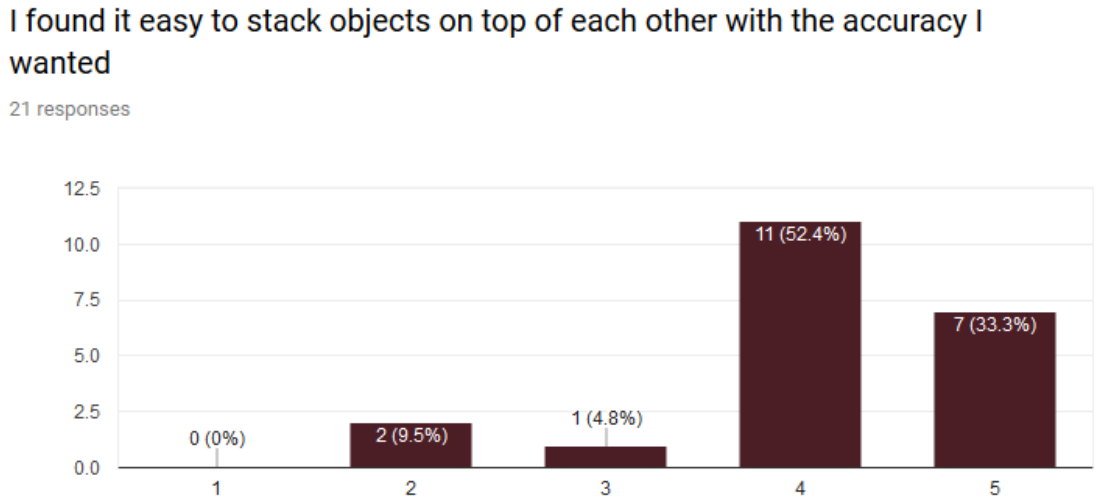


Figure 135: How users agree with the statement "I found it easy to place objects on top of each other with the accuracy I wanted". 1 being I Strongly Disagree and 5 being I Strongly Agree

When asked to comment on object manipulation in installed VR, two topics came up from multiple users:

1. The roomscale functionality was intuitive and made for a better experience when

moving objects.

2. The ability to hold an object in the hands and move/teleport to the place you wanted to place it made for a better experience.

This is highlighted by the following quote from the questionaire:

> *"It was a nice experience, especially because i could move around. It gave me a sense of freedom in the game. The controllers were very good because you could actually see them ingame."* - User on IVR object manipulation.

Lastly, when asked which platform the users preferred for object manipulation, 95.2% answered that they preferred the HTC VIVE (IVR), with 4.8% had no preference.

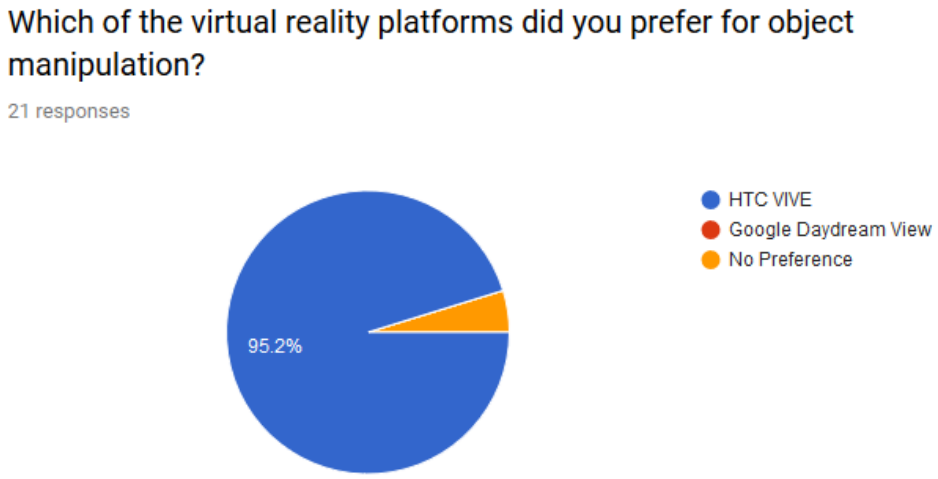


Figure 136: Distribution of preferred platform for object manipulation.

When asked to describe the greatest difference between the platform, users almost unanimously commented on the intuitiveness of the roomscale functionality.

> *"HTC Vive felt very natural to me, it was easy and intuitive. It was fairly easy to place object with high accuracy, especially because you could use your hands to pick up and place items. The google daydream was not a good experience in my opinion, as it was frustrating to place objects with high accuracy. "* - User on platform differences.

E.3.2 Measurement results

During the experiment we gathered measurements about accuracy in distance and orientation. The raw data for the experiment can be found in appendix I.7. To simplify

the readings, we have gathered all averages (total, and per room) in an excel file, which can be found in appendix I.8.

This section will highlight the results of the accuracy measurements. Table 19 shows the total averages for distance, orientation and completion time for the experiment.

Table 19 shows, that installed VR had an overall higher average orientation accuracy, as well as a higher distance accuracy, but that mobile VR had a faster completion time.

Platform	Total Average Orientation Accuracy (%)	Total Average Distance Accuracy (meters)	Total Average Completion Time (seconds)
IVR	93.02	0.424	273.854
MVR	90.07	0.621	269.748

Table 19: Total averages for MVR and IVR object manipulation

Table 20 shows the averages in distance, orientation, and completion time for the Macro room. From this, we see that installed VR has a higher orientation accuracy, whereas mobile VR was more accurate with distance, and has a faster completion time.

Platform	Macroroom Average Orientation Accuracy (%)	Macroroom Average Distance Accuracy (meters)	Macroroom Average Completion Time (seconds)
IVR	93.77	0.867	132.907
MVR	92.64	0.671	103.900

Table 20: Averages for the Macro room

Table 21 shows the averages for orientational and distance accuracy as well as completion time for the Micro room. From this, we see that installed VR has a higher orientation- and distance accuracy, as well as a faster completion time.

Platform	Microroom Average Orientation Accuracy (%)	Microroom Average Distance Accuracy (meters)	Microroom Average Completion Time (seconds)
IVR	92.64	0.203	140.935
MVR	88.79	0.597	165.845

Table 21: Averages for the Micro room

E.4 Measurement Model

For the Object Manipulation Experiment, a more extensive measurement model was used compared to the Navigation Experiment. This was for two reasons: getting interesting

data which can be used for further detailed analysis related to our problem statement, and two, to demonstrate the capabilities of the *Firebase Measurement Repository System* of the *The Experiment Framework*.

The classes comprising this measurement model can be seen in figure 137.

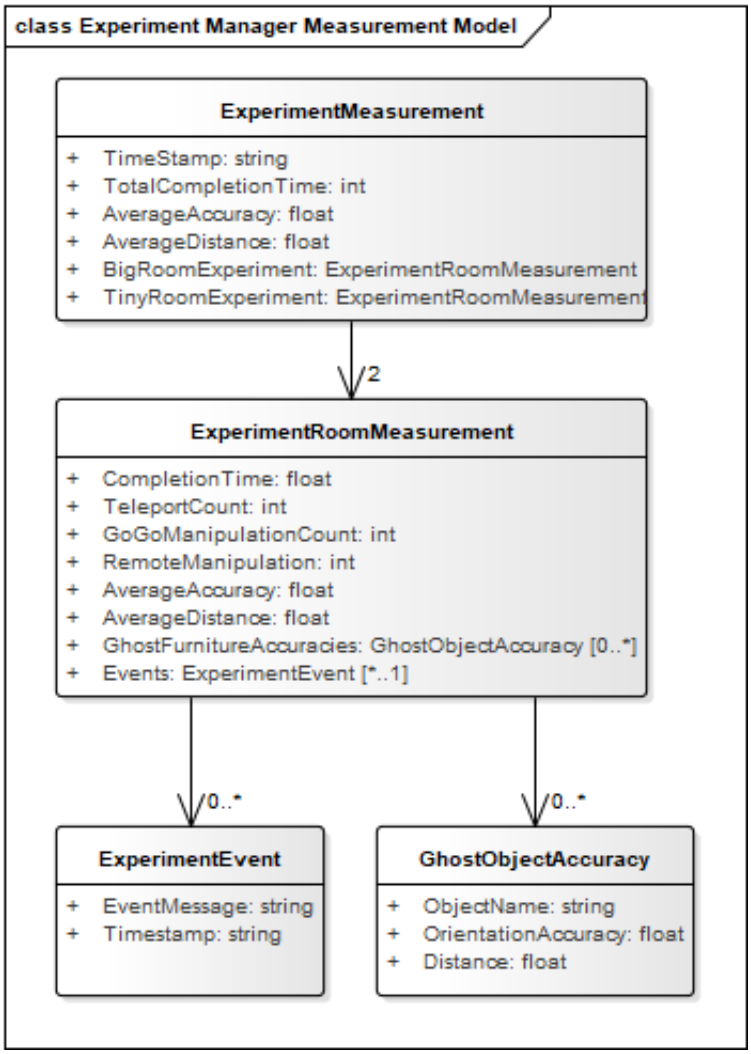


Figure 137: The classes making up the measurement ultimately saved to Firebase.

The class *ExperimentMeasurement* represents the measurement of a single run-through of the experiment on a single platform. Table 22 contains descriptions of each property.

Property	Description
TimeStamp	The TimeStamp of when the measurement was recorded.
TotalCompletionTime	The total duration, in seconds, that it took for a user to complete both rooms of the Object Manipulation Experiment.
AverageAccuracy	The total average orientation accuracy of all placed furnitures, in percentage, throughout both rooms of the Object Manipulation Experiment.
AverageDistance	The total average position accuracy of all placed furnitures, in Unity units, throughout both rooms of the Object Manipulation Experiment.
BigRoomExperiment	The measurement model of the Big Room of the Object Manipulation Experiment. This contains measurements related only to the Big Room.
TinyRoomExperiment	The measurement model of the Tiny Room of the Object Manipulation Experiment. This contains measurements related only to the Tiny Room.

Table 22: Description of the properties belonging to the measurement *ExperimentMeasurement*.

The class *ExperimentRoomMeasurement* represents measurements localized to a single room of the Object Manipulation Experiment, such as Big Room or Tiny Room. Table 23 contains descriptions of each property.

Property	Description
CompletionTime	The total amount of time, in seconds, it took for a user to complete the room by continuing to the next room.
TeleportCount	The total amount of times the user teleported in the room.
GoGoManipulationCount	The total amount of times the user grabbed an object with GoGo-Manipulation.
RemoteManipulation	The total amount of times the user grabbed an object with remote object manipulation.
AverageAccuracy	The average orientation accuracy of all placed furniture existing within the single room.
AverageDistance	The average position accuracy of all placed furniture existing within the single room.
GhostFurnitureAccuracies	A collection of ghost furniture accuracies. These can be used to see detailed accuracy information of each specific furniture placed in the room.
Events	A collection of experiment events. These can be used to see what sort of actions - such as teleportation and object manipulation - the user performed at specific times during the experiment.

Table 23: Description of the properties belonging to the measurement class *ExperimentRoomMeasurement*.

The class *ExperimentEvent* represents events which occurs during an experiment. Experiment Events consists of a simple message descripting the event which took place, coupled with a timestamp of when occurred. The properties are described in table 24.

Property	Description
EventMessage	The customized event message of the event which took place.
Timestamp	The time at which the event took place.

Table 24: Description of the properties belogning to the class *ExperimentEvent*.

The class *GhostObjectAccuracy* represents detailed measurements of each furniture placement within a room, relating to its ghost furniture. This allows you to have detailed insight for each furniture specifically. Table 25 contains description of each property.

Property	Description
ObjectName	The name of the furniture placed.
OrientationAccuracy	The orientation accuracy of the placed furniture.
Distance	The position accuracy of the placed furniture.

Table 25: Description of the properties belonging to the class *GhostObjectAccuracy*.

E.5 Live Experiment Data Smartphone Application

After having conducted the Navigation Experiment, documented in appendix D, we learned that it was difficult to gain an idea of what users were doing on the MVR platform. This is due to the fact that users are wearing a headset with no easy way for any spectators to watch along on a secondary live feed. Furthermore, we discovered that there was no easy way to reset experiments in preparation of the next user in line.

Because of this, we decided to extend the core system of our experiment, described in the system specification of section 4, with an Android smartphone application. This smartphone application is capable of receiving live data from ongoing experiments - such as the user's position within the experiment world space and the user's view direction - as well as communicate events from itself to an ongoing experiment. The idea behind this was to be able to follow along on a simplified 2D map representing the virtual world that the user was experiencing. It should also be possible to reset experiments by button presses within this application. A picture of the final app can be seen in figure 138 with indicators of the various elements maping up the app.

See appendix I.10 for a file path to a video demonstration of the app.

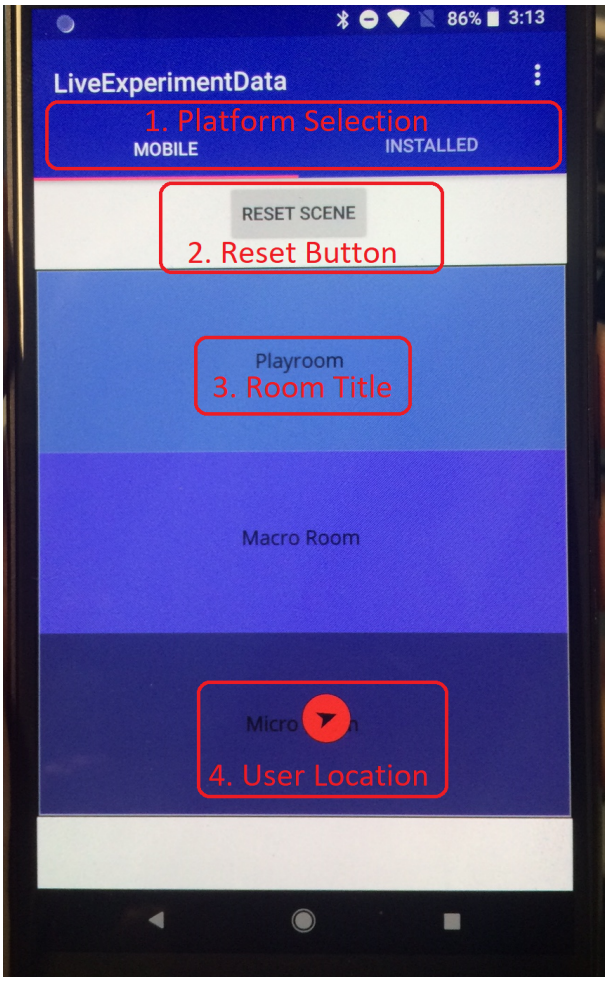


Figure 138: The Live Experiment Data Smartphone Application running on a Google Pixel.

On figure 138, the elements are:

1. Platform Selection - Through a tabbed menu it is possible to switch between monitoring an experiment currently running on the mobile- or installed- virtual reality platform.
2. Reset Button - The reset button will reset the currently running experient to the beginning, depending on the platform tab selected (mobile or installed).
3. Room Title - The room title indicates the room of the colored area. Each room have different colors to be able to distinguish between them.
4. User Location - The user location is represented by the red circle with a black arrow inside it. The user's view direction within the world is represented by the direction

of the black arrow.

Figure 139 presents the class diagram of the most relevant classes of the application.

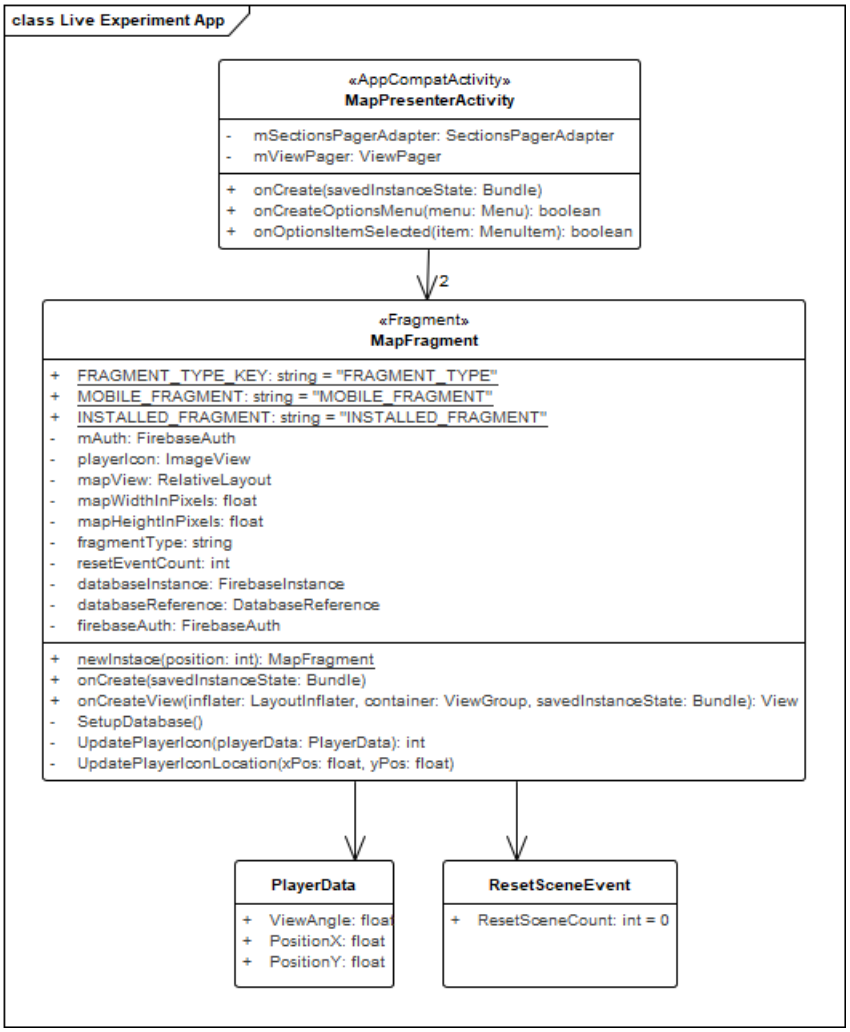


Figure 139: The Live Experiment Data Smartphone Application running on a Google Pixel.

MapPresenterActivity is the only Activity of the app. This is the Activity which is seen in figure 138. The Activity in itself holds none of the actual GUI elements seen in figure 138. Instead, it functions as a container for *MapFragments*, specifically 2 of these. One for each platform.

MapFragment is a Fragment containing the actual view of GUI elements. It contains the *Reset Scene* button, the 2D map of the Object Manipulation Experiment, in which *Room Titles* and the *User Location* is displayed.

Table 26 describes selected, most relevant methods of the *MapFragment* class.

Method	Description
SetupDatabase	This method sets up Firebase event listeners to relevant nodes of the app. This involves listening to a Firebase node which the Object Manipulation experiments will push live user data to, so that the app can update the location on the 2D map.
UpdatePlayerIcon	This method unpacks the PlayerData received from Firebase, and calculates a mapped coordinate from the received Unity units to the 2D map of the app.
UpdatePlayerIconLocation	This method animates the user icon on the 2D map to move to a newly received pair of mapped coordinates.

Table 26: Descriptions of selected, most relevant methods of the *MapFragment* class.

PlayerData is a data model of the relevant player data received by the app from ongoing Object Manipulation experiments. A description of the properties can be scene in table 27.

Property	Description
ViewAngle	This is the direction that the player is looking in degrees.
PositionX	The X-axis position of the player in the virtual environment of the Object Manipulation experiment. This is in Unity units, and will be used to position the user icon on the 2D map.
PositionY	The Y-axis position of the player in the virtual environment of the Object Manipulation experiment. This is in Unity units, and will be used to position the user icon on the 2D map.

Table 27: Properties of the *PlayerData* model class.

ResetSceneEvent is the data model for a reset scene event which the app will publish to a Firebase node which ongoing Object Manipulation experiments are listening in on. It contains a single property, *ResetSceneCount*, which is simply a count on the number of times the scene has been reset since the start of the Live Experiment Data smartphone application.

Figure 140 shows the two basic data flows between the app and the Object Manipulation Experiment. That is, the flow in which an ongoing Object Manipulation Experiment updates the app with live player data, and the flow where a user resets the experiment from the app.

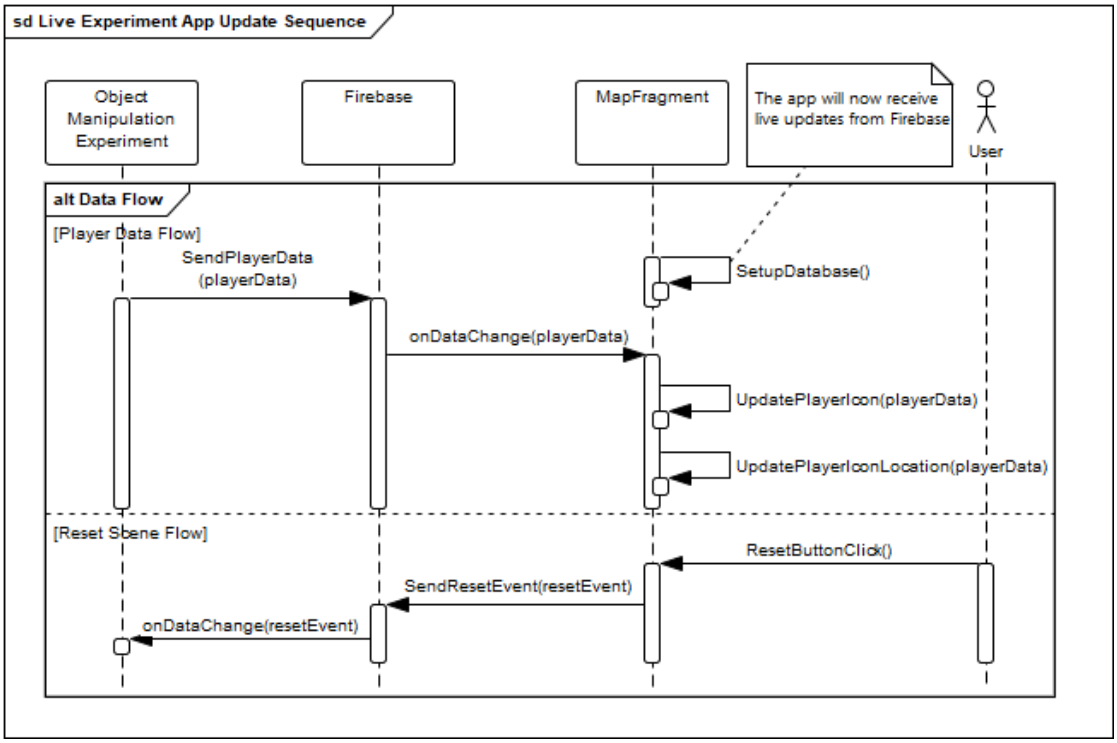


Figure 140: The ExperimentManager class diagram.

E.6 UI

The rooms of the Object Manipulation Experiment consists of various different UI elements. Each room consists of the same layout, and the same UI elements. An example from the Macro Room is given in figure 141.

