# Enhancing Virtual Reality Interactions with Modular Peripherals

A Major Qualifying Project Report
submitted to the faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree of Bachelor of Science by:

**Dominic Cascino**
Electrical and Computer Engineering

**Paulo Chow**
Electrical and Computer Engineering

**Project Advisor**:
Professor Xinming Huang

March 2020

Worcester Polytechnic Institute
Department of Electrical and Computer Engineering

# Abstract

Virtual Reality is an immersive and powerful technology which is already changing computing, entertainment, education, and social networking. Modern VR headsets are capable of comfortably delivering high-resolution, high-framerate content and providing fully mobile motion tracking. Consumer VR systems typically consist of a tracked headset and two tracked hand controllers. However, the system format and technology implementation of commercial VR headsets introduce limitations in the user experience. In this project, we identify three specific interaction limitations present in modern VR and devise a hardware solution for each. The three issues we aim to improve are finger presence, two-handed rigid virtual object interactions, and locomotion. The first interaction limitation, finger presence, is the sense of movement control of the virtual hand's fingers when in VR. Another issue arises when interacting with rigid two-handed virtual objects, as the user is typically using two separate controllers. The third limitation is in the ability to move large distances in virtual environments while constrained to a real-world area. The result of this project is a modular and wireless peripheral system that addresses each of the interaction limitations with a purpose-built peripheral. Each peripheral uses a combination of capacitive touch surfaces and general purpose inputs. The three peripherals we developed are the Multi-Digit Input Peripheral, the Hand Grip Position Peripheral, and the Step Detection Peripheral.

# Acknowledgements

We would like to acknowledge and thank:

*WPI for this opportunity*
*Xinming Huang for guidance*
*William Appleyard for supplies*
*Alex Camilo for a key recommendation*
*Our family and friends for support*
*Coffee for energy*

# Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Virtual Reality

Modern Virtual Reality (VR) is an immersive technology that places users in a virtual environment. VR can be used for games, immersive cinema, social platforms, education applications, prototyping visualizations, and other applications. VR systems are currently implemented as head-mounted displays, (HMDs) otherwise known as headsets, along with hand controllers. A VR system may track the head, hand controllers, facial expressions, hand gestures, or eye movements [12]. Two tracking configurations exist for tracking the headset and controllers. In an "outside-in" tracking system, the sensors are static and observe the motion of mobile emitters. Similarly, in an "inside-out" tracking system, the sensors are connected to the mobile system and the emitters are static [28]. Besides the tracking configuration, another differentiating factor is the location of processing. Standalone headsets refer to mobile headsets with on-board processing. One the other hand, tethered headsets are better defined as display peripherals that connect to external processing such as a PC or game console. High quality and immersive experiences can be built with current VR systems, but technical and interaction based issues still arise.

## 1.2 Interaction Limitations in VR

The system format of a headset and two hand controllers introduces limitations in the type of interactions a user may have in a virtual environment. This project identifies three specific user-facing limitations that may be immersion-breaking or discomforting. The **first** is the ability to show a representation of the user's finger movement in VR. This is otherwise known as finger presence, which can increase the immersion of the user and their acceptance of the virtual hands. Within VR, users are not able to fully interact with the environment due to the lack of finger presence provided by VR controllers, limiting common forms of real-world interactions, such as hand gestures and object manipulation. Several methods of VR finger tracking are optical, mechanical, magnetic, and capacitive. Products such as the Valve Index controllers, Leap Motion controller, and the Oculus Touch controllers use one of these methods. Each implementation has limitations in tracking ability, the number of fingers tracked, and cost.

The **second** interaction limitation of VR arises with virtual two-handed objects. Like the real-world, virtual objects consist of different size, texture, and shape. VR systems typically use two separate tracked controllers, which makes interacting with a virtual object with two hands non-trivial. Handling all these different objects with only two controllers makes the interaction with most objects less immersive and affects the user interaction with the virtual environment. Common software solutions are to use a distance threshold from the initial grab points or to not allow objects to be grasped with both hands. The distance threshold method can work, but the hands naturally separate during movement and can create the sensation of puppeting the object, as the user's hands are no longer in the same place as the virtual hands.

The **third** interaction limitation deals with locomotion, where movement within the virtual environment is not initiated by the user's real-world movement. Due to the VR system's limited hardware, the lower part of the human body is not tracked and the user is constrained within a setup area and a maximum distance for tracking. Any action involving this area of the human body is ignored in VR, resulting in a complex task of simulating any form of movement involving the lower half of the body, such as walking. There are two types of solutions to this limitation. The first set of solutions involves software such as teleportation, smooth movement, and climbing, which leads disorientation and motion sickness. The second solution is hardware, such as the VR treadmills and cybershoes, but they are not affordable for the average consumer and do not accurately represent a user's real-world movement in VR.

## 1.3 Proposed Solutions

We developed three peripherals to address the identified interaction limitations of the existing VR systems.

- The **first** peripheral is the Multi-Digit Input Peripheral (MDIP). The MDIP includes a 3D printed cup attached to the Oculus Touch Controller and a cord that straps around the user's hand when in use. It integrates three additional capacitive touch sensors to detect the presence of the user's remaining three fingers, middle, ring, and pinky finger.

- The **second** peripheral is the Hand Grip Position Peripheral (HGPP). This peripheral is composed of various 3D printed segments and a PCB integrated with an array of capacitive touch sensors to simulate the user's manipulation of two-handed rigid virtual objects. Through discrete reading, the array of capacitive touch sensors is able to detect and estimate both of the user's hands along the peripheral.

- The **third** peripheral is the Step Detection Peripheral (SDP). The SDP consists of a 3D printed base, a force sensor, and a strap that goes around the user's foot. The force sensor is sewed into the strap and hidden with an extra layer of elastic band to cover it. It is used as a button input to detect the user's steps in the real-world environment to simulate walking in the virtual environment.

# Chapter 2

# Background

## 2.1 Modern Virtual Reality

Virtual Reality (VR) technology aims to provide the user with a designed experience through artificial sensory stimulation. These experiences may be games, immersive cinema, social platforms, education applications, or prototyping visualizations. A common form of VR is the head-mounted display (HMD), made possible by the recent advances in technology such as microprocessors, displays, and sensors. A VR system consists of input, output, and computation. The outputs are typically a display and an audio source such as headphones, while the inputs are the position and orientation of the user [12]. A VR system is comprised of a tracked headset and tracked controllers.

The location of processing divides VR headsets into two main categories, PC VR and standalone VR. Devices such as the Oculus Rift S, Valve Index, Pimax headsets, Windows Mixed Reality, and Playstation VR can be considered display peripherals driven by a PC or game console through a wired or wireless connection. Standalone VR headsets do not require external processing and house the entire system in the headset. This began with smartphone-based VR headsets such as the Google Daydream and Samsung Gear VR, which are limited in tracking, computation, and thermal aspects. A standalone device that overcomes these limitations is the Oculus Quest, displayed in Figure 2.1.

The Oculus Quest is the most recent headset from the company, Oculus, founded by Palmer Luckey in 2012. From the launch of a development kit Kickstarter to an acquisition by Facebook, Oculus is credited with reviving VR in modern times [12]. After introducing two development kit headsets, Oculus released its first consumer PC VR headset, the Oculus Rift. One of Oculus's most recent products, the Oculus Quest has the same tracking capabilities as its PC VR equivalent, the Oculus Rift S, but does not require a PC to operate. However, at Oculus Connect 6, Oculus introduced a feature called Link, which allows the Quest to act as a PC VR headset through an USB 3 cable [16]. Link transforms the Quest into a powerful hybrid headset able to run applications natively or stream content rendered on a PC.

Figure 2.1: The Oculus Quest VR headset and controllers [15]

### 2.1.1 Tracking Methods

When a user of a VR system moves in the real world, the system must output the result of this movement in the virtual image and do so with minimal latency. The movement may be looking around a virtual environment, crouching, leaning, walking, or reaching out towards an object, all of which are a change in position and orientation of the headset or hand controllers. This requires the VR system to measure the movements of the user. A VR system may track the head, hand controllers, facial expressions, hand gestures, or eye movements [12]. There also exist solutions for tracking the entire body, such as the VIVE Trackers or the TESLASUIT. These however are beyond the normal consumer use case for VR, in which a system will usually have a headset and two hand controllers.

Tracking technologies for human motion tracking may be mechanical, magnetic, optical, acoustic, or inertial [28]. Another differentiating factor is the configuration of the sensors and emitters. In an “outside-in” tracking system, the sensors are static and observe the motion of mobile emitters. Similarly, in an “inside-out” tracking system, the sensors are connected to the mobile system and the emitters are static [28]. In the case of the “inside-out” configuration, the emitters may be recognizable static features in the world. The two tracking configurations are shown in Figure 2.2

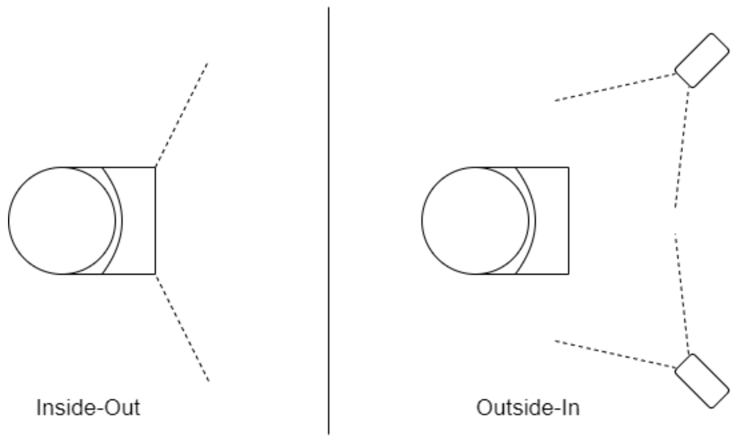


Figure 2.2: Motion tracking configurations

A tracking system for VR typically uses an inertial measurement unit (IMU) in combination with other optical components such as cameras. These components have become affordable and small, as they have been optimized by the smartphone industry. The IMU is used for accurate and highly sampled orientation tracking but cannot measure position alone due to a high drift error rate [12]. To accurately track the position and orientation of an object, another tracking system must supplement the IMU data. In the case of systems such as the Oculus Quest, Oculus Rift S, Valve Index, and HTC Cosmos, the supplementary tracking system uses optical elements such as cameras, lasers, or photodiodes.

For example, the Valve Index Base Stations use IR lasers that rotate at 100Hz to track photodiodes on the headset and controllers [27]. The current Oculus products, the Rift S and the Quest, use Oculus Insight for headset and controller tracking. A headset with Oculus Insight uses 4 or more ultra-wide-angle cameras and computer vision algorithms for tracking [15]. Oculus Insight computes a 3D map of the user's surroundings in real-time and uses simultaneous localization and mapping (SLAM) algorithms for tracking [6]. A visualization of the feature identification used to create the 3D map is illustrated in a visualization in Figure 2.3. The cameras on the headset optically track the controllers as well, which are constellation-based controllers [13]. That is, an array of active IR LEDs is placed within the tracked device behind IR sensitive plastic [12]. A visualization of the constellation-based controller tracking is shown in Figure 2.4.

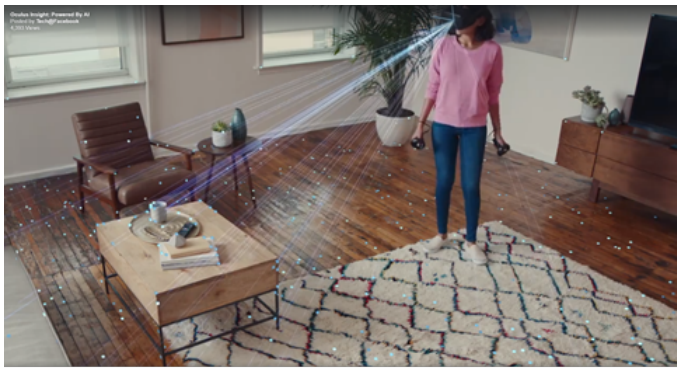


Figure 2.3: Oculus Insight headset tracking visualization [6]

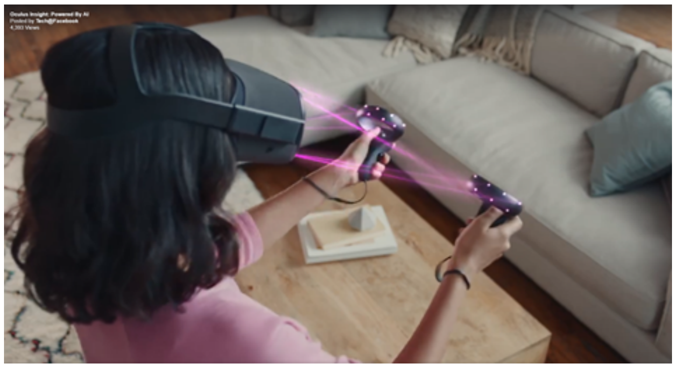


Figure 2.4: Oculus Insight controller tracking visualization [6]

## 2.2 Interaction Limitations

Virtual Reality is a relatively new technology. Although it is convincing enough to human sight and hearing, it has limitations that affect immersiveness and interaction. There are several companies developing new technologies that address interaction issues in VR. The following sections present relevant info regarding the particular interaction limitations addressed by this project, followed by the evaluations of current technologies and solutions that aim to solve these issues.

### 2.2.1 Finger Presence

For various games and social VR applications, the control of hands and fingers and the presence of haptics are important to interact with the virtual environment. Within VR, users are not able to fully interact with the environment due to the lack of finger presence provided by VR controllers. The absence of finger presence limits the use of hand gestures and manipulation of virtual objects. This is a problem in the user's immersiveness in VR since it is an important part of daily human interaction. There are several solutions to this problem, such as the optical finger tracking solution in the Leap Motion and the Oculus Quest, the magnetic solution presented by the HaptX glove, the mechanical solution shown in the Dexmo and the Wolverine, and the capacitive touch solution in the Oculus Touch and the Valve Index Controllers.

