Aaron McGaha

Virtual Reality Tracking and Robotic Arms

Helsinki Metropolia University of Applied Sciences

Engineering

Electronics Engineering

Thesis

9 November 2018

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Aaron McGaha Virtual Reality and Robotic Arms 23 pages + 9 appendices 9 November 2018
Degree	Engineering
Degree Programme	Electronics Engineering
Specialisation option	
Instructor(s)	Timo Kasurinen, Senior Lecturer

The goal of this thesis project was to create a proof of concept for the integration of virtual reality tracking systems and robotic control systems. Commercially-available virtual reality systems may be used to create human-machine interfaces which require less training time than traditional interfaces in many fields.

Accurate position and orientation data can be acquired from virtual reality systems at sufficient frequency to keep up with human motion. This data can then be used to control the movement of a robotic arm without the need for traditional controls such as buttons, joysticks, or pedals.

Position data from HTC Vive virtual reality tracked controllers is collected on a PC. This data is parsed by a Unity program on the PC, turned into gcode readable by the uArm Swift Pro robotic arm, and sent to the arm over a serial connection.

The result is a system where a user holds a tracked controller and the endpoint of the robotic arm matches the movement of the user. Further work can be done to improve this project in the arena of the firmware used by the robotic arm. The movement speed is limited by how the firmware executes movement commands.

Keywords	Robotics, VR, Microcontrollers

Contents

Abbreviations

VR Virtual Reality. Specifically the use of the HTC Vive. A headset containing a separate display for each eye along with sensors to determine its position and orientation so users can perceive and virtually move through 3D-rendered environments.

Arm uArm Swift Pro shortened. A small commercially-available robotic arm.

FPGA Field Programmable Gate Array. An integrated circuit which allows concurrent execution of logic for high frequency applications.

1 Introduction

This thesis report details the results of research and developmental exploration headinto the new possibilities offered by the advent of commercially-available virtual reality hardware combined with robotic systems.

The goal is to produce a proof of concept showing that it is possible to have a precise, intuitive, realtime, 1:1 control method for a robotic arm by utilizing VR tracking technologies. This is an alternative to current control methods which generally utilize buttons, sticks, and levers. These methods are functional with sufficient training, but their form is quite divorced from what they achieve. This thesis aims to clarify the possibilities VR offers to reduce training time and make operation of machinery and robotics more natural.

Innovations in tracking technology have now made their way into consumer products, but the full range of the possibilities VR and the tracking technologies they use have yet to be fully explored. Of particular note is the way these consumer products package the data they use. The HTC Vive, for example, has a controller which contains photodiodes, accelerometers, FPGAs, and assorted other complex electronics. Yet, the software development kit offered with this VR system allows developers to simply access coordinates as variables without any specific knowledge about how these electronics work. This is a huge advancement which cuts down prototyping time immensely and marks a stark contrast with previous decades in which such things were technically possible, but almost completely inaccessible outside of specially-equipped research and development laboratories staffed by experts in a variety of fields. This thesis report documents one area of these possibilities.

2 Virtual Reality In Robotics

In this section the research, work, and decisions leading up to the thesis work itself are described.

2.1 Problem Breakdown

The core problem examined in this thesis is one of human-machine interface. The human-machine interfaces used today are a product of many factors, technological limitations not the least among them. As we advance it is prudent to look back and re-examine old problems in order to find applications for new techniques, materials, technologies. Particularly relevant to this study are the complex control systems typically present in heavy machinery used in the construction and logging industry, as well as cargo transportation and other fields. These machines often require a high degree of skill and training to operate safely and effectively at speed. So much so that it is not unheard of to overhaul a cockpit to allow a skilled operator who has lost the use of his or her legs to continue working rather than hire a new operator. In addition, such mechanical systems necessitate the operator being in a cockpit physically attached to the machine. This is especially notable in the case of large crane operators, since simply getting from the ground to the cockpit and back is an ordeal not undertaken lightly.

2.1.1 Alternatives

It follows that alternatives to these complex industry-standard control schemes may be explored. One possibility considered is building a sleeve full of sensors and switches of various kinds. An operator might put an arm in this sleeve and move around normally to manipulate a crane arm or backhoe. While certainly possible to build, such systems tend to be quite bulky, uncomfortable, susceptible to slop, and require adjustment per operator to maintain accuracy due to human differences in size and proportion. Virtual reality and associated tracking systems may have more potential. These potential benefits are described elsewhere in this document since they were the selected solution.

2.2 Hardware Selection

As of April, 2018 there are four major virtual reality platforms commercially available - HTC Vive, Oculus Rift, Playstation VR, and Windows Mixed Reality. The absolute basics of each platform's operation will be listed here.

Figure 1: HTC Vive headset, two tracked controllers, and the two "lighthouse" base stations

HTC Vive

The HTC Vive, seen above in Figure 1, consists of a VR headset, tracked controllers, and two "lighthouses". The headset and controllers have many photodiodes arrayed on their surfaces. The lighthouses are small cubes, they are placed in opposite corners of the defined VR use space and angled downward somewhat. These lighthouses house an array of LEDs and a pair of mirrored spinners. The LED array flashes at 60Hz. This is used to synchronize timing for the headset and tracked controllers. Following a flash the horizontal and then vertical spinners pan another LED across the room. Each photodiode times how long it takes from seeing the synchronization flash to seeing the horizontal and vertical pan. By comparing timings to other photodiodes on the same object, both position and orientation of that object can be determined with a high degree of accuracy with low latency, this can be seen in Figure 2 directly below.

.