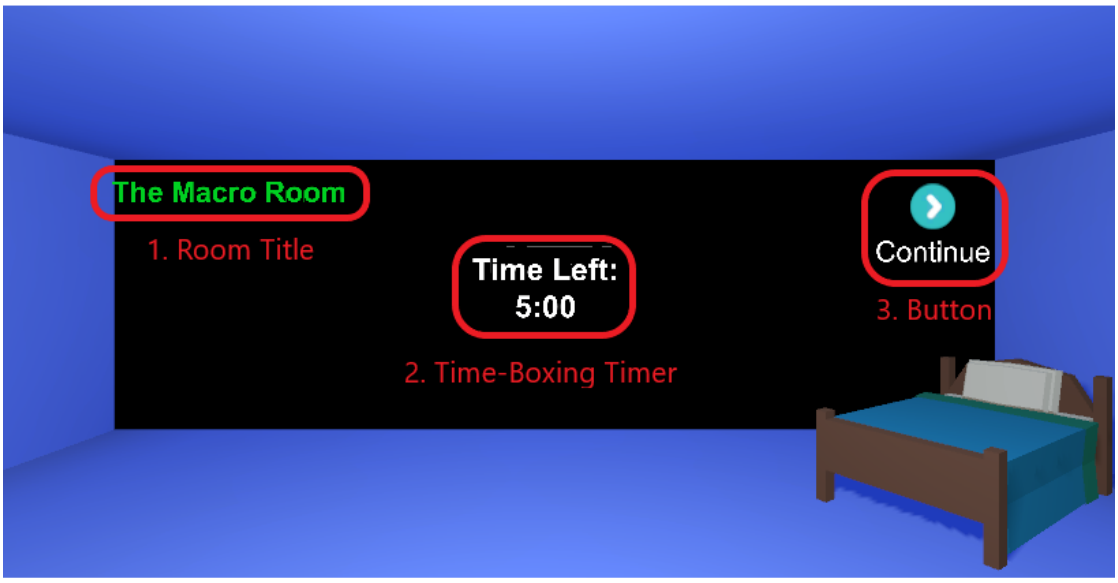


Figure 141: The ExperimentManager class diagram.

The elements, as taken from figure 141, are:

1. Room Title - This is a static text communicating the name of the current room to the user. This way it is always possible to know which room you are currently in.
2. Time-Boxing Timer - This timer is continuously updated with the time left in the room. It makes it possible for the user to know how much time is left before furniture interaction is disabled.
3. Button - The button makes it possible for the user to perform the next required action in the experiment sequence. The action of the button is context-specific, and is described in section E.7.

The UI is designed as taking up the entirety of a wall of the rooms. The background is black in order to highlight it as being different to the user.

E.6.1 Remote UI Interactable Script

The Button UI elements of the Object Manipulation Room experiment can be clicked in various ways, depending on the platform the experiment is executed on.

On an IVR platform, the button can be clicked either by using GoGo Manipulation or via raycasting. Figure 142 demonstrates a button of the Object Manipulation experiment being selected using racyasting on IVR.

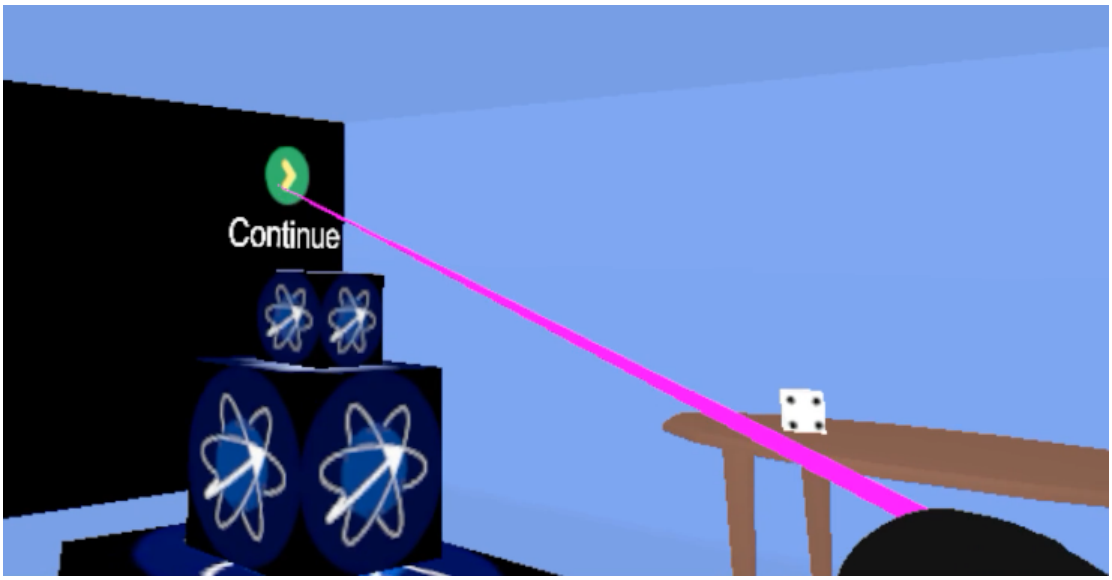


Figure 142: A button of the Object Manipulation Experiment UI being selected using raycasting on the IVR.

Unfortunately, there was no standard implementation for us to use in order to get button selection via raycasting on IVR. On MVR, Google Daydream had a standard implementation for their controller, so the functionality existed out of the box.

Therefore, in order to get raycasting button selection on IVR, we made our own script to handle this, which is the *RemoteUiInteractable* script. Figure 143 shows a class diagram of it.

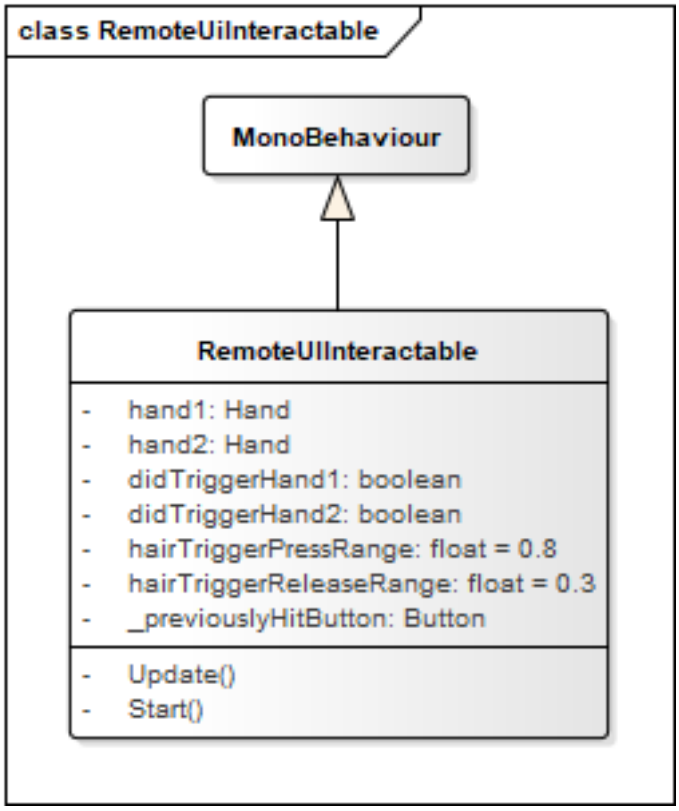


Figure 143: The RemoteUiInteractable script class diagram.

The essence of *RemoteUIInteractable* can be described by looking at a code snippet from the script. This code snippet is presented in listing 1.

```
1  RaycastHit hitHand1;
2  Vector2 hairTriggerRangeHand1 =
       hand1.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_SteamVR_Trigger);
3  if (Physics.Raycast(hand1.transform.position - hand1.transform.forward,
       hand1.transform.forward, out hitHand1, 20f))
4  {
5     if (hitHand1.transform.gameObject.CompareTag("Button"))
6     {
7        _previouslyHitButton = hitHand1.transform.GetComponent<Button>();
8
9        _previouslyHitButton.OnSelect(null);
10
11       if ((hairTriggerRangeHand1.x >= hairTriggerPressRange) && didTriggerHand1
             == false)
12       {
13          didTriggerHand1 = true;
```

```
14            _previouslyHitButton.onClick.Invoke();
15        }
16    }
17 }
```

Listing 1: Code snippet of the RemoteUIInteractable, demonstrating the essential raycast logic.

On line 3 of listing 1, a raycast is performed from, in this example, Hand 1 - Controller 1 - of the user. If this raycast hits a GameObject tagged *Button*, the script invokes the method *OnSelect* of said button. This will make the button highlight visually, so that the user can see the fact that they are pointing the raycast on it.

In line 11, it is checked if the user has also pressed down the trigger of the IVR controller far enough to count as a click. If this is the case, the button will have its *onClick* event invoked, and the button will have been registered as being clicked.

This same logic is carried out for both hands in the scene, so that the user is able to click buttons with both.

To get an overview of how the *RemoteUIInteractable* script fits in with the flow of the Object Manipulation Experiment, figure 149 of section E.7.

E.6.2 Interaction Experiment UI Handler Script

The *InteractionExperimentUiHandler* script has the responsibility of executing logic related to button clicks of the various buttons within the Object Manipulation Experiment.

A class diagram of the script is presented in figure 144.

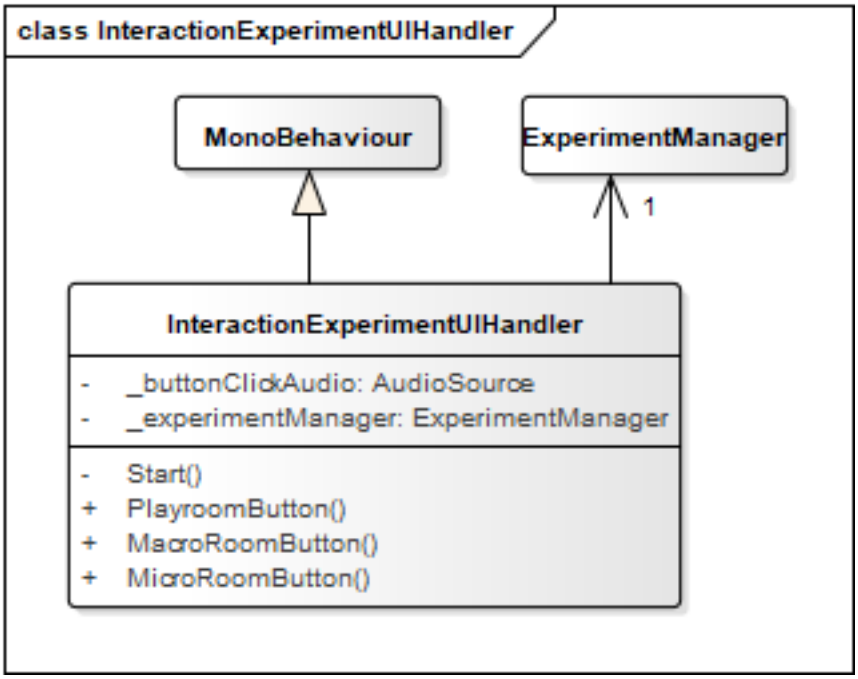


Figure 144: The InteractionExperimentUiHandler class diagram.

Table 28 descripes the properties of the *InteractionExperimentUiHandler*.

Property	Description
_buttonClickAudio	A reference to a Unity AudioSource for the button click sound. This is used by the script to play the button sound whenever a button is clicked.
_experimentManager	A reference to the Experiment Manager of the scene. This is used in order to change the state of the Object Manipulation Experiment based on the button that was pressed. The ExperimentManager is described in further detail in section E.7.

Table 28: The properties of the *InteractionExperimentUIHandler*.

Table 29 describes the most relevant methods of the *InteractionExperimentUIHandler*.

Method	Description
PlayroomButton	The logic to be executed when the Continue button of the Playroom is pressed by the user.
MacroRoomButton	The logic to be executed when the Continue button of the Big Room is pressed by the user.
MicroRoomButton	The logic to be executed when the Finish button of the Tiny Room is pressed by the user.

Table 29: Relevant methods of the *InteractionExperimentUIHandler*.

In order to see how the *InteractionExperimentUIHandler* is used in the flow of the Object Manipulation Experiment, see figure 149 of section E.7.

E.6.3 Playroom Button Enabler

The *Continue* button of the Playroom UI behaves a bit different from the other buttons, as this element is only shown when three orbitlab cubes have been placed correctly upon Ghost Objects in the room. This is achieved through the *PlayroomGhostListener* script.

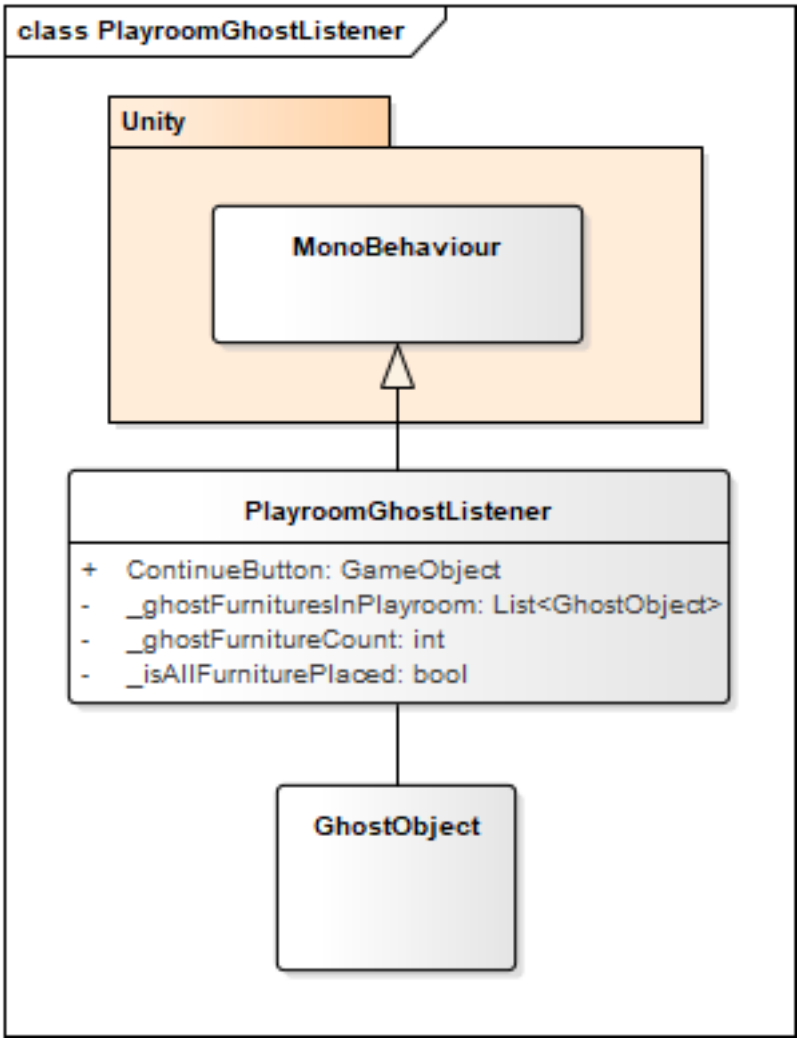


Figure 145: The PlayroomGhostListener class diagram.

The GhostObject class in the diagram on figure 145 is described in appendix F.8.

The *PlayroomGhostListener* script keeps track of ghost objects in the playroom, and as soon as all of them have been placed, it enables the continue button, allowing the user to proceed to the next room.

E.7 Experiment Manager

The experiment manager has the responsibility of orchestrating the entire flow and measurement gathering of an object manipulation experiment run-through.

Figure 146 presents a class diagram of the Experiment Manager script.

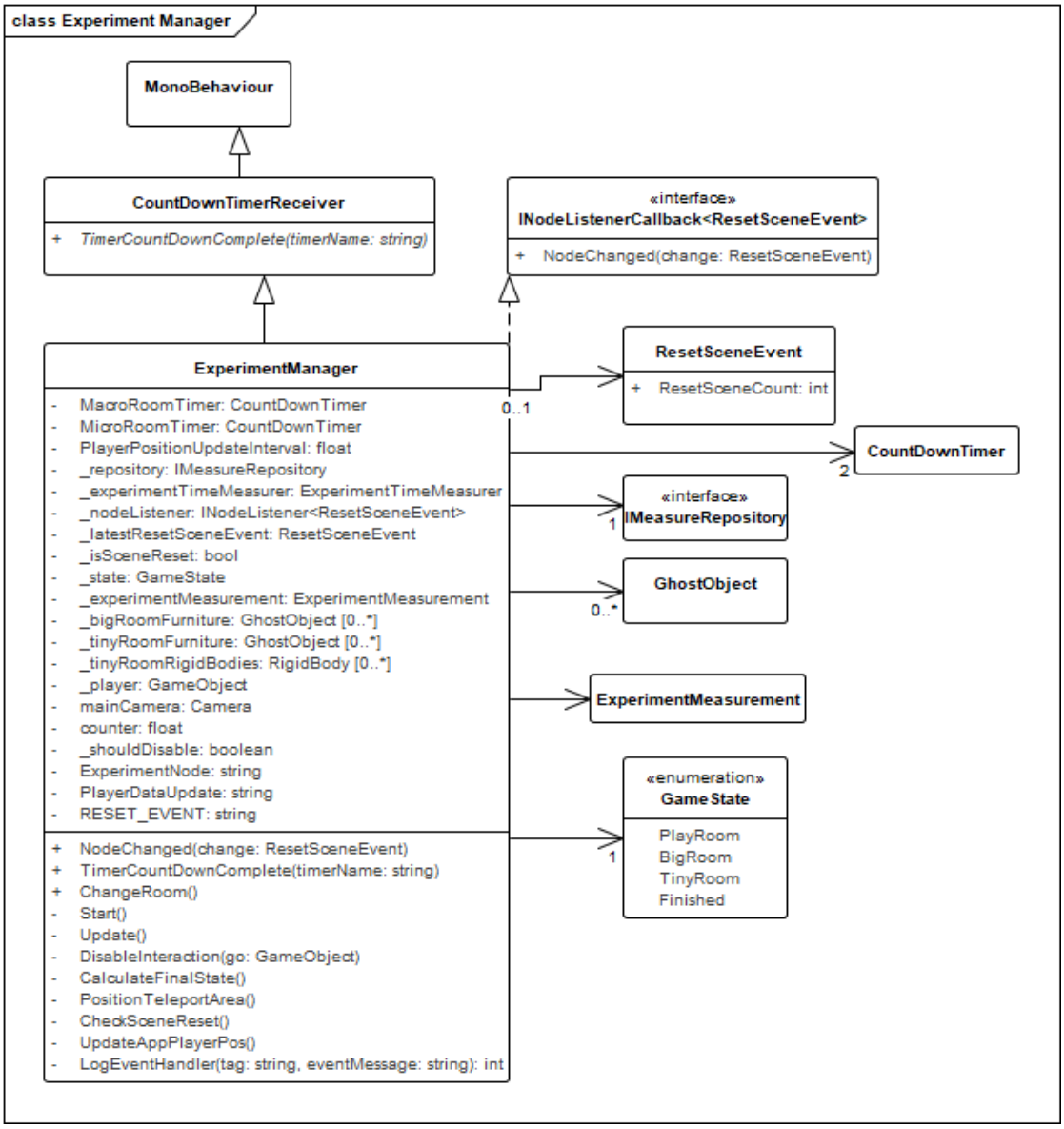


Figure 146: The ExperimentManager class diagram.

Table 30 gives a description of selected methods of the ExperimentManager. These methods have been chosen based on how much they help communicate the purpose of the script and its functionality.

Method	Description
ChangeRoom	ChangeRoom will examine the current Game State, and progress the player to the next relevant room based on that. Thus, if the player is in the Play Room and external scripts call ChangeRoom, the player will be moved to Big Room. ChangeRoom also creates the measurement relevant to the room the player just finished.
TimerCountDownComplete	When ExperimentManager receives the TimerCountDownComplete event call from a timer in the experiment, it will disable furniture selection and movement in the room which the player is currently in. For example, if the player runs out of time in the Big Room, ExperimentManager receives the TimerCountDownComplete event call and disables all further interaction with furniture in that room.
NodeChanged	The NodeChanged event method implementation of the INodeListenerCallback interface is used to detect when the experiment receives a reset event from the smartphone experiment app. If this occurs, ExperimentManager will reset the scene of the experiment.
DisableInteraction	This method is used to disable furniture interaction of a provided furniture GameObject. This method is used in conjuction with TimerCountDownComplete.
CalculateFinalState	This method has the responsibility of calculating the measurement statistics related to the entire experiment run-through of a player. This is stats such as the total completion time in seconds for the entire experiment, as well as total average orientation and position accuracy of all ghost furniture of the experiment.
PositionTeleportArea	This method has the responsibility of moving an associated teleport area to follow the player through the various experiment rooms. The purpose of this is it to disallow players to move outside the designated room.
UpdateAppPlayerPos	This method continuously saves the player's position and view angle to Firebase, so that the smartphone experiment app can receive live updates.
LogEventHandler	This method receives log messages from throughout the experiment and saves them to the measurement of the current experiment run-through.

Table 30: Descriptions of select methods of the ExperimentManager script.

The ExperimentManager script makes use of several tools from the *Experiment Framework*. These include:

- CountDown Timer System. See appendix F.1 for more technical details.
- Firebase Event Listener System. See appendix F.2 for more technical details.
- Firebase Measurement Repository System. See appendix F.4 for more technical details.
- Experiment Time Measurer. See appendix F.6 for more technical details.
- Experiment Logger System. See appendix F.7 for more technical details.
- Ghost Object System. See appendix F.8.

The ExperimentManager's basic update flow for each frame is represented in figure 147.

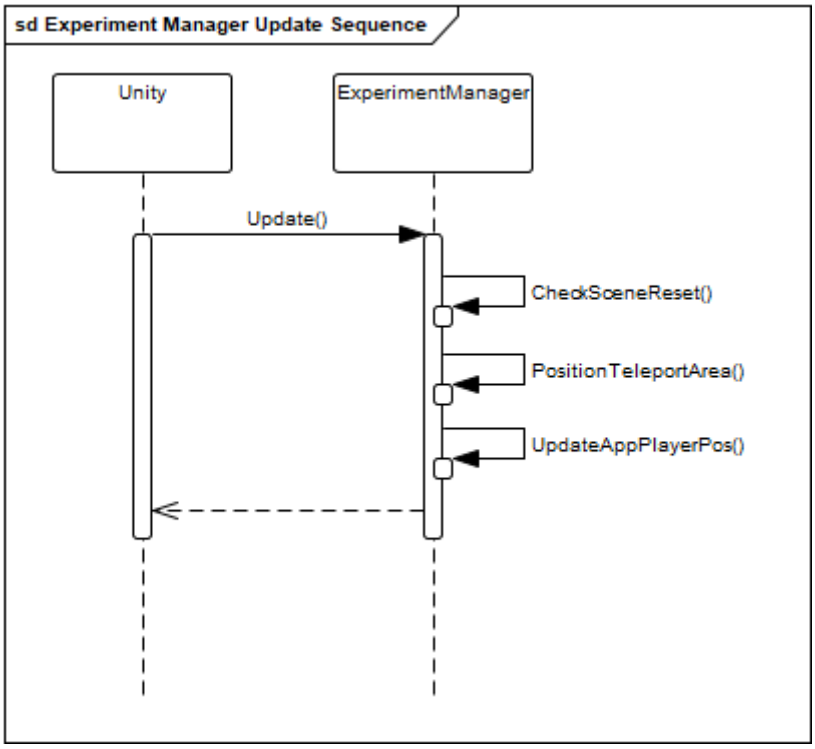


Figure 147: Basic update flow of the ExperimentManager, executed every frame.

