# Haptic Interactions in Virtual Reality MQP

**Major Qualifying Project Report:**

Submitted to the Faculty of

In partial fulfillment of the requirements for the

Degree of Bachelor of Science

In association with

WPI Automation and Interventional Medicine Laboratory

By


**Richard Matthew Rafferty** RBE

**Jiaqi Ren** RBE/ME


**Professor Gregory S. Fischer** Advisor

**Professor Robert W. Lindeman** Co-Advisor

# Table of Contents

## Abstract

Many possible systems exist that could benefit from Haptic Interactions, the communication of forces between a user and a system. Robotic assisted rehabilitation, interactive Virtual Reality media, and Telerobotics are some examples. However, due to simplified interactions methods, high costs, and lack of application development tools, haptic interaction with virtual environments has not reached its full potential.

As a solution towards these problems, the team developed a platform Haptic Interaction System, a platform capable of supplying Haptic Interactions between a user and hosted simulated environment and objects, along with the tools to enhance the system and develop applications based on Haptic Interactions. With this platform, future developments can be made, in hardware, software, and application design that can work to bring Haptic Interaction to its full potential and benefit for society.

# Executive Summary

## I. Background

Many possible systems exist that benefit from Haptic Interactions, the communication of forces between a user and a system. Robotic assisted rehabilitation, interactive Virtual Reality media, and Telerobotics are some examples.{Preusche, 2007 #1} However, due to simplified interactions methods, high costs, and lack of application development tools, such as with the CyberGlove Systems CyberGrasp and Novint Falcon, haptic interaction with virtual environments has not reached its full potential.

As a solution towards these problems, the team developed a platform Haptic Interaction System, capable of supplying Haptic Interactions between a user and hosted simulated environment and objects, along with the tools to enhance the system and develop applications based on Haptic Interactions

## II. Functional Requirements

In order to create this first iteration of the Haptic Interaction System platform, all key basic functionality would be needed. These functionalities were determined as the following functional requirements of the system.

- Apply forces on user's hand to simulate haptic interactions
- Track the position and orientation of the user's hand
- Present a simulated environment including virtual objects and user proxies
- Provide tools for developing applications with haptic interactions
- Construct a system that is low in cost, accessible, and expandable, for future system and application work by developers

## III. System Design

Based on this research, requirements, and design testing, the following system was created. On the user side, user interacts with virtual objects in Virtual Display through Force Exertion System. Via Leap Motion Controller, hand and fingers' pose data convey towards Simulation System and Application. Then Simulation System, based on hand pose data and sensed current from Force Exertion System, commands corresponding force feedback to Force Exertion System. In the meantime, Application displays interactions in Visual Display.

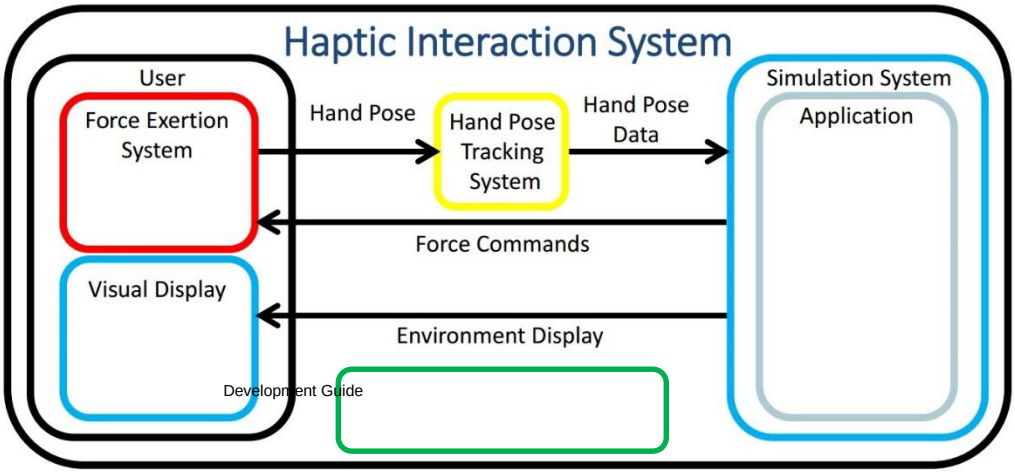



**FIGURE 1: HAPTIC INTERACTION SYSTEM SUB-SYSTEMS AND INFORMATION FLOW**

## IV. Results

Overall, the final iteration had been able to have Haptic Interactions between a user and a simulated environment, with a parts cost of approximately $400. The entire system as shown in Figure 2. For each finger, the Force Exertion System was able to apply 512 discrete forces, up to 4N on each direction. The Hand Tracking System could read hand and fingers' pose, velocity and gesture over an 8 cubic foot space. And the Simulation Environments could be generated for user proxies to interact with and generate force commands. Applications could be developed based on this iteration.

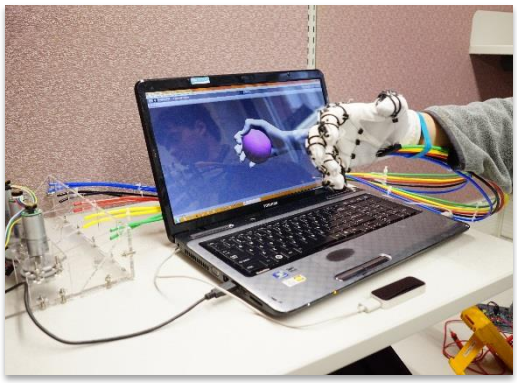


**FIGURE 2: PHOTO OF FULL SYSTEM**

## V. Discussion

In this iteration of system, each of the sub-systems met the requirements on functionality laid out for the Haptic Interaction System, with the exception of the command communication, which became unstable with integration of the other subs-systems. Together with these sub-systems it was possible to design virtual reality applications and perform haptic interactions with the objects held in their environments, so future projects can

- Enhance the systems for more functionality, such as full hand interactions
- Add systems for Virtual Reality presence, such as head mounted displays
- Develop applications using Haptic Interactions

# 1. Introduction

People interact with their environments through many senses. With the advancement of technology, people are able to expand their reach and have new types of experiences. Such is the case with Virtual Reality, a term invented over half century ago, and has been rapidly accepted by people with help from developing products like Oculus Rift and Sony Morpheus Project. An essential resource for people to interact with their surrounding in daily life is from sense of touch, which allows the understanding of texture and forces, collectively known as haptic information. Many possibilities exist in the combination of these virtual reality and haptics. However, many of these possibilities need tools and development in technology to be reached. A system combining these areas, to allow haptic interactions with virtual environments and the ability to produce applications based on such interactions could bring revolutionizing innovation to many problems of society. Integrating haptic and virtual information into applications provides a rich method of communication with the user. While development has been performed for applications with haptic and virtual interactions, it has often been restricted by costs and technologies.

Therefore, the goal of this project was to create a system to communicate haptic information between a user and a computer simulated environment, and allow for the development of applications that use haptic interactions. To achieve this goal, systems were made from commercially available parts. An exoskeleton hand was developed to allow forces to be applied to a user's fingertips. A computer system was developed to capture the user's hand movements, apply them to a simulated environment, and return the resulting forces back onto the user. To demonstrate this system an application based on learned non-use rehabilitation was developed. Therefore, a base system was developed for building and running applications using haptic interactions, to be used to develop future haptic interaction systems and applications.