The Leap Motion is a device that uses optical tracking to capture the movements of hands and small objects. An example of its use is shown in Figure 2.5 where the Leap motion tracks the movement of a hand. The Leap Motion contains three infrared (IR) light emitters and two IR stereo cameras. The IR emitters from the Leap Motion flood the scene with IR lights [22]. The device reads the data and sends it to the Leap Motion tracking software through USB. Since the Leap Motion tracks infrared lights, it produces grayscale images of the light intensities, shown in Figure 2.6. Then, a software in the computer, Leap Motion Service, processes these images, analyzes the images, and reconstructs a 3D representation of the scene [2]. The tracking layer extracts the tracking information, containing palm and finger position, direction, and velocity [22]. The tracking algorithm interprets the data and infers the position of the objects shown on the scene [2]. The detection accuracy of the Leap Motion is about 200$\mu$m, but with occlusion, the tracking data quality is affected [22]. Occlusion is the most severe issue with optical tracking methods, as the fingers are often occluded by the hands and the hands may be blocked by other objects in the environment.

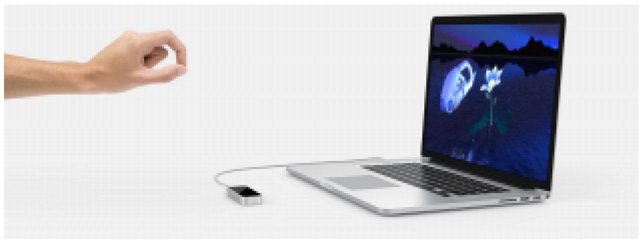


Figure 2.5: Leap Motion connected to a computer and tracking hand [22]

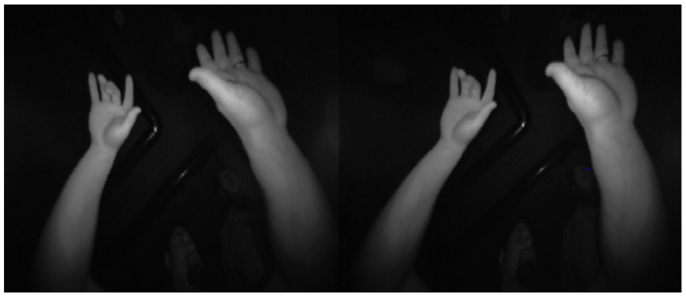


Figure 2.6: The Leap Motion produces grayscale images due to infrared tracking [2]

One of the latest updates released by Oculus on the Oculus Quest is the optical finger tracking released in December 2019. This update allows the Oculus Quest to track the position of the user's hands and fingers without the need for outside external sensors or cameras and tracked controllers. The hand-tracking system uses deep learning and model-based tracking to understand the position of the user's hands and fingers by using the monochrome cameras integrated within the Oculus Quest, shown in Figure 2.7. This system uses deep neural networks to predict the position of the user's hands and joints to reconstruct a 26 degree-of-freedom pose of the user's hands and fingers [8]. An example of the reconstruction of the user's hand model is shown in Figure 2.8.

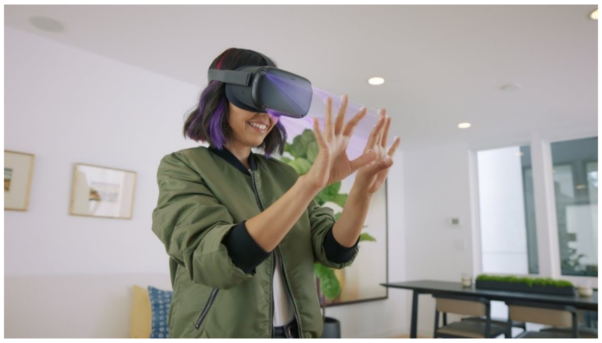


Figure 2.7: The Oculus optical finger tracking uses the headset's cameras to track the user's hands and fingers [8]

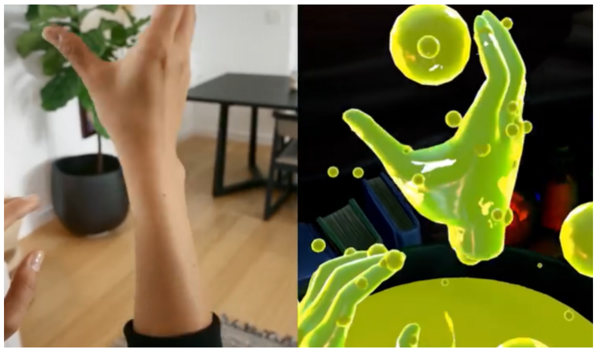


Figure 2.8: The user's hands and fingers are modeled by predicting the position of the user's hands and joints [8]

The HaptX glove is a device that provide precise haptic feedback and tracks finger movements by using magnetic sensors. The HaptX glove presents a smart silicon-based textile and integrated air channels that contain 130 pneumatic actuators throughout the entire glove to provide tactile feedback on the user's hand. It also contains an exoskeleton that provides resistive force feedback up to four pounds of force to each finger. The HaptX glove can track 6 DOF through its integrated motion capture that allows occlusion-free interaction [9]. An example of the HaptX glove can be shown in Figure 2.9.

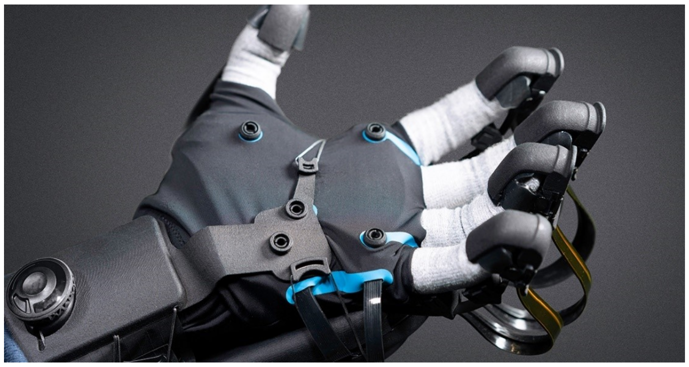


Figure 2.9: The HaptX glove [9]

The Dexmo is a wireless mechanical exoskeleton that provides both force feedback and motion capture. The most recent version of the Dexmo, shown in Figure 2.10, features servomotors that allow variable force feedback, which is limited to one DOF per finger [19]. It contains two rotational sensors that T the user's hand movement to build a virtual hand model corresponding to the user's hand motion. This device imitates the sensation of touching objects by using mechanical brakes to apply force to the user's fingertips [20].

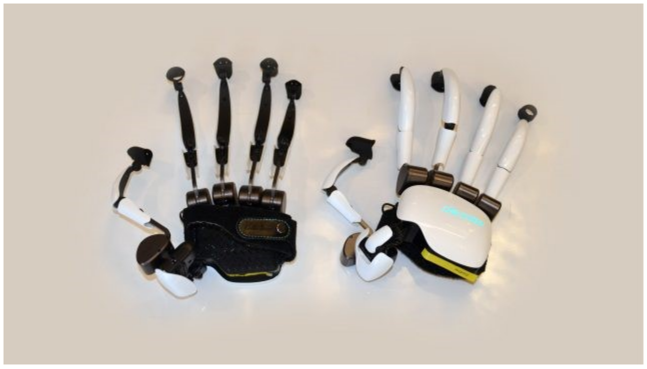


Figure 2.10: The Dexmo Gloves [21]

The Wolverine is an exoskeleton device that provides haptic feedback by simulating the grasp of virtual rigid objects. Sensors are integrated within the device to provide feedback control and user input by tracking the position of each finger and the overall orientation with an IMU. The Wolverine generates force between the thumb and three fingers from the hand to simulate the handling of any object. It uses a braked-based locking slider where the thumb is connected to the index, middle, and ring fingers through connected rods [1]. Figure 2.11 provides an example where the Wolverine is used to simulate the grasp of mug.

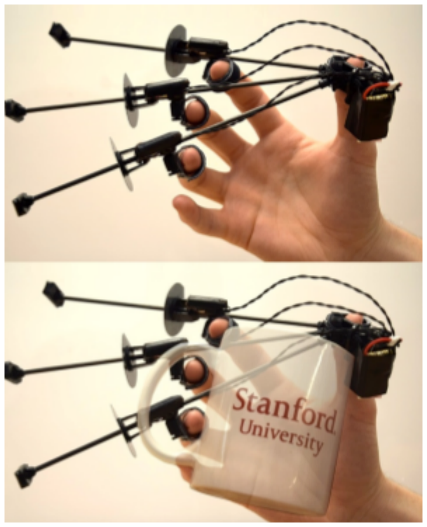


Figure 2.11: The Wolverine simulates the grasping of a mug [1]

The Oculus Touch controllers are tracked controllers that contain capacitive sensors that can detect the presence of the user's fingers. The most recent version of the Oculus Touch controllers used for the Oculus Quest and the Oculus Rift S, shown in Figure 2.12. Each controller includes an analog stick, a trigger for the index finger, a trigger for the middle finger, and two face buttons. The Oculus Touch controllers contain capacitive sensor in all its inputs except for the trigger for the middle finger. In other words, the Oculus Touch controller is only capable of detecting the presence of the thumb and the index finger of the user. An array of active IR LEDs is placed within the controllers to track its position [12].

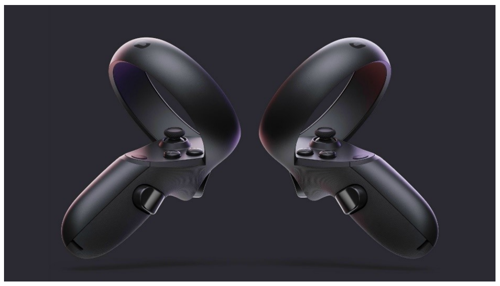


Figure 2.12: The Oculus Quest/Rift S controllers [15]

The Valve Index controllers use the same capacitive sensor solution as the Oculus Touch controllers. Unlike the Oculus Touch Controllers, the Valve Index Controllers contain capacitive sensors to detect the presence of all the user's fingers, which includes the pinky, ring, and middle fingers. A significant difference between the Oculus Touch and Valve Index controllers is that the Oculus Touch controllers uses the trigger for the middle finger to grab objects in VR, whereas the user can grab the objects with the Valve Index controllers by squeezing the grip. Figure 2.13 shows the Valve Index controllers.

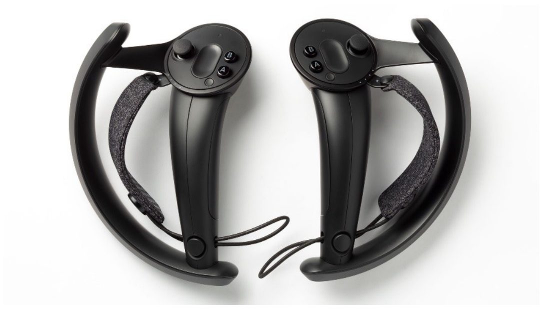


Figure 2.13: The Valve Index controllers [24]

### 2.2.2 Two-Handed Rigid Virtual Object Interactions

One of the most important aspects of user interaction within the virtual world is the manipulation and handling of virtual objects. Objects consist of different size, texture, and shape. Handling all these different objects with only two controllers makes the interaction with most objects less immersive and affects the user interaction with the virtual environment. This problem is most common in two-handed objects where people use both hands to interact with it. There have been a few solutions to deal with this problem. These solutions include software solutions, such as using one or two hands to handle virtual objects, and hardware solutions, such as the Haptics Link and the Dragon:on VR Controller.

A software solution to this problem includes the use of one hand to handle two-handed rigid virtual objects. By using only one hand, this would require the user to handle all objects with only one hand, being an easy and simple solution. If the user would use both hands to handle a two-handed rigid object, the object would only be attached on one hand under any circumstances. The second software solution to deal with two-handed rigid virtual objects is using both hands. This involves the use of a distance threshold relative to a point within the virtual object. The user would only be able to handle an object with both hands if both hands are within a distance between each other. If the distance between both hands is past the threshold, the object would be attached to only one hand instead of both. An example of this software solution can be shown in Figure 2.14, where a player within the game Echo VR is only able to handle an object with two hands if both hands are within a distance between each other. As one hand moves farther apart from the other hand, the object is affixed to only one hand.

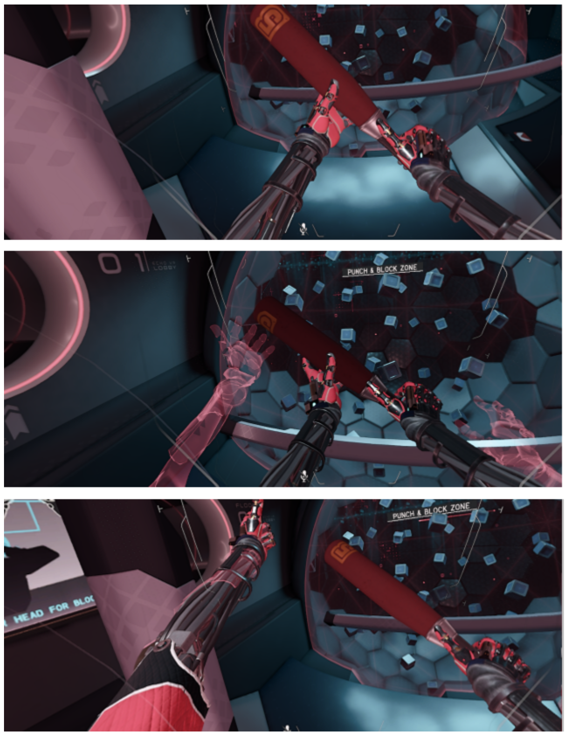


Figure 2.14: An example of virtual two-handed object interaction in Echo VR

One of the hardware solutions to deal with two-handed rigid objects is presented by the device Haptic Links. Haptic Links is a device that is electro-mechanically actuated by using physical connections between two controllers to simulate the stiffness and shape of different virtual objects and interactions. It locks both controllers depending on the configuration, constrain certain DOF or motion, and dynamically set the stiffness of the device. The type of configurations includes: Chain, Layer-Hinge, and Ratchet-Hinge. The Chain configuration uses an articulated chain composed of ball-and-socket elements with a cable along the length the of the chain and attached to a linear actuator on each end, extending or retracting the cable to increase or decrease the friction at each joint [23]. Figure 2.15 displays the elements that compose the Chain configuration. The Layer-Hinge configuration uses ball joints and hinge connected by rods to control the distance and rotation of both controllers. The hinge consists of a series of layers that are compressed by a bolt threaded through the layers to increase the friction of the hinge. This configuration has three distinct points of actuation that can lock a particular DOF based on the motion of the controllers [23]. Figure 2.16 displays the organization of each component that form the Layer-Hinge configuration. The Ratchet-Hinge configuration consists of ball joints and a dual-ratchet mechanism that contains two pawls set against a central gear at opposite angles. The Ratchet-Hinge configuration allows force-feedback configurations, but unlike the Layer-Hinge configuration, it provides no stiffness capability between the controllers [23]. Figure 2.17 provides a layout of the Ratchet Hinge configuration.