In order to comply with the experiment flow, the ExperimentManager has a state machine to indicate the current phase of the experiment. This is represented in the state machine diagram of figure 148.

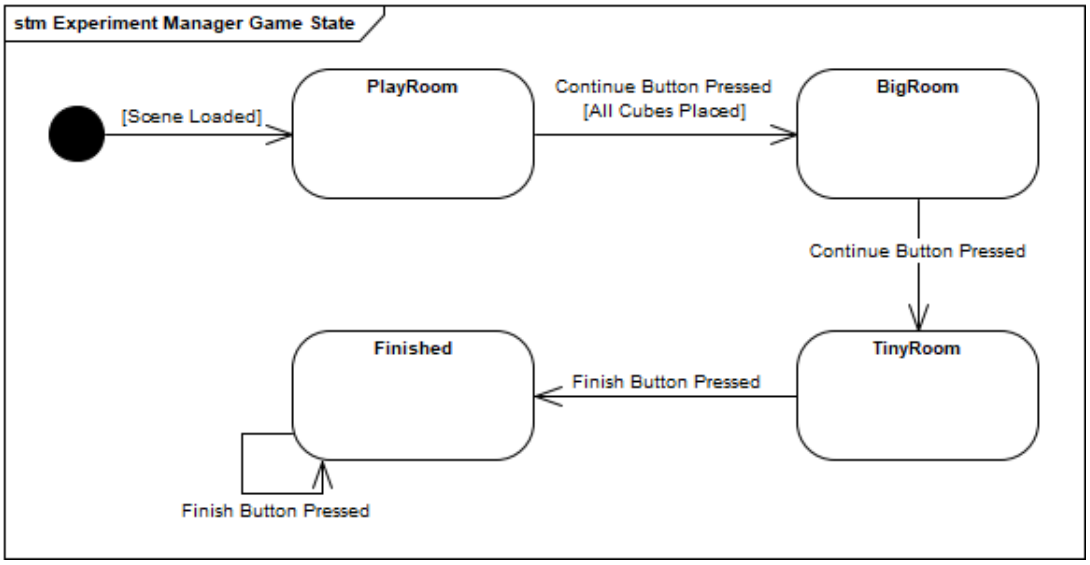


Figure 148: The various possible states of the ExperimentManager as the user progresses through the Object Manipulation Experiment.

Between the transitions of the rooms, the ExperimentManager constructs specific ExperimentRoomMeasurements - as described in figure 137 - for the room just completed, with the exception of the PlayRoom. This is represented by the sequence diagram in figure 149.

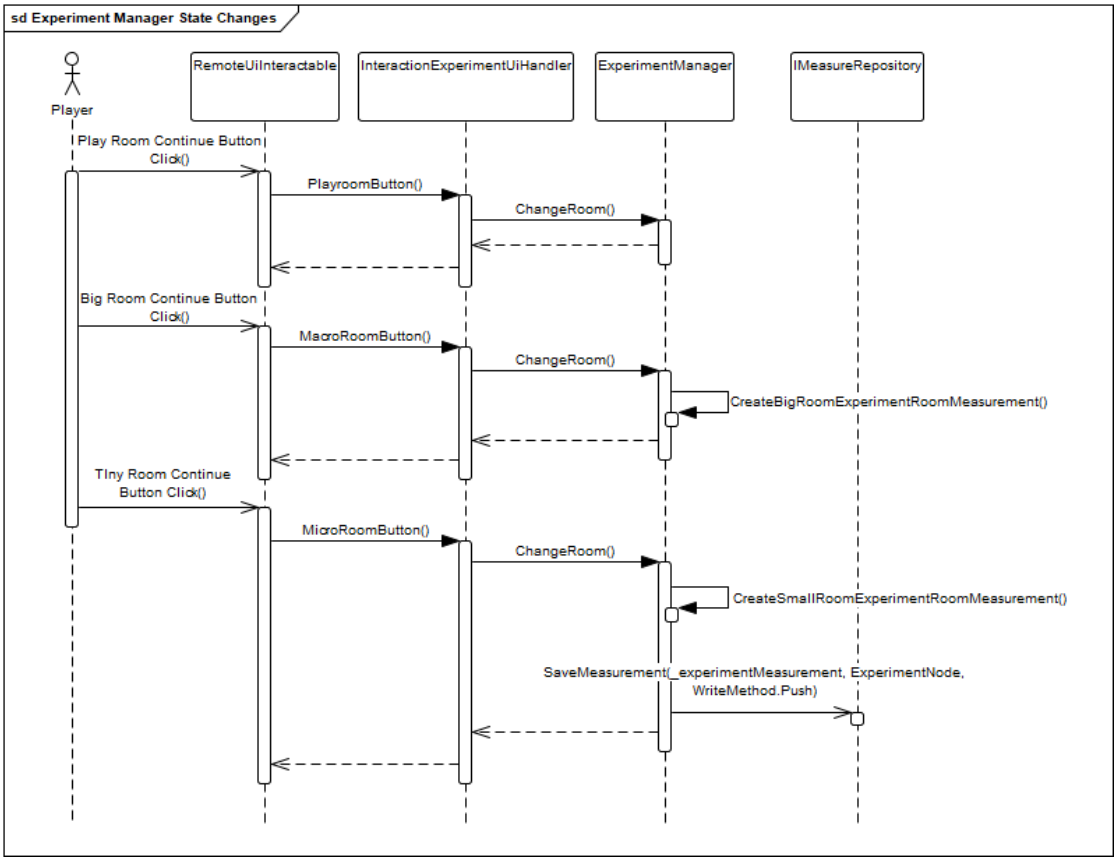


Figure 149: The sequence of events occurring between the user and the experiment system in order to trigger state changes of the ExperimentManager.

E.8 Ghost Object Integration

In the object manipulation experiment we made heavy use of the Ghost Object system described in appendix F.8. Mostly we made use of the functionality to get measurements between two objects, but in the Play Room we were interested in knowing when objects had been placed correctly in order to disable further movement.

To achieve this we implemented two platform specific scripts that hooked into the C# event emitted by the Ghost Object when the associated furniture had been placed. These two scritps, *IVRPlayroomFurniture* and *MVRPlayroomFurniture*, when placed on a furniture associated with a GhostObject, holds the functionality to listen for onObjectPlaced events and disable the platform specific object manipulation scripts. The class diagram for the script can be seen on figure 150.

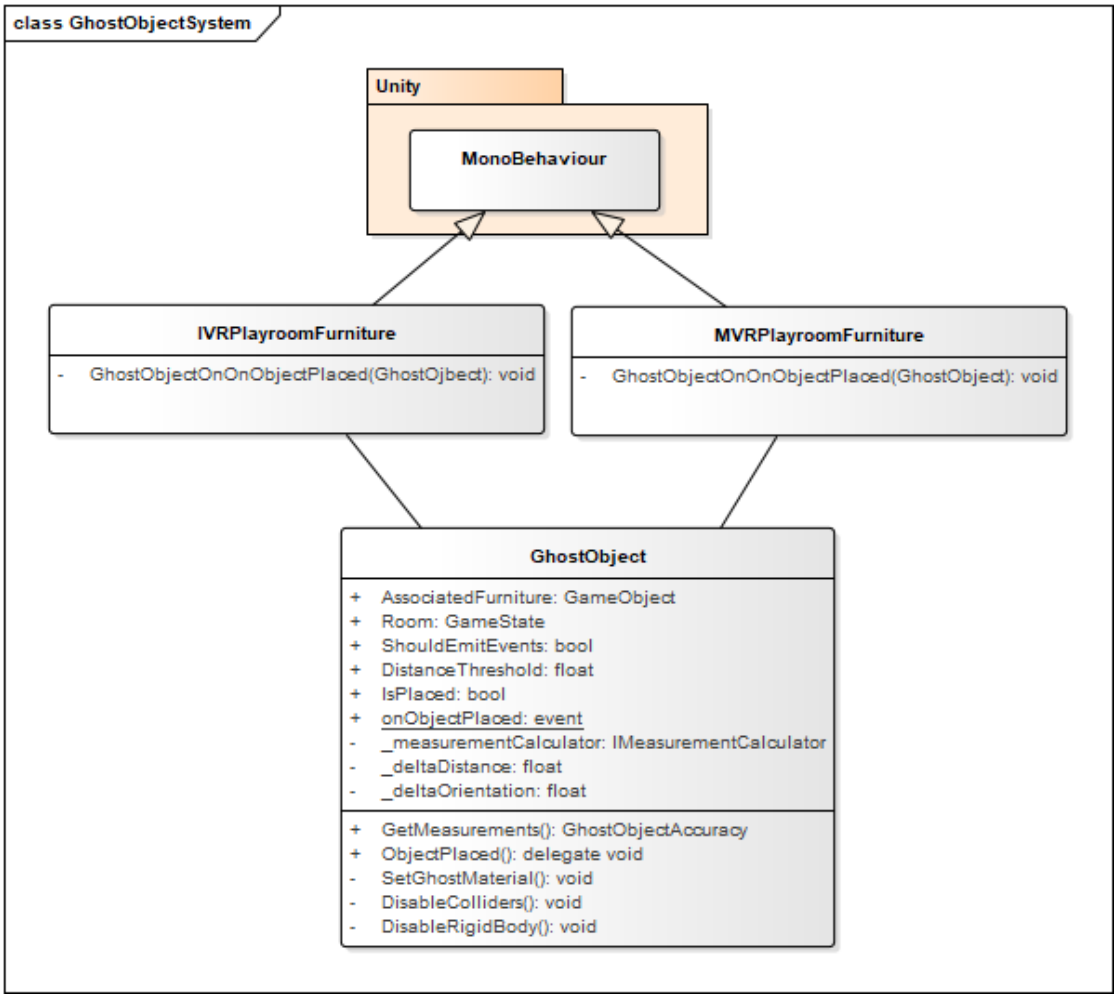


Figure 150: Class diagrams for the IVR/MVRPlayroomFurniture scripts

Both scripts are derived from Unity's MonoBehaviour script, and thus gain access to two important lifecycle methods; *Start()* and *OnDisable()*. It is in these two methods that the scripts hook into the C# events emitted from the ghost object.

```
private void Start ()
{
   GhostObject.onObjectPlaced += GhostObjectOnOnObjectPlaced;
}

private void OnDisable()
{
   GhostObject.onObjectPlaced -= GhostObjectOnOnObjectPlaced;
}
```

Figure 151: Code snippet of the Start and OnDisable methods in MVR/IVRPlayroom-Furniture scripts

When handling the subscribed events (see code snippets on figure 152 and 153) the scripts first check if the associated furniture of the ghost object sending the event, is the same furniture belonging to the IVR/MVRPlayroomFurniture script. If not, the event is ignored.

Next the script checks to see if the associated furniture is being manipulated by the user, as we don't want to disable object manipulation until they release the furniture.

```
private void GhostObjectOnOnObjectPlaced(GhostObject ghostObject)
{
   if(ghostObject.AssociatedFurniture.transform.GetInstanceID() !=
       transform.GetInstanceID())
   {
      Debug.Log($"Got placed event from
          {ghostObject.AssociatedFurniture.name}");
      return;
   }

   if (!ObjectManipulationPointer.IsObjectSelected())
   {
      ghostObject.IsPlaced = true;
      gshostObject.transform.GetComponent<MeshRenderer>().enabled = false;
      ghostObject.AssociatedFurniture.gameObject.GetComponent<Rigidbody>()
         .isKinematic = true;
      Destroy(ghostObject.AssociatedFurniture
         .GetComponent<MoveablePhysicsObject>());
   }
}
```

Figure 152: Code snippet of the event handler of MVRPlayroomFurniture script

```
1  private void GhostObjectOnOnObjectPlaced(GhostObject ghostObject)
2  {
3     if(ghostObject.AssociatedFurniture.transform.GetInstanceID() !=
          transform.GetInstanceID())
4        return;
5
6     if(!ghostObject.AssociatedFurniture.GetComponent<ROMInteractable>()
7           .IsAttached &&
8        !ghostObject.AssociatedFurniture.GetComponent<ROMInteractable>()
9           .IsSelected)
10    {
11       ghostObject.IsPlaced = true;
12       ghostObject.transform.GetComponent<MeshRenderer>().enabled = false;
13       ghostObject.AssociatedFurniture.gameObject.layer =
             LayerMask.NameToLayer("Ignore Raycast");
14       ghostObject.AssociatedFurniture.gameObject.GetComponent<Rigidbody>()
15          .isKinematic = true;
16       Destroy(ghostObject.AssociatedFurniture.GetComponent<Throwable>());
17    }
18 }
```

Figure 153: Code snippet of the event handler of IVRPlayroomFurniture script

Once the scripts receives an event from a ghost object associated with the furniture the MVR/IVRPlayroomFurniture script is on, and the user is not currently manipulating the furniture, the platform specific scripts on the furniture is disabled.

Appendix F The Experiment Toolbox

During the development of the object manipulation experiment, we discovered that many of the functionalities that we needed could be generalized to a plug-and-play Unity toolbox, which exposes easy to use functionality for VR experiments developed with Unity.

This section documents this toolbox in technical detail, and also explains how to use each component with the use of short step-by-step guides.

F.1 CountDown Timer System

The CountDown Timer System provides a countdown timer mechanism, which can be used for UIs in experiments. These timers are practical for time-boxing tasks. Figure 154 shows a screenshot of one of the timers used in the object manipulation experiment.

Figure 154: The CountDown Timer as used throughout the various rooms of the object manipulation experiment.

The countdown timer can be easily configured in the inspector of the Unity editor, as seen in figure 155.

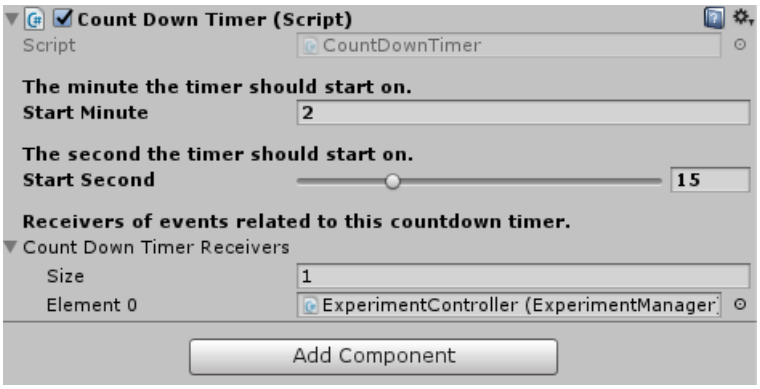


Figure 155: The CountDown Timer expose various settings to the inspector of the Unity editor.

As can be seen in figure 155, it is possible to configure the minutes and seconds that the timer should start counting down from.

Furthermore, it can also be seen that it is possible to specify *CountDown Timer Receivers*. These are scripts within the scene that will receive an event when the timer reaches zero. With this, it is possible for scripts to react and activate time-based logic. For example, in the object manipulation experiment, furniture is disabled once the timer reaches zero, in order to prevent users from interacting with furniture any longer.

Figure 156 shows a class diagram of the classes comprising the CountDown Timer System.

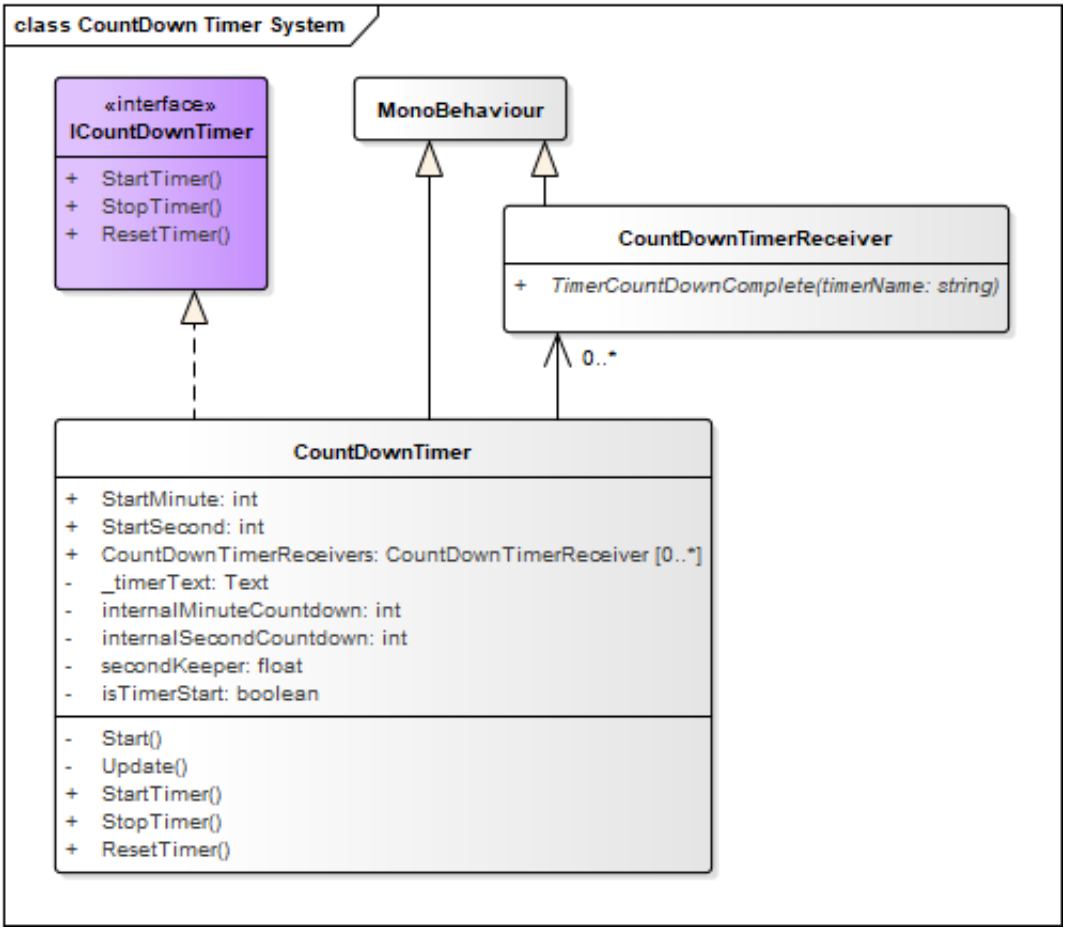


Figure 156: The class diagram of the CountDown Timer System.

The interface *ICountDownTimer* represents the various actions that can be performed on a countdown timer. External scripts can call these to perform actions such as starting the timer countdown, stopping the timer countdown, or resetting the counter to the initial starting time.

The class *CountDownTimer* is a monobehaviour, and contains the primary logic of the countdown timer. The public fields: *StartMinute*, *StartSecond* and *CountDownTimer-Receivers* are visible in the inspector of the Unity editor. It is these properties that a user can manipulate to configure the countdown timer to their liking.

CountDownTimerReceiver is an abstract class which scripts can inherit from if they want to receive the *TimerCountDownComplete* event. This method is called by *CountDownTimer* when its clock reaches zero.

Figure 157 shows the basic countdown flow between the various actors and components.

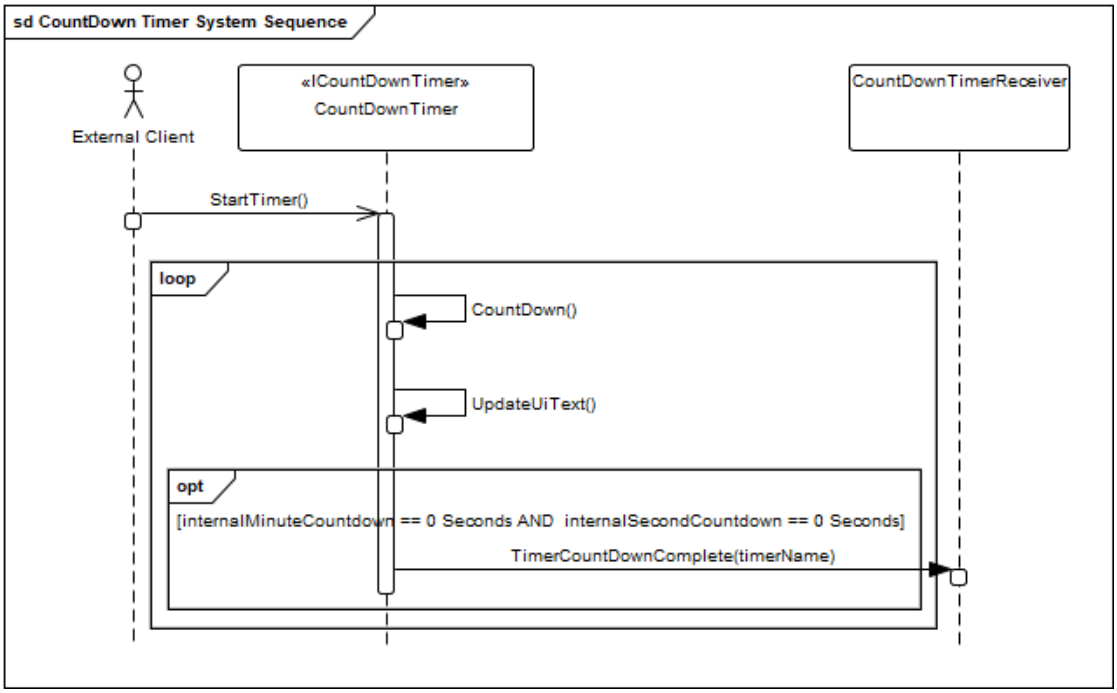


Figure 157: The basic countdown timer sequence betwen various actors and components.

F.1.1 How To Use

This is a short guide that shows the basic usage of a Countdown Timer.

For this guide, a new empty scene has simply been created. Here, the *CountDownTimer* prefab from the experiment toolbox has been dragged into the scene.

Next, a test script has been created, which can be seen in figure 158. This script inherits from *CountDownTimerReceiver* in order to be able to receive the *TimerCountDownComplete* event from the countdown timer. Additionally, it simply starts the timer in the scene on scene startup.

```
public class CountDownTimerTest : CountDownTimerReceiver
{
    private ICountDownTimer countDownTimer;

    // Use this for initialization
    void Start ()
    {
        // Find timer and start it
        countDownTimer = GetComponent<ICountDownTimer>();
        countDownTimer.StartTimer();
    }

    public override void TimerCountdownComplete(string timerName)
    {
        Debug.Log("Timer Reached Zero!");
    }
}
```

Figure 158: A basic example script demonstrating basic usage of the CountDown Timer system. This script simply starts the timer on scene startup, and prints a debug message when the countdown timer reaches zero.