# 2. Background Research

## 2.1. Virtual Reality

Virtual Reality, in its most general form, is the method of allowing the creation of interaction with artificial environments, through the use of computer modeling and simulation. The means of interaction between human and computer in virtual reality systems is directly through human senses, as opposed to traditional devices such as keyboards and mice. This definition is an overview and has been debated through the field's existence, and its history is need for further comprehension of what is and is not virtual reality.{Newton, 2015 #16}

The specific definitions of virtual reality systems depend on the type of environment being simulated, purpose of the system, and what senses it communicates through. For example, many virtual reality systems have focused only on visual and audio communication in their environments, such as with head mounted video displays (HMDs) that contained audio speakers and would alter their display based on the user's orientation. Additionally other systems focused on capturing the motions of physical interactions, such as with data gloves, a device that would measure and capture hand position and flexion to apply to virtual reality interactions, an example of which is shown in Figure 3.{Grifantini, 2009 #2}


**FIGURE 3: ACCELEGLOVE, A PROGRAMMABLE GLOVE THAT RECORDS HAND AND FINGER MOVEMENTS**

**IMAGE**:http://www.technologyreview.com/articl

Virtual reality has gone through rises and falls with it use, consumer popularity, and research attention. This history is captured in the recounts of many of early virtual reality researchers and prominent developers, in Voices from a Virtual Past.{Newton, 2015 #16} It started around in the 1950s and 60s by a few pioneers. First was filmmaker Mort Heilig who created what he called the "Sensorama" a sit down device consisting of a 3D display screen, vibrating chair, and scent produce as seen in Figure 4. The Sensorama would use combinations of these features to commune simulated experiences to its users. Heilig envisioned his device to be of a line of systems, creating the" cinema of the future."

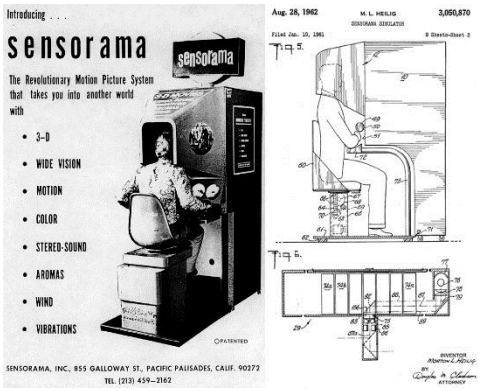



**FIGURE 4: THE SENSORAMA, ONE OF THE EARLIEST KNOWN EXAMPLES OF VIRTUAL REALITY DEVICE**

**IMAGE**:http://tallera3bandas.blogspot.com/2012/04/sensorma-simulator.html

Additionally around the development of 3D computer graphics, to go along with this new technology Ivan Sutherland proposed 'The Ultimate Display'.{Sutherland, 1964 #4} This was to be what is now considered a virtual reality system where "the computer can control the existence of matter" according to Sutherland. In 1968 he created and demonstrated a preliminary system that consisted of a head mounted video display, with the ability to track the user's head motion. This system was called "The Sword of Damocles" due its size and weight requiring it to be supported by the ceiling when in use. {Sutherland, 1965 #5}

The field of virtual reality continued from there, with much research and development being performed by different people of many different areas. Much of virtual reality's public interest was drawn in by the possibilities people imaged could be achieved through simulated environments. This early age of development often contained statements and promises of the future, in similar theme to Heilig's future cinemas and Sutherland's envisioned computer. Tom Zimmerman and Jaron Lanier founded the company VPL, most well-known for creating hand worn flexion sensors to capture hand poses, called the "DataGlove." Jon Walden founded the company W Industries, later called Virtuality, which developed head mounted displays for virtual reality systems and first debuted in 1990. {Zimmerman, 1987 #6}

**FIGURE 5: DATAGLOVE WITH OUTTER GLOVE REMOVED**

**IMAGE: {Zimmerman, 1987 #6}**

This high period in virtual reality eventually fell through, due to bankruptcies, a market shift towards home gaming systems, which could not support the approximate $60,000 systems of the time, and the simultaneous release of the internet. Thus virtual reality stopped being in public focus and research and advancements dropped significantly. During the late 1990s and early 2000s virtual reality research and use was most strongly done in the military field, for training and simulation.

Eventually, in 2012, with the release of the Oculus Rift head mounted display, public interest in virtual reality was brought back. In the meantime, the creator of Oculus Rift, Palmer Luckey, had worked at the Institute for Creative Technologies with many pioneers of previous era of virtual reality systems. Along with the research by Mark Bolas, Luckey was able to create this new system. As of 2015, there are many competing companies creating virtual reality systems, such as Sony Corporation's Project Morpheus, Oculus VR's third prototype Crescent Bay, and many others.

**FIGURE 6: OCULUS RIFT DEVELOPER KIT 3 AND SONY PROJECT MORPHEUS**

**IMAGEB**: http://www.erectronics.com/2014/03/24/oculus-rift-vs-sony-project-morpheus/

In this past year 2014, there had been several trade and acquisitions in Virtual Reality Industry such as GoPro Inc. purchased a spherical video company, Kolor, as it first step into the Virtual Reality Industry in April.{Booton, 2015 #7} In the projection of the Virtual Reality Industry, KZer estimate 2015 sales at 5.7 million units rising to 23.8 million units in 2018, as show in in Figure 7. Total cumulative units sold from 2014 to 2018 equals 56.8 million devices.{WorldWide, 2015 #8} The history shows the diverse field, wide interest, wild and expansive ideas Virtual Reality can inspire, and the past and future needed to be comprehend in order for progress in virtual reality to precede.

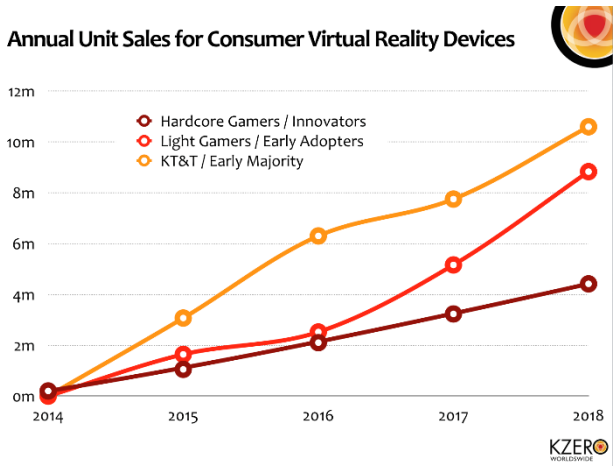



**FIGURE 7: KZER'S FORECAST OF VIRTUAL REALITY INDUSTRY**

**IMAGE: KZERO.CO.UK/BLOG/CONSUMER-VIRTUAL-REALITY-MARKET-WORTH-13BN-2018/**

Many applications currently exist that utilize virtual reality. One of the most prominent applications is in entertainment and game media. This is the primary focus with many of the technologies being developed in the resurgence of virtual reality technology. This again includes the devices from the company

Oculus VR, Sony's Project Morpheus, but was also the focus of many devices in the early era of virtual reality. Such products included Nintendo Co.'s Virtual Boy, a home gaming console composed of a stand mounted goggle style screen display and Virtuality's VR arcade pods, large sit-down style systems with head mounted displays for commercial arcades. {Newton, 2015 #1}