Another hardware solution is the Drag:on VR Controller. The Drag:on VR controller is a device that uses a dual folding fan-like system that outputs different configurations based on the surface

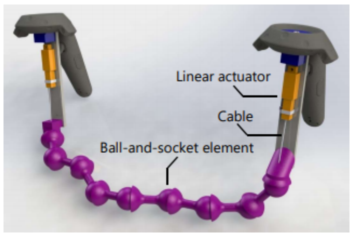


Figure 2.15: The chain configuration [23]

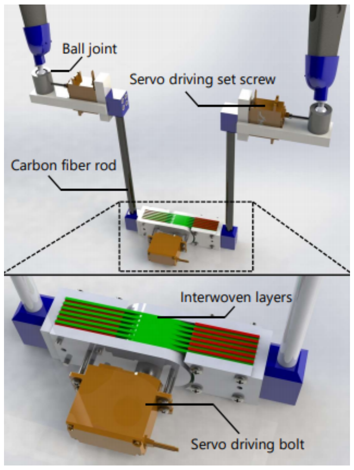


Figure 2.16: The layer-hinge configuration [23]

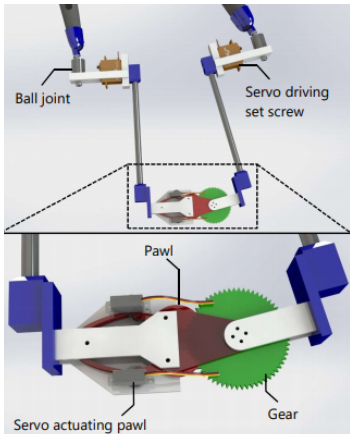


Figure 2.17: The ratchet-hinge configuration [23]

area adjusted by each fan providing a dynamic passive haptic feedback based on air resistance and weight shift [29]. It provides different levels of haptic feedback based on the change in drag and rotational inertia provided by the controller [29]. Figure 2.18 shows the different configuration outputs of the Drag:on VR controller.

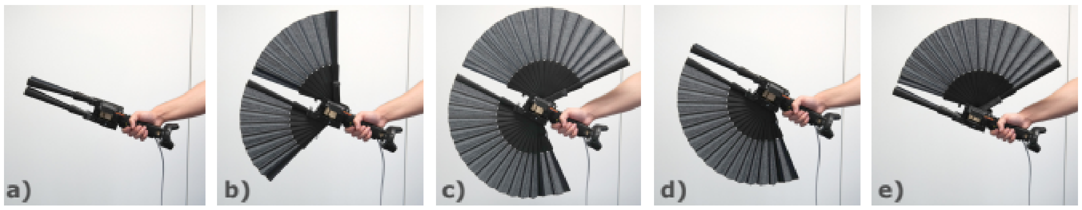


Figure 2.18: Different Output Configurations of the Drag:on VR controller [29]

### 2.2.3 Locomotion

In the VR industry locomotion is defined as "Acceleration, rotation, or movement not initiated by real-world movement of a user" [14]. Users of VR are limited by the dimensions of the setup area and the maximum distance for tracking. The maximum tracking distance is especially important for headsets that require external sensors or emitters, such as the Valve Index. The Valve Index Base Station 2.0 tracking devices can track within a 10 square meter area when using 4 Base Stations [27]. The limitations in physical and tracking area dimensions means that VR applications must implement a form of locomotion to move through a vast virtual environment. Locomotion methods can be differentiated by their use of external hardware.

Locomotion methods that use only the data from the headset and controllers are the most common form of locomotion. Some forms of locomotion are teleportation, smooth movement, climbing, and jogging on the spot [14]. Teleportation is a comfortable movement method in which the user indicates a location with a visual marker and teleports to that destination. Teleportation avoids motion sickness inducing acceleration, but significant distance changes may be disorienting [18]. Teleportation also has the downside of not feeling like realistic and immersive movement [14]. Smooth movement is another common form of locomotion in which the user provides a speed input from the controllers and moves in the direction of either the headset facing direction or hand facing direction. Although more immersive, the smooth camera movement creates a disconnect between the visual and vestibular system, potentially inducing motion sickness. Climbing or grabbing the world is a situation specific method that is immersive and natural feeling but is not a general solution for moving around a virtual world [14]. Jogging in place involves physical movement to convince the vestibular system that actual walking is taking place through the small accelerations of the head while walking or jogging in place. The direction of movement is determined from a joystick or similar input of the controllers [14]. Another method of moving in VR is room-scale movement. Although technically not considered locomotion, room-scale movement is the lack of locomotion, in which the user physically walks in the play area. This is the most immersive but distance limited form of movement in VR [14]. Additional solutions to VR locomotion involve external hardware other than the headset and controllers. The Cybershoes are mobile treadmill devices worn on the feet. Each device has a roller for measuring foot movement on the ground and an IMU for estimating foot orientation. The user must sit down on a swivel stool and use an appropriate carpet underneath the stool [3]. A user wearing the Cybershoes in a VR game can be seen in Figure 2.19.

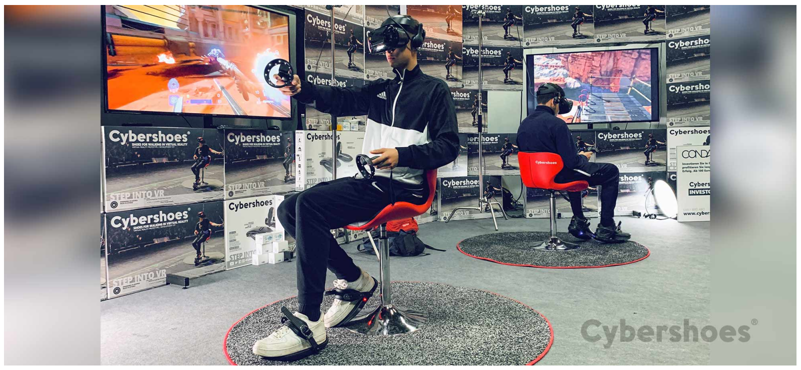


Figure 2.19: VR users wearing the Cybershoes [4]

Another class of VR movement devices is treadmills, which are either active or passive. Passive VR treadmills have curved, low friction surfaces for walking and a harness to hold the user upright. Such devices are the KAT Walk and the Virtuix Omni, as shown in Figure 2.20. Active VR treadmills are those that counter user motion with a moving platform. The Infinadeck is an omnidirectional treadmill that does not require a harness and allows for a natural walking movement [10]. A user on the Infinadeck can be seen in Figure X. Note that VR treadmills, whether active or passive, are marketed towards the enterprises and are out of reach of the average consumer in terms of size and cost.

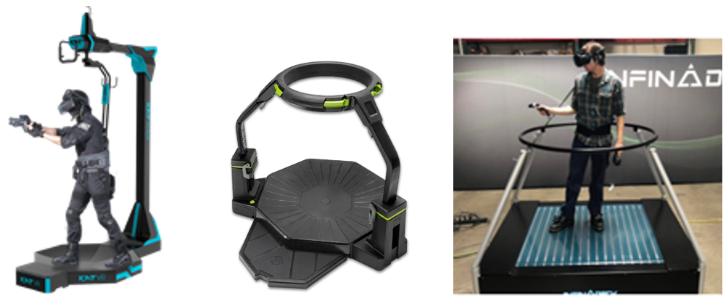


Figure 2.20: The KAT Walk (left) [11], Virtuix Omni (middle) [17], and Infinadeck (right) [10]

# Chapter 3

# Technical Design

To solve interaction limitations with current VR external hardware devices are needed to augment the capabilities of the VR system. In this chapter we detail our solution, a wireless and modular peripheral system that connects to the Oculus Quest VR headset. We targeted three specific interaction problems to address, which are finger presence, two-handed object manipulation, and locomotion. The peripherals may be rigidly connected to the controllers for tracking and each use a main module for computation and communication. Each peripheral uses the same sensing technologies which are capacitive touch and button-like inputs. The input types are used in various ways and customized for each application. We use the Unity game engine to develop demo applications for each peripheral.

## 3.1 Core Module

The core module houses the power, sensing, and communications for each peripheral. Adhering to the design paradigm of modularity, the core module is reused for each peripheral. Most of the peripherals require left and right versions, so two identical core modules are used in each peripheral. A modular system is advantageous because the same core module and sensing capabilities can be reused by peripherals that solve many different interaction issues. This also opens up the possibility for user-created peripherals.

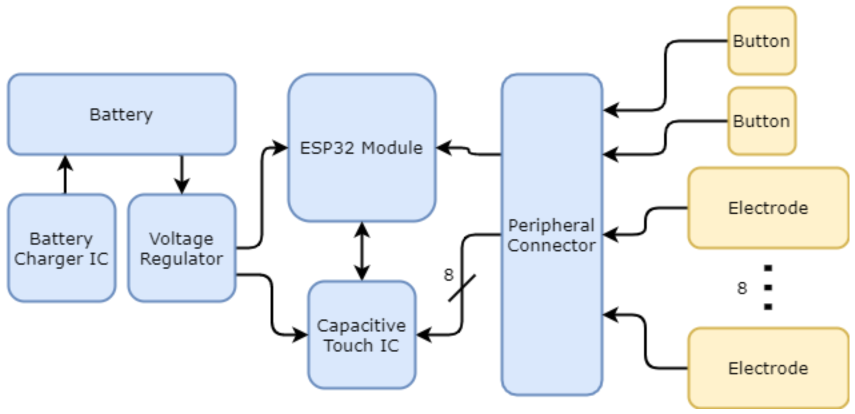


Figure 3.1: System level diagram of the core module

The system level of the core module is displayed in Figure 3.1. Its schematic can been in Appendix A. Starting with the power system, each module is powered by a small 400mAh lithium ion polymer battery which is regulated down to 3.3V for the ESP32 module and other components. For processing and BLE communications we use the ESP32-WROOM-32D module from Espressif Systems, detailed in Figure 3.2.

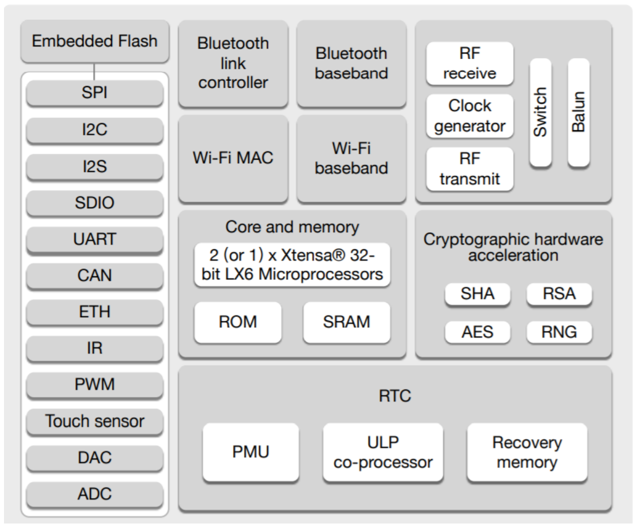


Figure 3.2: ESP32 functional block diagram [5]

The ESP32 module is also responsible for configuring and reading data from the capacitive touch IC. The core module supports up to eight capacitive touch inputs and two general purpose inputs (GPI). These inputs as well as a debug port are routed through the peripheral connector. The peripheral connector is the standard connection to each peripheral, though each peripheral may not use all of the inputs. Using Autodesk Eagle, a PCB layout design tool, we prototyped the board for the core module and arrived at the device in Figure 3.3. The board is a small 66 by 24 mm two-layer board manufactured at Oshpark. It features a USB micro-b connector for charging and a series of pin headers for the peripheral connector. The peripherals themselves feature a small pcb and female pin header connector, shown in Figure X, in which each core module attaches to. The connector PCB has pads exposed for the electrode and button wires on the bottom. Depending on the peripheral, the pads may not be used, which allows for multiple peripherals to use the same connector but for customized input types.