This script was simply attached on the CountDownTimer prefab. Then, the script was added to the list of Countdown timer receivers of the Countdown Timer. This configuration can be seen in figure 159.

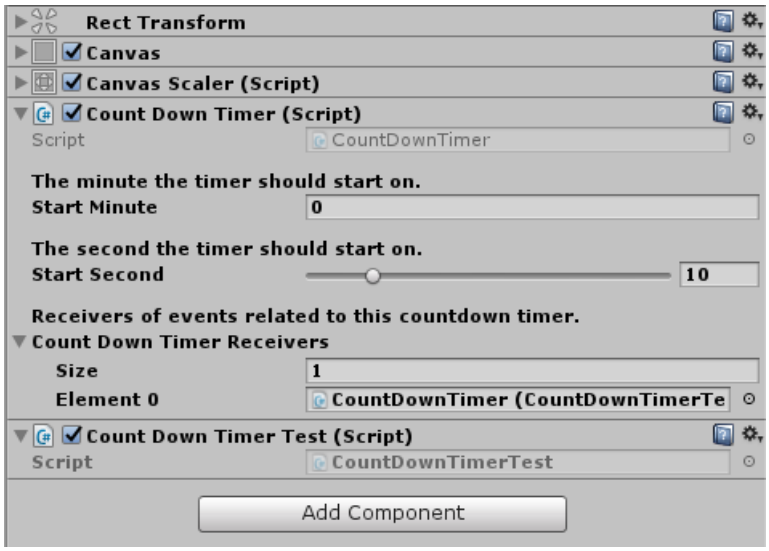


Figure 159: The configuration of the CountDownTimer prefab with the test script added.

This shows simple usage of the CountDown timer.

F.2 Firebase Event Listener System

The Firebase Event Listener System allows experiment applications in Unity to receive updates when nodes change on Firebase. This allows for two-way communication between external tools and the experiment applications.

Figure 160 shows the class diagram for the Firebase Event Listener System.

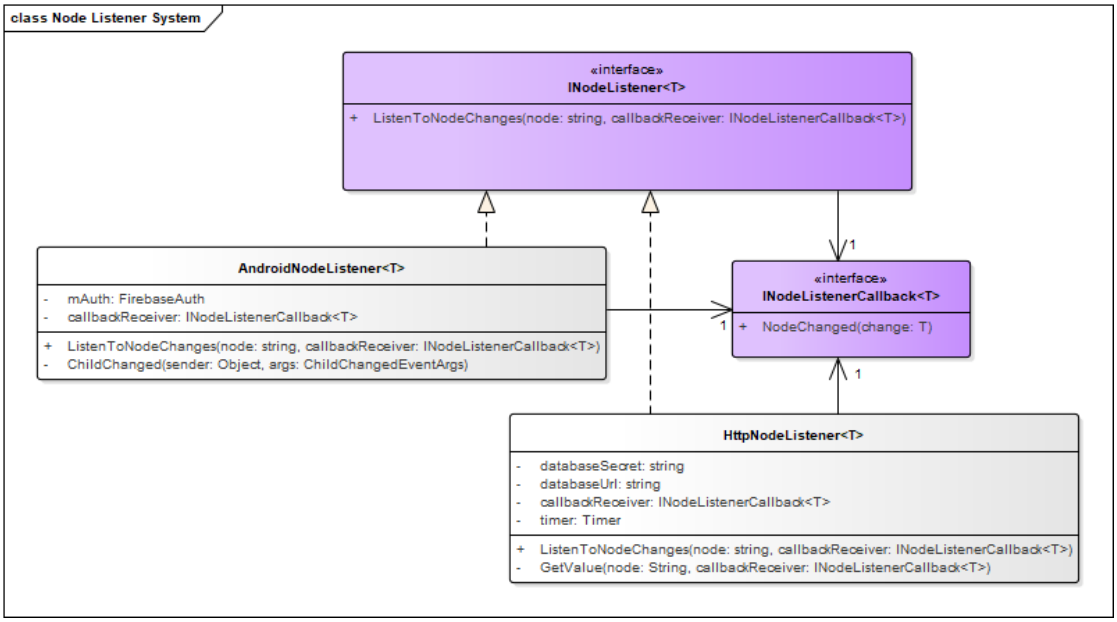


Figure 160: The classes comprising the Firebase Event Listener System.

The system consists of two primary concepts: *Node Listeners* and *Node Listener Callbacks*.

Node Listeners are technology-concrete implementations of the Firebase event listeners. These classes are responsible for detecting changes on a specific node on Firebase, and at these event detections, call a *Node Listener Callback* with these changes. *Node Listener Callbacks* are external clients implementing the *INodeListenerCallback* interface. These are interested in getting a call when a specified node changes. They must call node listeners in order to start receiving these changes.

Importantly, the system works on the concept of generics. Experiment applications must contain model classes representing the equivalent nodes on Firebase. When concrete implementations of *INodeListener* detects a change on Firebase for a specified node, they will deserialize the received JSON to the type specified by the generics.

Figure 161 shows the basic flow between firebase events, *NodeListeners* and *NodeListener-Callback*.

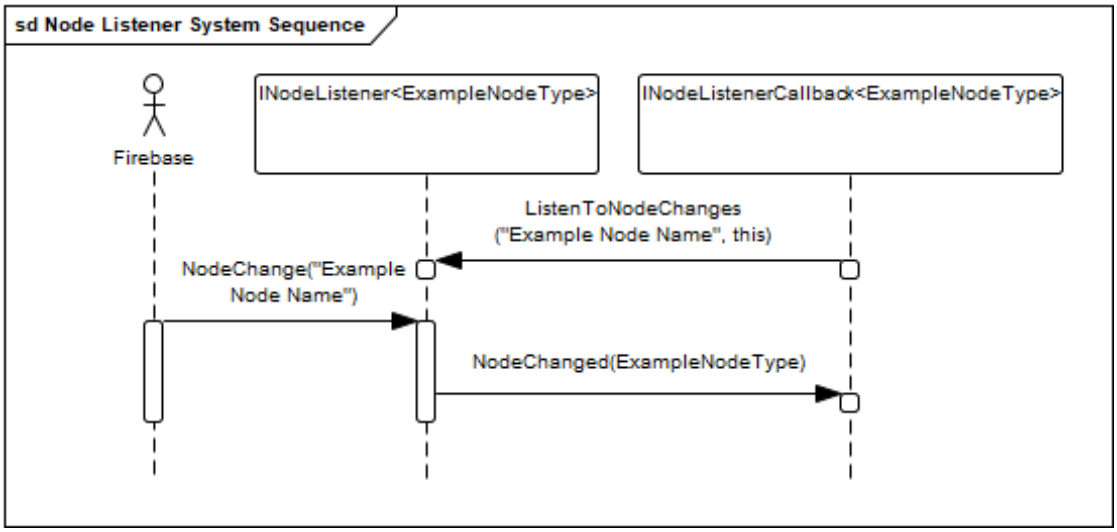


Figure 161: A basic example flow of the Firebase Event Listener System.

In figure 160, the class *AndroidNodeListener* is a concrete realization of *INodeListener*. This class makes use of the Firebase Plugin for Unity to listen for Firebase node changes. It can only be used on the Android platform.

The class *HttpNodeListener* is a concrete realization of *INodeListener* which makes use of the HTTP REST API of Firebase to retrieve node changes. This implementation can be used on all platforms supporting HTTP.

F.2.1 How to use

This is a short guide that shows the basic usage of the Firebase Event Listener System.

In this guide, we will be listening for changes of a *HelloWorld* firebase node, having the structure seen in figure 162.

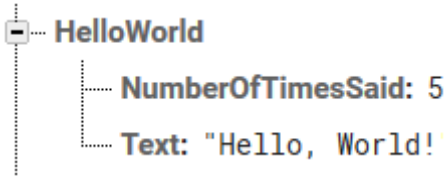


Figure 162: The firebase node which will we will use to listen for changes.

In order to achieve this, the model class presented in listing 2 have been created.

```
1 using System;
```

```
2
3  [Serializable]
4  public class HelloWorldModel
5  {
6     public string Text;
7     public int NumberOfTimesSaid;
8  }
```

Listing 2: Simple model class of the *HelloWorld* firebase node which should be listened to.

Next, it is required to create a script implementing the *INodeListenerCallback* interface using the *HelloWorldModel* class just created in listing 2. This script should also start listening for changes of the node by calling an implementation of the *INodeListener* interface. Such an example script is presented in listing 3.

```
1  using UnityEngine;
2  using Zenject;
3
4  public class NodeListenerTest : MonoBehaviour,
       INodeListenerCallback<HelloWorldModel>
5  {
6     [Inject]
7     private INodeListener<HelloWorldModel> nodeListener;
8
9     // Use this for initialization
10    void Start()
11    {
12       nodeListener.ListenToNodeChanges("HelloWorld", this);
13    }
14
15    public void NodeChanged(HelloWorldModel change)
16    {
17       Debug.Log("HelloWorld Node Changed! The values are:");
18       Debug.Log("Text: " + change.Text);
19       Debug.Log("NumberOfTimesSaid: " + change.NumberOfTimesSaid);
20    }
21 }
```

Listing 3: Demonstration script which begins listening to node changes and receives all subsequent changes.

In listing 3, notice that dependency injection is used through the Zenject framework in order to get an implementation of a node listener. This can be seen in line 7. This is not required. You are free to create the instance yourself. See section C.2.1 about our use of Zenject for a greater explanation of Zenject and our usage of the framework.

In the *Start* method of the test script presented in listing 3, it can be seen that the script starts listening to changes for the *HelloWorld* firebase node by calling *ListenToNodeChanges* of a node listener implementation, giving the node name of a parameter, and the script instance itself as the node listener callback.

All subsequence changes to the *HelloWorld* firebase node will thus be given to the test script by getting its implemented *NodeChanged* method called. An example change is illustrated in figure 163.

Figure 163: An example change of the *HelloWorld* firebase node.

The change in figure 163 will be detected by the node listener and routed to the demonstration script of listing 3.

An example of the log output after this change can be seen in figure 164.

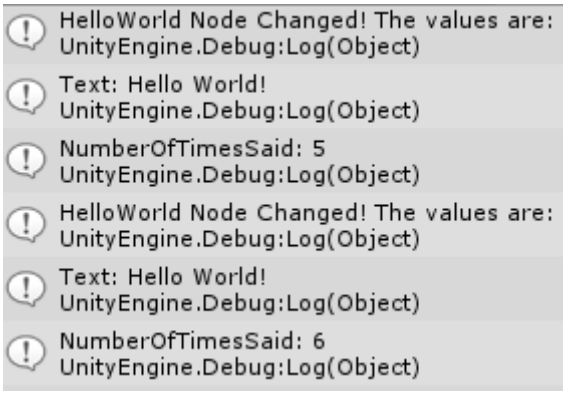


Figure 164: The change was received by the demonstration script, and the output logged to the Unity console.

The configuration of the test scenes used in this guide can be seen in figure 165

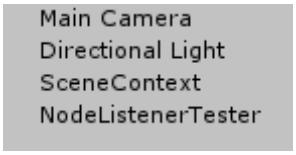


Figure 165: The configuration of the scene used for the node listener guide.

SceneContext is the GameObject used for the dependency injection framework Zenject. This GameObject is configured to use the Zenject Installer used throughout our project. In order to read more about this configuration, see section C.2.1 on our usage of Zenject.

NodeListenerTester is the GameObject which has the node listener demonstration script from listing 3 attached to it. The configuration of this GameObject can be seen in figure 166.

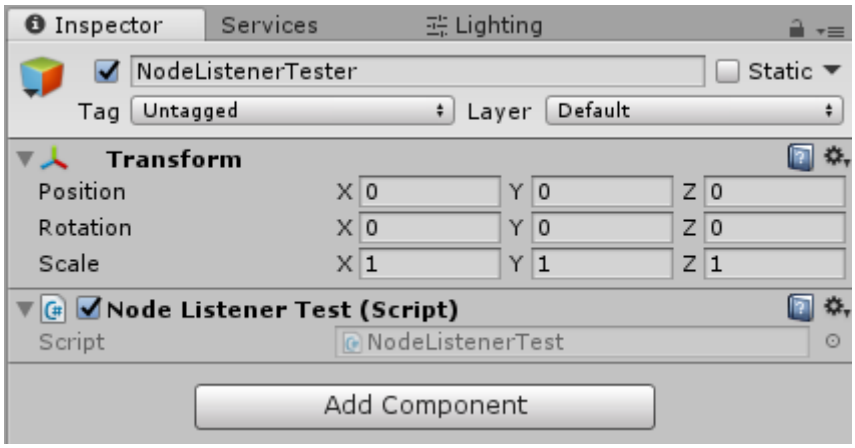


Figure 166: The configuration of the *NodeListenerTester* GameObject of the node listener demonstration scene.

This concludes the guide of Firebase Event Listener System.

F.3 Installed VR Remote Object Manipulation System

The Installed VR Remote Object Manipulation (ROM) System can be used with installed VR platforms to achieve remote object manipulation.

The ROM supports pulling objects closer or further away from the user through vertical swipe functionality. Additionally, it supports left-right rotation of the selected object.

Furthermore, it is integrated to work with SteamVr's interaction system. This means that users can seamlessly switch between ROM mode and SteamVr's hand-oriented interaction mode by pulling objects close enough to the visual hands and picking them up. That, or using one of the virtual hands to pick up an object already selected through the ROM system of another hand. An example of this is demonstrated in figure 167.

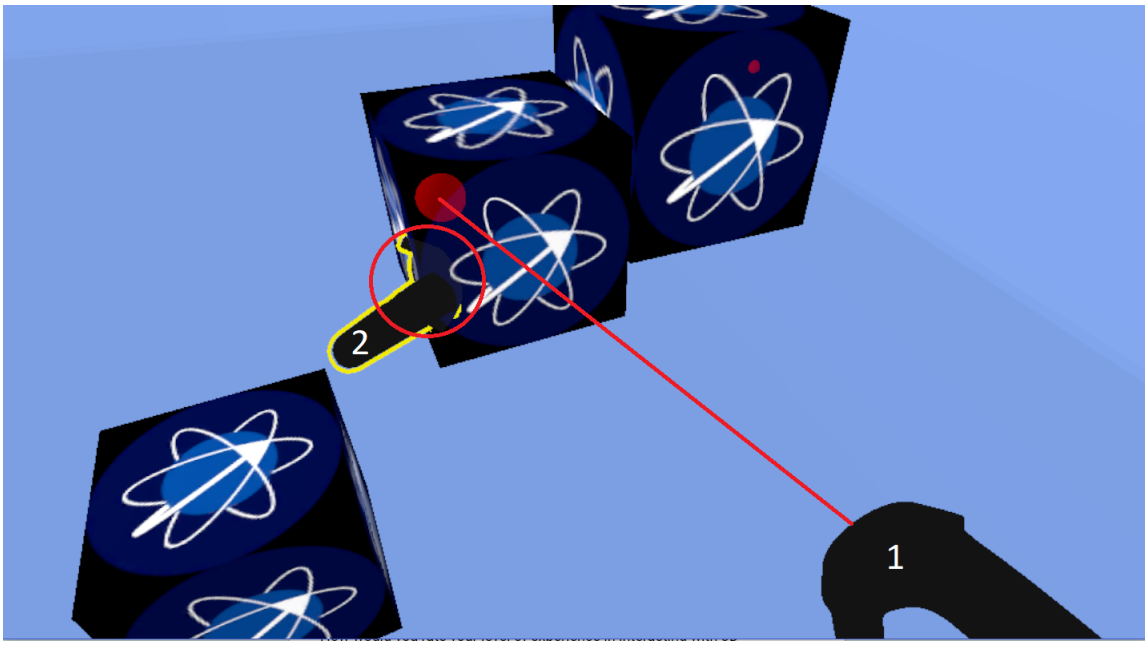


Figure 167: Demonstration of our ROM system working seamlessly with SteamVr's interaction system.

In figure 167, hand nr. 1 has the cube selected using our ROM system, illustrated by the red raycast extending to the cube. Hand nr. 2 is hovering on the selected cube, and the user is free to pick it out of hand nr. 1. SteamVr's interaction system thus works well with our ROM system.

Figure 168 shows the scripts comprising the ROM system.

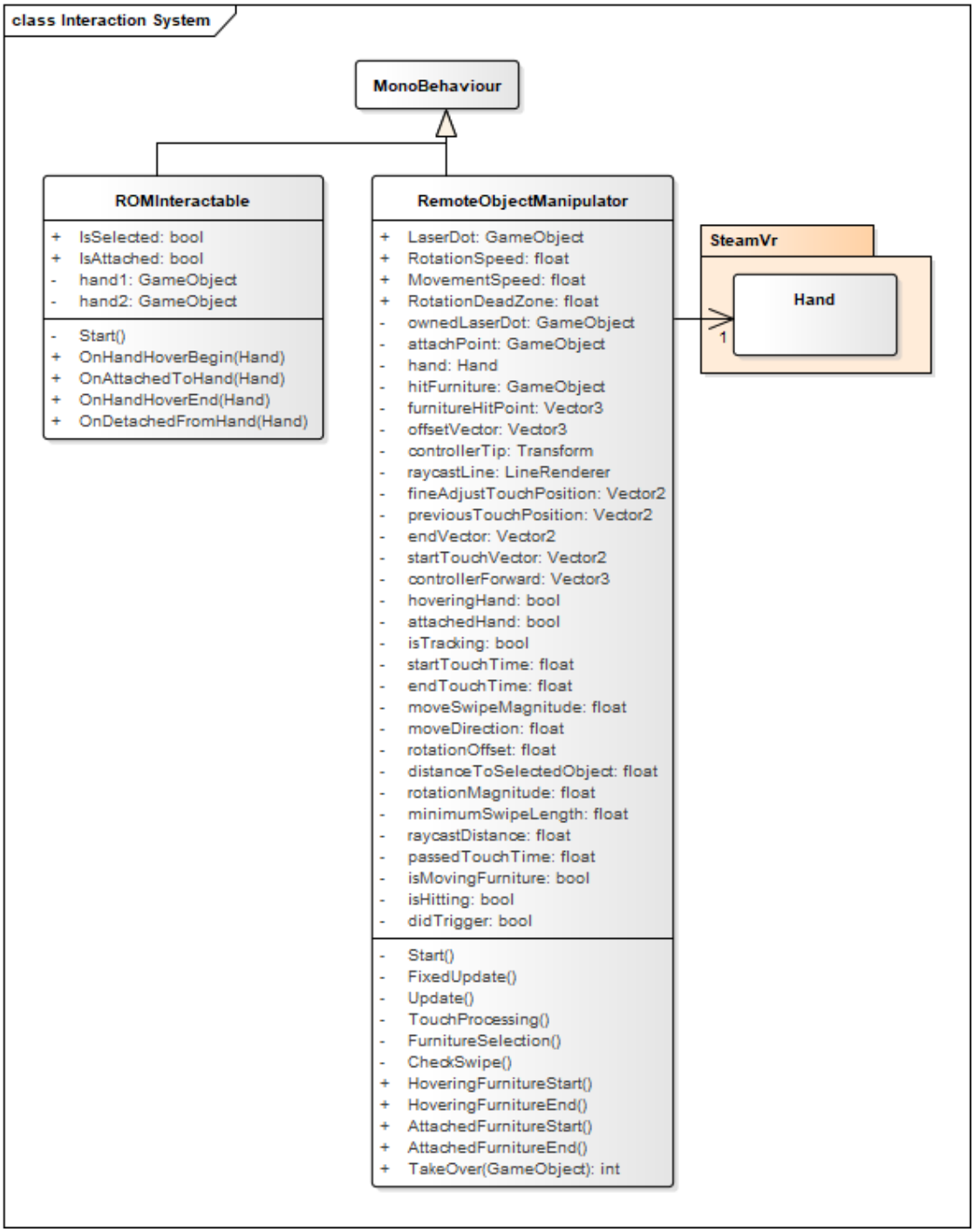


Figure 168: The scripts comprising the ROM system.

The *RemoteObjectManipulator* is the primary script of the ROM system. It handles all

the logic of: raycasting from the controller(Hand) it is attached to, input reading from controller hardware (this includes swipe detection) and moving and rotating the selected object based on this input. The *RemoteObjectManipulator* script is attached to the SteamVr hand GameObject it should work with.

The *ROMInteractable* script needs to be attached to all objects in the world which one would like to use with the ROM system. Objects in the world that does not have this script will not be able to be picked up by the ROM system. The *ROMInteractable* handles its own state of being either attached to hands or selected by the ROM system. This state is used by *RemoteObjectManipulator* in order to prevent conflicts between SteamVr's interaction system and our ROM system. *ROMInteractable* is also responsible for communicating important events to *RemoteObjectManipulator*.

The *ROMInteractable* and *RemoteObjectManipulator* has some involved event flows that enable seamless integration to SteamVr's interaction system. These flows are best described through sequence diagrams. Thus, the most important of these will now be shown.

Figure 170 shows the flow related to disabling object selection of the ROM system, if the virtual hand of the user is hovering on an object already selected by it. This is a safety mechanism that makes sure our ROM system does not interfere with SteamVr's interaction system. Figure 169 shows an example of this case.

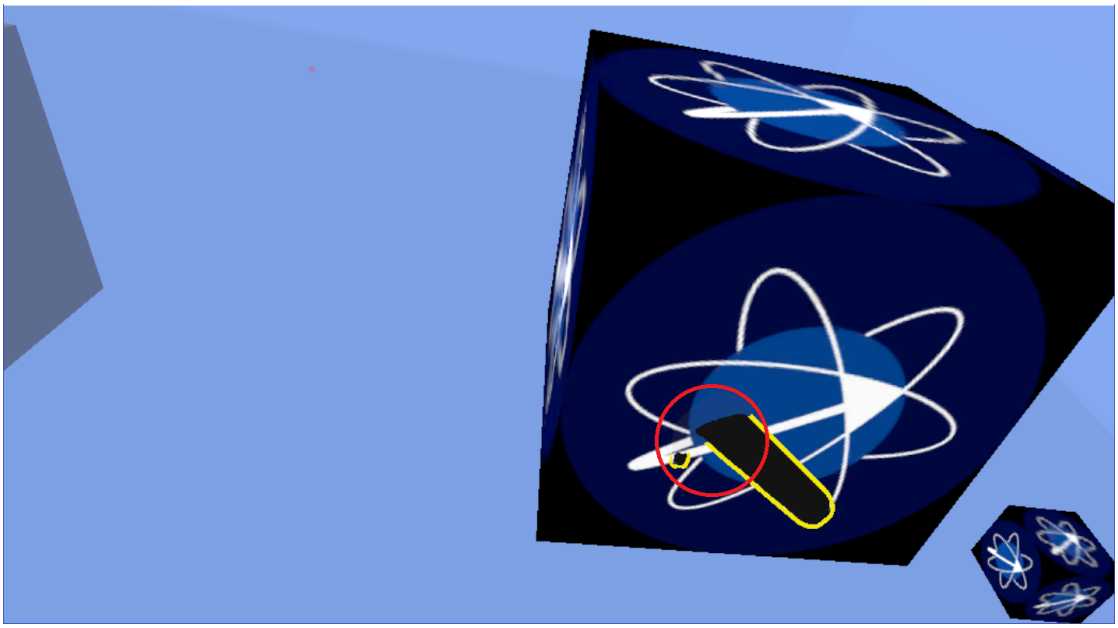


Figure 169: Demonstration of the ROM system, where SteamVr's hand is hovering on the selected object.