While entertainment has often been the medium through which the public eye sees virtual reality, its applications include many other beneficial areas. For example, in the medical field, virtual reality is used to improve many people's lives. It is helpful in cases of people with phantom limb pain, the condition where people with amputated limbs feel pain as if it is originating from their missing limb. By controlling a virtual limb through their sensors attached to their nervous system and showing the result as if attached to the person's body the patient is able to be relieved of some of their pain and be trained for using prosthetics. Another set of uses for virtual reality is in simulations. In medical fields, virtual reality systems can be used for patients to confront phobias or counter post-traumatic stress disorder. By using technologies such as head mounted displays patients can be exposed to simulated environments in safe and controlled situations, to confront their source of phobia or PTSD environments. This way over time, they can be cured of their phobia or PTSD and be able to live life as they desire. Another application related to simulation is in training.{Seymour, 2002 #19} Many systems are used to train military and medical professionals for example. Some include the Dismounted Soldier Training System, which allows army units to practice in simulated environments, using head mounted displays. Another is the dv-Trainer by mimic technologies, show in Figure 8, a system that provides training opportunities for using the Di Vinci robotic surgery device, without risk of damage to or taking time from the actual surgical system.{Lerner, 2010 #9}

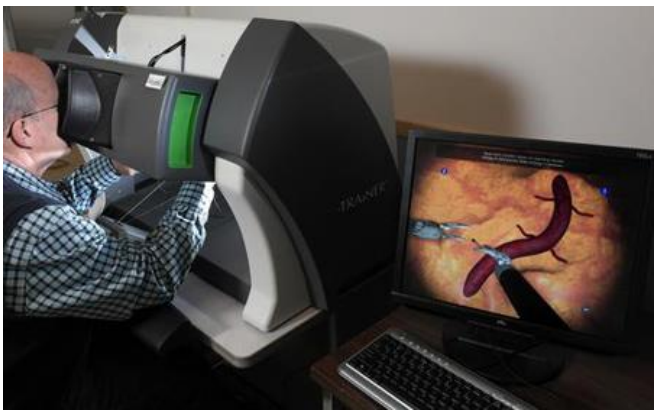


**FIGURE 8: TRAINING IN SIMULATION WITH SURGERY ROBOT**

**IMAGE:FORGEFX.BLOGSPOT.COM/2013/04/TRAINING-SIMULATORS-CRITICAL-TO-ROBOTIC.HTML**

Through this, the meaning of virtual reality, its complicated history, and some of its many potential uses have been shown. With this comprehension, the requirements of a virtual reality system and the many

ways it can be utilized can be known allowing more effective and beneficial systems to be created for society.

## 2.2. Haptics

Haptics is the field of study related to the sense of touch. In reference to humans, haptics has two primary areas, tactile and kinesthetic feedback.{Berkley, 2003 #3} Tactile feedback is the information received about an object's texture. In humans this sensed through the skin, and most prominently through the finger pads.{Massie, 1994 #13} These react to many stimuli, such as light and heavy contact, vibration, temperature, and pain, relaying the information back to the brain where the texture is perceived. Kinesthetic feedback itself has multiple aspects. In humans this is the result of proprioceptors, a series of receptors for muscles, carrying signals about changes in muscle length and tension to the somatosensory region of the cerebral cortex. These data provide a sense of body position and movement, body relative object position, and interaction forces.

It is tactile and these interaction forces, or force feedback, that is of most interest and research in human-computer interaction haptic feedback.{Srinivasan, 1995 #14} Many commercial devices exist that allow for interactions that provide these form of sensory inputs, such as vibration on mobile devices and force feedback controllers. The main focus of haptic devices is to allow for haptic information to be provided to users in order to enrich interactions, provide feedback to their interactions, and allow force communication in interactions.

In order for haptic interactions with virtual environments, three factors about interactions with environments drive the haptic device design process, according to the PHANToM haptic interface development team from MIT. {Massie, 1994 #13} This team was the original design team of a system that lead to the PHANTOM line of tool based haptic interaction devices, an example of which is shown in in Figure 9. First, the most important haptic information to provide to users is force and motion. This is because this information allows for not just human comprehension of an object and creation a spatial map, but also how it reacts to interactions and therefore how tasks can be performed with the objects. Second, many meaningful haptic interactions have little to no torque. Most basic

**FIGURE 9: PHANTOM OMNI® HAPTIC DEVICE**

**IMAGE:DENTSABLE.COM/HAPTIC-PHANTOM-OMNI.HTM**

interactions with few agents can be represented by linear forces, such as moving a weighted object from one place to another. Third, many meaningful interactions require only a small amount of space and are centered on the movement of users' fingers. Therefore, only a work space based to the human hand for unrestricted finger and wrist motion is needed for many basic, but impactful, interactions.

Furthermore, for a haptic interface to be effective in its interactions it must follow three criteria. {Massie, 1994 #13} First, free space in the virtual environment must feel free to the user. For this, the haptic device must not encumber the user such that the simulated free space interactions do not mirror unrestricted movements in real space. Second, solid objects must feel solid to the user. For this, the control loop algorithm for the haptic device must provide a suitable amount of stiffness to convey solid objects in interactions. From the PHANToM team's research, they determined that a stiffness of 20N/cm was able to convince the majority of users that an object was solid in virtual interactions.{Sutter, 1989 #22} Third, virtual constraints must not be saturated easily. The haptic interaction device must be able to supply resistance such that the user cannot easily overpower it, and cause them to pass through what is to be a solid object. From the PHANToM team's research, while the maximum exert-able human finger force is on the order of 40N of force, in the majority of interactions people do not exert more than 10N, with the time average force on the order of 1N. Therefore, by following these observations and criteria a simplistic but useful and effective haptic interaction device can be designed.

The other side of haptics, in respect to the simulation of haptic interactions, is the simulation of objects to interact with a user. One of the many names for such simulated objects is Virtual Fixtures.{Bowyer, 2014 #15} Virtual fixtures are control algorithms that react to human controlled systems and relegate their performance in simulation to mechanical fixtures.{Bowyer, 2014 #15} Many different methods of representing the space and response of virtual fixtures exist, such as linear constraints or polygonal meshes and guidance or forbidden region virtual fixtures. One particular method for representing a simulated object as a virtual fixture, when interacting with a real space system, is proxy-based constraint enforcement.{Bowyer, 2014 #15} Here, a proxy represents the real system, and follows it through space, however it is inhibited by a mesh representing the simulated object. Based on the error in position between the real system inside the virtual fixture and the proxy, an elastic force is applied on the real system in the direction of rectifying the error, or penalty.

## 2.3. Rehabilitation

Of the many uses of virtual reality and haptic interaction devices, their possible applications to the field of medical rehabilitation provide new and alternate opportunities. Rehabilitation is any of the many techniques to increase and restore abilities and quality of life in people after injury or due to disability. While many alternate techniques exist, such as phobia exposure for such mental issues, the primary area of the field is physical therapy and physiotherapy.{Physiotherapy, 2015 #23} These are the technique of rehabilitation through movement and exercise. This includes activities such as core muscle strengthening for people with back pain and range of movement and muscle strength training for post stroke patients.