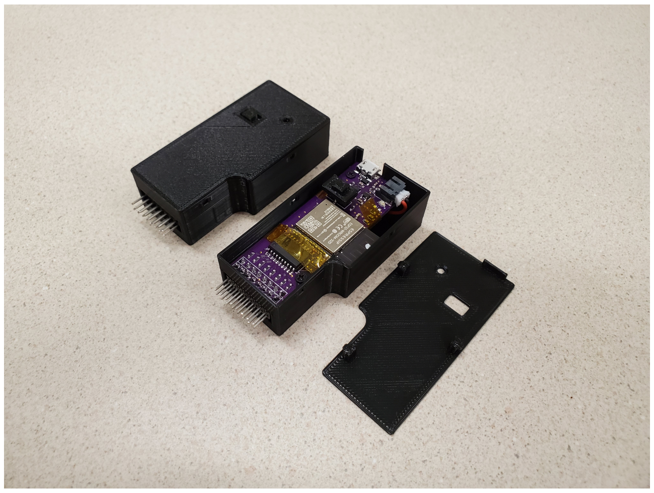


Figure 3.3: The core modules shown in housing cases

### 3.1.1 Software Implementation

We developed an embedded BLE application on the ESP32 with the ESP-IDF from Espressif Systems. The ESP32 uses FreeRTOS, a popular and open source real-time operating system for microcontrollers that is distributed under the MIT license [7]. The application configures the module as an HID gamepad. Using HID for the core module allows it to be easily recognizable as an input device and does not require any BLE specific custom application on the Oculus Quest. Since the Oculus Quest runs Android, HID gamepads, keyboards, and mouses are natively supported. The final version of the core module has 10 inputs, specifically 8 capacitive touch inputs and 2 buttons. Earlier versions of the core module board supported 20 inputs broken down into 16 capacitive touch inputs and 4 buttons inputs. When developing peripherals with the 20 input core module, the latency of the inputs was noticeable relative to the native Touch controller inputs. However, the final core module with only 10 inputs does not exhibit noticeable latency. In HID, the report descriptor describes to the host the type of input device, type of inputs, and number of inputs. For the core module we designed the report descriptor in Figure 3.4.

```
1  static const uint8_t hidReportMap[] = {
2      0x05, 0x01,  // Usage Page (Generic Desktop)
3      0x09, 0x05,  // Usage (Gamepad)
4      0xA1, 0x01,  // Collection (Application)
5      0x85, 0x01,  // Report Id (1)
6      0xA1, 0x00,  //   Collection (Physical)
7
8      0x05, 0x09,  //     Usage Page (Buttons)
9      0x19, 0x01,  //     Usage Minimum (01) - Button 1
10     0x29, 0x02,  //     Usage Maximum (2) - Button 2
11     0x15, 0x00,  //     Logical Minimum (0)
12     0x25, 0x01,  //     Logical Maximum (1)
13     0x95, 0x02,  //     Report Count (2)
14     0x75, 0x01,  //     Report Size (1)
15     0x81, 0x02,  //     Input (Data, Variable, Absolute) - Button states
16
17     0x75, 0x06,  //     Report Size (6)
18     0x95, 0x01,  //     Report Count (1)
19     0x81, 0x01,  //     Input (Constant) - Padding or Reserved bits
20
21     0x05, 0x09,  //     Usage Page (Buttons)
22     0x19, 0x01,  //     Usage Minimum (01) - Button 1
23     0x29, 0x08,  //     Usage Maximum (8) - Button 8
24     0x15, 0x00,  //     Logical Minimum (0)
25     0x25, 0x01,  //     Logical Maximum (1)
26     0x95, 0x08,  //     Report Count (8)
27     0x75, 0x01,  //     Report Size (1)
28     0x81, 0x02,  //     Input (Data, Variable, Absolute) - Button states
29     0xC0,        //   End Collection
30     0xC0,        // End Collection
31 };
```

Figure 3.4: Core Module HID Report Descriptor

This report descriptor sets the usage of the ESP32 as a gamepad, meaning any connecting device will use it as an gamepad input device. The series of lines from 8-15 detail the two button inputs as one bit each, followed by six bits of padding in lines 17-19. Lines 21-28 describe the eight capacitive touch inputs as buttons, due to the binary nature of touched/not touch. Once again, each HID button is a single bit, so the eight capacitive touch inputs are defined as a full byte. Once this report descriptor is known to the host device, it can interpret the reports sent by the input device.

The core module sends the sensor and button data to the Oculus Quest through HID reports. This process happens mainly between two FreeRTOS tasks. Task1 is responsible for reading the state of the two buttons on the GPI pins and reading the AT42QT touch sensing IC through I2C. Once both input types are read, the data is place appropriately into a 16-bit unsigned integer and put on a FreeRTOS queue. Task2 then reads from the queue and if any of the 10 inputs changed state, Task2 sends an HID report. This process can be seen in Figure 3.5.

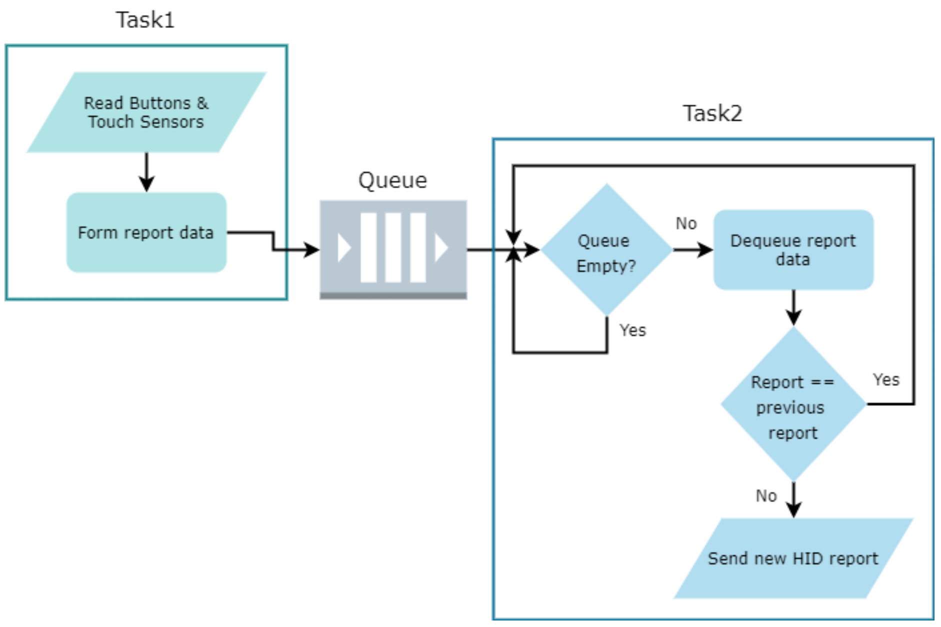


Figure 3.5: Core module task flowchart

Once the Oculus Quest receives an input report, the Unity application maps these inputs to a virtual joystick. Each core module is mapped to a joystick in the order in which Unity first received input. For example, if core module 2 sends input to the Unity application before core module 1, core module 2 will be mapped to joystick 0. It is then necessary to devise a way to order the core modules so that input from a specific device may be checked. The ordering of core module to peripheral to joystick is solved by connecting the core modules to certain peripherals by convention. For example, if a peripheral has a left and right version, core module 1 is connected to the left and core module 2 is connected to the right. Then by using the method from the Unity Input API, *GetJoystickNames*, which returns an array of the device names in the order of the joystick assignment, the full input mapping is complete [25]. Once the device assignment is known, we use it in the input manager layer, another layer of virtual inputs we built that combines a mapping of the inputs from the core modules and the Touch controllers. Applications are then able to use the input manager to check sensor states.

## 3.2 Multi-Digit Input Peripheral

A common limitation in VR is finger presence due to the tracking absence of all the fingers within the VR system. This limits the user's interaction with the virtual environment by limiting common forms of human interaction such as hand gestures. The Multi-Digit Input Peripheral (MDIP) solves this issue. It extends the user's finger presence in VR by adding three capacitive touch sensors in addition to the sensors present in the Oculus Touch controller. The sensors in the buttons and the index trigger of each Oculus Touch controller only read the user's thumb and index finger. The MDIP allows additional discrete finger input for the user by reading the three remaining fingers, middle, ring, and pinky finger. Unlike the Oculus Optical Finger Tracking, the MDIP keeps the presence of both Oculus Touch controllers' inputs for the user. This extends the input capabilities the user can use within the VR system.

### 3.2.1 MDIP Hardware Implementation

The system level for the MDIP is shown in Figure 3.6. The MDIP receives input from three electrodes attached to the peripheral by sensing the user's finger placement, middle, ring, or pinky finger, on each sensor. The electrodes act as capacitive touch sensors for the MDIP. Each of these are made up of copper covered with polyamide tape to prevent the copper from oxidizing. The input received from each capacitor is read by the core module. Based on the input received by the electrodes, the core module calculates the state of each finger, opened or closed, based on the previous state of each corresponding finger. This data is then outputted to the Unity application through HID over BLE. Along with the data obtained from the sensors in the Oculus Touch controller, within Unity, each finger input from the MDIP and the Oculus Touch controller is mapped to the Oculus Quest's inputs through the input manager. By mapping these inputs, we are able to move each finger of the hand model representation made in Unity based on the discrete readings received by the MDIP and the Oculus Touch controller of each of the user's fingers.

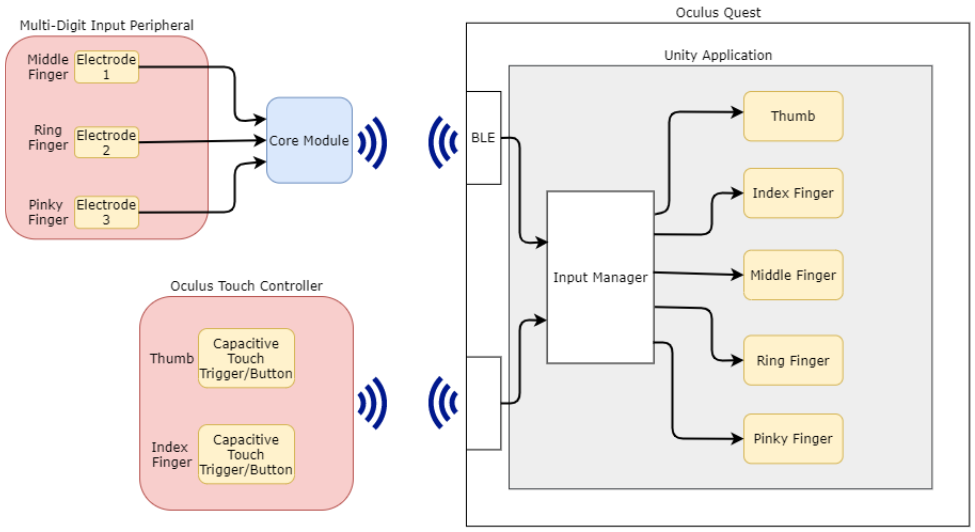


Figure 3.6: System level diagram of the Multi-Digit Input Peripheral

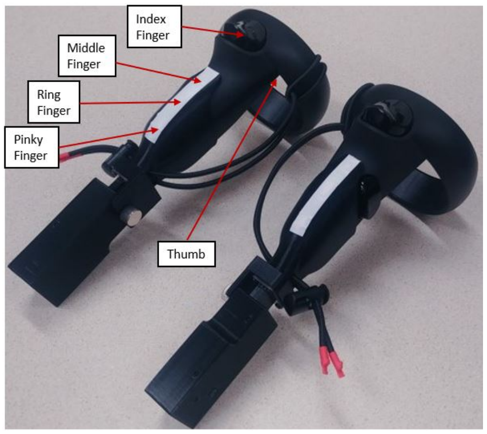


Figure 3.7: Multi-Digit Input Peripheral attached to the Oculus Touch Controllers

The hardware for the MDIP is shown in Figure 3.7. The MDIP consists of a 3D printed sleeve, three electrodes, and a cord. By using Autodesk Fusion 360, we designed and modeled a tight fit sleeve for each Oculus Touch controller. The MDIP sleeve serves as the base for the Oculus Touch controller and it is where the other components of the MDIP and the peripheral connector are attached to. It keeps the shape of the Oculus Touch controller's handle. The three electrodes that work as capacitive touch sensors for the MDIP are embedded through a small gap at the front of the 3D printed sleeve. The length and width of each electrode is 10mm and 5mm, respectively. Each one is covered with polyamide tape to prevent the copper from oxidizing. To hide the electrodes from the user's sight, we covered the electrodes with white adhesive vinyl. Since the electrodes are also accessible within the sleeve, each electrode is soldered and wired within the peripheral to the peripheral connector, which is attached at the base of the sleeve. Additionally, each Oculus Touch controller contains a cord that runs from a hole at the base of the MDIP sleeve to and round the ring of the controller and back to a second hole at the base of the sleeve. This allows the user to safely keep both controllers on each hand, including when all the user's fingers are fully extended or open.

### 3.2.2 MDIP Unity Application

We created a demo application in Unity to test the functionality of the MDIP and represent the user's finger placement on the device within the virtual environment, displayed in Figure 3.9. Within the MDIP demo, the input states of each finger is read in each frame from a boolean array where the state of each finger is represented as either true or false, opened or closed. During each frame, the code checks if a finger has changed its state by comparing its current state with its previous state. For any input that has changed its state, a coroutine for the finger movement animation in the correct direction is executed to rotate each finger individually. This process is shown in Figure 3.8. The complete Unity script for MDIP demo application can be seen in Appendix B.1.

```
// Update is called once per frame
    void Update()
    {
        // read all input states
        read_inputs(ref finger_states);
        for (int i=0; i<10; i++)
        {
            if (finger_states[i] != prev_finger_states[i] && !cr_running[i])
            {
                // Turn towards our target rotation.
                float dir = finger_states[i] ? -1f : 1f;
                StartCoroutine(AnimateFinger(dir, i));
                prev_finger_states[i] = finger_states[i];
            }
        }
    }
```

Figure 3.8: Update function for MDIP application

Within the coroutine, the overall rotation of each finger is divided into three separate rotations corresponding to the three joints of each finger. During the coroutine, each individual joint is rotated by a certain angle and speed for each frame until it reaches a predefined target rotation. The overall rotation of a finger is completed after all the joints of the finger reach its target rotation, the joint located in the knuckle being the last one.