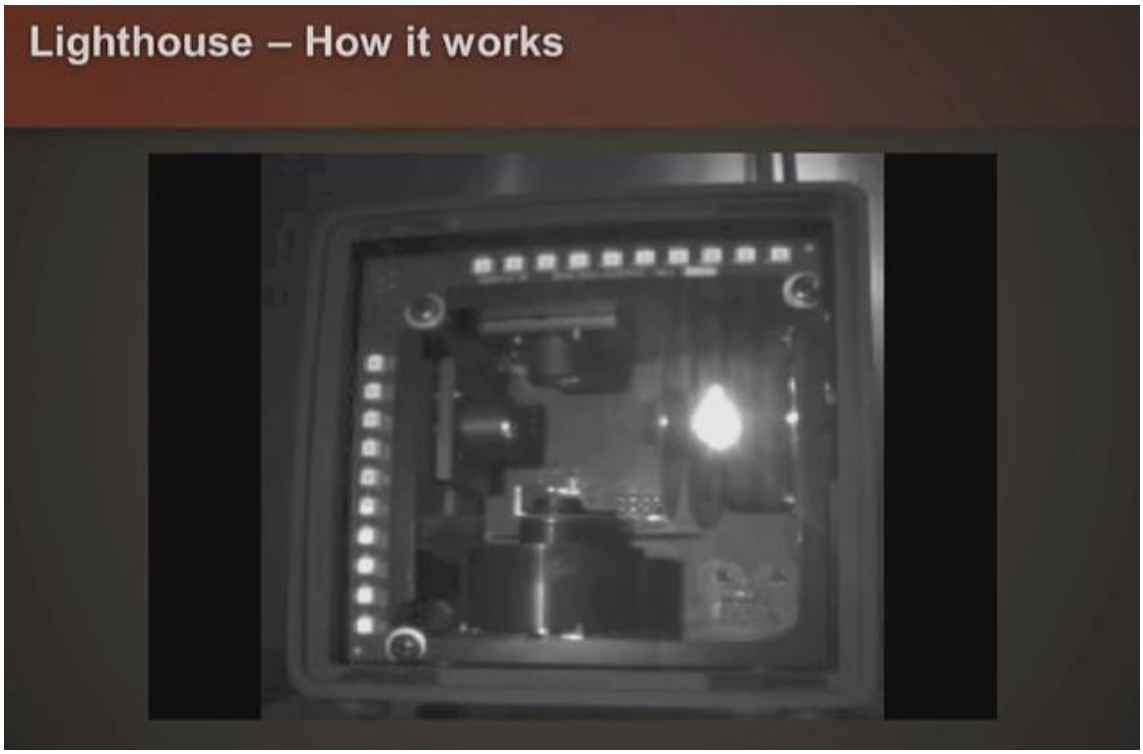

Figure 2: HTC Vive "lighthouse" with front cover removed. Image in infrared.

Figure 3: Oculus Rift headset, two tracked controllers, and two "constellation" sensors

Oculus Rift

The Oculus Rift, seen in Figure 3, consists of a VR headset, “constellation” sensor(s), and optional “touch” tracked controllers. The headset and controllers have an array of infrared LEDs on their surfaces. The “constellation” sensor is in essence an IR camera. Since many IR LEDs are visible to the camera no matter the orientation of the headset or controller, and their positions on the tracked object are both known and fixed, position and orientation can be determined. While under optimal conditions the accuracy and latency of tracking is similar to that of the HTC Vive, the range at which tracking quality starts to degrade is significantly shorter since the resolution “constellation” sensor is limited.

Figure 4: Playstation VR headset, “VR camera”, and two tracked controllers

Playstation VR

Playstation VR, seen in Figure 4, operates similar to the Oculus Rift, except that it uses fewer IR LEDs. Instead, each IR LED is polarized so that its orientation can be determined by the IR camera. This gives more data from which position and orientation can be inferred per IR LED. In addition, only one stereoscopic IR camera is used, making Playstation VR a “front facing” only system since turning away from the IR camera results in intermittent loss of tracking due to the user’s body occluding IR LEDs. Playstation VR must be connected to a Playstation 4.

Figure 5: Windows Mixed Reality headset and two tracked controllers, this model made by Asus

Windows Mixed Reality

Windows Mixed Reality, an example model seen above in Figure 5, consists of a VR headset and optional tracked controllers. Utilizes an “inside-out” tracking system which requires no external devices. This gives the advantage of not needing to be confined to a predefined VR use space. It works by using stereoscopic cameras and laser rangefinders mounted in the headset to scan the world around around the user and build a simplified 3D model of it. However, this means that anything outside the current visible arc of the headset cannot be tracked or updated. The tracked controllers cannot function outside of user view.

Since Windows Mixed Reality only became available after this thesis project was started it was disqualified from selection. Playstation VR was disqualified because the requisite hardware was not available for use. Also, since it is tied to a Playstation 4 the processing power available is rather limited compared to PC. The HTC Vive was selected for two reasons - First, the tracking seems to be a little more solid than the Oculus Rift. Second, a HTC Vive was available, and ready for use.

All of the above described VR systems are vulnerable to reflective surfaces and use areas should be not have mirrors or exposed windows to minimize tracking errors. In addition, all systems include accelerometers and gyroscopic sensors for “dead reckoning” estimations in case the primary tracking system loses a tracked object briefly.

Figure 6: uArm Swift Pro with various end effectors and accessories

As for the robotic arm, after some research into options available the uArm Swift Pro, seen in Figure 6, was selected for its familiar Arduino-based internals and simply because that is what the school offered. It should be noted, however, that the control methods discussed in this thesis report should function with any kind of robotics.

2.3 Previous Work

Before this project could begin a number of questions needed to be answered. The whole project hinges on some assumptions which needed verification, or at least preliminary exploration which would imply that the core idea was possible. This was done through use of a HTC Vive, Arduino microcontrollers, and the Unity development environment. Previous experience in these areas lead to a concept for an applicable design, and innovation project.