A large area in physical therapy is in the rehabilitation of stroke victims. A stroke is an incident where blood flow to a person's brain is cut off, most often due to a blood clot.{cdc.gov, 2015 #20} This results in many issues for the victim, such as loss of motor skills and mobility, speech, and brain functions, and learned non-use of their limbs. Strokes are a very prominent issue, with many people affected, and has therefore had a lot of attention in the rehabilitation field. According to the World Health Organization, 15 million people have strokes planet wide each year, of which 5 million die and 5 million have permanent destabilization. The United States center for disease control and prevention reports over 795,000 people have strokes each year with 130,000 people dying from Stokes, one nineteenth of all deaths in the United States. {cdc.gov, 2015 #20} Stroke costs approximately 36.5 billion USD in the United States, including medical care, medicines for treatment, and missed work. Therefore, with the large amounts of victims, delimiting effects, and high costs, stroke is a high concern in the field of rehabilitation. {cdc.gov, 2015 #20}

A novel approach to rehabilitation is through robotic assisting devices, introducing elements of virtual reality and haptic interaction to rehabilitation applications. An example of which is the ARMin Robot for stroke rehabilitation, shown in Figure 10. Based on previous observation research done by Neurorehabilitation scientists, the use of hand robot-assisted therapy in stroke patients could improve the hand mobility recovery.{Sale, 2012 #21} Through movement of disabled limbs after injury, such as stroke, motor ability can be restored, preventing learned non-use of the limb. Such robotic therapy devices work by providing guidance to patients' limb exercises, requiring voluntary and unrestricted participation by the patient to make productive movements, while preventing unproductive movements in the exercises. An often misconception in early devices was that exercising against constraints would be beneficial. As such, robotic therapy devices were created that provided this resistance in exercises, however this method was determined to be ineffectual, about half as effective as traditional rehabilitation methods.{Buerger, 2010 #17}

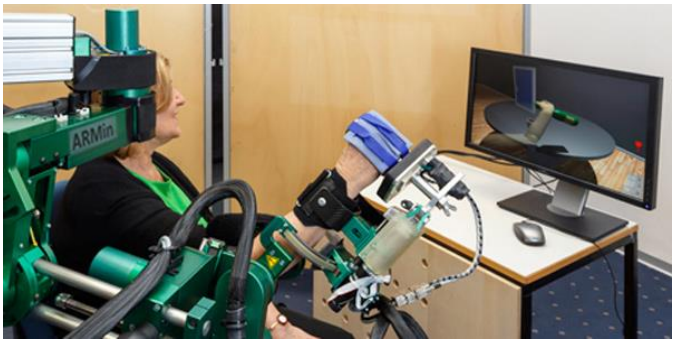



**FIGURE 10: STROKE REHABILITATION WITH ARMIN ROBOT**

**IMAGE: MEDGADGET.COM/2014/01/ROBOTIC-THERAPY-SHOWN-EFFECTIVE-FOR-STROKE-REHAB.HTML**

In order to create successful and efficient robotic therapy devices, or any haptic interface device designed to communicate high forces, the following six capabilities.{Buerger, 2010 #17} First, the device must have the capacity to deliver large haptic forces which means either sufficient to move human limbs actively against patients' muscular effort or able to support substantial fractions of body weight, depending on the devices specific purpose. Second, the ability to represent controlled virtual environments, including virtual objects such as solid boundaries and resistive forces. This is to allow for the device to provide guidance, as needed assistance, resistance, and to communicate haptic information. Third, the device must be back drivable or be able to simulate free unrestricted space. The patient must be able to perform productive desired movements without unintended resistance for effective rehabilitation. Additionally, this is to prevent inhibiting patients and therapists from moving or observing the outcome of rehabilitation. Fourth, a large workspace is required. This is an operating range of the device on the magnitude of one cubic meter, to allow for full range of human limbs. Fifth,

the device must have degrees of freedom on scale to small and large levels of human movement. The device must be able to operate with both levels of movement and match the movement capabilities of human limbs. Sixth, the device must guarantee safety and stability in haptic force exchanges. The devices will be applied to and used by non-expert engineering personnel, and therefore must be self-reliant in protecting these potentially impaired and unpredictable people.

Therefore from this, the field of rehabilitation, some of the critical medical issues it is used to treat, and the requirements to robotic therapy devices is known. This allows for a comprehension of the effective market for rehabilitation devices and how to create such devices that can be effective and helpful to society.{Buerger, 2010 #17} Some systematic review of robotic stroke therapy has unveiled the robot-aided therapy is a promising therapy for upper-limb rehabilitation after stroke.{Prange, 2006 #18}

## 2.4. Human Hand Dynamics

One way for simulated haptic interactions to be meaningful, is for them to be compatible with humans most sensitive and most often used tools for interaction, the hand. Therefore, the requirements, abilities, and constraints of the human hand is needed.

The first point is the amount of force the human hand is able to exert. If haptic interaction is limited to the fingers, then only forces exert-able by such muscle systems alone are need for the haptic device design. From research by the Department of Biomedical Engineering at John Hopkins University, the maximum strength of the human index finger is 40N, independent of flexion angle.{Sutter, 1989 #22} The PHANToM haptic interface design team, from the Massachusetts Institute of Technology, has gathered further data on forces exerted during haptic interactions. The PHANToM development team was a group who developed a device for communication focused haptic interactions with simulated environments. Their research found that 10N is the maximum force exerted in the majority of hand scale simulated haptic interactions, with only approximately 1N average during most interactions.{Massie, 1994 #13}

## 2.5. Current Solutions

In examining the commercial field of haptics interaction devices, it was determined that there were a nominal amount of attempts at different forms of haptic interaction devices, but none that suited the purposes of this project. The devices and software found often had issues such as high cost, limited availability, or over simplification of haptic interactions. In addition, commercial ready and available

systems built to work with natural hand interactions for the currently available virtual reality head mount displays were not found.

### 2.5.1. CyberGrasp and Virtual Hand SDK by CyberGlove Systems{CyberGlove, 2015 #10}

The most developed devices found through this research were the systems by the company CyberGlove Systems. For a mechanical system the CyberGrasp (CyberGrasp , CyberGlove Systems, 2157 O'Toole Avenue, Suite L San Jose, CA 95131) was a data glove, for tracking hand poses, and a system of motor actuators on each finger, to allow for force feedback. In addition there was the CyberForce (CyberForce, CyberGlove Systems San Jose, California), an armature manipulator that could apply forces to the user's wrist in space, for full hand haptic interactions. For software to implement the CyberGrasp, CyberGlove has the VirtualHand SDK (VirtualHand SDK , CyberGlove Systems,San Jose, California). This was a C++ development library with collision detection, hand interaction, and force feedback, along with tools for connecting to CyberGlove products. However some issues with applying this system were in cost and accessibility. The CyberGrasp alone was $39,000, had issues with friction decreasing mechanical bandwidth, and CyberGrasp not updating their business since the year 2013. {Grigore C. Burdea, 2003 #11} For this project's intended system, the cost would be much cheaper, on the scale of being easily purchasable by an individual, and also available for purchase through commercial means.