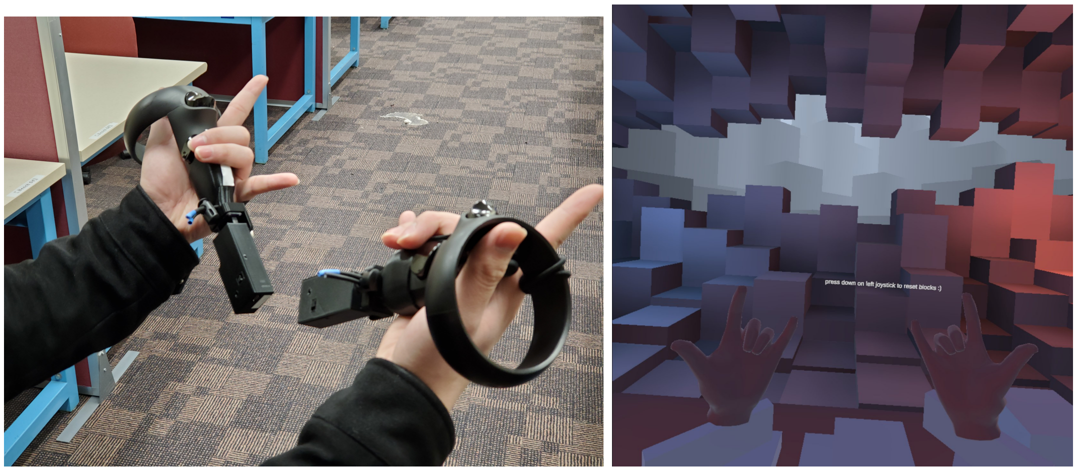


Figure 3.9: User with the MDIP (left) and the demo representing the user's finger placement (right)

## 3.3 Hand Grip Position Peripheral

Virtual Reality applications typically involve manipulating virtual objects with the hands. Objects held with one hand work well, but two-handed virtual object interaction is non-trivial due to the user's hands holding two separate controllers. Developers make a design choice to either not allow objects to be held with more than one hand, or to set a distance threshold from the initial grab points of a two-handed object. The former is not a common implementation, though it can be done with some success. The ideal solution in terms of immersion is to have the user grip a real world two-handed object and see a virtual object in VR, which is what the Hand Grip Position Peripheral (HGPP) achieves. The HGPP is cylindrical handle object with a Touch controller mounted to it for motion tracking. The HGPP then detects the number of and position of the hands along the sensing area. With this, the device is capable of providing a rigid object to grasp with a visual representation of the grip position. The HGPP is designed to be a generic controller for a variety of virtual two-handed objects such as a sword or tennis racket.

### 3.3.1 HGPP Hardware Implementation

On a system level, the HGPP uses the 8 capacitive touch inputs from two core modules, for a total of 16 touch sensors, as shown in Figure 3.10. It also has two buttons for general input, though the Unity application we developed does not use the buttons. The 16 touch sensors form a large touch sensitive area on the handle to sense the hand grip position. The core modules send the touch sensor data to the Unity application where they are mapped and processed to determine the grip positions.

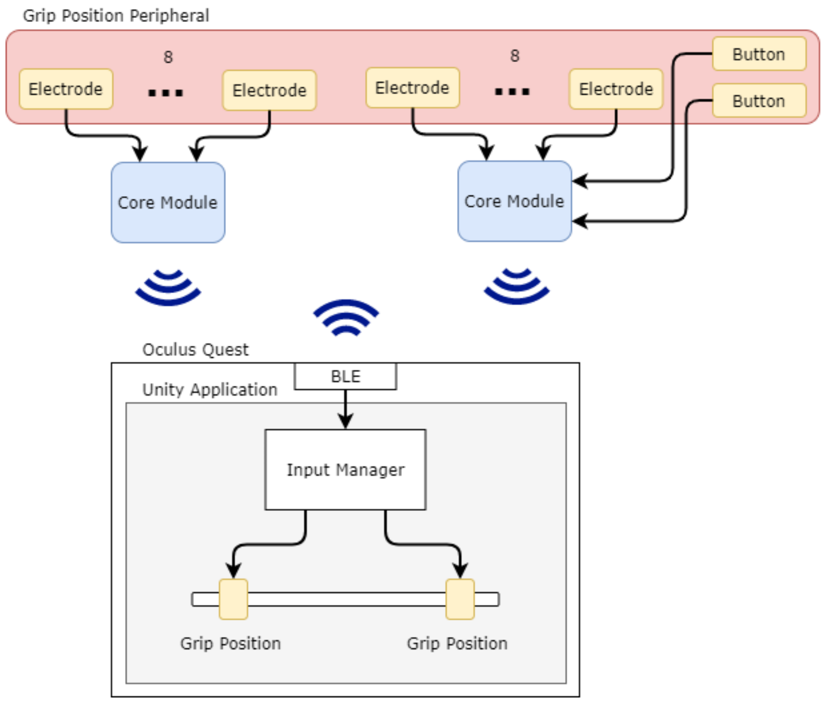


Figure 3.10: System level diagram of the Grip Position Peripheral

The HGPP, shown in Figure 3.11 is built upon a 300mm long piece of extruded aluminum. We designed a stacking sleeve system to hold the sensor array PCB and conceal wires. The ends of the aluminum are capped with 3D printed mounts for the core modules and Touch controller. One core module is placed at the top of peripheral close to the Touch controller, with the second mounted to the bottom of the peripheral. Note that the Touch controller is only used for tracking and haptic vibration feedback in this peripheral. The Touch controller sits in a cup and is secured

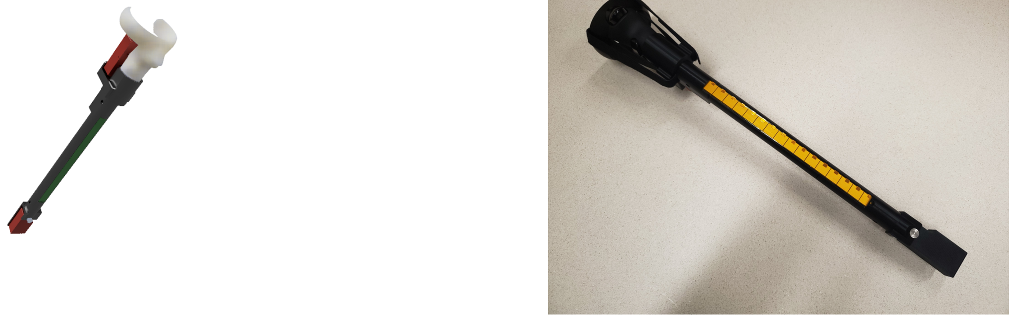


Figure 3.11: The HGPP design (left), and completed version (right)

down with hook and loop fastener strips. The two buttons are located at the top of the peripheral below the core module mount and are accessible by the thumb when the hand is at the top of the sensor array. As in all of the peripherals, the core modules are secured by thumb screws, which is especially important for this peripheral since the peripheral connectors may be subject to relatively large forces. The PCB was designed in Autodesk Eagle and cut at WPI's Foisie Innovation Studio Prototyping Lab. It is a single sided PCB with 16, copper pads that serve as the touch sensors. The PCB is covered with polyamide tape to protect the exposed copper.

### 3.3.2 HGPP Unity Application

We developed a demo application on the Oculus Quest to show a potential use case for the HGPP. The demo shows the peripheral as a two-handed sword, displays the user's current grip position, and produces acceleration-based haptic feedback. As shown in Figure 3.10, the application first receives input from the device and sends it through the input manager code. The input manager creates another layer of virtual input mappings for the core modules. After the inputs have been mapped, the position of the user's hands is estimated. First, the data from the capacitive touch inputs are stored in an array, with 0 being not touch and 1 being touched. With this format, a hand gripping near the middle of the device may look like the following.

$$[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]$$

To recognize groups of ones and determine the centers of the groups, 1D convolution is performed on the array. Note that the array is padded to 18 elements for convolution. Convolving this example array with a filter of $[1, 1, 1]$ yields the following.

$$[0, 0, 0, 0, 0, 1, 2, 3, 3, 3, 2, 1, 0, 0, 0, 0, 0, 0]$$

Consecutive threes in the array are matches for the filter of $[1, 1, 1]$ and denote areas near the center of the group. The filter was chosen as $[1, 1, 1]$ through trial and error, as hand grip activations can be groups of 3-7 elements. After convolution, the number of groups of threes and their lengths are determined. Doing so allows the system to differentiate between array states of one hand, two hands separate, and two hands combined, as shown in Figure 3.12. The presence of two groups correlates to two separate grips while one group may be one hand grip or two hands close together. In the case of one group, a threshold of the group length determines the number of hands forming the group. After the grip case has been determined, the position of the hands is calculated as the center of the group, if applicable. Note that the position of the hands in the case of two hands combined is calculated as 1/3 and 2/3 of the full length of the group.

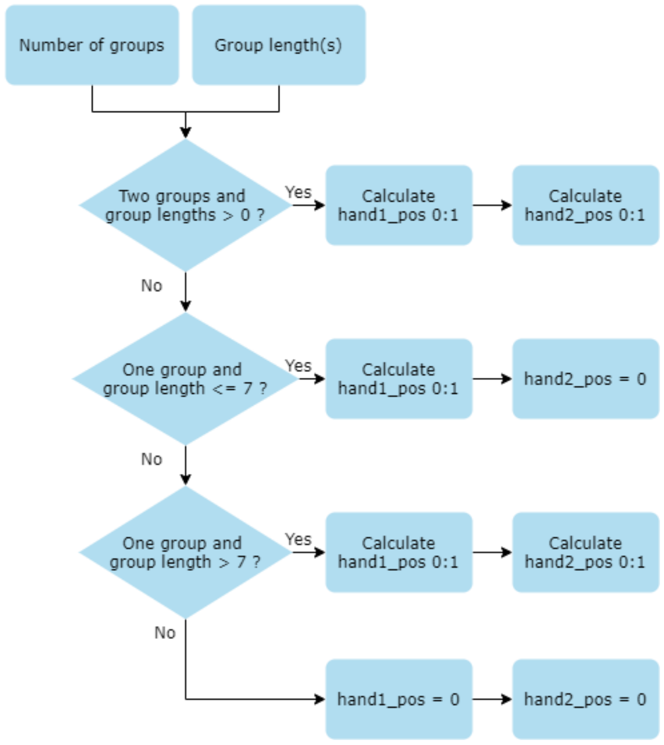


Figure 3.12: Hand activation group cases

At this point, the system is aware of the number of hands on the device and where the hands are placed, though the results can be noisy. The noise is reduced with a moving average filter with samples N=5. The filter does introduce a slight latency but the response time of a change in hand position is still acceptable. So far the system has read the sensor states, convolved the sensor data, computed hand positions and updated the moving average. This occurs in lines 3 through 9 in the frame update function shown in Figure 3.13. The last task is to display a meaningful representation of the user's hands along the virtual two-handed object. That is the purpose of lines 11-12 in Figure 3.13, which pass the hand positions to a vertex displacement shader. Appendix B.2 provides the Unity script used to develop the HGPP demo application.

```
void Update(){
        // read sensors into an array
        byte[] raw = GetRawSensors();
        // convolve the data to look for 111
        byte[] feature_map = Convolve(raw);
        // compute hand position(s) 0.01:0.99
        Vector2 pos = GetPos(feature_map);
        // update the moving average
        UpdateMA(pos);
        // show visual feedback of hand positions
        hand1.GetComponent<Renderer>().material.SetFloat("_distance", pos.x);
        hand2.GetComponent<Renderer>().material.SetFloat("_distance", pos.y);
    }

```

Figure 3.13: Update function for HGPP application

Using Unity's built-in visual shader programming tool, Shader Graph, we designed a shader that moves a mesh along its local y-axis by some input distance. The shader also styles the mesh as a transparent blue object for a 'hologram' effect. The mesh in particalar is a symetric hand model, since this peripheral cannot distinguish the left hand from the right hand. The vertex displacement section of the shader is shown in Figure 3.14 where the input scalar *dist* is built into a vector3, scaled by some distance, and the vertices are moved relative to object space.

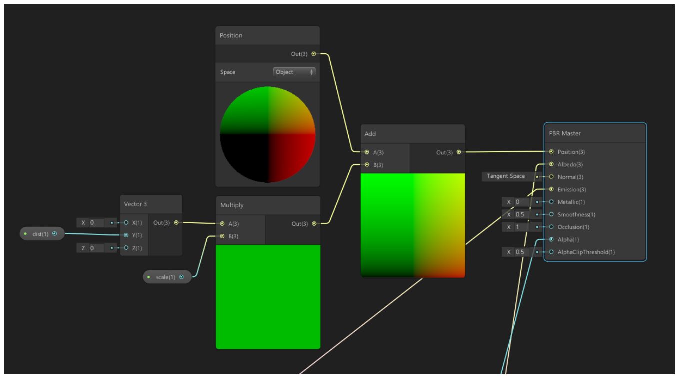


Figure 3.14: Vertex displacement in Unity Shader Graph

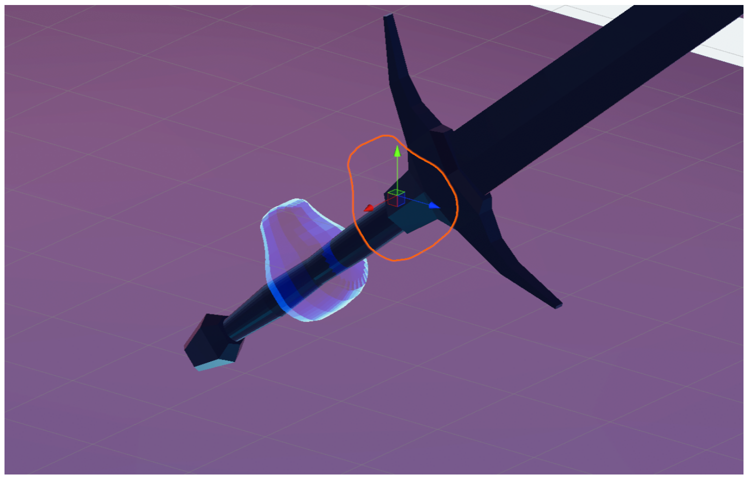


Figure 3.15: Hand grip visualization

The effect created from this can be seen in Figure 3.15, where the hand is shown gripping a sword. In this visual, the *dist* parameter is defaulted to 0.5 and the effect of the vertex displacement is shown by change in position from the objects local origin. Note that two hands are actually displayed in the same place in Figure 3.15. For the demo application we mapped a two-handed sword to the dimensions of the handle peripheral. The user is able to see a representation of their hand positions as shown in Figure 3.16.