In figure 169, the cube is already selected by our ROM system. However, it is pulled so

close to the user that SteamVr's hand is hovering on it. In this case, our ROM system should not interfere with SteamVr's interaction system when the user then decided to pick the cube up in the SteamVr hand.

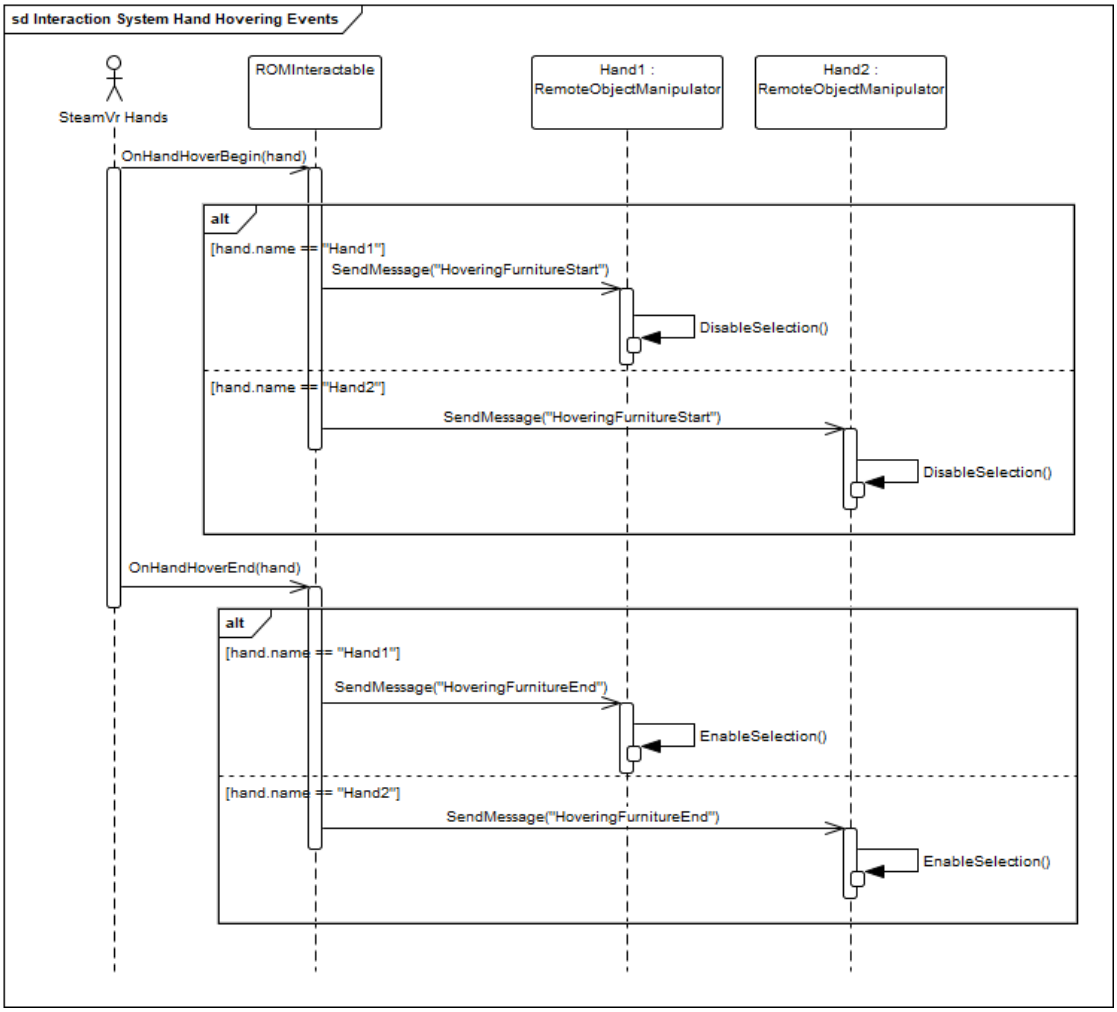


Figure 170: The scripts comprising the ROM system.

Here, it can be seen that hands of the SteamVr system notifies the *ROMInteractable* of a *OnHandHoverBegin* event once a hand is hovering on an object. Once the event is received, it relays this event to the *RemoteObjectManipulator* of the relevant hand. This *RemoteObjectManipulator* will then know that SteamVr's interaction system should take over, and it will thus disable object selection on itself so as to not interfere.

Once the hand stops hovering on an object, the event *OnHandHoverEnd* is sent to the *ROMInteractable*. It then relays this message to the *RemoteObjectManipulator* of the relevant hand, so that it can enable object selection again, since there is now no danger

of interfering with SteamVr's interaction system.

Figure 171 shows the flow related to the take-over mechanism when going from manipulating an object using our ROM system, to manipulating it with SteamVr's interaction system. This happens when the user pulls an object so close that it hovers on SteamVr's virtual hand, and then picks it up with that hand. It can also happen when the user has an object selected with the ROM system in one hand, and then picks up the object using the other hand. This case is demonstrated figure 167.

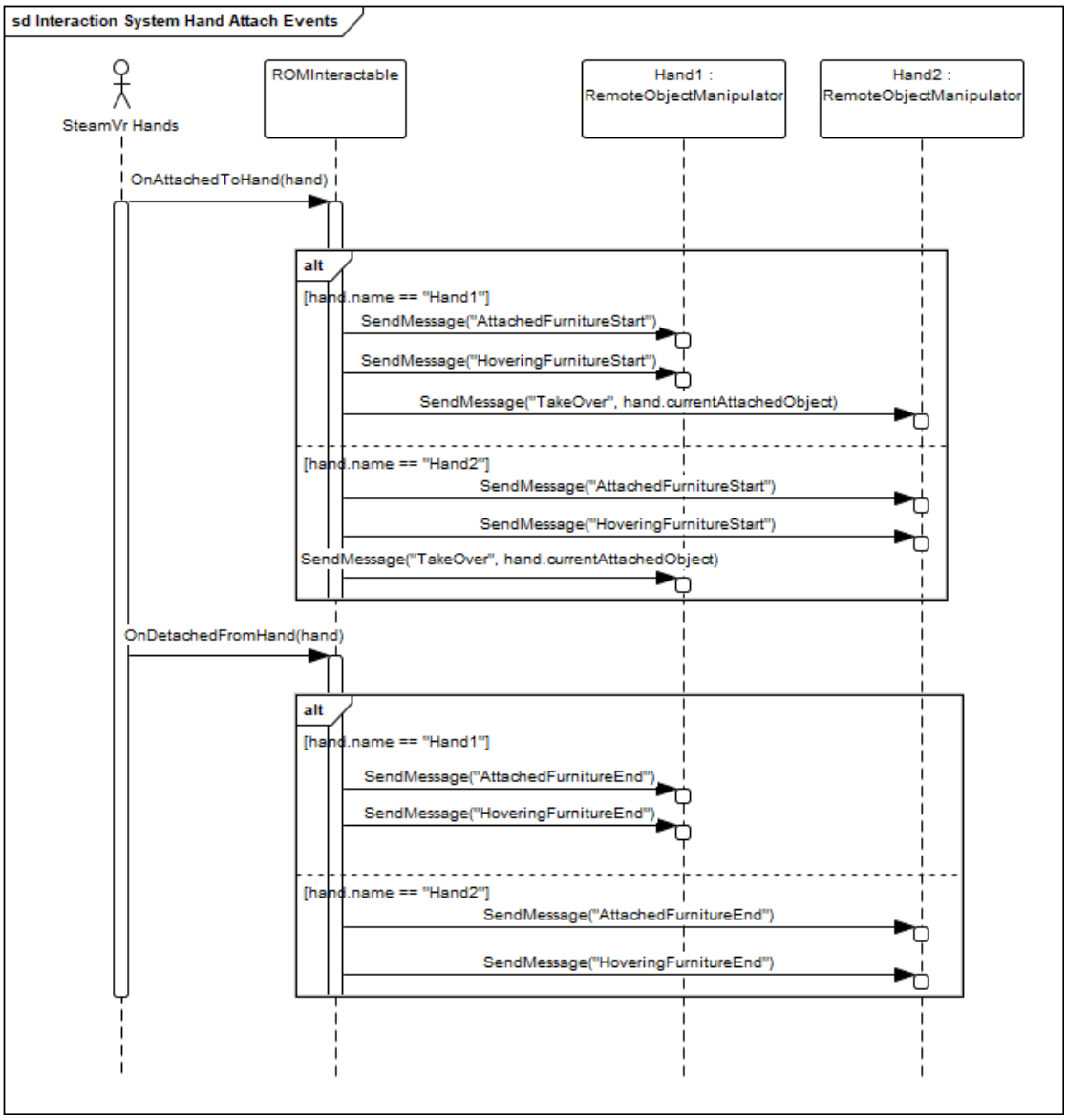


Figure 171: The scripts comprising the ROM system.

F.3.1 Swipe Functionality

During development of the ROM system, a considerable amount of time was spent creating swipe functionality which worked well. The swipe functionality includes the user being able to swipe vertically on the controller touchpad in order to move selected objects further away or closer.

This section will highlight important code snippets of the swipe functionality for a more in-depth technical discussion. It is the hope that this technical showcase can serve as a starting point for others in need of developing a custom swipe mechanism.

As a pre-requisite, it is useful to know that the touchpad of a VR controller in Unity is considered a 2D surface in which touches are recorded as positions using 2D vectors. Figure 172 shows a mapping of the touchpad from the point of view of Unity.

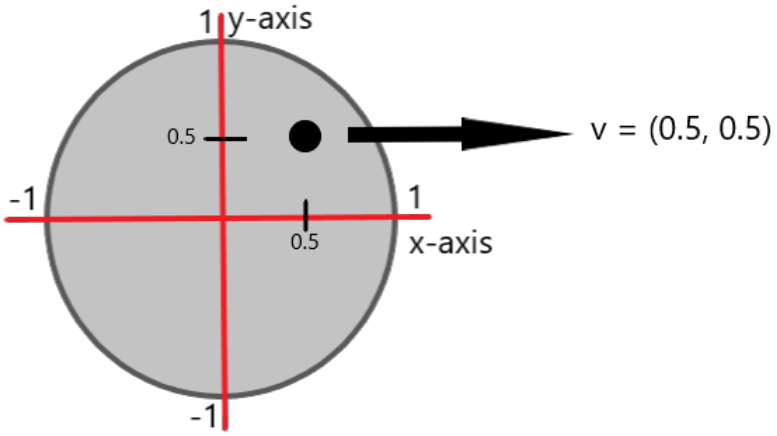


Figure 172: The mapping of a VR controller touchpad in Unity.

In the example of figure 172, a touch in the middle of the upper left corner of the touchpad would be recorded as a 2D vector with its x- and y-components equal to 0.5.

The majority of swipe functionality takes place within the private method *TouchProcessing*, as presented in the class diagram of figure 168. This method is executed in every update loop of the Unity game engine.

Two pieces of logic within this method drives the swipe functionality. One is the detection of the user touching the touchpad (without pressing it). The other is detection of the user letting go out the touchpad.

Listing 4 shows touchpad touch down detection. Listing 5 shows touchpad touch release detection.

```
1  if (hand.controller.GetTouchDown(SteamVR_Controller.ButtonMask.Touchpad))
2  {
3     startTouchTime = Time.time;
4     startTouchVector =
          hand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_SteamVR_Touchpad);
5     fineAdjustTouchPosition =
          hand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_SteamVR_Touchpad);
6  }
```

Listing 4: Code Snippet of controller touch down detection of the ROM system.

```
1  if (hand.controller.GetTouchUp(SteamVR_Controller.ButtonMask.Touchpad))
2  {
3     passedTouchTime = 0.0f;
4     endTouchTime = Time.time - startTouchTime;
5     rotationMagnitude = 0.0f;
6
7     if (endTouchTime <= maximumSwipeTime)
8     {
9        rotationMagnitude = 0.0f;
10       CheckSwipe(startTouchVector, endVector);
11    }
12 }
```

Listing 5: Code Snippet of controller touch release detection of the ROM system.

For the swipe functionality, the position of the touch on the touchpad is recorded when the user first puts a finger on it. Furthermore, the time of the touch is also recorded. This happens in listing 4.

When the user releases the finger from the touchpad, the code in listing 5 is executed. Here, the total time since the user first touched the touchpad is calculated using the recorded start time from the code in listing 4. This time is used to determine if the user's finger was on the touchpad within an acceptable interval to be considered a swipe. If the user have spend too much time in contact with the touchpad, the swipe functionality will not be executed, as it won't be considered fast enough for a swipe.

If it is determined that a swipe was detected, the method *CheckSwipe* is executed. Here, the position of the first touch recorded from listing 4 is used in conjunction with the position of the last touch recorded before the user let go of the touch as parameters to the *CheckSwipe* method. This can be seen in line 10 of listing 5.

The code of the CheckSwipe method can be seen in listing 6.

```
private void CheckSwipe(Vector2 startVector, Vector2 endVector)
{
   var finalVector = endVector - startVector;
   var yStartVector = new Vector2(startVector.y, 0);
   var yEndVector = new Vector2(endVector.y, 0);

   if (Vector2.Distance(yStartVector, yEndVector) >= minimumSwipeLength)
   {
      moveSwipeMagnitude = Vector2.Distance(yStartVector, yEndVector);
      moveDirection = Mathf.Sign(finalVector.y);
   }
}
```

Listing 6: Code Snippet of controller CheckSwipe method of the ROM system.

The purpose of the *CheckSwipe* method is to calculate the distance between two points - in this context, the start position and the end position of the user's touches - in order to determine if the finger movement was far enough along the y-axis of the touchpad to be considered a vertical swipe. If this is indeed the case, that distance will be applied as a movement vector to the selected object of the ROM system.

F.3.2 Fine Adjustment Functionality

The Fine Adjustment functionality is related to the swipe functionality discussed in section F.3.1. Read that section in order to be introduced to relevant pre-requisite knowledge and code snippets related to swiping functionality in general.

Fine adjustment allows the user to move his or her finger slowly vertically across the touchpad in order to make minor position changes to the selected object of the ROM system. This is useful in cases where more accuracy is needed.

Fine adjustment processing takes place within the *TouchProcessing* method. Listing 7 shows the code relevant to fine adjustment. Please notice that code not related to fine adjustment has been omitted from this snippet.

```

if (hand.controller.GetTouch(SteamVR_Controller.ButtonMask.Touchpad))
{
   passedTouchTime += Time.deltaTime;

   if (passedTouchTime >= fineAdjustUpdateInterval)
   {
```

```
8       fineAdjustTouchPosition =
            hand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_SteamVR_Touchpad);
9
10      passedTouchTime = 0.0f;
11  }
12
13  endVector =
        hand.controller.GetAxis(Valve.VR.EVRButtonId.k_EButton_SteamVR_Touchpad);
14
15  CheckSwipe(fineAdjustTouchPosition, endVector);
16 }
```

Listing 7: Code Snippet of fine adjustment logic of the *TouchProcessing* method.

The code in listing 7 is executed so long as the user has a finger touching the touchpad.

The main idea behind the fine adjustment logic is that it still makes use of the *CheckSwipe* method - introduced in section F.3.1 and listing 6 - in order to apply a movement vector to the selected object. However, instead of making use of the larger vector used in the swipe functionality of section F.3.1, fine adjustment makes use of a continuously updated small movement vector. In line 6 through 11 in listing 7, it can be seen how a finger's touch position on the touchpad is sampled at the interval specified by *fineAdjustUpdateInterval*. This is a small value, in the range of milliseconds. This position is used in conjuction with the *endVector* of line 13 when calling *CheckSwipe*. The small update interval of the variable *fineAdjustUpdateInterval* limits the size of the movement vector to always be relatively small. In this way, fine adjustment is achieved.

F.3.3 How To Use

This is a short guide that shows the basic usage of the ROM system.

The ROM system works in conjunction with SteamVr. Thus a pre-requisite for this guide is that the scene you are working in is already set up with SteamVr's prefabs.

In order to enable the ROM system for SteamVr's controllers, you have to add the *RemoteObjectManipulator* script to one or both of SteamVr's hands.

Figure 173 shows an example configuration of a SteamVr hand object with the *RemoteObjectManipulator* added.

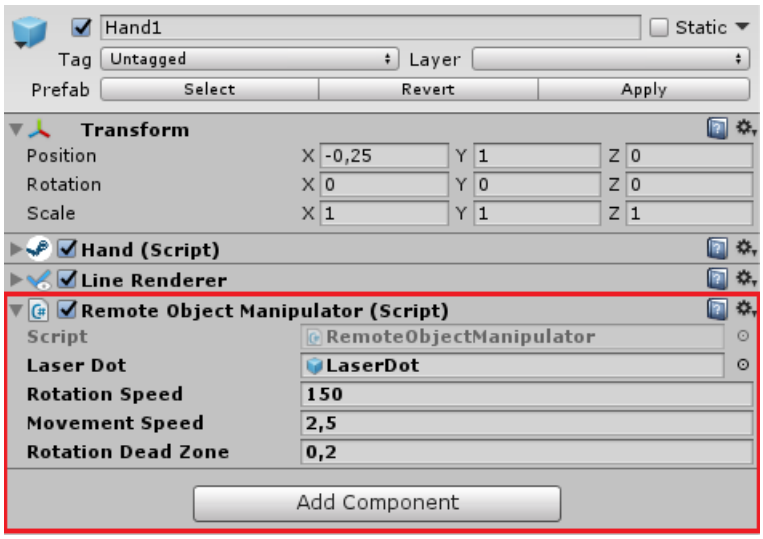


Figure 173: Example configuration of the RemoteObjectManipulator script on a SteamVr hand object.

In figure 173, four important properties can be adjusted from the inspector: *Laser Dot*, *Rotation Speed*, *Movement Speed*, and *Rotation Dead Zone*.

For *Laser Dot*, you provide a prefab that represents a laser dot. This will be used at the end of the ROM system's raycast in order to give the user a visual cue as to where they are aiming in the world. An image of the laser dot used in this configuration can be seen in figure 167.

Rotation Speed determines the maximum speed of which the user can rotate selected objects with.

Movement Speed determines the maximum speed at which users can move objects to and from themselves.

Rotation Dead Zone determines the dead zone before touchpad input will begin rotating the selected object. The bigger this value is, the further to the left and right side of the touchpad the user will have to move in order to begin rotating the object.

A required script of the *RemoteObjectManipulator* is the *Line Renderer* seen in figure 173. This script comes standard with the Unity framework and will be automatically attached to the hand if you have not already done so yourself. This line renderer controls the configuration for the visual raycast line of the ROM system. Figure 174 shows an example configuration of this. However, this configuration can be done in any way you like, to achieve personalized raycast visuals.

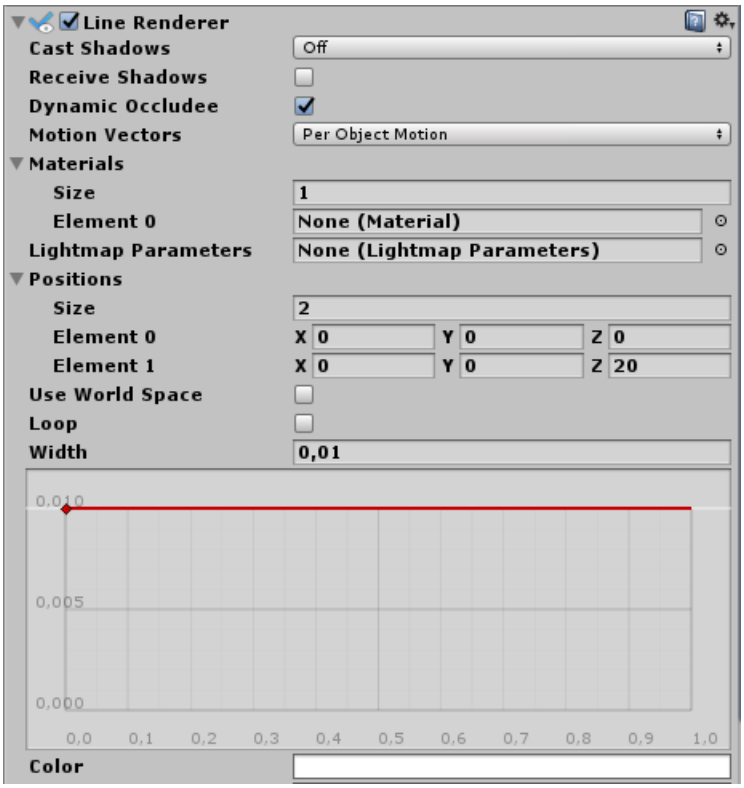


Figure 174: Example configuration of the line renderer used to render the raycast coming from *RemoteObjectManipulator*.

In figure 175 a screenshot can be shown of how this line renderer configuration looks during the object manipulation experiment.

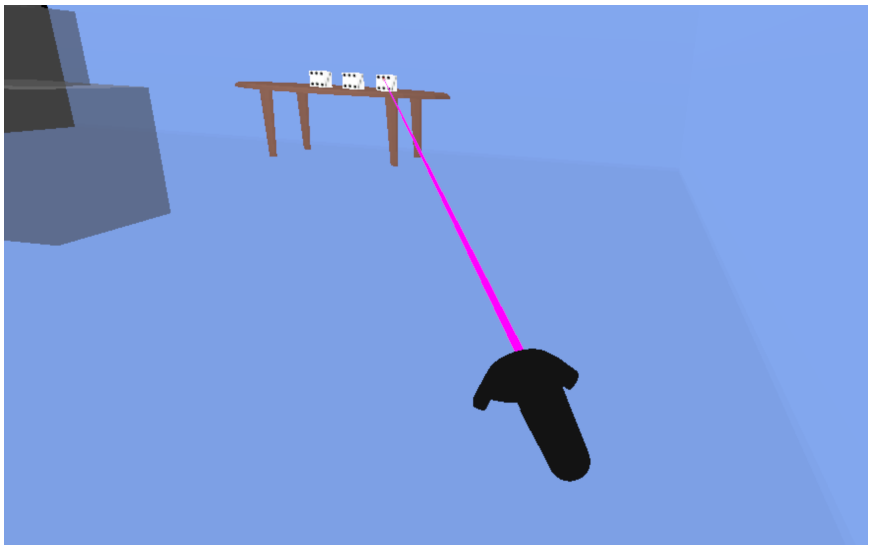


Figure 175: Example of the raycast visuals of the *RemoteObjectManipulator* line renderer configuration.

Lastly, any GameObject which you would like to be enabled for the ROM system should have the *ROMInteractable* script attached to it. No further configuration is required after attaching it.

This conludes the guide of the *Installed VR Remote Object Manipulation System*.