**FIGURE 11:CYBERGRASP WITH ATTACHED CYBERFORCE ARMATURE BY CYBERGLOVE SYSTEMS**

**IMAGE: HTTP://WWW.SENZTECH.CC/SHOWPROS.ASPX?PROID=83**

### 2.5.2. Falcon by Novint {Novint, 2015 #12}

**FIGURE 12: FALCON HAPTIC CONTROLLER BY NOVINT SYSTEMS INC. IMAGE: : HTTP://WWW.NOVINT.COM/INDEX.PHP/**

The most accessible system found in this research was the Falcon by Novint (Falcon, Novint Technologies, Inc, Albuquerque, New Mexico). This was a single point the user grabbed, built on parallel linkages connected to a base. These linkages allowed for the communication of force and weights to the user. The Falcon also came with an SDK for developing interactions, mainly for use with video games.

While this device only cost $249.95, it had a small range of movement for interactions, and reduced interactions down to one point of communication. This project's intended system would supply forces with more representation, to be similar to a user's hand for more natural interactions.

### 2.5.3. Geomagic Haptic Devices {Geomagic, #25}

Commercial consumer haptic feedback controllers were available from the company Geomagic. Their line of devices were based on simulating hand interactions with pens and similar tools, for applications such as 3 D modeling and scalpel based surgical applications. The line of devices were the former PHANTOM device series, with new names. The series ranged in features from simple 3DOF position sensing and feedback, to 6DOF position and orientation sensing and forcing models, and models for high magnitude forces or large workspaces. While some of this line of devices were similar in scale and function to the Falcon by Novint, purchasing any requires contacting the Geomagic sales division, to even be informed of the latest

**FIGURE 13: GEOMAGIC TOUCH X HAPTIC DEVICE ( FORMALLY SENSABLE PHANTOM DESKTOP) IMAGE: HTTP://WWW.GEOMAGIC.COM/EN/PRODUCTS/PHANTOM-DESKTOP/OVERVIEW/**

pricing and availability.  This project's intended system would be openly visible in cost and acquirement, whether it be in plans for the ordering of parts and building a system, or somehow distributing built systems.

# 3. Methodology

## 3.1. Project definition

This background research on the fields related to the idea of this project, a system for the creation and use of haptic integrations with virtual environments, provide support to the direction of the project. From this research the state of the potential uses, or market, for such a device, its related technological history, design guidelines and requirements have been determined. By following this information in project design, the resulting device can be effective, useful, and have high beneficial impact to society.

The planned limited timeline of this project naturally forms two distinct types of goals, the ultimate goal that this work aims towards and the direct goal, that which is to be completed in the timeline of this project instance. The ultimate goal is to create a system that allows for the full development of applications based on users having haptic interactions with virtual environments. The direct goal of this instance of the project is to create an initial version of such a system. This system is to be complete with sub-systems giving basic capacity in all the different features that would be needed for a perfect system, and to set in motion further iterative system development, with guidelines and tools need for such advancements.

There are four different main subsystems to the haptic interaction system.

1. A simulation system to host virtual environments and manage the interactions between the virtual environment and the user's virtual representation. The basic version of this system is to host the interaction systems, applications, and tools for future development.
2. A force exerting system. The initial version is to be focused on the user's hand, allowing for the most quantity of meaningful simulated haptic interactions. The system is to be able to generate simulated forces that can either induce movement or prevent movement, based on solid boundaries.
3. A system to track the user’s positions and orientations. This allows such data to be captured and reproduced in the simulation environment to then generate haptic feedback based on the users input and the environment's structure. The basic version of the system is to again track just the user's hand, including its position and orientation over a limited space, along with the pose of the user's fingers.

4. A display of the virtual environment to the user. This allows the user to see the environment they are interacting with, in a perspective simulating their presence in the environment. The basic version of this system is to display the environment to the user.

Two other key components are to be included in this project, a demonstrative application and guide for future development.

5. The demo application is to show to other potential application and system developers the features of the system and some of its potential applications.
6. The guide is to be a comprehensive manual on how to use the system, how the different sub-systems integrate and communicate, and how to further improve and extend the abilities of the system towards the ultimate goal.

The components and information exchanges of this system is shown below through Figure 14.

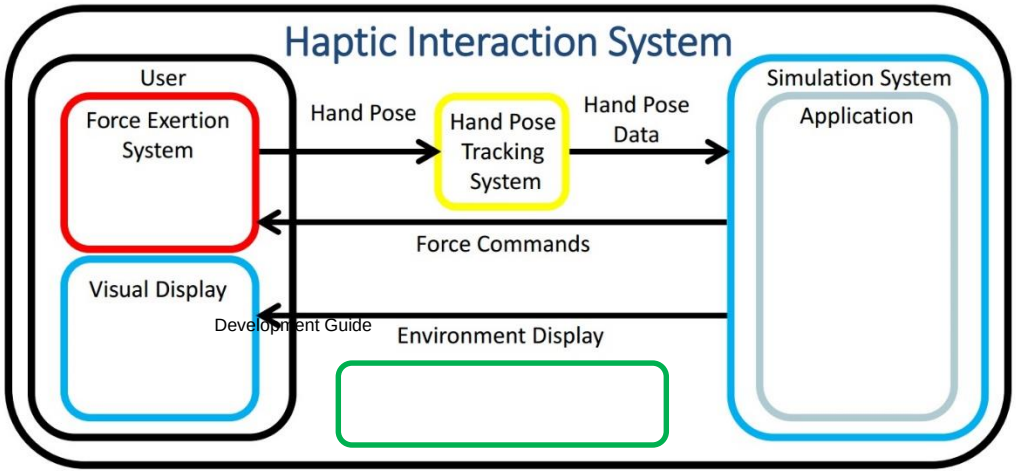



FIGURE 14: OVERALL SYSTEM DIAGRAM SUB-SYSTEMS AND INFORMATION EXCHANGE

## 3.2. System Requirements

This project premise defines requirements needed for this first iteration of the haptic interaction system. These requirements follow the same division as the systems features above.

Simulation System:

- Present the following interactions to the user
    - Immovable barriers
    - Assistive and resistive forces
    - Unrestricted space
- Hold custom virtual environment, objects, and user representations
- Simulate physics interactions and agents
- Connects with visualization system and hand pose tracking system
- Communicate and instruct exoskeleton system
- Detect object collisions and generate resultant forces
- Update sensory information to user fast enough that user does not experience mental disconnect from the simulated environment

Force Exertion System:

- Exert fingertip perpendicular forces over five fingers of the human hand, with forces on the scale of one handed interactions' forces
- Individually apply halting, opposing, or no force on each finger, in either direction
- Not exceed the limit of the user's comfortable hand flexion angle
- Not interfere with hand pose tracking

Hand Pose Tracking System:

- Track the pose of the user's hand in a region to scale with stationary body one handed interactions, with suitable resolution and low noise
- Not be susceptible to devices expected in a rehabilitation or game session environment

Virtual Environment Viewing System:

- Display virtual environment to user
- Not break user presence with visual lag

Demonstrative Application:

- Display the capabilities of the Haptic Interaction System
- Show a potential beneficial use for the system, related to target market of rehabilitation or entertainment

Development Guide:

- Provide information explaining the structure, functions, and design decisions for the system
- Provide all code, programs, and resources
- Provide instructions on:
    - how to build and set up the system
    - how to develop applications for system
    - how to update the guide for future system iterations

## 3.3. Sub-Systems Design Decisions

The initial design of the system was determined by researching available solutions and methods for the different sub-systems needed. Each sub-system had multiple methods proposed, and were picked based on analysis and evaluation.