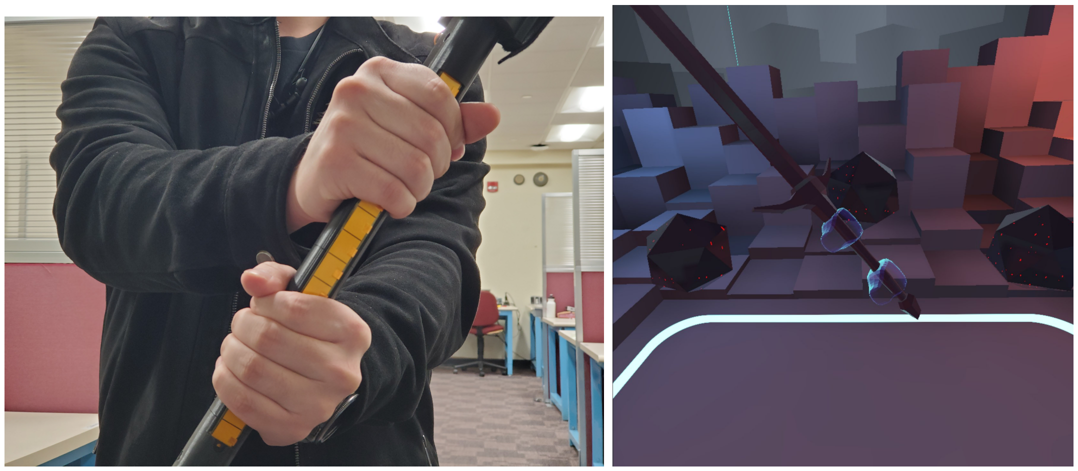


Figure 3.16: User with the HGPP (left) and the demo representing the user's hand positions (right)

## 3.4 Step Detection Peripheral

The user's real-world movement in VR is mainly limited due to the current hardware in the VR system, a headset and two controllers. This leads to a tracking absence of the user's lower half of the body as well as a maximum tracking distance and a restricted setup area for the user. Any action involving this area of the human body is ignored in VR, resulting in a complex task of simulating any form of movement involving the lower half of the body, such as walking. With all these constraints, there are a few solutions to this problem, such as teleportation and smooth movement, but such solutions also come with their own problems, disorientation and motion sickness. Due to these problems, a better solution would be to develop a device that could track the user's movement of the lower part of the body within a restricted tracking area. In other words, the user would have to move without displacing from its current position in the real world. Few solutions, such as the cybershoes and the VR treadmills, have addressed this issue, but they also have some limitations themselves. The cybershoes do not accurately represent real-world movement in VR, whereas VR treadmills are not affordable for average customers. Our device, the Step Detection Peripheral (SDP), solves the constraints imposed by the current VR system. The SDP allows the user to move within its current position while accurately representing the user's real-world movement in the virtual environment and being affordable for the average customer.

### 3.4.1 SDP Hardware Implementation

The system level diagram of the SDP, shown in Figure 3.17, includes one force sensor and one core module for each SDP. In the SDP, each force sensor is located under the user's foot and works as a button input to detect any of the user's steps. In other words, any force sensor data read under a certain force threshold is detected as 0, whereas force sensor data above the threshold is detected as 1. The core module on each SDP sends the force sensor data to the Unity application. The inputs are mapped within the Unity application to determine any step detection from the SDP of either user's foot.

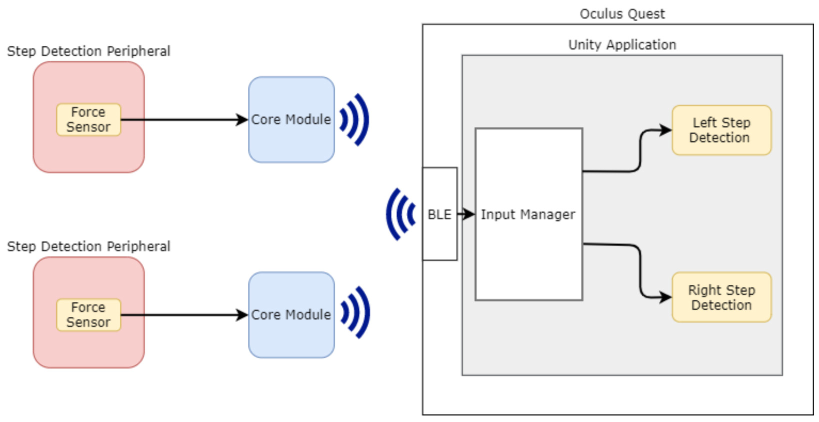


Figure 3.17: System level diagram of the Step Detection Peripheral

The SDP is a small device that is placed on top of the user's foot and is secured by using a custom-made strap that wraps around it, shown in Figure 3.18. The SDP consists of a 3D printed base, a force sensor, and a strap made up of hook-and-loop fastener and elastic band. The 3D print is about 85mm long and 37mm wide and houses the core module for the SDP. The 3D print is curved underneath the base which can be seen in Figure 3.18 and covered with foam to adjust to the user's foot while taking into account different foot sizes. The top of the 3D print contains a hole that is aligned with the hole present in the core module. This is to affix the core module to the 3D print with a thumbscrew. The 3D print contains two other holes that are essential for the SDP. The first hole runs through the length of the SDP about 2.50mm above the curved base of the 3D print. This allows the strap to go through the 3D printed base and secure the SDP to the user's foot. The second hole is located at the side of the SDP where it initially merges with the strap hole and runs through to the inside of the 3D printed base where the peripheral connector, attached to the 3D print of the SDP with bolts, and the core module are located. The second hole allows us to connect the force sensor to the peripheral connector through a wire which is soldered to the pads of the peripheral connector. The strap of the SDP consists of segments of hook-and-loop fasteners sewed into the ends of an elastic band. The force sensor is sewed in the middle of the strap and covered with an additional layer of elastic band to hide the force sensor from the user's view. Due to the elasticity of the strap, we also took into account adding slack to the wire. This allows the user to keep the force sensor centered underneath their foot, shown in Figure 3.18.

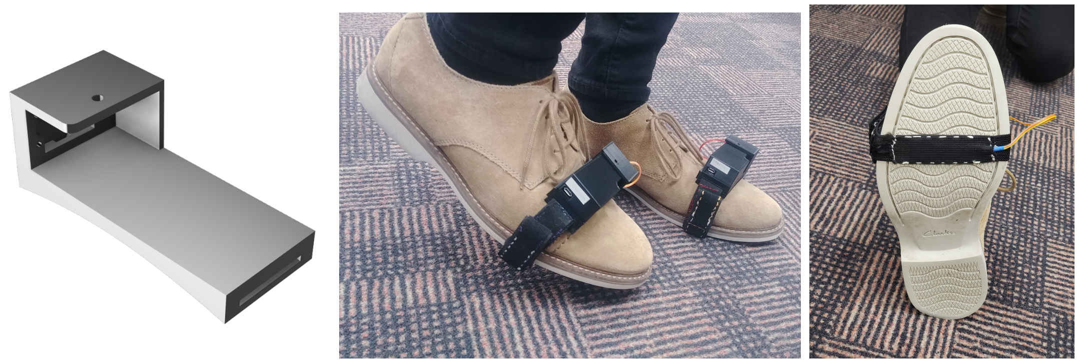


Figure 3.18: The SDP design (left), user with SDP (center), and force sensor location (right)

### 3.4.2 SDP Unity Application

We created a demo application in the Oculus Quest to test the SDP's functionality. This test allows us to verify that the SDP is able to recreate the user's real-world movement in the virtual environment. Within the SDP demo application, detailed in Appendix B.3, the state of each foot is read during each frame with a boolean variable where false means that the user's foot is off the ground and true means that the foot is in contact with the ground. The user is able to move within the virtual application regardless of which foot the user steps with. This user's movement in the virtual application depends on a single boolean variable that represents this situation. During each frame, a comparison case is done between the current state and the previous state for each foot to determine if the user has stepped with either foot, shown in lines 6-9 and 13-17 from Figure 3.19. After comparing the current and previous state, if it is determined that the user has stepped with either foot, the boolean variable is set to true and the user moves within the virtual environment.

```
void Update(){
        bool either_lift = false;
        bool left_lift = input_manager.GetButton("button2", "VRCORE1");
        if (!left_lift && prev_left_lift) { // new step
            either_lift = true;
        }
        prev_left_lift = left_lift;
        bool right_lift = input_manager.GetButton("button2", "VRCORE2");

        if (!right_lift && prev_right_lift) { // new step
            either_lift = true;
        }
        prev_right_lift = right_lift;

        if (either_lift){
            gameObject.GetComponent<Rigidbody>().AddForce(torso.forward.normalized *
    speed, ForceMode.VelocityChange);
        }
    }

```

Figure 3.19: Update function for SDP application

The user's movement in the virtual environment is based on the the forward direction of the torso based on the user's head and hands. A model of the user's arm was created in the SDP demo application to represent the user in the virtual environment. By using an Inverse Kinematics (IK) system in the demo application, the user moves within the virtual environment based on the user's tracked head and hands. In other words, the position and orientation of the Oculus Quest VR headset and touch controllers serve as the inputs for the IK system. The system determines the front-facing direction of the torso. This is done by having the torso direction following the head when the hand are near the player and following the hands when the arms are extended. Within the system, there are three vectors that are taken into consideration: the head's constant magnitude forward vector and two vectors starting from the head to one in each hand. After resolving the each vector, the system calculates the weighted combination of the vectors based on magnitude to determine the forward direction of the torso. The system avoids any incorrect torso direction by ignoring the hands whenever they are located behind the user's head.

# Chapter 4

# Conclusions

## 4.1 Project Contributions

In this MQP, the team identified three interaction limitations with current VR products and devised solutions for each limitation. The result is a wireless and modular peripheral system that can be extended to create other interaction devices. The three peripherals realized in this project are the Multi-Digit Input Peripheral (MDIP), Hand Grip Position Peripheral (HGPP), and the Step Detection Peripheral (SDP).

The MDIP aims to provide an increased sense of finger presence with the Oculus Touch controllers by sensing the state of all fingers. This allows users to form basic hand gestures using all fingers and more immersive grabbing/releasing of virtual objects. Open-palm interactions are also possible, such as pushing an object with the palms of the hands. The demo application for the MDIP maps the 10 finger inputs to tracked hand models and animates each finger according to the device input. The MDIP is a balanced combination of solutions for finger presence in VR. Optical solutions can provide quality tracking but are heavily affected by occlusion and may have a low tracking frequency. Additionally, controllers are not held with optical methods, which can create interaction issues when grabbing objects. For example, reaching out and grabbing a virtual cube can feel strange because the hand does not grab onto anything physically. When holding a controller, this same action can be more immersive due to the presence of the controller, which provides a form of passive haptic feedback. In essence, the controller can feel like many different objects when grabbing onto it. The MDIP uses this observation and provides basic finger tracking simultaneously. Another design decision and benefit of the MDIP is that the controller remains held in the normal position so that the controller inputs may still be used.

The second peripheral, the HGPP, is a solution to the two-handed virtual rigid object problem. Commercial VR systems typically use two separate tracked controllers, which makes interacting with a virtual object with two hands non-trivial. One common software-only solution is to use a threshold for the relative distance change from the initial grab points. The effect is that two handed interactions become possible, but can feel like the user is puppeting the object, since the hands are bound to move away from the initial grab points during movement. Another common solution is to not allow objects to be held with two hands at a time. A true solution to this problem is to create a real-world rigid constraint between the hands, which is the approach taken by the HGPP. The HGPP is a cylindrical peripheral with 250mm of touch sensitive pads for tracking the grip position of both hands. A Touch controller is mounted to the top of the peripheral for 6 DOF tracking. It also includes two buttons near the top of the touch sensitive area for general input. The HGPP is a generic device that can become many two-handed virtual objects and display the approximate position of where the hand grip the device. For example, the HGPP can become a virtual sword, tennis racket, or magical staff. The demo we created for the HGPP showed a sword in place of the peripheral with the handle of the sword mapped to the peripheral's length. The user can see representations of their hand position on the handle in real time.

The final peripheral developed in this project is the SDP. The SDP is a solution to the walking movement problem in VR, in which the user must be able to traverse large virtual distances while remaining in a small real-world play area. The most immersive movement is room-scale movement, in which the user physically walks within the tracked space. Software devised locomotion methods allow the user to move larger distances but may be disorienting or motion-sickness inducing. Common locomotion methods are teleportation, smooth movement, and world-grabbing. The SDP is a device worn on each foot that detects foot lift. The SDP only measures the act of stepping while the direction of movement is estimated as the direction of the torso, which is derived from the head and hand positions. To move in the virtual world, the user walks or jogs in place, creating natural-feeling movement that is initiated by the legs instead of hand controller input. The head bobbing introduced from the motion of walking in place can also reduce motion sickness as the vestibular system experiences real-world motion and reduces the visual acceleration disconnect.

## 4.2 Recommendations

Although we were able to address the identified VR interaction limitations, our system could still be improved to better solve these limitations. The system itself and the three peripherals in this project, Multi-Digit Input Peripheral (MDIP), Hand Grip Position Peripheral (HGPP), and the Step Detection Peripheral (SDP), have room for improvements.

Our system is based on the concept of modularity where the core module is able to interact with the different peripherals to solve interaction limitations. Instead of having purpose-built peripherals, an improvement to our system is using interlocking components, a method similar to The Lego Group's interlocking blocks. These interlocking components can be used to build the different peripherals and recreate the same functions. The current system can also be expanded to include more inputs for each core module and use proximity sensors to detect the user's finger and hand movement along the peripheral. Another improvement would be to expand the data type to include a more analog user input from the current discrete button and capacitive touch input, such as integrating a capacitive touch slider instead of an array of capacitive touch sensors to detect the user's hand movement along the peripheral.

The improvements for the core module include its electrical and mechanical connections. Due to prototype purposes, the core module uses pin headers for electrical connection. An improvement to this would be to use a better and more reliable connector, such as USB-C or pogo pins. An improvement for its mechanical connection would to be securely insert the core module into the peripheral instead of fastening it to the peripheral with a thumbscrew.