F.4 Firebase Measurement Repository System

The Firebase Measurement Repository System can be used as an easy way to save measurement objects from Unity scripts to specified firebase nodes.

Measurement objects can be any serializable model class. This means that one is free to save any custom model.

The classes and interfaces making up this system can be seen in figure 176.

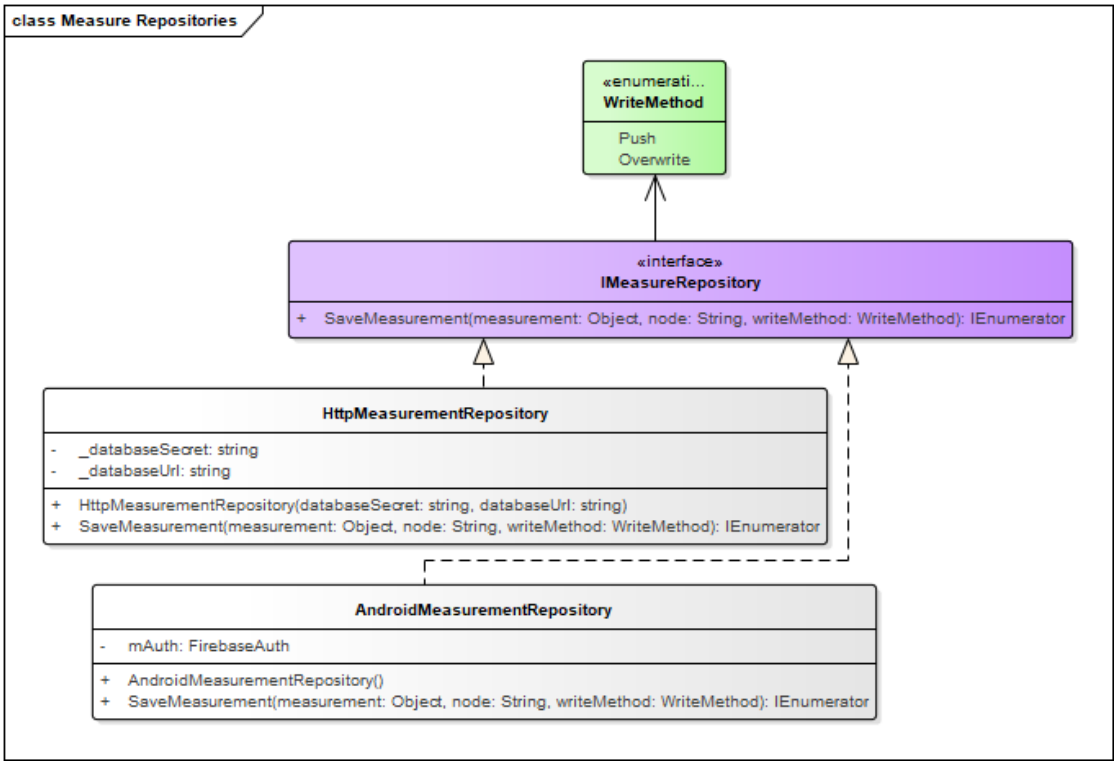


Figure 176: The classes and interfaces comprising the Firebase Measurement Repository System.

The system consists of the primary interface *IMeasureRepository*. Clients saves measurement objects by calling the *SaveMeasurement* method of this interface.

The *measurement* parameter of the *SaveMeasurement* method specifies the object that should be serialized and saved to a firebase node. The *node* parameter specifies the name of the node on Firebase in which the object should be saved to. The *WriteMethod* parameter is used to specify whether the measurement object should be pushed to the node as a new child, or if it should overwrite whatever data that is specifically in the node. Using this parameter, it is possible to control whether to save measurements to a collection of measurements under a single node, or whether or whether to have the measurement be a single node that is overriden.

The classes *HttpMeasurementRepository* and *AndroidMeasurementRepository* are two concrete realizations of this interface provided by the toolbox.

HttpMeasurementRepository is an implementation that can be used on all platforms supporting Http, as this class saves data to Firebase through the official Firebase REST API.

AndroidMeasurementRepository is an implementation that can only be used on the Android

platform. This is because the implementation makes use of the official Firebase Unity package.

F.4.1 How To Use

This is a short guide that shows the basic usage of the Firebase Measurement Repository System.

In this guide, we will save a simple model class to a firebase database. The code for the model class can be seen in listing 8.

```
using System;

[Serializable]
public class HelloWorldModel
{
   public string Text;
   public int NumberOfTimesSaid;
}
```

Listing 8: Simple example model class to be saved to Firebase using the Firebase Measurement Repository System

In order to do this, the simple script of listing 9 has been created.

```
using UnityEngine;
using Zenject;

public class MeasurementRepositoryTest : MonoBehaviour
{
   [Inject]
   public IMeasureRepository measurementRepository;

   // Use this for initialization
   void Start () {
      HelloWorldModel helloWorldModel = new HelloWorldModel
      {
         Text = "Hello, World!",
         NumberOfTimesSaid = 5
      };

      StartCoroutine(measurementRepository.SaveMeasurement(helloWorldModel,
          "HelloWorld", WriteMethod.Overwrite));
   }
```

```
19 }
```

Listing 9: Simple example script which demonstrating how to save an instance of a model class to Friebase.

In listing 9, notice that dependency injection is used through the Zenject framework in order to get an implementation of a measure repository. This can be seen in line 7. This is not required. You are free to create the instance yourself.

In the start method, the *HelloWorldModel* introduced in listing 8 is used by creating an instance with some specified values for its properties.

The crucial part is line 17, where the *SaveMeasurement* method of the measure repository is called. Here, the instance of the *HelloWorldModel* is provided. Furthermore, it is specified that it should be saved to a node in Firebase by the name of ḦelloWorld̈. The write method is specified as *Overwrite*, meaning that any existing data on that node will simply be overriden. Please also notice that the *SaveMeasurement* method is executed using Unity's *StartCoroutine* method. This is *required*. If you do not call *SaveMeasurement* using this method, it will not work. The measure repository implementations of this project are coroutines, which are used in Unity to model behaviour that is potentially executed over several frames (such as doing HTTP calls).

The configuration of the test scenes used for this example can be seen in figure 177.

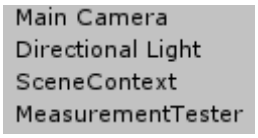


Figure 177: The GameObjects of the scene used for this guide.

The important GameObjects in this scene to pay attention to is *SceneContext* and *MeasurementTester*.

SceneContext is the GameObject used for the dependency injection framework Zenject. This GameObject is configured to use the Zenject Installer used throughout our project. In order to read more about this configuration, see section C.2.1 on our usage of Zenject.

MeasurementTester is the GameObject created specifically for this demonstration. To this GameObject is attached the test script presented in listing 9. This configuration can be seen in figure 178.

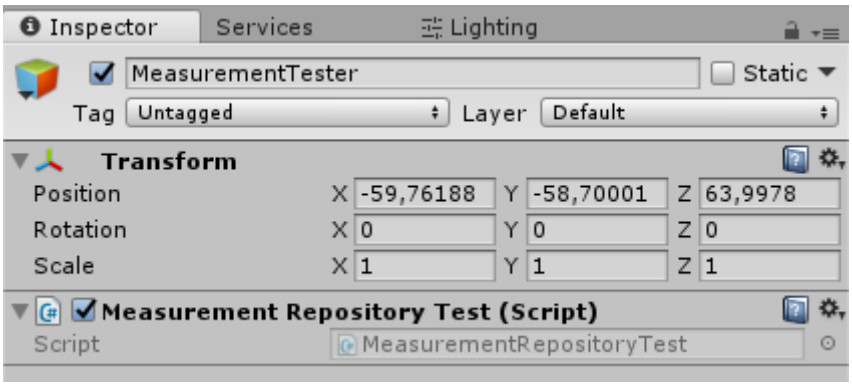


Figure 178: The configuration of the *MeasurementTester* GameObject created for demonstration purposes.

Running this scene, the test measurement created in listing 9 will be saved to Firebase. A screenshot of the result on firebase can be seen in figure 179.

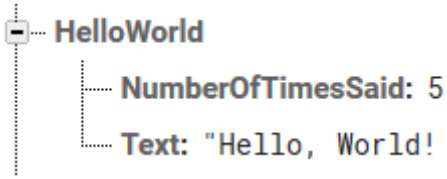


Figure 179: The result of saving the HelloWorld model instance to Firebase through the use of the Firebase Measurement Repository System.

This conludes the guide of the Firebase Measurement Repository System.

F.5 Measurement Calculator System

The Measurement Calculator System is used to calculate the objective measurements of *accuracy* related to the problem statement defined in section 2.3. That is, it can calculate accuracy of how well a user placed an object with respect to orientation and distance. For a greater description of this, see section 3.

The Measurement Calculator System is comprised of the classes and interfaces shown in the class diagram of figure 180.

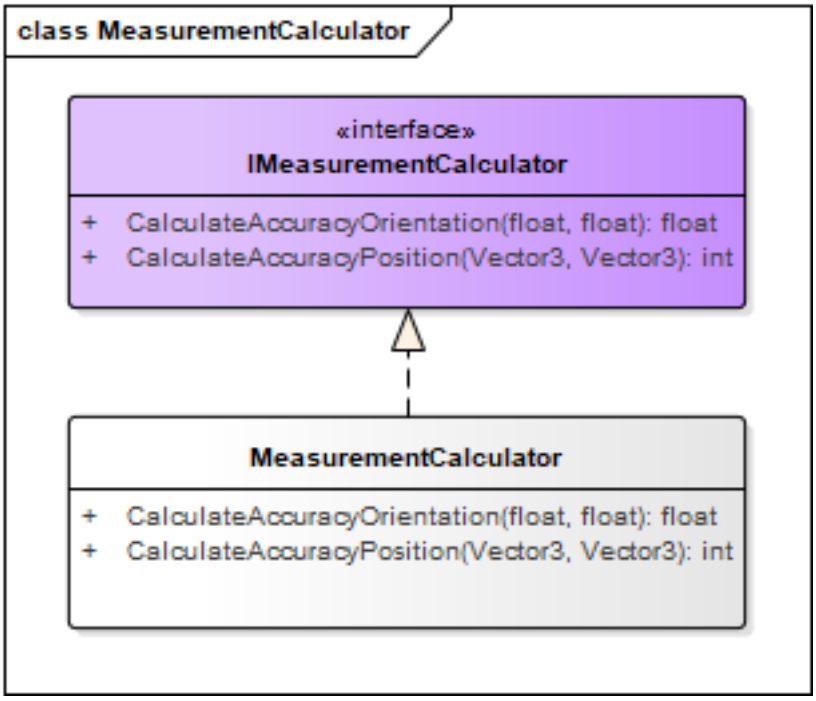


Figure 180: The classes and interfaces comprising the Measurement Calculator System.

The method *CalculateAccuracyOrientation* calculates a percentage value from 0% to 100%. 100% represents two perfectly aligned objects in regards to orientation. The parameter is two euler angles that should be compared. The percentage is calculated using a *shortest distance* method between the two compared euler angles. This means that the maximum possible difference between the two euler angles is 180 degrees. Thus, a 0% accuracy will be a difference of 180 degrees. A 100% accuracy will be a difference of 0 degrees.

The method *CalculateAccuracyPosition* calculates a distance in Unity units between the position of two objects. The parameters are two 3D vectors representing the 3D position of the two objects you would like the distance between.

F.6 Experiment Time Measurer

The *Experiment Time Measurer* is a script which can do simple time keeping for any type of experiment.

It is described in the class diagram of figure 181.

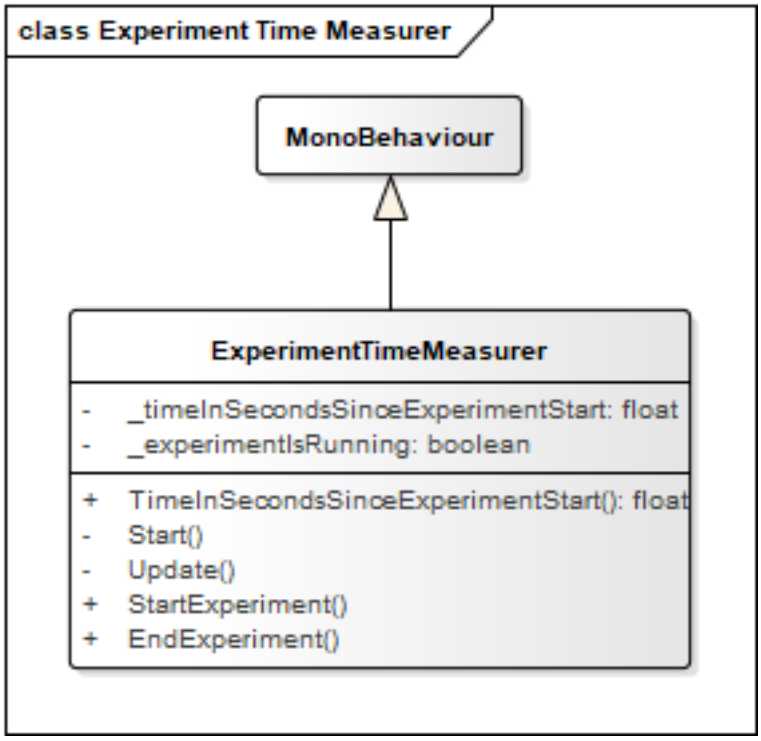


Figure 181: The class diagram of the *ExperimentTimeMeasurer*.

The most relevant methods for clients wishing to make use of this script is *StartExperiment*, *EndExperiment* and *TimeInSecondsSinceExperimentStart*.

The method *StartExperiment* will simple start the time keeping. *ExperimentTimeMeasurer* will from this point on start to count the amount of seconds passed. This method should be called when an experiment is started.

The method *EndExperiment* will stop pause the time keeping. This method should be called when the experiment is done.

The property *TimeInSecondsSinceExperimentStart* can be called in order to get the amount of seconds passed between calling the *StartExperiment* and *EndExperiment* methods.

F.7 Experiment Logger System

The Experiment Logger System can be used by clients in order to implement a loosely coupled logging system.

The class comprising the system can be seen in the class diagram of figure 182.

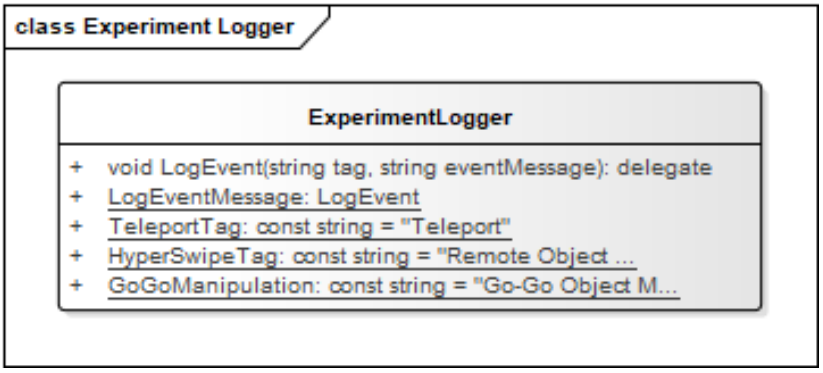


Figure 182: The class comprising the Experiment Logger System.

The *LogEvent* delegate defines the signature of the method which subscribers of the *LogEventMessage* event must implement if they want to listen to it.

The *LogEventMessage* event is used by scripts when they want to log a specific event. Listeners of the event will then receive the log statement.

The static constant properties: *TeleportTag*, *HyperSwipeTag*, and *GoGoManipulation* are pre-defined tags which clients can use for typical use-case scenarios of Virtual Reality navigation modes.

Thus, the typical way of using the Experiment Logger System is by having scripts logging events by calling *LogEventMessage*. Clients can then implement scripts which listen to this event and then process the received logged events.

For a pratical demonstration of this, the context of the Object Manipulation Experiment described in section E is used.

In the Object Manipulation Experiment, various scripts log events which might be interesting to save as part of the experiment measurements for further analysis. One such script is the *TeleportLocomotion* script which simply logs an event when the user teleports using the Google Daydream controller.

The *ExperimentManager* script of the Object Manipulation Experiment, described in appendix E.7, listens to any such *LogEventMessages* and saves them as part of the measurement to be stored on Firebase.

The sequence diagram of figure 183 demonstrates this flow.

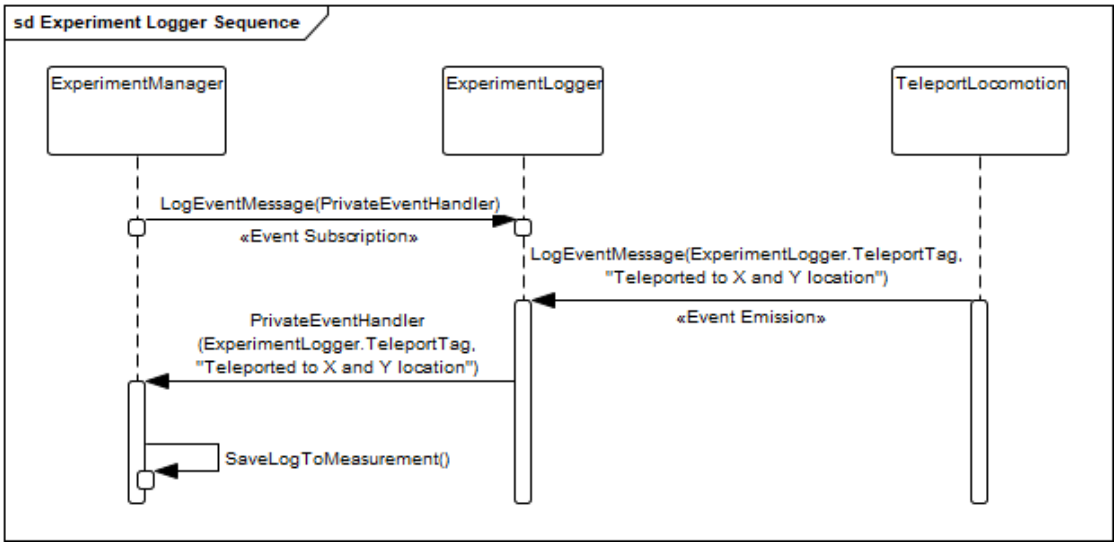


Figure 183: A typical use-case sequence of event logging. This is an example taken from the Object Manipulation Experiment.

In this way, the Experiment Logger System can be used as a way to gather important log data from many disconnected parts of an experiment system.

F.7.1 How To Use

This is a short guide that shows the basic usage of the Experiment Logger System.

For this demonstration, two simple scripts have been created. One script, shown in listing 10, will emit a log message. Another script, shown in listing 11, will receive log messages and print them to the Unity console.

```
1  using Assets.Scripts.Experiment;
2  using UnityEngine;
3
4  public class ExperimentLogEmitterTest : MonoBehaviour {
5     // Use this for initialization
6     void Start ()
7     {
8        ExperimentLogger.LogEventMessage("ExampleTag", "Hello, World!");
9     }
10 }
```

Listing 10: Simple script which emits a log message through the Experiment Logger System.

```
using Assets.Scripts.Experiment;
using UnityEngine;

public class ExperimentLogHandlerTest : MonoBehaviour {
   void Awake()
   {
      ExperimentLogger.LogEventMessage += LogHandler;
   }

   private void LogHandler(string tag, string logMessage)
   {
      Debug.Log("Log Message Received!");
      Debug.Log("The Tag Was: " + tag);
      Debug.Log("The Message Was: " + logMessage);
   }
}
```

Listing 11: Simple script which handles log messages of the Experiment Logger System.

These two scripts have been applied to the same GameObject in a test scenes for simple demonstration purposes. Notice that this is not at all a requirement. The scripts do not have to be on the same GameObject in order to work.

Notice in the log handler of listing 11, that it subscribes to the *LogEventMessage* event with its own private *LogHandler* method. This method complies with the signature of the delegate described in the *ExperimentLogger* class. Notice also that the subscription of the event occurs within the *Awake* method of the MonoBehaviour. It is recommended that you do so, in order to be completely sure that all logged events from other scripts are caught if they perform a log message operation in their *Start* method, such as our example log emitter of listing 10.

Figure 184 shows the output produced by running the scene.

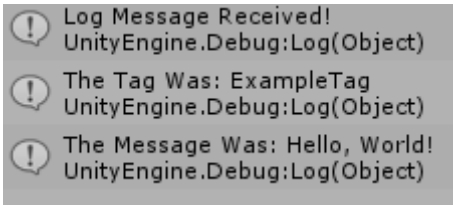


Figure 184: The output produced when running the Experiment Logger demonstration scene.

Here, it can be seen that our event handler received the log event emitted by our log emitter.

F.8 Ghost Object System

One of the most critical features of the experiment toolbox is the ghost object script. The script is used a lot in the object manipulation experiment (see appendix E) and its main functionality is to be a reference point for the user when trying to place associated furniture. It also holds functionality to measure distance and orientational accuracy. It can be configured to send events when the associated furniture has been placed within a configurable margin of error. An example of the usage of ghost objects can be seen in figure 185.

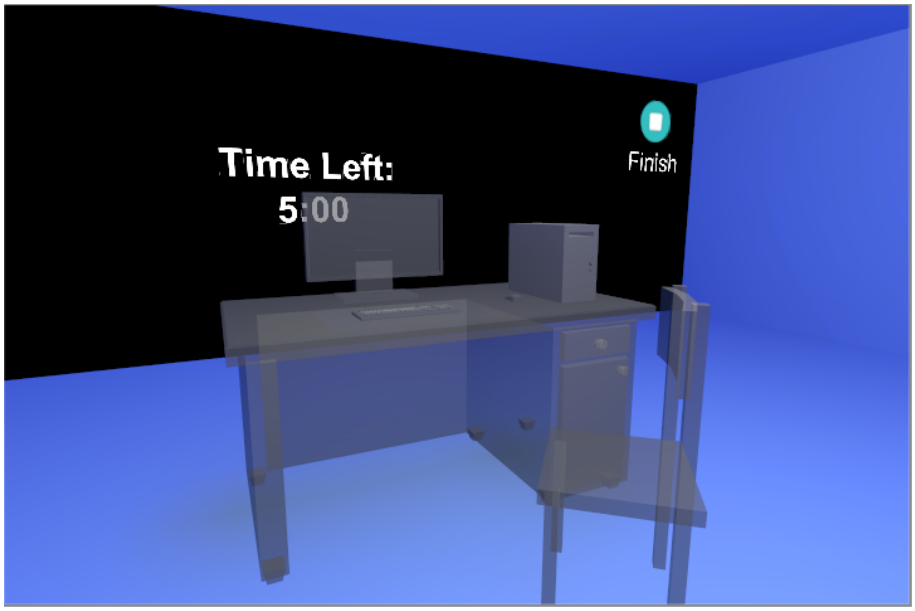


Figure 185: An example of Ghost Objects from the Object Manipulation experiment

The settings for ghost objects that can be configured can be seen on figure 186 and their description in table 31

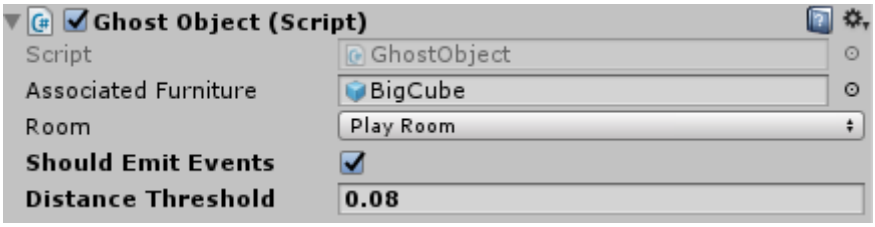


Figure 186: An example of Ghost Object settings from the Object Manipulation experiment

Setting	Functionality
Associated Furniture	This is the furniture that the ghost object is comparing itself to when calculating differences in distance and orientation.
Room	This is a setting that helps the experiment manager distinguish which phase of the experiment the ghost object belongs to.
Should Emit Events	A simple True/False value, that determines if the ghost object should emit events when the associated furniture is within the distance defined in Distance Threshold
Distance Threshold	The distance between ghost objects and associated furniture that counts as the object being placed.