2.4 Innovation Project - “FLOOR CLEAN ROBOT!!”

The innovation project which laid the groundwork for this project was completed in the summer of 2017. FLOOR CLEAN ROBOT!! is a small, two-wheeled, Arduino-based robot which has no onboard sensors. It simply runs a listen server and executes motor instructions received over wifi. A Unity program running on a PC with a HTC Vive handles all navigation decisions and sends simple commands such as “move forward” or “turn left” to the robot through the listen server over wifi. In one relevant mode, a Vive controller is placed on top of the robot while the other is held by the user. Calculations are made comparing the direction the robot is facing (known by the orientation of the controller resting atop it) to the desired direction (calculated by comparing the positions of the two controllers) which are used to decide what action to take next. The goal is for the robot’s position to be close to the user, but not underfoot.

2.4.1 Operation

FLOOR CLEAN ROBOT!! operates with simple DC motors. It lacks the more sophisticated servo motors of the uArm. It does not care how much or how fast it moves, and it is incapable of losing calibration. It simply rotates toward the desired direction. If it overshoots, it rotates back the other way. This loop continues until a heading within tolerances is achieved. It then drives forward until either heading leaves tolerances, or position is close enough to user.

The result of this project was quite fast and responsive, lending evidence to the idea that VR tracking solutions may be used to operate robotics at any scale, regardless of the nature of movement.

2.5 What VR Offers

Recent advances in commercially available VR tracking systems have the potential to solve some longstanding problems with robotic control systems. It is very difficult for a free-roaming robot to really know where it is. For example, there are many commercially-available robots available designed to clean floors. The simplest, cheapest varieties typically work by driving forward until an object is struck. This is known by a button placed on the front of the robot which is pressed by the act of driving into a wall. On button press, the robot will reverse briefly, then turn a random amount and drive forward again. This repeats and should eventually provide near-total floor coverage. This is inefficient. More sophisticated models may instead drive in spirals and make use of ultrasonic and/or laser rangefinding sensors to roughly map out the area in which it operates. This extra data paired with some clever math can be a big improvement in efficiency over the basic models, but the fact remains that these robots do not really know where they are at any given time.

3.5.1 VR Advantages

VR tracking gives FLOOR CLEAN ROBOT!! a strong navigational advantage over the commercially-available floor cleaning robots described above. The VR tracking system provides exact coordinates and orientation of the robot at all times, and these virtual coordinates are mapped one to one with real, physical space. This means that FLOOR CLEAN ROBOT!! can be programmed to traverse the tracked space with very little retread of previously-covered floorspace in a logical manner (no bouncing around the room randomly). This also means “keepout” areas can be defined in software without a need for additional hardware.

2.5.2 VR Possibilities

In order to further explore the possibilities offered it was decided to continue using the HTC Vive but this time see what could be done with a robotic arm. It should be quite possible to control this arm with direct one-to-one movements of a VR tracked controller. Furthermore, there seems to be no reason why such technology could not be scaled up or down freely. This may allow heavy machinery (for example, a crane or backhoe) or micro robotics (as used in medical applications) which typically have complicated control schemes requiring a great deal of training and skill to operate

proficiently to instead be operated with intuitive, natural movements as simple and relatable as picking up a television remote.

2.6 Other Works

When this project began there was not much yet published on the subject of interactions between the HTC Vive and robotics. However, checking again at the conclusion of the project reveals that a number of similar projects have been published in the intervening months. For example - SRI International's Taurus Dexterous Robot was recently integrated with the Oculus Touch [1], or University of Tokyo's JAXON robot which was recently integrated with the HTC Vive [2]. Another partial example already in use is Hiab's HiVision [3] system for logging trucks which uses an Oculus Rift headset paired with traditional crane controls.

3 Development

3.1 Requirements and Test Specifications

Given the stated intent of this study, it was decided that the general development goal would be to achieve the highest speed and accuracy of uArm movement matched to user movement without altering the default firmware or hardware. This means that, whatever the avenue of communication, what the uArm receives are gcode commands. This choice was made based on keeping development time to a reasonable level, and due to the risk of damage to the uArm inherent to working outside its intended use.

3.2 uArm Familiarization

In the second week of October 2017 a uArm Swift Pro was borrowed from Metropolia for this project. After reading the start guide [4] and installing the associated uArm Studio [5] software initial explorations began. The uArm Studio quickly proved to be completely inadequate for this project, lacking any capability to interact with other programs or devices. It seems to be aimed primarily at entry-level hobbyists only interested in recording and playing back predefined movements.

3.2.1 Systems Setup

After giving up on uArm Studio, tests were conducted using a serial monitor to manually send gcode commands [6] over serial using the USB connection between PC and uArm Swift Pro. These tests proved that the arm works much like a 3D printer, as the documentation suggests. Next, development moved to Unity where serial communications were initiated using Alan Zucconi's tutorial on serial communications between Unity and Arduino [7] as a starting point, and code written to create a button which sent predefined gcode commands over serial. With that in place the SteamVR Plugin was added from the Unity asset store. The SteamVR Plugin [8] provides access to tracking data from the HTC Vive. This data was applied to dynamically build gcode commands.

3.2.2 First Version

While the command is updated on every loop of the code, at this point it was only sent to the uArm Swift Pro when a button was pressed for sake of testing stability. Once stable functionality was confirmed a simple system for automatic updates was written. This was disastrous because it sent a new command on every update, and updates occur at 60Hz. While initially it would seem to work, the arm quickly fell behind a backlog of commands and became unresponsive until reset. Also, in the case that the command instructs the arm to move to an out-of-bounds area, it immediately responds with 'ok' (and an error code) which floods the serial connection both ways.

3.2.3 Bad Fixes

As a quick temporary fix some bounding math was added to the command generation code to prevent bad commands from being built. This limited the arm's range of movement, as it was a simple cube. These bounds were applied exclusively to the Absolute Movement mode described below, and eventually removed in favor of refined command-sending code. The first solution to this problem was to reduce the update rate to a sane level. This solution worked somewhat, but resulted in very clunky, stilted operation.