### 3.3.1. Hand Pose Tracking System

Different options were considered for the purpose of tracking the position and orientation of the user's hand and fingers. The purpose of this information was to apply the spatial movements of the user to the virtual environment hosted by the simulation system, so the reactionary forces can be generated to complete the interactions.

Two methods were researched for the hand pose tracking system. First was a sensor and emitter system consisting of small sensors placed on the hand, which would be tracked in space by a receiver using electromagnetic signals. These would provide a large range to the user's motion, on the space of a human arm's full workspace. However, these sensors were not accurate enough for the precise finger movements needed and would be distorted by any metallic or electronic systems nearby. Therefore this device would not be usable in the office computer setting the Haptic Interaction System is designed for and would place unnecessary constraints on the other sub-system designs. The second method considered was a commercially available infrared hand tracking sensor, called a Leap Motion. This sensor works with infrared emitters and cameras to detect and measure the positions of hands and

fingers in its view. It also came with a software development kit and application program interface, along with integration with some computer application development programs. This sensor could capture hand and finger positions in detail and precompute the received data into a hand pose form. However, it only had a range of about 2 feet above the sensor and a field view of 150 degrees. The sensor could also be incorrect at times when faced with fast movements or parts of the user's hands being occluded by itself. Between both of these options, the Leap Motion sensor was chosen. This was due to its relatively larger amount of resources about its use and setup, its commercial availability, ability to be placed in an office environment without loss of accuracy, and provided tools for building applications with it.

### 3.3.2. Simulation System

Many options were considered for the simulation system, and was heavily debated due to its centrality to the interaction system and high difficulty to replace in future iterations. The criteria for each potential system to be graded on were as follows:

1. Connect ability with Leap Motion hand tracking device and supporting resources
2. Connect ability with Arduino microprocessors and supporting resources
3. Capable frame rate
4. Connect ability with head mounted displays and supporting resources.
5. Collision detection ability
6. User market availability
7. General supporting resources
8. compatible programming languages
9. compatibility for future project work

For the simulation system, commercial computer application systems were considered. Building a new engine was not considered feasible, due to the time constraints of the project, the limited members, and the expertise of the group being in robotics and mechanical engineering, not purely computer science. Additionally, such a system would not have the support base and resources dedicated to upkeep that commercial game development engines, physics simulators, and other simulations programs have. The considered systems were the game development engine C4, the software framework Robotic Operating System (ROS) with 3D simulation program Gazebo, and the game development system Unity.

The grading for these systems was determine by first ranking the importance of each of the criteria, from 1 to 5, and then determining how well each method performed that criteria, from 1 to 3. Then, the score for each method was determined by multiplying each criteria score by its ranking importance and summing these values. The final break down is in Table 1 below.

| | Criteria | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Total Score |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Methods** | Importance Rank | 3 | 2 | 4 | 3 | 5 | 5 | 3 | 3 | 5 | |
| | Methods | | | | | | | | | | |
| | Unity | 2 | 2 | 2 | 3 | 3 | 3 | 2 | 1 | 2 | **76** |
| | C4 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 1 | 2 | **55** |
| | ROS | 1 | 2 | 2 | 1 | 3 | 3 | 2 | 2 | 1 | **65** |

Table 1: Simulation system methods scoring matrix to aim towards fully perfect device

Therefore, Unity 3D was chosen, for it high amount of resources, free cost for commercial use, and connect ability to other systems.

### 3.3.3. Force Exertion System: Mechanical Structure

The mechanical system of the force exertion system is comprised of two components, the force applying mechanism and the motors that drive it.

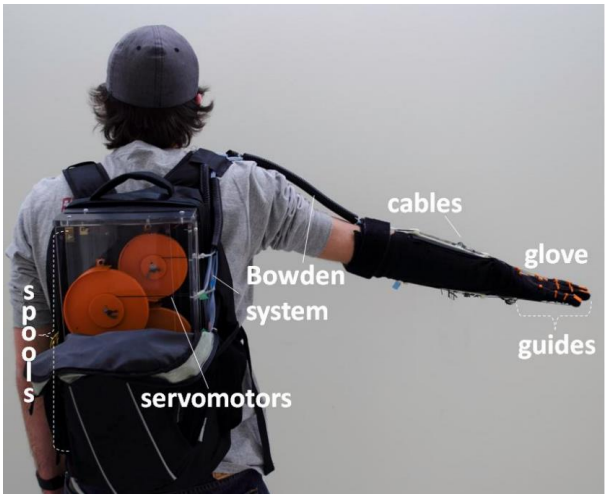



**FIGURE 15: BOWDEN SYSTEM DEVELOPED BY AIM LAB IN WORCESTER POLYTECHNIC INSTITUTE**

**IMAGE: (DELPH, 2013 #24)**

The force applying mechanism, due to being based off of previous systems and work by WPI's AIM lab, an early device shown in Figure 15, had a relatively less extensive design analysis, however different methods were considered along with what features were to be implemented.{Delph, 2013 #24} For the mechanism, research for commercial systems and possible methods to implement with custom hardware were considered. For the commercial systems, few options could be found. Most options, such as the Novint Falcon, were of a style that consisted of only a point or a pen interface, reducing the methods of haptic interactions down to a single point, as opposed to a hand as desired by the project. While these types of devices were the most commonly available on the market, and in a medium cost range from $200 to $300 for some systems, the reduction of interaction does not meet the needs of this project to allow rehabilitation or immersive media experiences.

Other commercial systems existed that allowed interactions in the style of using a person using their hand in daily activities, such as the CyberGrasp by Cyber Glove Systems. These fell into two categories. The first being systems developed by haptic controller companies that were highly priced due to their age and being targeted only for research laboratories, and often backed by defunct businesses. The second being systems that were developed in research labs for haptic interactions, but never made available for commercial production and use. While many in these categories could provide well some functionality that this project needed, due to the systems being developed in the past with less effective technology the results were not perfect, often leading to all virtual objects feeling soft and deformable. Additionally, the cost and lack of availability of these systems caused them to be unsuited towards this project.

Therefore a custom system would be needed for the force applying mechanism. To keep this system from having the same issues as the above options, it was to be designed with simplicity and low cost of manufacturing. Some different designs were considered. The most typical and direct method would be actuators attached directly to each joint of the user's hand. While this would allow for the most simple and compact system this method would have issues. The motors on the hand could impede the user's motions and be heavy and tiring to wear over time and this method was the most susceptible to the soft and deformable virtual objects issue, in the analyzed commercial systems. Another method was to have actuated pistons placed between the user's finger tips and palm. While this could have less deformability issues, the usable range of the user's hand would be greatly decreased from normal and the weight from the components could still be tiring. The last method considered was one based on the exomuscular glove and sleeve stroke rehabilitation systems developed by WPI's AIM lab. These systems were based on a series of motor actuated pulleys which would pull the user's finger tips with a series of Bowden cables, inelastic cords inside plastic tubes to allow cord tension regardless of hand location. This would remove heavy weights from the users hand and not create artificial range limits. The cables being of an inelastic material would also prevent the deformability issue, or at least reduce it down to the slack in the motors which could be optimized since they were not constrained by being on the user's hand. Additionally, the designs were readily available and many parts could be 3D printed or were common enough to be easily available. Therefore, with the ease of building and material availability, the un-inhibiting design, low cost, and hand based interactions, the AIM lab's Bowden cable based design was chosen for the force applying mechanism.