Table 31: Ghost object settings description

The class diagram for the ghost object script can be seen on figure 187. The description of the methods in the class can be found in table 32.

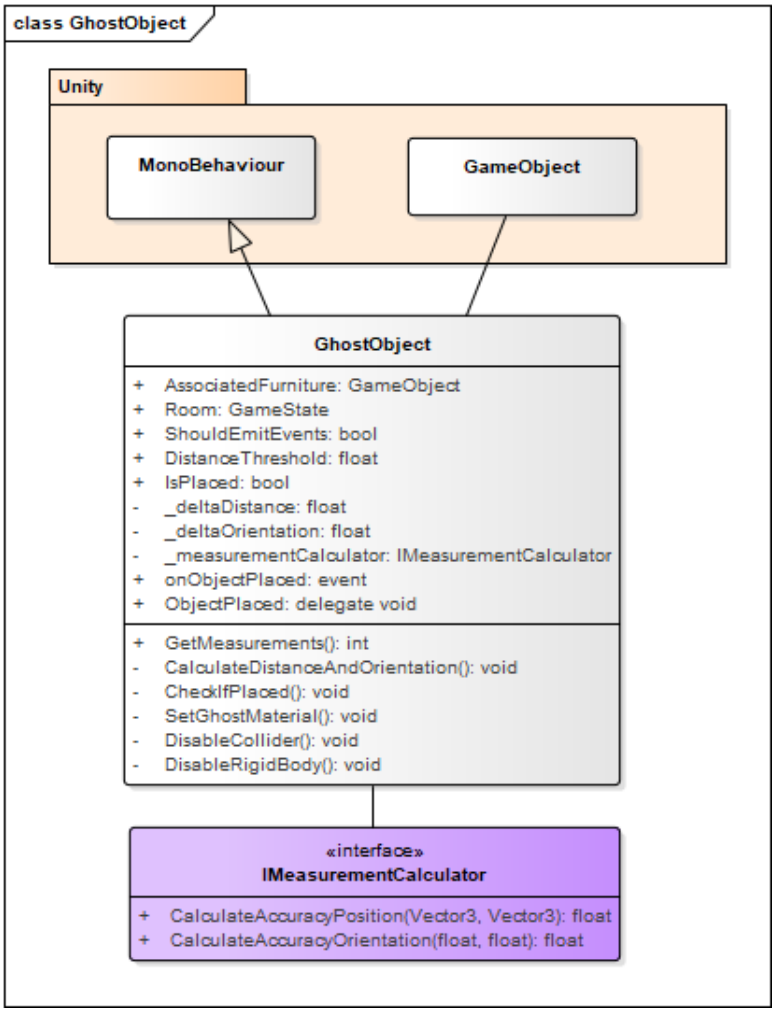


Figure 187: Class diagram for the ghost object system

Method	Functionality
void CalculateDistanceAndOrientation()	This private method calculates the distance and orientation between the ghost object and the associated furniture.
GhostObjectAccuracy GetMeasurements()	This public method returns the distance and orientation as a GhostObjectAccuracy.
void SetGhostMaterial();	This private method sets the material of the GameObject that the ghost object script is applied to.
void DisableRigidBody();	This private method disables rigidbodies that is on the GameObject that the ghost object script is applied to.
void DisableCollider();	This private method disables all colliders that is on the GameObject that the ghost object script is applied to.

Table 32: Description of methods in the Ghost Object script

Since the GhostObject class is a MonoBehaviour class, it has 2 important Unity-specific functions; the Start and Update methods. The Start method is where the object can do initialization, and is where we call the DisableColliders(), DisableRigidbody(), and SetGhostMaterial() methods to ensure that the furniture behaves as we expect.

The update method calculates the distance and orientation between ghost object and associated furniture, and emit events in accordance with the sequence diagram on figure 188.

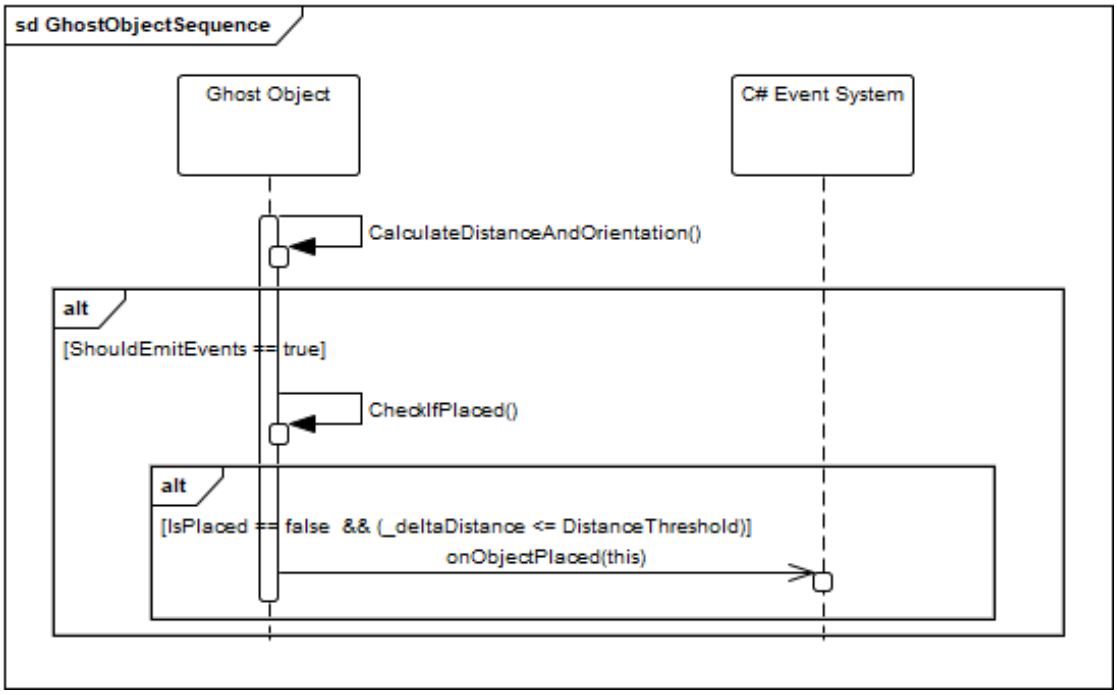


Figure 188: Sequence diagram for the update method of the ghost object class

Appendix G Virtual Reality Prototypes

A lot of the technologies which we worked with during the development of this project were new to us. Because of this, and our focus on learning and building described in our project process of section 5.1, we made extensive use of prototyping throughout the project. The key idea behind this was to gain a proper understanding of what we were about to work with, in order to gain valuable insight into potential problems and possibilities. This helped us remove as much uncertainty and risk as possible.

G.1 Mobile

The very first mobile VR prototype was a simple demo scene provided by Google in their official GoogleVR Unity package. A screenshot of this demo scene can be seen in figure 189.

Figure 189: A screenshot from the initial mobile vr prototype. Here is a simple 3D render of the scene as seen from the headset.

This same scene can be seen from the Unity editor in figure 190.

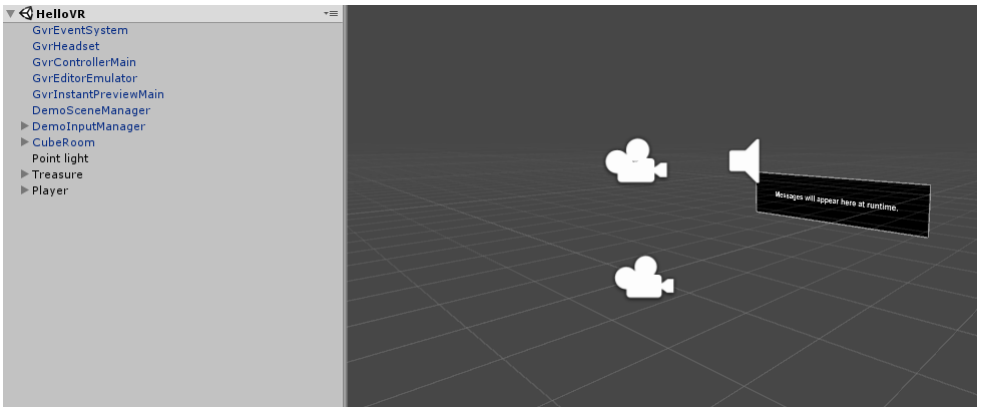


Figure 190: A screenshot from the initial mobile vr prototype. This perspective is from inside the Unity editor.

As can be seen, this is a very primitive scene essentially consisting of nothing but the ability to look around with the HMD.

The purpose of this prototype was simply to gain experience with the build workflow in Unity, when building the game to an Android smartphone.

Doing this prototype was a great learning experience, as some difficulties was discovered regarding the build process from Unity to the smartphone. Poorly documented setup of the Android SDK - which is required when building the game to an Android smartphone - meant that we had to spend quite a bit of time figuring out errors reported from within Unity when building the project. The importance of us discovernig this early was that we were able to spend the required time fixing a central issue to the rest of the project development.

G.2 Installed

We made a quick prototype of our virtual reality application on HTC Vive in order to gain a better understanding of what is required to set it up properly in Unity. A screenshot of this prototype running on a desktop computer with HTC Vive can be seen on figure 191.

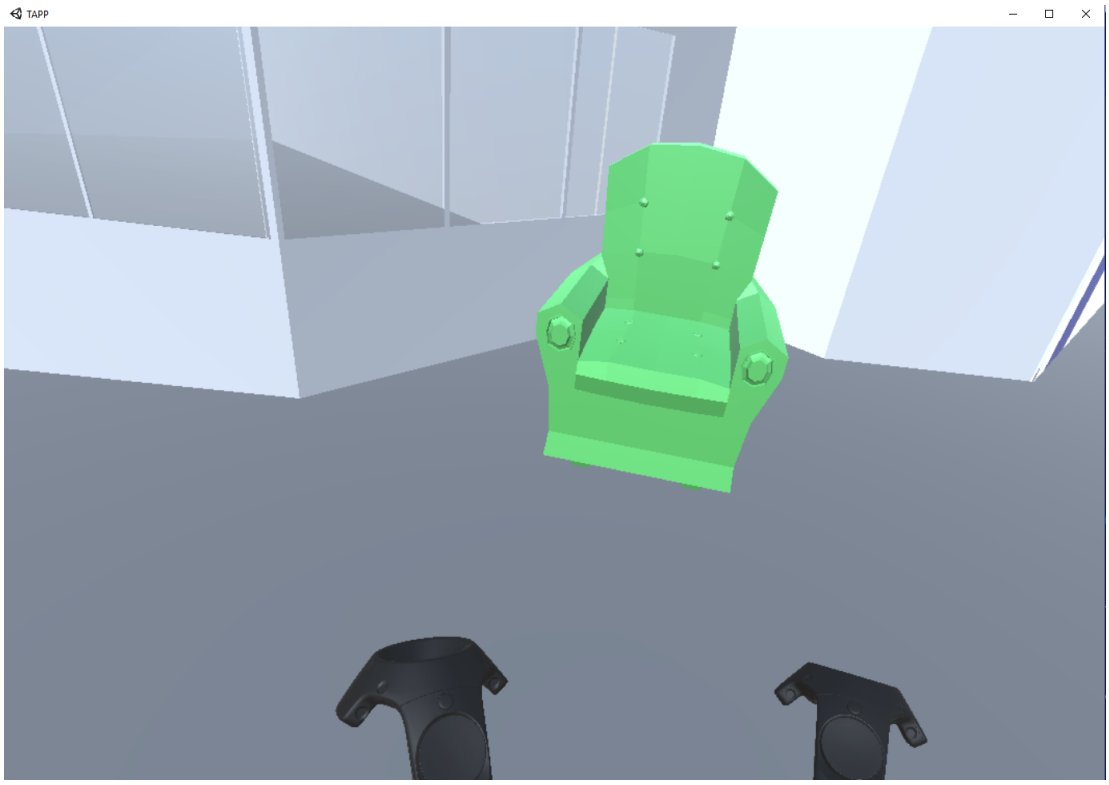


Figure 191: A screenshot from the installed VR prototype

As can be seen from figure 191, the prototype comes with fully modelled HTC Vive controllers tracked in real time. The prototype takes place within an early virtual environment of Orbit Lab.

What we learned from the prototype was the basic requirements for setting up HTC Vive in a Unity project.

First of all, *Valve* supplies the *SteamVR* Unity asset through the Unity Asset Store [82]. By importing this package to your Unity project, you are supplied with documentation on how to implement basic installed VR functionality such as head-tracking and tracking of controllers. Furthermore, the package includes prefabs for these essential things so that it works out of the box in a configuration which Valve recommends. Figure 192 shows a screenshot of the many different folders of prefabs that comes with it.

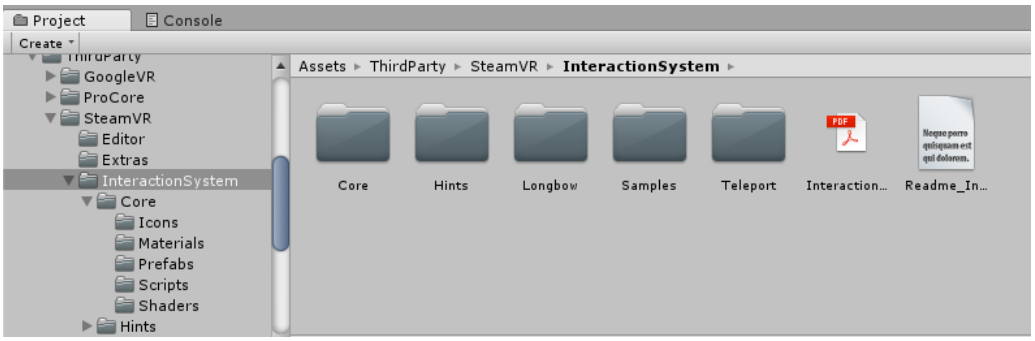


Figure 192: A screenshot of the many folders of provided prefabs that ships with the SteamVR asset

One thing that we did learn from this initial prototype was that the included documentation on the various prefabs was a bit sparse. In the *Core* folder seen in figure 192, for example, SteamVR provides a *Player* prefab which is supposed to be a prefab representing a player in the virtual space, with head-mounted tracking and two tracked controllers. We used this prefab in our prototype, and found out that it is actually configured incorrectly. The prefab consists of a nested, outdated, prefab called *DebugUI*. This can be seen in figure 193.

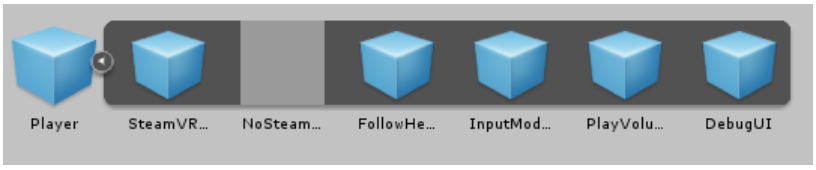


Figure 193: A screenshot of the Player prefab, with its nested prefabs folded out.

The *DebugUI* prefab causes rendering artifacts when playing the game on a desktop computer. The artifact consists of incorrectly projecting 3D models of the environment, resulting in them being rendering as flat 2D objects on the head-mounted display when looked at from specific angles. Tracking down this bug meant that we managed to identify a potentially disturbing bug early on in the process.

G.3 UI Prototypes

We knew that developing the experiments of the project meant that some type of UI would have to be implemented in our virtual reality applications.

Therefore, before implementing our experiments, we made prototypes of how to do UI programming with virtual reality on the installed and mobile platforms. We did this in order to gain a quick understanding of what is required in order to do it, and if we would bump into potential difficulties in doing so.

G.3.1 Mobile

For UI programming with Google Daydream, we made a scene dedicated solely to UI prototyping. Figure 194 shows a screenshot of us playing this scene.

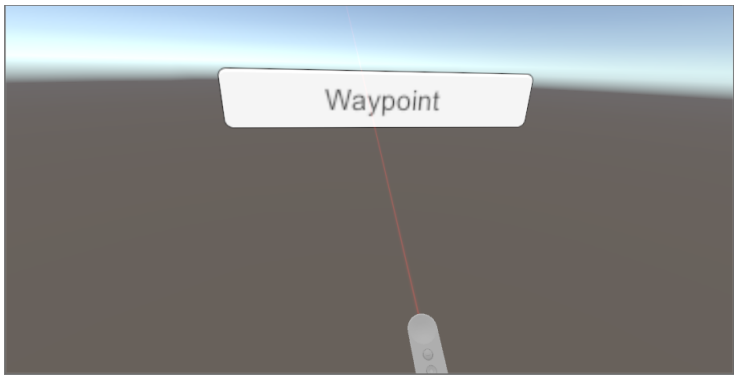


Figure 194: A screenshot from the Google Daydream UI prototype scene.

In figure 194, it can be seen that the scene contains only a button. This was all that was required in order to play around with UI programming, specifically how to detect the player pointing on the button, and how to detect click events.

In figure 195 it can be seen that the button is being highlighted, as the laser pointer is hitting it.

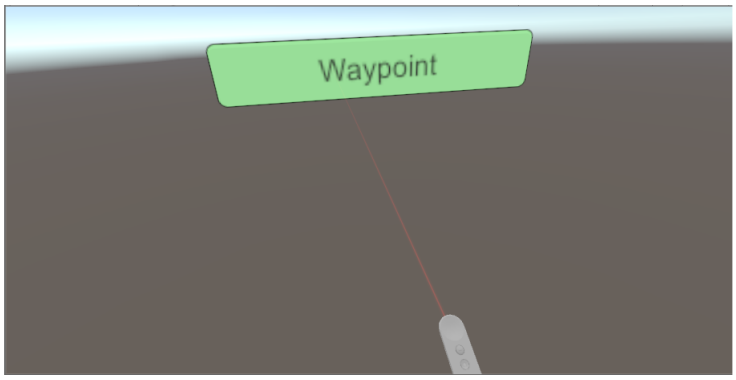


Figure 195: A screenshot from the Google Daydream UI prototype scene. Here, the button is highlighted as the laser pointer is hitting it.

It turned out that doing UI programming with Google Daydream was relatively easy. In order to get buttons to work, you need to follow a specific structure to the GameObjects composing the menu. Essentially, you need a GameObject with the *Canvas* component on. This is a component provided by Unity for UI programming. On this same GameObject,

you also need to add the *GvrPointerGraphicRaycaster* component. This is a component provided by the Google Daydream SDK, and makes sure that the raycast from the Google Daydream controller is detected when pointing on the buttons.

Nested within this GameObject, you then place all the buttons comprising the menu. These GameObjects need no Google Daydream specific components. Instead, they only need the *Button* component provided by Unity.

A screenshot of this structure within one of our Unity scenes can be seen in figure 196.

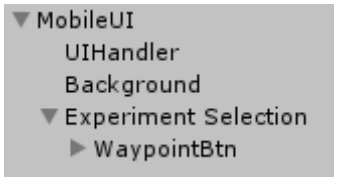


Figure 196: A screenshot of the GameObject structure of a UI menu for Mobile VR.

Here, the *Experiment Selection* GameObject is contains the *Canvas* and *GvrPointerGraphicRaycaster* components. Nested within this GameObject is the *WaypointBtn*, which is the actual button of the UI.

One thing to point out is that we were unable to find any official documentation on doing this. What we had to do was examine demo scenes which came with the *Google VR SDK* package described in section C.2. Some of these demo scenes had menus in them, and from that we were able to deduce the setup required.

We learned that in order to handle click events, the Unity component *Button*, which in this case is added to the *WaypointBtn* GameObject, contains an OnClick configuration, in which you can specify a method on a script that should be executed when a click occurs. A screenshot of this can be seen in figure 197. In this case, we have specified that the method *TeleportationNavigationMode* should be executed once this button is clicked. This method is contained within our own *UIHandler* script.

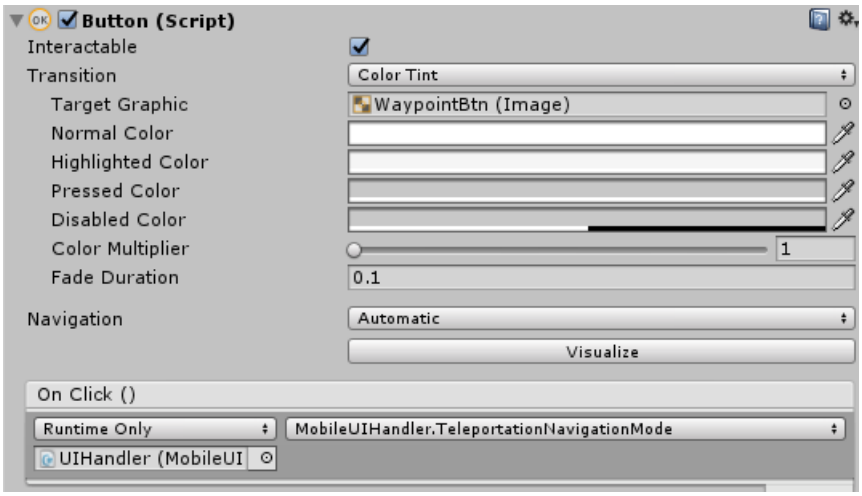


Figure 197: A screenshot of the GameObject structure of a UI menu for Mobile VR.

In the end, we got the prototype to register clicks. This prototype was helpful for us in that we learned the essentials of UI programming for Google Daydream, and found out that it did not have any initial problems. We could thus use this knowledge for the real implementations within the experiments during our project.

G.3.2 Installed

For UI programming with the HTC Vive, we made a scene dedicated solely to UI prototyping.

As we made the UI prototype for Google Daydream first, we had already gained some initial knowledge of the Unity-specific components related to UI programming. Specifically *Button* and *Canvas*. Thus section G.3.1 can be seen for a description of this setup. These components are required in the same way when doing HTC Vive UI programming.

What was relevant to this prototype was figuring out if any HTC Vive specific components would have to be used for UI programming on the desktop platform.