3.2.4 Good Fixes

At this point the idea occurred to make use of the ‘ok’ acknowledgement the uArm Swift Pro sends back after each command is completed. Code was written so that a new commands are sent automatically, but only after an ‘ok’ has been received. Problem solved quite nicely, but in doing so reveals another one: tested independently - the arm refuses to accept new commands while one is already in progress.

3.2.5 Bad Commands

This line of research satisfied for the moment, development turned to further investigation into possible gcode commands. Searching revealed two promising commands- M112 (emergency stop) and M410 (quick stop). These sounded useful for interrupting a movement command and sending a new one, but both were quickly abandoned. M112 put the arm into an unresponsive state requiring a power cycle to restore from. M410 does successfully interrupt a command in progress, however it results in complete loss of position calibration in the uArm Swift Pro. This means that sending commands after a M410 sends the arm into a self-destructive seizure as it attempts to twist itself beyond its physical constraints. Needless to say, development moved on to other avenues immediately.

3.2.6 Alternate Movement Modes

The next thing developed was the Relative Movement mode. Since Absolute Movement mode is unresponsive during long movements, Relative Movement mode sought to provide improved tracking by “nudging” the arm small distances As with the other two modes, it is described in detail below. After Relative Movement mode proved to merely trade one set of shortcomings for another, Compromise Movement mode was developed. This mode utilized the lessons learned from the previous two modes to achieve a relatively happy medium between speed and responsiveness. See Figure 7 below for a flow chart of the Unity program.