Only a few choices had to be made to determine the design of the Bowden cable force applying mechanism, since it was a complete system, however some design choices were needed as to which features were to be included. The basic structure of the AIM Lab's Bowden system is as follows. A glove with 3D printed cable guides sewn on, cables connecting the top and bottom of each fingertip to a pulley, plastic tubes surrounding the cables and connecting to the cable guides to keep tension, and a series of servos connected to the pulleys and a motor controller. All of these design aspects were desired and kept as specified by the previous systems, except for using DC motors instead of servos, in order to perform current control. The AIM Lab systems also had additional features, such as bioelectric sensors to read user's nerve signals and pressure sensors on the finger tips. These were not kept in the design because assistive motion based on nerve signals was outside of the scope of the basic version of the Haptic Interaction System and the pressure sensors to record force were not previously successful enough to be useful. While force sensors could be useful to the system, it was decided that the system would not use them in this iteration in trade of completing all the sub-systems for a full base Haptic Interaction System.

The glove used by the force applying mechanism also had to be chosen by additional requirements. The glove still had to support the forces of the system but now also had to be viewable by the Leap Motion device for the Hand Pose Tracking System. For this the glove needed to reflect infrared light. To avoid having to build custom gloves, the Leap Motion was brought to a commercial sporting goods store and tested with available gloves until one was found that responded as well as a glove-less hand. A pair of white Foot Joy faux leather golfing gloves was chosen from this exercise.

The motors chosen to replace the servos of the force applying mechanism had many crucial requirements to fulfill. They needed to be able to resist the up to forces the average human hand could apply, not inhibit free motion when no force is being applied, be capable of running in stall, and not exceed the current limitations of the possible motor controllers available.[36] This was a difficult decision due to how these requirements countered each other. To have free motion, a low gear ratio is needed, but then this would need a higher current per force than a high gear ratio motor. Multiple motors were examined, but few fit these constraints. To better determine the gear ratio versus free motion problem, a motor type used in the team's previous work was tested. This motor was a Pololu 34:1 25D DC brushed gearmotor, which was used with a pulley from the force applying mechanism, in order to determine its

36

resistance to free motion that would be applied to the hand. With the lever arm provided by the pulley, the motor was able to be moved comfortably to provide low restriction to the user's hand in free movement mode, even with the relatively high gear ratio. Therefore, the Pololu 34:1 brushed gearmotors was chosen due to it fitting the needed requirements and being readily available through online motor resources.

### 3.3.4. Control System

To control forces, a basic control loop process is as follows. A desired force is received from the simulation system, which is then applied based on a model of the system. Finally, the force output is measured and the output is corrected based on any sensed errors. The control system for the Force Exertion System itself includes two main components, the processing device and additional sensing hardware.

For the processor, commercially available microprocessors and computers were considered. Again, a completely custom microprocessor was not considered feasible, due to its lack of easy availability and the support base needed by future application developers. Having the Force Exertion System being directly managed by the computer running the simulation system was also deemed non-ideal. This would require a complex custom circuit to interface with the exoskeleton's motors and additional hardware, along with potentially limiting the activity of the system's running applications. The Arduino series of microprocessors was chosen, over other similar devices such as the BeagleBone and Raspberry Pi lines. Arduino was chosen due to its large support base, ease of commercial procurement, low level hardware, and familiarity to this and possible future development teams from the WPI Robotics curriculum. With these aspects, the Arduino would provide the best experience, being easy to get and use for haptic interaction application developers and not have an unneeded operating system that only introduces more complexity. Next was to decide what model of Arduino microcontroller to use, but this was dependent on the additional hardware used.

While the microprocessor would handle instructions for the Force Exertion System, it alone lacked the ability to command motor states and read encoder counts without over tasking the system nor could it sense motor torques. Methods of motor commanding considered included custom circuits, and Arduino and Adafruit brand motor controller add-ons for Arduino microprocessors. Again custom circuits would not be feasible due to procurement and use by application developers. While the Arduino add-on, or shield, would keep the hardware to one resource and allow for current sensing, meeting the motor torque

sensing requirement as well, it was not fitting to the system. The Arduino shield had a limit of 2 amps draw per motor, lower than required by motors that can work on scale of human hand capable forces. Also, it could only control two motors per shield, requiring three shields total reaching $68.28. The Adafruit motor controller shield however could run four motors and handle currents at the needed level. To cover its lack of current sensing, five commonly available current sensing chips were added to the system design with the two shields, totaling $44. Therefore the Adafruit shield with current sensing chips was deemed the best option.

To meet the requirement of managing the Force Exertion System system's motor encoders, both direct measurement and subsidiary encoder counter chips were considered. The encoder counting chips were chosen because direct measurement would over task the microprocessor with encoder update interrupts, leaving no ability for other tasks. Therefore commonly available encoder counter chips were added to the system design.

Therefore, with the additional hardware determined, an Arduino model was needed to be chosen. Based on the needed capabilities of the processor only being digital I/O, analog measurement, and serial communication, the basic line or Arduino microprocessors was studied for options. These options included the Uno and Mega models. While either model met the processing and I/O ability, the analog input and serial communication structure of the Uno eliminated it as an option. The Uno had six analog measurement ports, but two of them were also the serial communication data and clock lines used to command the PWM motor control chips on the Adafruit shields. Therefore there were not enough ports remaining for the five current sensing chips needed for the motor to each of the user's fingers. Even though two Uno models could be used to control the force communication system, this would introduce much complexity in communication with the simulation system. This would cause larger processing times to send commands and gather user state data and keep track of frame IDs over multiple sources. Therefore the Mega, with its 16 analog measurement ports and serial communication dual functioned with the abundant I/O ports, was chosen.

### 3.3.5. Communication Design Decision

A needed element of the Haptic Interaction System is the communication between the Simulation System and Force Exertion System. This would handle generating commands for the Force Exertion system and getting the hand state information to the Simulation System. From previous decisions, the communication would be between Unity (Simulation System) and Arduino Mega (Force Exertion System).

To determine how to achieve this communication, the functional requirements for the system were determined as follows:

- Command Forces to Force Exertion System
- Retrieve force and pose data from Force Exertion System

Multiple methods were available for these functional requirements, but the main aspects consisted of one choice, the use of a middleware program to handle the communication messages. This middleware would consist of a C++ or c# standalone executable file that would catch messages sent to and from the systems and forward them along as needed.

With this distinctions the possible methods for the functional requirements were as follows:

- Command Forces to Force Exertion System
  1. Without Middleware
     a. Unity and Arduino communicate messages directly over a communication (COM) port. Unity sends a force command which Arduino processes before each control loop.
  2. With Middleware
     a. Unity and Arduino communicate to middleware, through server client socket and COM port respectively. When middleware receives a force command from Unity, it forwards it to Arduino at the beginning of its check for messages function. Arduino then reacts at the beginning of its next control loop.
- Retrieve force and pose data from Force Exertion System
  1. Without Middleware
     a. Unity sends a request for the hand state to Unity directly over a COM port. After the next control loop Arduino checks for this request and complies if one is there.

b. Arduino sends hand state to Unity over COM port at a regular interval. When this information is needed Unity reads the newest state update and clears the serial buffer.