What we learned was that documentation on this aspect was very sparse. The documentation which follows with the *SteamVR Plugin* package described in section C.2 essentially says nothing about UI programming.

Thus, in order to learn about UI programming for the HTC Vive, we had to examine demo scenes that were included with the SteamVR Plugin package. From doing this, we were able to deduce the setup required.

Figure 198 shows a screenshot of us playing the scene.

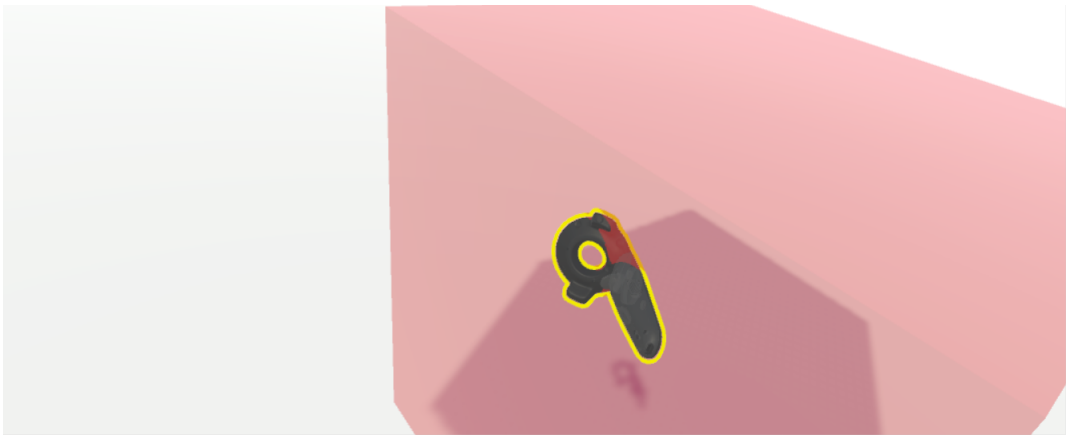


Figure 198: A screenshot of the HTC Vive UI prototype scene. Here, it can be seen that the HTC Vive controller is highlighted as we touch the red cube.

In the end, we learned that UI programming for HTC Vive was relatively simple. In order to interact with buttons, we found that two components were required: *Interactable* and *UI Element*. These components are part of the *SteamVR Plugin* package. They are added to the same GameObject which contains the Unity-specific *Button* component.

G.4 Installed Teleportation Prototype

When it came to virtual reality programming of the HTC Vive, we wanted to explore the possibilities of navigation through teleportation.

G.5 Mobile Remote Object Manipulation Prototype

Before starting the primary *Base VR Apps* development lane for the mobile VR application, we spend a few days developing a prototype for it.

The purpose of this prototype was to get familiar with Unity development on a smartphone in relation to Virtual Reality with Google Daydream. We wanted to gain early experience with potential difficulties associated with the official Unity SDK provided by Google for Google Daydream.

Figure 199 and figure 200 show the final results of this initial mobile VR prototype.

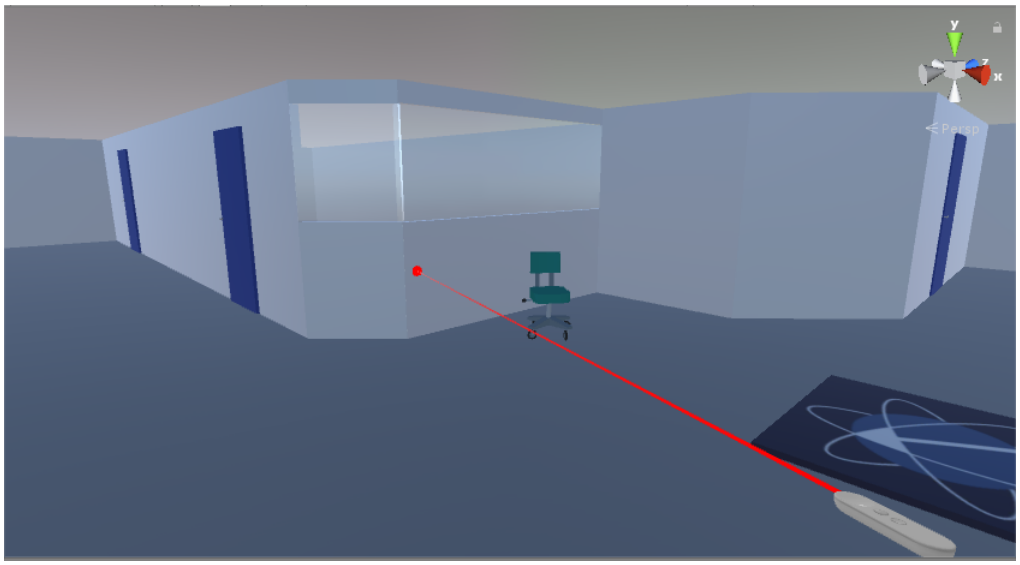


Figure 199: A screenshot from the mobile VR prototype with the Google Daydream controller prefab

On figure 199, a screenshot is shown from the user's perspective. Here, the virtual environment is an early prototypical model of Orbit Lab. In the bottom right corner of the screen it is possible to see the offical Google Daydream controller prefab in use. The 3D model is thus provided by Google, as well as all the script logic associated with orienting and moving the controller in the 3D world.

Also visible is the laser point projecting forward from the controller. It's also possible to see the hit location of this laser pointer on the wall, shown as the red dot on the wall opposite of the controller. The laser pointer and the underlying raycast logic to hit the surrounding virtual environment was scripted developed by the team. The point here was to get a feel of the possibilities and potential difficulties with creating laser pointer logic.

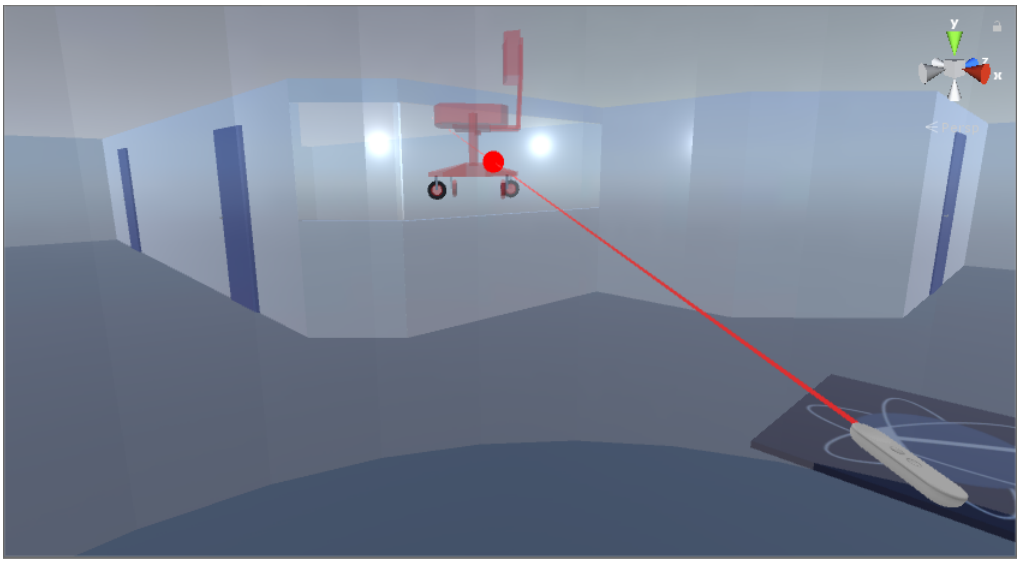


Figure 200: A screenshot from the mobile VR prototype in which the user has selected, and is moving, a furniture

In figure 200, a screenshot is shown of the same user perspective, this time where a furniture is being highlighted by the controller's laser pointer, as the user is pointing on it and moving it. The purpose of this was to develop a quick object manipulation model, to develop an idea of potential difficulties in Unity's 3D programming when coupled with virtual reality.

Overall, this initial prototype gave us a great idea of the baselines workflow of Google Daydream development with Unity. Additionally, we were able to get a feeling for the 3D programming aspect of Unity, and test it out with primary features of our final product.

Finally, from this prototype, we also realized how much work that goes into making remote object interaction which feels right and natural. So, this prototype also helped us determine that it would make the most sense to use Google Daydream's *Elements*, which has a best-practice remote object interaction model that fits well with the requirements for our project.

G.6 Installed Remote Object Manipulation Prototypes

During the development phase of the final experiment, we spend quite a while investigation the possibilities for remote object manipulation on the installed platform.

What we ultimately found was that there were no ready-made professional solutions that covered our requirements for remote object manipulation. This investigation was conducted by prototyping some simple scenes in Unity using various toolkits from the

Unity Asset Store - namely *Virtual Reality Toolkit* and *NewtonVr*. These prototypes will be presented here.

G.6.1 Virtual Reality Toolkit Prototype

Virtual Reality Toolkit (VRTK) is a toolkit on the Unity Asset Store which includes various cross-platform VR components typically used in games. This includes things such as: Teleportation, touchpad movement, 3D controls (buttons, levers, drawers, knobs, etc...), ready-made menus and so forth. Our idea was that this toolkit might also have some ready-made functionality for remote object manipulation which we could use.

Figure 201 shows a screenshot of our remote object manipulation prototype using VRTK.

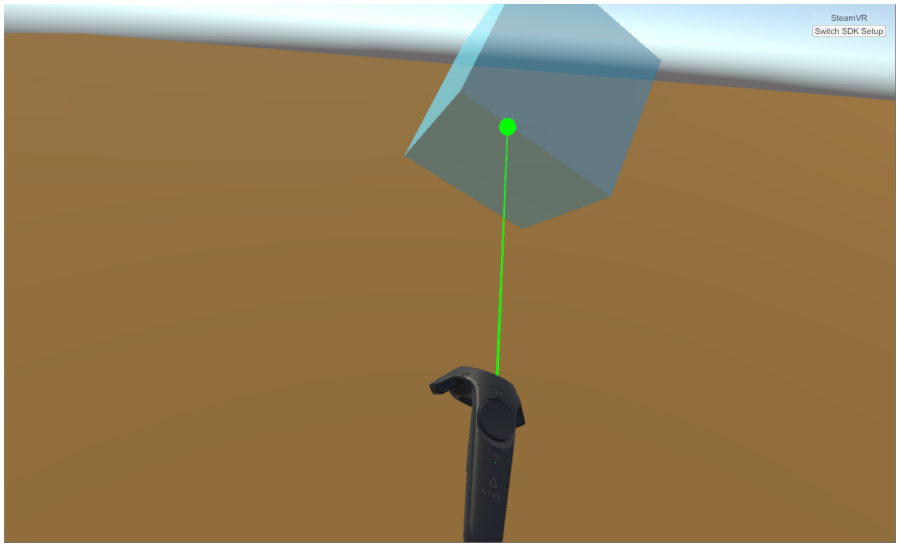


Figure 201: A screenshot from the remote object manipulation prototype using VRTK.

In figure 201 it can be seen that we did get remote object manipulation to work with VRTK. The toolkit included ready-made functionality for it. Unfortunately, the functionality present was too simplistic for our needs. There was no possibility of, for example, rotating the object currently selected using the touchpad. It was also not possible to adjust the distance between you want the selected object. This distance was fixed based on the length between you and the object at the point of picking it up with the laser pointer.

We also experimented with simply extending the already existing functionality of VRTK with our own custom scripts, however we found that this was not practical. The toolkit did not have the needed extensibility that we needed for doing what we wanted with remote object manipulation.

Thus it was decided that VRTK was not going to be a practical option for us.

G.6.2 NewtonVr Prototype

NewtonVr is a cross-platform VR interaction toolkit on the Unity Asset Store. Just like VRTK, it includes various cross-platform VR components typically used in games, however NewtonVr focuses primarily on interaction systems. Our idea was that this toolkit might include the type of remote object manipulation that we needed.

Figure 202 shows a screenshot of a pre-made demo scene included in the NewtonVr Unity package. The purpose of this scene was to demonstrate the various interaction systems provided by the toolkit.

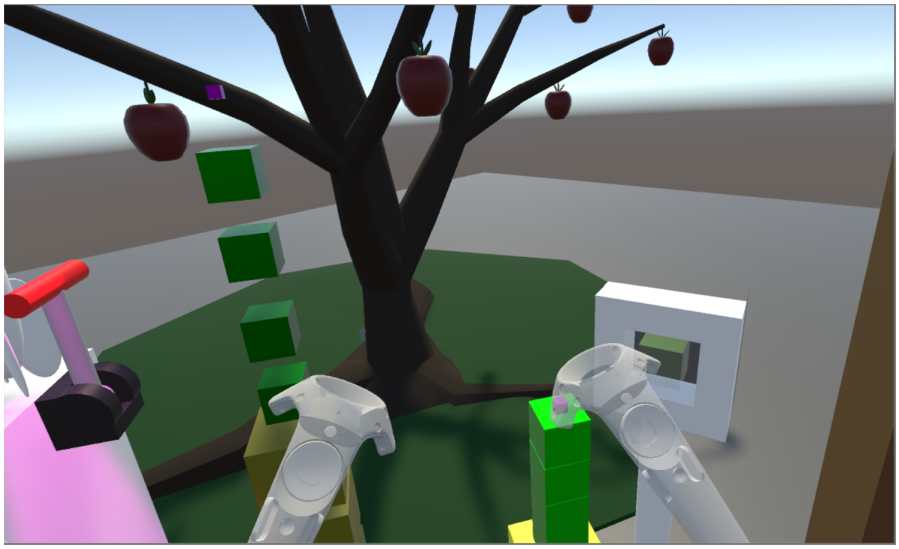


Figure 202: A screenshot of the included demo scene of NewtonVr.

In figure 202, it is possible to see some of the interaction systems provided by NewtonVr, such as the lever seen on the left, or the physics-based apples on the tree which can be plucked. Besides this, the scene also included demonstrations of doors, drawers, menus, and more. Unfortunately, this toolkit seemed to have to focus on remote object manipulation. Therefore, there was no functionality within the package that was deemed fit for our needs.

Thus it was decided that NewtonVr would not be a practical option for our project.

Appendix H Process Description

This appendix will highlight the process used in the project, and comment on some of the tools used in the process.

H.1 The Team

The team consists of two ICT-Engineer students from Aarhus University School of Engineering, who have been working together during many projects before joining forces for the bachelor project.

The team did not spend time to create an official co-operation agreement, but simply agreed on a work schedule, as seen in table 33.

Monday	Tuesday	Wednesday	Thursday	Friday
Part-Time Job	KPU and bachelor work	Part-time Job	AMS and bachelor work	KPU and bachelor work

Table 33: Work schedule for the bachelor project

Here, KPU and AMS refers to courses which both team members had during the semester.

H.2 Meetings and Project Management

The project did not have a single project manager, and the responsibilities for making decisions and pushing the project forward was a joint venture between the team members.

This method of project management has the possibilities of failure, as the project can become unstructured, but due to the high sense of ownership from both team members, the project got carried out without any major issues.

The team would have weekly or bi-weekly meetings with their supervisors, to keep them updated with the progress of the project, as well as gain valuable feedback. The supervisors were used as partners for discussion to bounce off ideas. This was especially helpful as the format of the project process was new for the team.

H.3 Development process

As described in the main report, the team made use of a lean-inspired development process. The build-measure-learn feedback loop helped them to continually improve on their experiments, both from a software and a user experience perspective. This in turn increased the quality of the gathered results from the experiments, and as a whole the team believes the process to be a good choice for their project.

H.4 Project Type

The team have struggled with the type of the project, as it is more of a research-oriented project rather than more conventional product development - something the team has never tried before. It was especially the concept of not having the software of the project

being the actual final answer that the team had troubles coping with. In the end, however, the group learned a lot about the research-driven approach to a project, and also gained a lot of valuable insight into experiment design, with all the careful considerations that follows along with it. Overall, the team felt that the challenge of this new type of project was a great experience with positive learning benefits. It was a good way to get a taste of the academic world.

H.5 Process Tools

To help achieve a better end product, the team made use of different tools to help them facilitate their process. Some of these will be highlighted here.

H.5.1 Time planning

To help structure the project, the team created a timetable, see figure 34, at the beginning of the project. This was meant to help us structure the entire project, and despite our limited knowledge of the field at the time, the group actually followed it through to the end.

Week	Area of Focus
1 - 5	Analysis
3 - 5	Problem Statement
6 - 14	Report Writing
6 - 14	Documentation
6	VR Application Prototyping
7 - 9	Navigation Technique
10 - 12	Object Manipulation
13 - 14	Object Manipulation Experiment Application
15 - 16	Object Manipulation Experiment Execution
15 -18	Report Writing

Table 34: The timetable for the project.

It was a major advantage to have the topics of the project mapped out, as it helped the group time-box the different development tasks, such as the setting up basic VR applications and running the navigation experiment. Due to the well-defined nature of the time-boxing, it was easy for the group to see if they were well ahead of the timetable, or if they, at times, needed to move ahead a bit faster.

H.5.2 Version Control

The team made use of version control in the form of Git and Github in this project. As the team only consists of two members, who often used pair programming, it didn't make much sense to use Git's feature-branching.

H.5.3 Continuous Integration

In order to ensure continuous health of our virtual reality applications, and to manage a build- and test pipeline, we decided to use Continous Integration(CI).

The purpose of CI is to automatically test the codebase for build- or unit-test errors, and provide the developers with a notification and report when tests or builds fails.

As the virtual reality applications were developed using Unity, we decided to use *Unity Teams* [83] as our CI tool. Unity Teams is a commercial subscription-based cloud Service provided by Unity. It provides different features for a development team, such as collaborative features for sharing and syncing projects between multiple team members, and also *Cloud Build*, which is the service we use for CI.

The Cloud Build feature has both an interface for the Unity-Editor, as well as a web-interface, from which the user can get a quick report of the build for both current and previous builds.

Besides offering automatic build and test processes, Unity Cloud Build also makes deployment easier. Successful builds allows developers to download the build as an executable binary, depending on the platform that it build for.

For a more detailed look at the unity cloud build feature, see appendix C.4 - Continuous Integration

H.5.4 Unit Testing

In order to ensure software quality of critical components in the system, we decided to make use of unit testing. Unit testing was a natural addition to the philosophy behind continuous integration which we also chose as a process, described in the previous section.

Unity has built-in support for unit testing via the Editor. Unity scripts can be made which tests code you have written, and these tests will then be exposed through the editor. The Unity Test Runner window from our project can be seen in figure 203.

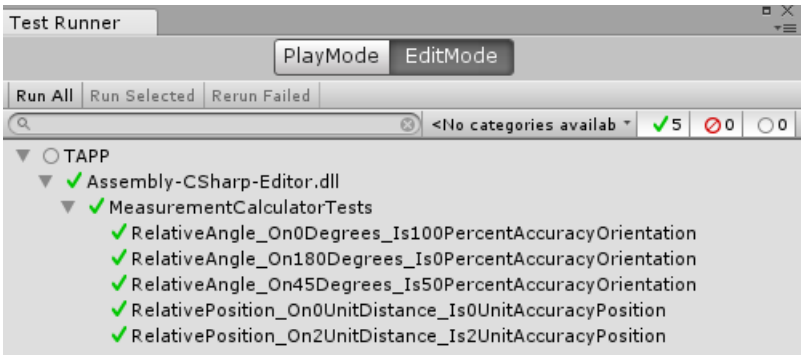


Figure 203: Unity Test Runner window from our project

In figure 203 the unit tests from our project can be seen. From this window, it is then possible to run the tests and get instant feedback on succeeded and failed tests.

In the project, we tested the *MeasurementCalculator* class described in appendix F.5. This was done as the class is responsible for critical measurement calculations in regards to the object manipulation experiment.

Appendix I External Appendices

This appendix will be a lookup table for the external appendices such as the raw database data, the video recordings or the questionaire pdfs, all found in files outside of this document.

I.1 Navigation Questionaire

The questionaire which was used for the navigation experiment can be found in *MiscellaneousFiles/Questionaires/NavigationQuestionaire.pdf*.

I.2 Raw Navigation Questionaire Data

The reply data from the navigation experiment questionaire is split up into two files based on the platform which the experiment was run on (MVR or IVR). Therefore the data can be found in the two files:

MiscellaneousFiles/Questionaires/NavigationExperimentsIVRAnswers.xlsx and *MiscellaneousFiles/Questionaires/NavigationExperimentMVRAnswers.xlsx*.

I.3 Raw Json Navigation Database Dump

The measurement data gathered from users completing the navigation experiment can be found in the json file *MiscellaneousFiles/RawMeasurements/RawDatabase.json*

The measurements are split up into two nodes, *NavigationExperimentInstalled* and *NavigationExperimentMobile*, respective of the platform which the measurement was taken from.

I.4 Navigation Experiment Time Averages

The time average calculations for the navigation experiment can be seen in *MiscellaneousFiles/Calculations/NavigationTimeAverages.xlsx*.

I.5 Object Manipulation Experiment Questionaire

The questionaire which was used for the object manipulation experiment can be found in *MiscellaneousFiles/Questionaires/ObjectManipulationQuestionaire.pdf*.

I.6 Raw Object Manipulation Questionaire Data

The reply data from the object manipulation experiment questionaire can be found in *MiscellaneousFiles/Questionaires/ObjectManipulationExperiment.csv*

I.7 Raw Object Manipulation Database Dump

The measurement data gathered from users completing the object manipulation experiment can be found in the json file *MiscellaneousFiles/RawMeasurements/RawDatabase.json*.

The object manipulation measurements can be found in the json nodes *INSTALLED* and *MOBILE*, depending on the platform you are interested in examining.

I.8 Simplified Object Manipulation Measurements

The simplified object manipulation measurements is a tab called *Raw Data* in the excel file:
MiscellaneousFiles/Calculations/AccuracyAndEfficiencyNumbers.xlsx

I.9 Video Recordings of Object Manipulation Experiment

The video recordings can be found in the attached zip file of *ExperimentRecordings.zip*.

The recordings have been divided into two folders, based on the platform which the experiment was run on (Mobile and Installed). Each video file have been named after the corresponding measurement in the Firebase database.

I.10 Live Experiment Data Smartphone Application Video Demonstration

A video demonstration of the experiment controller Live Experiment Data App can be found in: *MiscellaneousFiles/DemonstrationVideos/LiveExperimentDataApp.mp4*.

I.11 Remote Object Manipulation Video Demonstration

A video demonstrating the remote object manipulation system for IVR can be found in:
MiscellaneousFiles/DemonstrationVideos/RemoteObjectManipulationSystem.mp4.

I.12 Object Manipulation Experiment Analysis and Calculations

The full calculations and further analysis of the raw data from the measurements of the object manipulation experiment can be found in:
MiscellaneousFiles/Calculations/AccuracyAndEfficiencyNumbers.

It is split into three tabs: *Raw Data*, *StudentT Removed Outliers*, and *StudentT Full Data*.

I.13 Executables

The folder: *MiscellaneousFiles/Executables*

Contains the subfolder *TAPP*, which has the PC executable for the object manipulation experiment.

In the executables folder the apk file *LiveExperimentApp* can also be found, which is the Android executable for our Live Experiment Data App.