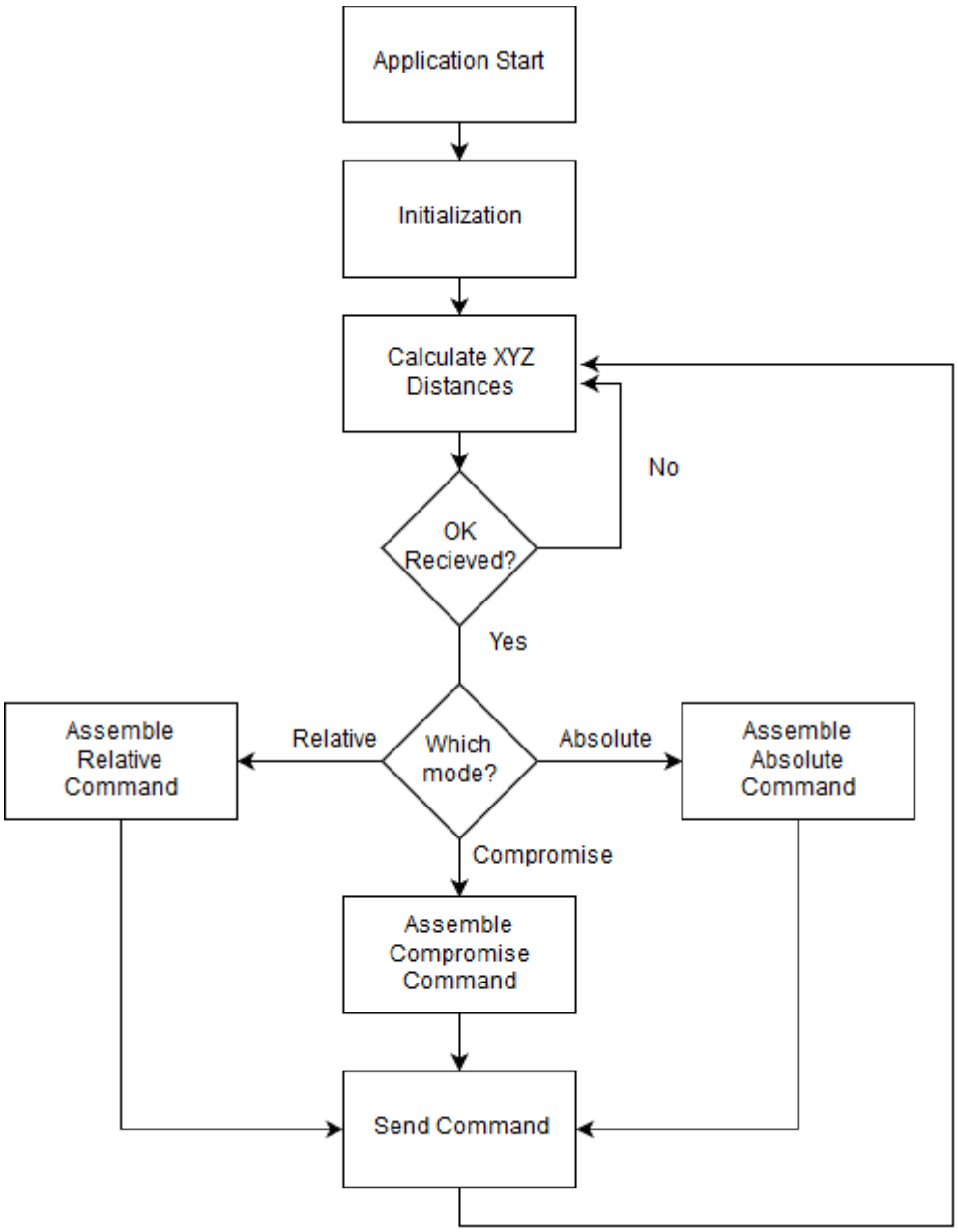

Figure 7: Flow chart for Unity program

3.3 Results

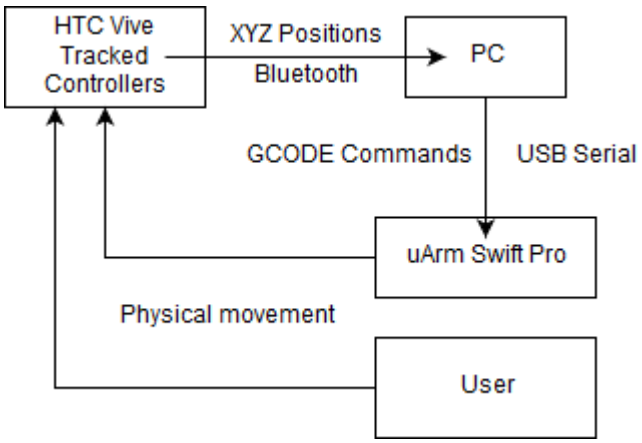

Figure 8: Block diagram of the system

Programming in this thesis project was done in the Unity development environment using C# and gcode. C# is a programming language developed by Microsoft in common use. gcode is a programming language commonly used by machine tools and, in recent years, consumer-grade 3D printers. The specific subset of gcode commands used by the uArm Swift Pro are 3D printer commands, since its firmware is a modification of the Marlin open source 3D printer firmware. [9] The uArm Swift Pro is connected to the PC by a USB cable which provides both power and an avenue of communication. At the start of the program serial communications between the PC and uArm. Once communications are established the program sends serial commands formatted in gcode, which the uArm understands natively. In order to prevent a buffer of commands from building up in the case that the program generates commands faster than the arm can execute them the program is limited by only sending a new command after an “ok” message is received from the arm. This “ok” message is sent by the arm after a command has completed execution, or when an “unreachable” command was sent. The block diagram for the whole system may be seen in Figure 8 above. The complete C# code used for this project may be seen in Appendix 1.

3.3.2 Absolute Movement

Figure 9: uArm Swift Pro and HTC Vive tracked controllers, positioned for use in Absolute Movement mode

Absolute Movement mode does not stray far from the uArm's roots in 3D printing software. In fact, it uses very standard 3D printing gcode commands to tell the arm where to go. In this mode one tracked controller is placed next to the base of the arm. The other tracked controller is held by the user. In operation the Unity program uses the relative positions of the tracked controllers to tell the arm where to go. This results in the arm moving to match the position of the held controller with a small offset controlled by how far from the base the resting tracked controller is placed. Tracked controller positioning for Absolute Movement mode may be seen above in Figure 9.

Absolute Movement mode utilizes the G0 gcode command. Its format is

```
#___ G0 X___ Y___ Z___ F___
```

Listing 1. Example gcode command with blanks

Where the ___ blanks seen in Listing 1 above are filled by some number. #___ is any number. It can be used for error tracking as “unreachable” messages are accompanied by it, though it is unused in this thesis project. G0 identifies what command this is. There are variants to it, but again they aren’t used in this thesis project. X___ tells the arm what x-position to move to, measured in millimeters. Y and Z are the same for their respective axes. F___ is a speed setting. It doesn’t seem to have a unit, but developer comments suggest that 30000 is the maximum speed. For the purposes of this project faster movement is always better.

In this project the X Y and Z variables are found with the code in Listing 2.

```
Xposition = Mathf.RoundToInt((Shoulder.position.x - transform.position.x)*1000);
```

Listing 2. Calculation for X position of arm in Absolute Movement mode

Mathf.RoundToInt takes a float and rounds it to the nearest integer value. Shoulder.position.x is the x position of the stationary tracked controller. transform.position.x is the x position of the held tracked controller. By subtracting them from each other we get a value for their relative positions. Since Unity operates in meters and the uArm operates in millimeters, this value is multiplied by 1000.

These variables are then fed into a string concat function in order to generate the complete gcode command as seen in Listing 3.

```
string.Concat ("#25 G0 X", Xposition, " Y", Zposition, " Z", Yposition, " F30000")
```

Listing 3. Complete gcode command generator

3.3.2.1 Absolute Limitations

This mode functions adequately, but has some inherent limitations. The most glaring problem is that due to how the arm functions a command cannot be interrupted without the arm losing track of where it is. So, any changes in position the user may make while the arm is in motion are lost. This is especially visible if one makes a large sweeping motion from the left side of the arm’s reach to the right. It is not especially fast, so there is a second or two during which the user may wave the tracked controller around freely with no response from the arm. It was initially thought that perhaps the “quickstop” gcode command could be used to interrupt movement commands to

update them with new coordinates, but this merely results in the arm trying to fold in on itself due to loss of calibration. Another problem can be found by, during a long movement, moving the held controller outside the movement range of the arm. This will return “unreachable” errors until the held controller is returned to a place the arm can reach.