The improvements for the MDIP include sensor type, strap, and sensor configuration. The MDIP could use proximity sensors, substituting the capacitive touch sensors to detect the user's finger movement. The configuration of these sensor could also be improved by integrating a flexible PCB containing these sensors instead of individual sensors attached to the peripheral. Another improvement is replacing the MDIP's cord with a band as a strap.

The improvements for the HGPP include its grip position tracking, number of modules, and peripheral configuration. The current HGPP uses an array of capacitive touch sensors to approximate the user's hand position. A better implementation of this function would be to use a touch pad as a continuous capacitive touch slider. An important improvement for the HGPP is to reduce the number of modules to make the peripheral simpler. Another improvement to consider would be able to configure the peripheral. This involves having detachable segments for the HGPP, being able to change the configuration of two handed objects.

Finally, the improvements for the SDP include size and weight, orientation tracking, and unwanted movement suppression. The current SDP is still big and its size and weight can be reduced by reducing the size of the core module. Another improvement would be to include an IMU in the SDP to detect the orientation of the user's feet. The SDP could also ignore unwanted movement suppression by detecting any successive steps of a single foot or its unrealistic orientation while walking.

# Bibliography

[1] Choi, I., Hawkes, E., Christensen, D., Ploch, C., & Follmer, S. (2016). Wolverine: A wearable haptic interface for grasping in Virtual Reality. *IEEE*, 986 - 993

[2] Colgan, A. (2014, August 9). How does the Leap Motion Controller work? Retrieved January 26, 2020, from http://blog.leapmotion.com/hardware-to-software-how-does-the-leap-motion-controller-work/

[3] Cybershoes. (2019). Cybershoes. Retrieved January 27, 2020, from https://www.cybershoes.io/

[4] Cybershoes. (2019). Cybershoes + Cybercarpet. Retrieved January 27, 2020, from https://www.cybershoes.io/product/cybershoes-cybercarpet/

[5] Espressif Systems. (2019). ESP32-WROOM-32D & ESP32-WROOM-32U datasheet. Retrieved February 15, 2020, from https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32d_esp32-wroom-32u_datasheet_en.pdf

[6] Facebook Technology. (2019, August 22). From the lab to the living room: The story behind Facebook's Oculus Insight technology and a new era of consumer VR. Retrieved January 26, 2020, from https://tech.fb.com/the-story-behind-oculus-insight-technology/

[7] FreeRTOS. (n.d.). The FreeRTOS Kernel. Retrieved February 9, 2020, from https://www.freertos.org/RTOS.html

[8] Han, S., Liu, B., Ho Yu, T., Cabezas, R., Zhang, P., Vajda, P., Isaac, E., & Wang, R. (2019, September 25). Using deep neural networks for accurate hand-tracking on Oculus Quest. Retrieved January 26, 2020, from https://ai.facebook.com/blog/hand-tracking-deep-neural-networks

[9] HaptX. (2019, February 20). Technology. Retrieved January 27, 2020, from https://haptx.com/technology/

[10] Infinadeck. (2020, February 2). Infinadeck. Retrieved January 27, 2020, from https://www.infinadeck.com/

[11] KATVR. (2020). KAT Walk. Retrieved January 27, 2020, from https://www.kat-vr.com/products/kat-walk-vr-treadmill

[12] LaValle, S. M. (2017). Virtual Reality. *Cambridge University Press*

[13] Melim, A. (2019, November 4). Tracking technology explained: LED matching. Retrieved January 27, 2020 from https://developer.oculus.com/blog/tracking-technology-explained-led-matching/

[14] Oculus Developer Center. (n.d.). Locomotion. Retrieved March 17, 2019, from https://developer.oculus.com/design/bp-locomotion/

[15] Oculus VR. (2018, September 26). Introducing Oculus Quest, out first 6DoF all-in-one VR system, launching spring 2019. Retrieved January 25, 2020, from https://www.oculus.com/blog/introducing-oculus-quest-our-first-6dof-all-in-one-vr-system-launching-spring-2019/?locale=en_US

[16] Oculus VR. (2019, November 18). Play Rift content on Quest with Oculus Link, available now in beta! Retrieved January 26, 2020, from https://www.oculus.com/blog/play-rift-content-on-quest-with-oculus-link-available-now-in-beta/?locale=en_US

[17] Omni by Virtuix. (n.d.). Virtuix Omni. Retrieved January 27, 2020, from https://www.virtuix.com/product/virtuix-omni/

[18] Pattuzzi, S. (2018, June 14). Developer perspective: Designing awesome locomotion in VR. Retrieved January 26, 2020, from https://developer.oculus.com/blog/developer-perspective-designing-awesome-locomotion-in-vr/

[19] Perret, J., & Vander Poorten, E. (2018, June 25). Touching virtual reality: A review of haptic gloves. *VDE*, 1-5

[20] PR Newswire. (2019, June 11). Dexta Robotics announces force feedback gloves Dexmo Enterprise Edition, radically improving the quality of virtual training. Retrieved January 27, 2020, from https://www.prnewswire.com/news-releases/dexta-robotics-announces-force-feedback-gloves-dexmo-enterprise-edition-radically-improving-the-quality-of-virtual-training-300864420.html

[21] Road to VR. (2019, August 12). Hands-on: Dexmo haptic force-feedback gloves are compact and wireless. Retrieved January 27, 2020, from https://www.roadtovr.com/dexta-dexmo-vr-gloves-force-feedback-haptic-hands-on/

[22] Shao, L. (2016). Hand movement and gesture recognition using Leap Motion Controller. *EE Stanford*, 1-5

[23] Strasnick, E., Holz, C., Ofek, E., Sinclair, M., & Benko, H. (2018, April). Haptic links: Bimanual haptics for Virtual Reality using variable stiffness actuation. *ACM*, 1-12

[24] Techspot. (2020). Valve Index. Retrieved January 27, 2020, from https://www.techspot.com/products/audio-video/valve-index.205736/

[25] Unity. (2019). Input.GetJoystickNames. Retrieved February 16, 2020, from https://docs.unity3d.com/ScriptReference/Input.GetJoystickNames.html

[26] Urtans, E., & Nikitenko, A. (2016, May 27). Active infrared markers for Augmented and Virtual Reality. *Latvia University of Agriculture*, 1018-1029

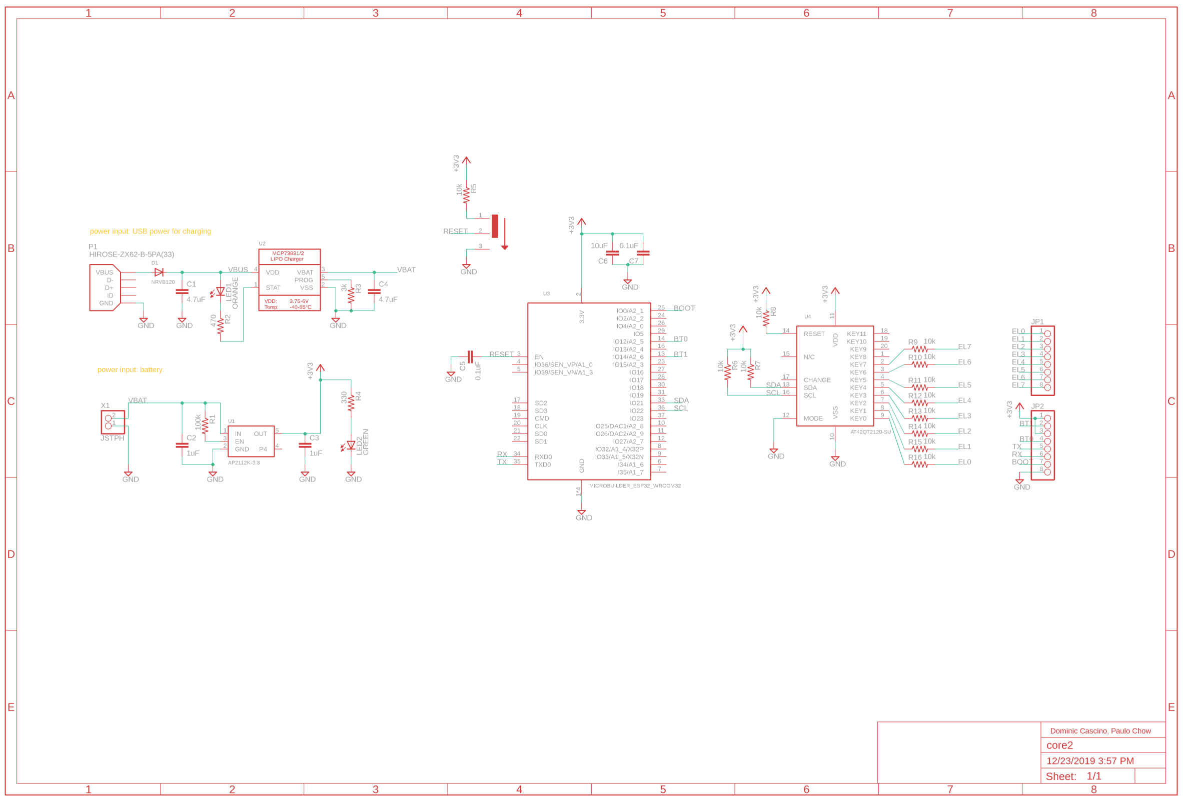
[27] Valve Index. (n.d.). Base stations. Retrieved January 27, 2020, from https://www.valvesoftware.com/en/index/base-stations

[28] Yun, X., & Bachmann, E. (2006, December). Design, implementation, and experimental results of a quaternion-based Kalman filter for human body motion tracking. IEEE, 22(6), 1216-1227

[29] Zenner, A., & Krüger, A. (2019, May). Drag:on - A Virtual Reality controller providing haptic feedback based on drag and weight shift. *ACM*, 1-12

# Appendix A

# Core Module Schematic



# Appendix B

# Unity Scripts

## B.1 MDIP

handAnimationController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class hand_ani_controller : MonoBehaviour
{
    public Transform thumb;
    public Transform index;
    public Transform middle;
    public Transform ring;
    public Transform pinkie;
    public Transform right_thumb;
    public Transform right_pointer;
    public Transform right_middle;
    public Transform right_ring;
    public Transform right_pinkie;

    private input_mapper input_manager;
    private Transform[] finger_transforms;
    private bool[] cr_running;          // array for coroutine status for each finger
    private bool[] finger_states;       // array for finger states
    private bool[] prev_finger_states;  // array for previous finger states to detect
     state transitions

    IEnumerator AnimateFinger(float dir, int idx)
    {
        cr_running[idx] = true;
        float rotation_speed = 300f;
        Transform joint1 = finger_transforms[idx];
        Transform joint2 = joint1.GetChild(0);
        Transform joint3 = joint2.GetChild(0);
        Quaternion target1 = new Quaternion();
        Quaternion target2 = new Quaternion();
        Quaternion target3 = new Quaternion();

        // different rotations for each finger
        switch (idx)
        {
            case 0: // thumb
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 7.5f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 25f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 15f * dir));
                break;
            case 1: // pointer
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 30f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 25f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 15f * dir));
                break;
            case 2: // middle
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 40f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 30f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 20f * dir));
                rotation_speed = 600f;
                break;
            case 3: // ring
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 40f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 30f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 20f * dir));
                rotation_speed = 500f;
                break;
            case 4: // pinkie
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 40f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 30f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 20f * dir));
                rotation_speed = 500f;
                break;
            case 5: // thumb
```

```
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 7.5f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 25f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 15f * dir));
                break;
            case 6: // pointer
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 30f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 25f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 15f * dir));
                break;
            case 7: // middle
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 40f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 30f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 20f * dir));
                rotation_speed = 600f;
                break;
            case 8: // ring
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 40f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 30f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 20f * dir));
                rotation_speed = 500f;
                break;
            case 9: // pinkie
                target1 = joint1.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 40f * dir));
                target2 = joint2.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 30f * dir));
                target3 = joint3.localRotation * Quaternion.Euler(new Vector3(0f, 0f, 20f * dir));
                rotation_speed = 500f;
                break;

            default:
                break;
        }

        while (Quaternion.Angle(joint1.localRotation, target1) > 0.1f)
        {
            float step = rotation_speed * Time.deltaTime;
            if (idx == 0)
            {
                step *= 0.5f;
            }
            joint1.localRotation = Quaternion.RotateTowards(joint1.localRotation,
    target1, step);
            joint2.localRotation = Quaternion.RotateTowards(joint2.localRotation,
    target2, step);
            joint3.localRotation = Quaternion.RotateTowards(joint3.localRotation,
    target3, step);

            yield return null;
        }

        joint1.localRotation = target1;
        joint2.localRotation = target2;
        joint3.localRotation = target3;
        cr_running[idx] = false;
    }

    void read_inputs(ref bool[] states)
    {

            states[0] = input_manager.ReadTouchController("x_button_cap", "left") ||
                    input_manager.ReadTouchController("y_button_cap", "left") ||
                    input_manager.ReadTouchController("joy_cap", "left");
            states[1] = input_manager.ReadTouchController("front_trigger", "left");
            states[2] = input_manager.ReadModule("top", "VRCORE1");
            states[3] = input_manager.ReadModule("middle", "VRCORE1");
            states[4] = input_manager.ReadModule("bottom", "VRCORE1");
        states[5] = input_manager.ReadTouchController("a_button_cap", "right") ||
                    input_manager.ReadTouchController("b_button_cap", "right") ||
                    input_manager.ReadTouchController("joy_cap", "right");
        states[6] = input_manager.ReadTouchController("front_trigger", "right");

        states[7] = input_manager.ReadModule("top", "VRCORE2");
        states[8] = input_manager.ReadModule("middle", "VRCORE2");
        states[9] = input_manager.ReadModule("bottom", "VRCORE2");
    }

    // Start is called before the first frame update
    void Start()
    {
        // init
        prev_finger_states = new bool[] { true, false, false, false, false, true,
    false, false, false, false};
        cr_running = new bool[] { false, false, false, false, false, false, false,
    false, false, false};
        finger_states = new bool[] { true, false, false, false, false, true, false,
    false, false, false};
        finger_transforms = new Transform[] { thumb, index, middle, ring, pinkie,
                right_thumb, right_pointer, right_middle, right_ring, right_pinkie};
        input_manager = gameObject.GetComponent<input_mapper>();
    }