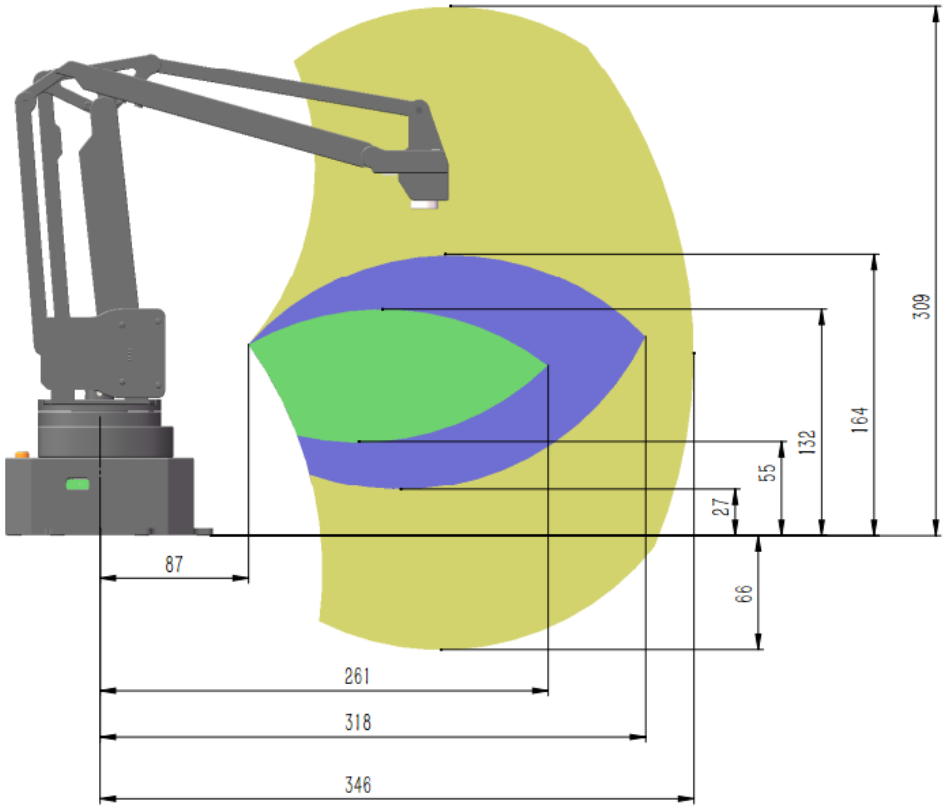

Figure 10: uArm Swift Pro movement range as viewed from the side, values in millimeters. Colors probably indicate either speed or accuracy in different zones, but the original documentation presents the image without comment. [10]

As one may infer from Figure 10 above, a simple bounding box for coordinates would make the usable movement range quite small.

3.3.3 Relative Movement

Figure 11: uArm Swift Pro and HTC Vive tracked controllers, positioned for use in Relative Movement mode

The Relative Movement mode operates on the principle of “nudging” the arm toward a destination. In this mode one tracked controller is held in the arm’s gripper claw while the other is held by the user as seen in Figure 11 above. At any time during operation a button may be pressed to set the desired “zero” position between the arm and the user controller. The arm continually makes very small movements which takes it toward this “zero” position. This serves to bypass the primary problem inherent in the Absolute Movement mode of being unable to adjust course during large movements.

It uses command G2204 which adjusts the position of the arm by XYZ millimeters. It is otherwise identical in structure to command G0 used by the Absolute Movement mode. This may be seen below in Listing 4.

```
#___ G2204 X___ Y___ Z___ F___
```

Listing 4. Relative Movement mode gcode command format

This mode makes use of the same Xposition variables as Absolute Movement mode, but applies them differently. When the “zero” button is pressed the current values of Xposition is stored in another variable called XpositionZero. To assemble the gcode command this zero is compared to the current position. For example, if Xposition is larger than XpositionZero then Xnudge will be set to 5. This means that the arm will move 5 millimeters to the right. Conversely, if Xposition is smaller than XpositionZero then Xnudge will be set to -5 and thus the arm will move 5 millimeters to the left. This is, of course, repeated for the Y and Z axes.

3.3.3.1 Relative Limitations

Relative Movement mode successfully circumvented Absolute Movement mode’s shortcomings in the realm of command update speed, but at the cost of actual movement speed. Since the motors are constantly starting and stopping it takes much longer to travel the same distance.

3.3.4 Compromise Movement

The Compromise Movement mode attempts to merge the speed of Absolute Movement mode with the agility of Relative Movement mode. The idea is to use the same commands Relative Movement uses, but rather than “nudging” the arm very small amounts toward the desired location it calculates how far the desired position is and then moves exactly halfway there. The hope being that this would allow larger distances to be covered fairly quickly, but the arm would not be stuck executing a command (and thus not listening to new input) for as long as proved problematic in Absolute Movement mode. The Compromise() function is nearly identical to Relative() with the sole difference being the if/else structure to determine if an axis is set to +5, 0, or -5 has been replaced with a single line for each axis.

```
Xnudge = Mathf.RoundToInt(Xposition - XpositionZero)/2;
```

Listing 5. Calculation for X axis movement distance in Compromise Movement mode

This line rounds the distance between where the arm is and where it should be to a whole number, then divides that number by two.

3.3.4.1 Compromise Limitations

In practice, Compromise Movement mode is true to its name. It does improve on previous modes in some ways, though not without making sacrifices in other areas. It is much faster in operation than Relative Movement mode, but it does have (though reduced) the lockup problem of Absolute Movement mode. A stutter is visible at each halfway point where the arm stops to accept the next command, and when making small adjustments very close to the target location it does resemble Relative Movement mode. This is still preferable to the CONSTANT stuttering of Relative Movement mode at every distance.

3.4 Discussions Overview

The purpose of this section is to discuss the results of the thesis work. The primary results are the three movement modes developed in an attempt to best match the movements of the uArm Swift Pro to the user's hand (holding an HTC Vive controller). However, before comparing movement modes the hardware should be discussed.

The uArm Swift Pro is certainly capable of moving point-to-point at a reasonable speed compared to other consumer devices currently available. It is well-made and since it uses a modified 3D printer firmware the movements are easily accurate enough for this application.

On the tracking side of things the HTC Vive met all the needs of this project, able to supply accurate location data far faster than the arm could possibly use.

3.4.1 Discussions Mode Comparisons

The main tradeoffs between the three modes are first the “point-to-point speed” defined here as the speed at which the arm moves from a starting position to single new position and second, the “agility” defined here as ability to handle updates to the target arm position. In terms of point-to-point speed the Absolute Movement Mode is the clear winner - moving swiftly and accurately to a specific point is what 3D printers are made for, after all. On the other end of the spectrum is Relative Movement Mode which, since it breaks one large movement into dozens of small movements, is extremely slow. This is because each small movement needs to be bookended by a “start” and a “stop”. Each of these takes a fraction of a second, but if one is needed for every 10 millimeters moved the delay adds up. Right in the middle for point-to-point speed is, as its name suggests, Compromise Movement Mode. Since this mode moves halfway its destination rather than a set few millimeters it takes far fewer movements to cover the same distance compared to Relative Movement Mode and thus less time. These starts and stops are still present, however, so it can never match the speed of Absolute Movement Mode.