    // Update is called once per frame
    void Update()
    {
        // read all input states
        read_inputs(ref finger_states);
        for (int i=0; i<10; i++)
        {
            if (finger_states[i] != prev_finger_states[i] && !cr_running[i])
            {
                // Turn towards our target rotation.
                float dir = finger_states[i] ? -1f : 1f;
                StartCoroutine(AnimateFinger(dir, i));
                prev_finger_states[i] = finger_states[i];
            }
        }
    }
}


```

## B.2 HGPP

gripPositionController.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class input_mapper : MonoBehaviour
{
    [HideInInspector] public int[] mapping;
   // public TextMeshProUGUI log1;
   // public TextMeshProUGUI log2;
    //  public Transform debug_object1;
    // public Transform debug_object2;
    public GameObject hand1;
    public GameObject hand2;

    private Vector2[] ma_buffer; // stores the moving average, latest value is at index 0
    private string[] joys;
    private string[] prev_joys;
    private int frame_count = 72;
    private input_mapper input_manager;
    private const int NUM_SENSORS = 16;
    private const int NUM_MA = 5;
    private readonly string[] handle_raw_mapping = { "13", "2", "3", "14", "4", "5",
    "6", "7", "7", "6", "5", "4", "14", "3", "2", "13" };

    private void Log(string mesg, TextMeshProUGUI log)
    {
        log.text = mesg;
    }

    public bool ReadTouchController(string input_name, string controller)
    {
        bool state = false;
        string left_mapping = "joystick " + mapping[0].ToString() + " button ";
        string right_mapping = "joystick " + mapping[1].ToString() + " button ";
        if (controller.Equals("left"))
        {
            switch (input_name)
            {
                case "x_button":
                    state = Input.GetKey(left_mapping + "2");
                    break;
                case "x_button_cap":
                    state = Input.GetKey(left_mapping + "12");
                    break;
                case "y_button":
                    state = Input.GetKey(left_mapping + "3");
                    break;
                case "y_button_cap":
                    state = Input.GetKey(left_mapping + "13");
                    break;
                case "joy_button":
                    state = Input.GetKey(left_mapping + "8");
                    break;
                case "joy_cap":
                    state = Input.GetKey(left_mapping + "16");
                    break;
                case "front_trigger":
                    state = Input.GetKey(left_mapping + "14");
                    break;
                case "middle_trigger":
                    state = Input.GetKey(left_mapping + "4");
                    break;
                case "menu_button":
                    state = Input.GetKey(left_mapping + "6");
                    break;
                default:
                    break;
            }
        }
        else if (controller.Equals("right"))
        {

            switch (input_name)
            {
                case "a_button":
                    state = Input.GetKey(right_mapping + "0");
                    break;
                case "a_button_cap":
                    state = Input.GetKey(right_mapping + "10");
                    break;
                case "b_button":
                    state = Input.GetKey(right_mapping + "1");
                    break;
                case "b_button_cap":
                    state = Input.GetKey(right_mapping + "11");
                    break;
                case "joy_button":
                    state = Input.GetKey(right_mapping + "9");
                    break;
                case "joy_cap":
                    state = Input.GetKey(right_mapping + "17");
                    break;
                case "front_trigger":
                    state = Input.GetKey(right_mapping + "15");
                    break;
                case "middle_trigger":
                    state = Input.GetKey(right_mapping + "5");
                    break;
                default:
                    break;
            }
        }
        return state;
    }

    public string ReadModuleAll(string module_name)
    {
        string module_1_mapping = "joystick " + mapping[2].ToString() + " button ";
        string module_2_mapping = "joystick " + mapping[3].ToString() + " button ";
        string module = "";
        if (module_name.Equals("VRCORE1"))
        {
            module = module_1_mapping;
        }
        else if(module_name.Equals("VRCORE2"))
        {
            module = module_2_mapping;
        }
        string pressed = "";
        for (int i=0; i<20; i++)
        {
            if(Input.GetKey(module + i.ToString()))
            {
                // Log(i.ToString());
                pressed = module + i.ToString();
                break;
            }
        }
        return pressed;

    }

    private void Remap()
    {
        //Log("remap");
        string[] devices = Input.GetJoystickNames();
        for (int i = 0; i < devices.Length; i++)
        {
            switch (devices[i])
            {
                case "Oculus Quest Controller - Left":
                    mapping[0] = i + 1;
                    break;
                case "Oculus Quest Controller - Right":
                    mapping[1] = i + 1;
                    break;
                case "VRCORE1":
                    mapping[2] = i + 1;
                    break;
                case "VRCORE2":
                    mapping[3] = i + 1;
                    break;
                default:
                    break;
            }
        }
    }

    private byte[] GetRawSensors()
    {
        string module1 = "joystick " + mapping[2].ToString() + " button ";
        string module2 = "joystick " + mapping[3].ToString() + " button ";
        byte[] sensors = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
        string raw = "";
        for (int i=0; i<16; i++)
        {
            if (i<=7)
            {
                if (Input.GetKey(module1 + handle_raw_mapping[i]))
                {
                    sensors[i] = 1;
                }
                else
                {
                    sensors[i] = 0;
                }
            }
            else
            {
                if (Input.GetKey(module2 + handle_raw_mapping[i]))
                {
                    sensors[i] = 1;
                }
                else
                {
                    sensors[i] = 0;
                }
            }
            raw += sensors[i].ToString();

        }
       // Log(raw, log1);
        return sensors;
    }

    private byte[] Convolve(byte[] raw)
    {
        byte[] filter = new byte[] { 1, 1, 1};
        byte[] expanded = new byte[18];
        byte[] convolved = new byte[16];
        string output = "";
        raw.CopyTo(expanded, 1);
        for (int i=0; i<NUM_SENSORS; i++)
        {
            convolved[i] = (byte) (expanded[i] * filter[0] + expanded[i + 1] * filter[1] + expanded[i + 2] * filter[2]);
            output += convolved[i].ToString();
        }
      //  Log(output, log2);

        return convolved;
    }

    private Vector2 GetPos(byte[] feature_map)
    {
        Vector2 hand_pos = new Vector2();
        int count = 0;
        int groups = 0;
        float[] starts = new float[2];
        float[] lengths = new float[2];
        for (int i=0; i<NUM_SENSORS; i++)
        {
            // count consecutive threes
            // count groups
            // count total number of threes
            if (feature_map[i] == 3)
            {
                if (count == 0) // this is a new edge, mark the index
                {
                    starts[groups] = i;
                }
                count++; // count number of threes
            }
            else if(count > 0)
            {
                lengths[groups] = count;
                count = 0; // reset count
                groups = 1; // increment groups, only expect 2
            }
        }

        // cases are one hand, two hands, and two hands together
        float pos1 = 0;
        float pos2  = 0;
        if (lengths[0] > 0 && lengths[1] > 0) // two hands seperate
        {
            pos1 = (starts[0] + ((lengths[0] - 1f) / 2f)) * 1f / 16f;
            pos2 = (starts[1] + ((lengths[1] - 1f) / 2f)) * 1f / 16f;
            hand_pos.x = pos1;
            hand_pos.y = pos2;
        }
        else if (lengths[1] == 0 && lengths[0] <= 7) // one hand
        {
            pos1 = (starts[0] + ((lengths[0] - 1f) / 2f)) * 1f / 16f;
            pos2 = 0f;
            hand_pos.x = pos1;
            hand_pos.y = pos2;
        }
        else if (lengths[1] == 0 && lengths[0] > 7) // two combined
        {
            pos1 = (starts[0] + ((lengths[0] - 1f) / 4f)) * 1f / 16f;
            pos2 = pos1 + ((lengths[0] - 1f) / 2f * 1f / 16f);
            hand_pos.x = pos1;
            hand_pos.y = pos2;
        }
        hand_pos.x = Mathf.Clamp(hand_pos.x, 0.01f, 0.99f);
        hand_pos.y = Mathf.Clamp(hand_pos.y, 0.01f, 0.99f);
        return hand_pos;
    }

    private void UpdateMA(Vector2 last_pos)
    {

        Vector2 newest = new Vector2();
        newest = (last_pos + ma_buffer[0] + ma_buffer[1] + ma_buffer[2] + ma_buffer
    [3] + ma_buffer[4]) / (NUM_MA + 1);
        newest.x = Mathf.Clamp(newest.x, 0.01f, 0.99f);
        newest.y = Mathf.Clamp(newest.y, 0.01f, 0.99f);

        // shift the array right and update newest value
        ma_buffer[4] = ma_buffer[3];
        ma_buffer[3] = ma_buffer[2];
        ma_buffer[2] = ma_buffer[1];
        ma_buffer[1] = ma_buffer[0];
        ma_buffer[0] = newest;
      //  Log(newest.x.ToString() + " " + newest.y.ToString(), log2);
    }

    public bool GetButton(string button_name)
    {
        bool btn = false;
        string module1 = "joystick " + mapping[2].ToString() + " button ";
        if (button_name.Equals("button1"))
        {
            btn = Input.GetKey(module1 + "0");
        }
        else if (button_name.Equals("button2"))
        {
            btn = Input.GetKey(module1 + "1");
        }
        return btn;
    }
    // Start is called before the first frame update
    void Start()
    {
        ma_buffer = new Vector2[NUM_MA];
        mapping = new int[] { 1, 2, 3, 4 };
        frame_count = 0;
        joys = Input.GetJoystickNames();
        prev_joys = joys;
        Remap();
    }

    // Update is called once per frame
    void Update()
    {
        if (frame_count == 72)
        {
            joys = Input.GetJoystickNames();

            bool input_dif = (
                             !prev_joys[0].Equals(joys[0]) ||
                             !prev_joys[1].Equals(joys[1]) ||
                             !prev_joys[2].Equals(joys[2]) ||
                             !prev_joys[3].Equals(joys[3])
```

```
                              );
            if (input_dif)
            {
                Remap();
                prev_joys = joys;
            }
            frame_count = 0;
        }
        frame_count++;
        // read sensors into an array
        byte[] raw = GetRawSensors();
        // convolve the data to look for 111
        byte[] feature_map = Convolve(raw);
        // compute hand position(s) 0.01:0.99
        Vector2 pos = GetPos(feature_map);
        // update the moving average
        UpdateMA(pos);
        // show visual feedback of hand positions
        hand1.GetComponent<Renderer>().material.SetFloat("_distance", pos.x);
        hand2.GetComponent<Renderer>().material.SetFloat("_distance", pos.y);
    }
}


```

## B.3 SDP

torsoDirectionEstimation.cs

```
using UnityEngine;

public class TorsoRot : MonoBehaviour
{

    public float pole_dist;
    public float speed = 4.0f;

    public Transform head;
    public Transform neck;
    public Transform lhand;
    public Transform rhand;
    public Transform pole;

    private Quaternion torso_rotation;

    private void Start()
    {
        torso_rotation = new Quaternion();
    }

    private void Update()
    {
        // head facing vector
        float head_ang = Vector3.Angle(Vector3.up, head.forward);
        Vector3 head_look = new Vector3();
        if (head_ang > 135.0f)
        {
            head_look = head.position - transform.position;
        }
        else if (head_ang < 45.0f)
        {
            head_look = -(head.position - transform.position);

        }
        else
        {
            head_look = pole.position - head.position;
        }
        float relative_pole_magnitude = Mathf.Sqrt(Mathf.Pow(head_look.x, 2) + Mathf.
    Pow(head_look.z, 2));
        Vector2 scaled_pole_2d = new Vector2(head_look.x * pole_dist /
    relative_pole_magnitude, head_look.z * pole_dist / relative_pole_magnitude);
        Vector3 pole_3d = new Vector3(scaled_pole_2d.x + head.position.x, transform.
    position.y, scaled_pole_2d.y + head.position.z);

        // hand average vector
        Vector3 lhand_relative = lhand.position - head.position;
        Vector3 rhand_relative = rhand.position - head.position;

        Vector3 hand_avg = (lhand_relative + rhand_relative) / 2.0f;
        Vector2 avg_hand_2d = new Vector2(hand_avg.x, hand_avg.z);

        float w_hand = avg_hand_2d.magnitude / (avg_hand_2d.magnitude +
    scaled_pole_2d.magnitude);
        float w_head = scaled_pole_2d.magnitude / (avg_hand_2d.magnitude +
    scaled_pole_2d.magnitude);

        // determine if hands are behind the user's head
        if (Vector2.Angle(scaled_pole_2d, avg_hand_2d) >= 90.0f)
        {
            w_hand = 0.0f;
            w_head = 1.0f;
        }

        hand_avg += head.position;

        // new rotation
        Vector3 new_torso_facing = new Vector3(pole_3d.x * w_head + hand_avg.x *
    w_hand, transform.position.y, pole_3d.z * w_head + hand_avg.z * w_hand);

        // update
        torso_rotation.SetLookRotation(new_torso_facing - transform.position);
        transform.rotation = Quaternion.Lerp(transform.rotation, torso_rotation, Time
    .deltaTime * speed);
        transform.position = neck.position;


    }

}




```

MovePlayer.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MovePlayer : MonoBehaviour
{
    public Transform torso;
    public input_mapper input_manager;
    public float speed = 1f;
    private bool prev_left_lift = false;
    private bool prev_right_lift = false;



    // Start is called before the first frame update
    void Start()
    {

    }

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

        bool either_lift = false;

        bool left_lift = input_manager.GetButton("button2", "VRCORE1");
        if (!left_lift && prev_left_lift) // new step
        {
            either_lift = true;
        }
        prev_left_lift = left_lift;

        bool right_lift = input_manager.GetButton("button2", "VRCORE2");
        if (!right_lift && prev_right_lift) // new step
        {
            either_lift = true;
        }
        prev_right_lift = right_lift;

        if (either_lift)
        {
            gameObject.GetComponent<Rigidbody>().AddForce(torso.forward.normalized *
    speed, ForceMode.VelocityChange);
        }



    }
}


```