Conversely, in terms of agility Absolute Movement Mode performs very poorly. As covered in the Results section, the arm’s firmware effectively prohibits interrupting a movement command as would be desirable to update the destination to keep up with the user’s movements. This means that if the user moves to the left, the arm will move to the left, but if the user moves to the right during the arm’s leftward movement this new destination will be ignored until the previous destination on the left has been reached. Since a new command cannot be accepted until the previous command completes, and Relative Movement Mode can make multiple small movements per second its agility is the best of all three modes. Compromise Movement Mode’s agility rates somewhere between the other two modes. With small user movements its performance is visually indistinguishable from Relative Movement Mode, and with large movements it better than Absolute Movement Mode by a large margin.

While far from perfect, Compromise Movement Mode is clearly the best of the three.

3.5 Conclusions

The goal of this project was the produce a proof of concept which shows that it is possible to use virtual reality tracking systems to effectively control robotics. This goal was achieved, showing good accuracy. However, the physical speed of the uArm Swift Pro is not enough to keep pace with normal human arm movements, so the practical usefulness of this particular hardware is limited. Despite hardware limitations, the concept is sound and there is no reason to think that, with faster robotic hardware, good 1:1 movement could not be achieved.

As mentioned above, there are inherent speed limitations in the uArm Swift Pro firmware. Because of how it handles calibrated servo-driven movement, it needs to come to a complete stop at the end of every movement instruction. No matter how optimized the Unity command structure becomes, this start-stop nature puts a hard limit on improvement. Therefore, while it is outside the scope of this thesis, a probable next step would be to write new firmware for the uArm Swift Pro.

This new firmware could abandon the calibrated servo movement and instead rely on Vive tracking to build a safe bounding box for movement. Since this new firmware would not include the (now unnecessary) start-stop of the default firmware, it should be capable of significantly faster navigation. While certainly possible, writing this firmware would be a large undertaking and carry significant risk of damage to the (quite expensive) uArm Swift Pro during development as the official documentation regarding safe movement range is rather lacking. However, even with firmware better-suited for the task it is unlikely to be able to reach desired speeds given observations of the arm during long, uninterrupted lateral movements (the highest speed observed) using the current firmware.

4 References

1 VRscout News. Dieter Holger. “This Robot Disarms Bombs Through Virtual Reality” URL: https://vrscout.com/news/taurus-bomb-robot-virtual-reality Accessed 27 October 2018.

2 VRscout News. Kyle Melnick. “Researchers Command Adult-Size Robot Using Vive Controllers” URL: https://vrscout.com/news/adult-size-robot-vive-controllers Accessed 27 October 2018.

3 HIAB. HiVision LOGLIFT - JONSERED. Product description. URL: https://www.hiab.com/en-GB/pages/loglift-jonsered/HiVision/ Accessed 27 October 2018.

4 uArm Swift Pro Quick-Start Guide V1.0.15. Previously http://www.ufactory.cc/#/en/support no longer available.

5 http://www.ufactory.cc/#/en/support uArmStudio software. UFACTORY. Version 1.1.18. 2018

6 uArm Swift Pro Quick-Start Guide V1.0.15 pages 47-54 Previously http://www.ufactory.cc/#/en/support no longer available.

7 Alan Zucconi. “How To Integrate Arduino With Unity” http://www.alanzucconi.com/2015/10/07/how-to-integrate-arduino-with-unity/ Accessed October 27 2018.

8 https://assetstore.unity.com/packages/templates/systems/steamvr-plugin-32647 Unity plugin. SteamVR Plugin. Valve Corporation. 2017

9 http://marlinfw.org/ Base for uArm Swift Pro firmware. Marlinfw. Marlin. 2017

10 uArm Swift Pro Quick-Start Guide V1.0.15 page 5 Previously http://www.ufactory.cc/#/en/support no longer available.

Appendix 1 Code - ArduinoConnector

```
using UnityEngine;
using System;
using System.Collections;
using System.IO.Ports;
using UnityEngine.UI;
using System.Net;
using System.Net.Sockets;

public class ArduinoConnector : MonoBehaviour {

    /* The serial port where the Arduino is connected. */
    [Tooltip("The serial port where the Arduino is
connected")]
    public string port = "COM9";
    /* The baudrate of the serial port. */
    [Tooltip("The baudrate of the serial port")]
    public int baudrate = 115200;
    public int Xposition = 180;
    public int Yposition = 0;
    public int Zposition = 150;
    public Text Readout;
    public Text WhatArmSay;
    public Transform Shoulder;
    //int honkhonk = 0;
    string armdo = "$25";
    string thingdo = "#25 G0 X";
    //bool donedid = false;
    int lastcheck = 0;
    int XpositionZero = 0;
    int YpositionZero = 0;
    int ZpositionZero = 0;
    string thingdoRelative = "hoop hoop";
    int Xnudge;
    int Ynudge;
    int Znudge;
```

```
    public int modeselect = 0;

    private SerialPort stream;

    public void Open () {
        // Opens the serial port
        stream = new SerialPort(port, baudrate);
        stream.ReadTimeout = 50;
        stream.Open();
        //this.stream.DataReceived += new
SerialDataReceivedEventHandler(DataReceivedHandler);
        Debug.Log("didit");
    }

    public void WriteToArduino(string message)
    {
        // Send the request
        stream.WriteLine(message);
        stream.BaseStream.Flush();
    }

    public string ReadFromArduino(int timeout = 100)
    {
        stream.ReadTimeout = timeout;
        try
        {
            return stream.ReadLine();
        }
        catch (TimeoutException)
        {
            return null;
        }
    }


    public IEnumerator
AsynchronousReadFromArduino(Action<string> callback,
```

```
Action fail = null, float timeout =
float.PositiveInfinity)
    {
        DateTime initialTime = DateTime.Now;
        DateTime nowTime;
        TimeSpan diff = default(TimeSpan);

        string dataString = null;

        do
        {
            // A single read attempt
            try
            {
                dataString = stream.ReadLine();
            }
            catch (TimeoutException)
            {
                dataString = null;
            }

            if (dataString != null)
            {
                callback(dataString);
                yield return null;
            } else
                yield return new WaitForSeconds(0.05f);

            nowTime = DateTime.Now;
            diff = nowTime - initialTime;

        } while (diff.Milliseconds < timeout);

        if (fail != null)
            fail();
        yield return null;
    }
```

```
    public void Close()
    {
        stream.Close();
    }

    public void Start()
    {
        Open();
    }

    public void Update()
    {
        //Universal mathing

        //Limits seemed to cause problems for Relative mode and just generally need refinement, so commented out for now
        Xposition = Mathf.RoundToInt((Shoulder.position.x - transform.position.x)*1000);
        /*if (Xposition < 100) {
            Xposition = 100;
        }
        if (Xposition > 350) {
            Xposition = 350;
        }*/
        Yposition = Mathf.RoundToInt((Shoulder.position.y - transform.position.y)*1000);
        /*if (Yposition < -100) {
            Yposition = -100;
        }
        if (Yposition > 100) {
            Yposition = 100;
        }*/
        Zposition = Mathf.RoundToInt((Shoulder.position.z - transform.position.z)*1000);
        /*if (Zposition < 100) {
```

```
            Zposition = 100;
        }
        if (Zposition > 350) {
            Zposition = 350;
        }*/
        //Xposition                                          =
Mathf.RoundToInt(transform.position.x * 100);
        //Yposition                                          =
Mathf.RoundToInt(transform.position.y * 100);
        //Zposition                                          =
Mathf.RoundToInt(transform.position.z * 100);

        //Close gripper
        if (Input.GetKeyDown (KeyCode.Keypad1)) {
            WriteToArduino ("M2232 V1");
        }

        //Open gripper
        if (Input.GetKeyDown (KeyCode.Keypad2)) {
            WriteToArduino ("M2232 V0");
        }

        if (modeselect == 0) {
            Absolute ();
        } else if (modeselect == 1) {
            Relative ();
        } else if (modeselect == 2) {
            Compromise ();
        }
    }

    public void Selector(int selection)
    {
        modeselect = selection;
    }

    public void Absolute()
```

```
    {
        thingdo = string.Concat ("#25 G0 X", Xposition, "
Y", Zposition, " Z", Yposition, " F30000");
        Readout.text = thingdo;
        if (Input.GetKeyDown (KeyCode.Space))
        {
            armdo = ReadFromArduino ();
            WhatArmSay.text = armdo;
        }

        if (Input.GetKeyDown (KeyCode.Keypad0)) {
            Debug.Log (thingdo);
            WriteToArduino (thingdo);
        }

        armdo = ReadFromArduino ();
        WhatArmSay.text = armdo;

        if (lastcheck == 1) {
            WriteToArduino (thingdo);
            lastcheck = 0;
        }
        if (armdo != null) {
            lastcheck = 1;
        }
    }

    public void Relative()
    {
        //Set zero
        if (Input.GetKeyDown (KeyCode.Keypad3)) {
            XpositionZero = Xposition;
            YpositionZero = Yposition;
            ZpositionZero = Zposition;
        }

        //Changed from 10 to 5
```

```
        if (Xposition > XpositionZero) {
            Xnudge = 5;
        } else if (Xposition == XpositionZero) {
            Xnudge = 0;
        } else {
            Xnudge = -5;
        }

        if (Yposition > YpositionZero) {
            Ynudge = 5;
        } else if (Yposition == YpositionZero) {
            Ynudge = 0;
        } else {
            Ynudge = -5;
        }

        if (Zposition > ZpositionZero) {
            Znudge = 5;
        } else if (Zposition == ZpositionZero) {
            Znudge = 0;
        } else {
            Znudge = -5;
        }

        thingdoRelative = string.Concat ("#37 G2204 X",
Xnudge, " Y", Znudge, " Z", Ynudge, " F30000");

        armdo = ReadFromArduino ();
        WhatArmSay.text = armdo;
        Readout.text = thingdoRelative;

        if (lastcheck == 1) {
            WriteToArduino (thingdoRelative);
            lastcheck = 0;
        }
        if (armdo != null) {
            if (armdo.Contains("ok")){
```

```
                lastcheck = 1;
            }
        }

        if (Input.GetKeyDown (KeyCode.Keypad0)) {
            Debug.Log (thingdoRelative);
            WriteToArduino (thingdoRelative);
        }
    }

    public void Compromise()
    {
        //So, like, do the thing
        //The thing with nudging halfway to destination

        //Set zero
        if (Input.GetKeyDown (KeyCode.Keypad3)) {
            XpositionZero = Xposition;
            YpositionZero = Yposition;
            ZpositionZero = Zposition;
        }

        //blrgbllebl bl
        Xnudge      =      Mathf.RoundToInt(Xposition      -
XpositionZero)/2;
        Ynudge      =      Mathf.RoundToInt(Yposition      -
YpositionZero)/2;
        Znudge      =      Mathf.RoundToInt(Zposition      -
ZpositionZero)/2;

        thingdoRelative  =  string.Concat  ("#37  G2204  X",
Xnudge, " Y", Znudge, " Z", Ynudge, " F30000");

        armdo = ReadFromArduino ();
        WhatArmSay.text = armdo;
        Readout.text = thingdoRelative;
```

```
            if (lastcheck == 1) {
                WriteToArduino (thingdoRelative);
                lastcheck = 0;
            }
            if (armdo != null) {
                if (armdo.Contains("ok")){
                    lastcheck = 1;
                }
            }

            if (Input.GetKeyDown (KeyCode.Keypad0)) {
                Debug.Log (thingdoRelative);
                WriteToArduino (thingdoRelative);
            }
        }
    }
```