2. With Middleware

a. Arduino sends hand state to middleware at a faster rate than method 1b. Middleware updates its holding value for the hand state each time a new one is available. When Unity requests hand state middleware sends it the latest values without calling Arduino

For the Command Forces to Force Exertion System function requirement, it was only a question of using the middleware. The middleware gave no benefit to this functional requirement, and actually would slightly delay the command due to having to wait for both the middleware and Arduino to loop around to the check for the command.

For the Retrieve force and pose data from Force Exertion System functional requirement the analysis was more complex. Without middleware method 1a would have a delay in the simulation system, due to Unity having to wait for the message to be received by Arduino, noticed, processed, and replied to, before the simulation system could advance. However it would be the most accurate state information due to its recent publication. Method 1b would not have such a delay, due to the information being already available, but it would require some more processing by Unity and Arduino and be slightly less accurate due to the time passed since the message publication. Method2a brings the most benefits from the middleware. This would allow a faster state publication by Arduino, due to the middleware being able to address the serial buffer more often than Unity. Therefore the messages would be less off due to time passing and Unity would not have to wait for the data to be processed before it can proceed.

Introducing the middleware also had further aspects that were considered in our analysis. Primarily, exporting the executable to other systems, that lack the developmental version of the middleware code, is complex. This often encounters difficulty in installing needed libraries and directing the executable towards them, which is most often handles by an installer and can require licensing agreements for some of the needed tools. These executables and installers are also often limited in the computing systems they can run on. This would introduce more difficult tasks for future users of the Haptic Interaction System, and could limit the possible computer systems that it could run on.

Between the extra complexities of distributing the middleware, it being only slightly beneficial to the Retrieve force and pose data from Force Exertion System functional requirement and slightly hindering

to the Command Forces to Force Exertion System function requirement, it was decided not to use the middleware approach.

Instead, the determined method was methods 1a for the first functional requirement and 1a and 1b for the second functional requirement. This way, sending force commands to Arduino was as unlimited as possible and getting hand state data has the choice of waiting for the most accurate values or getting old values without waiting, depending on the current needs of the system. This trade off in hand state efficiency was considered acceptable, for it is considered slightly less important than sending force commands and having the Haptic Interaction System be easy to use and mass applicable.

### 3.3.6. Virtual Environment Viewing System

There were two directions possible for the Virtual Environment Viewing System, either a head mounted display or a monitor screen. Each of these methods would be fitting to different application. The head mounted display towards classical virtual reality type applications, based on giving the user presence in a simulated environment, and the monitor screen towards where the haptic interaction is the only focus and presence is not needed or introduces difficulty in the process. Examples of these could be a virtual museum tour and a strengthening rehabilitation tool respectively. With both of these methods being suitable, design considerations were made for each.

For the head mounted display, basic head mounted displays were first considered. Many options existed that just consist of two displays, one per eye, showing slightly different images to convey a 3D visual input. These ranged between about $100 and $999 and gave different types of features, such as level of resolution, orientation sensors, and overlaying the user's vision or not. However, most of these were standalone generic device for viewing 3D media, lacking integration with applications or development tools. The other option were head mounted displays specifically for interactive mediums such as gaming. Many of these types of devices are currently in development, with the forerunner being the Oculus Rift by Oculus VR. The Oculus Rift, while only having development models available, includes the same features as a basic head mounted display, along with detailed position and orientation tracking and application development tools for programs such as Unity, for the cost of $350. With these options, for applications with head mounted displays, the Oculus Rift was chosen, due to its commercial availability, relatively low cost for available features, and its application development tools.

For applications without head mounted displays, there were few available options. While more exotic visualization methods exist, such as projecting environments onto the walls of a room, these still have limited applications and are not easy to obtain by the average developer. Therefore, a computer monitor was considered. A monitor is very easy to obtain for a developer and already has support for the visualization of applications. A monitor would also not interfere with some of the applications where a head mounted display would be unfavorable. For example, in an application for in home hand strengthening for stroke victim rehabilitation, a head mounted display would be difficult for the patient to put on and use.

Therefore, with the wide possibility of applications the Oculus Rift and computer monitor functionality would be the methods for the virtual environment viewing system. However, due to the limitations of

this iteration of the Haptic Interaction System, only the monitor would be implemented. This is due to the complexities of managing frames of reference between the Leap Motion and the Oculus Rift and the determined demo application not using the Oculus Rift.

### 3.3.7. Simulation System and Force Exertion System Communication

Given the determined method for communication, message types had to be made in order for Unity and Arduino to know how to process received messages. The following were determined:

- **Update Finger Force**: commands Force exertions system to set a finger for a given type of control at a given target value
- **Update All Finger Forces**: commands force exertion to all fingers with a given type of control and target value
- **Update All Fingers Separately**: commands force exertion to each finger with a different specified force type and target value
- **Send Hand State**: commands force exertion system to report the latest hand force types, finger forces applied, and finger curl amount

| Command Type | Command Type Code | Message Size (Bytes) | Message Format |
|---|---|---|---|
| **Update Finger Force** | 00000001 | 5 | Command Type Code; Finger Id; Force Control Type; Force Control Value, Force Direction |
| **Update All Finger Forces** | 00000010 | 5 | Command Type Code, All Fingers Id, Force Control Type, Force Control Value, Force Direction |
| **Update All Finger Forces Separate** | 00000100 | 17 | Command Type Code, All Fingers Id, Force Control Types (5 Bytes), Target Values (5 Bytes), Force Directions (5 Bytes) |
| **Send Hand State** | 00000011 | 16 | Commands Type Code, Force Control Types (5 Bytes), Target Values (5 Bytes), Force Direction (5 Bytes) |

Table 2: Command Type codes, Size and Message Format

The multiple ID and type commands have the following definitions. Finger IDs were 1 through 5 representing the fingers from thumb and pinky, with 0 representing all fingers. For Force Types, unrestricted movement force type is represented by 0, where any force control value is ignored, and the forces type is represented by 1, for applying forces to the user's fingers. Direction referees to the opening and closing of the user's fingers, with 1 dictating an opening force and 2 dictaing a closing force.

## 3.4. Demo Application: Rehabilitating Hand Learned Non-Use through Visual Feedback Distortion

The demonstrative application for the haptic interaction system is to be a procedure tool for the rehabilitation of hand learned non-use. Through the use of visual feedback distortion about their task performance, patients are to incrementally be tasked to perform past their self-believed limits, without their awareness. By this distortion patients will work past their learned limits, and with the proper rehabilitation regiment regain ability in their impaired hand.

In this application, the process will be modeled as a series of trials at grasping a virtual rehab stress ball. With each grasp the patient will be directed to perform a ball grasp of a specific force and report once they feel they have performed it. Each trial will have a known amount of distortion in the required force, based on a fraction of the patient's Just Noticeable Difference (JND) force. As the series progress this distortion value will increase the force required for specific level grasp, by increments below the patient's noticeable force threshold, showing less performance in the visual representation of the patient actions. Therefore the patient will perform past their learned limits and enforce this ability.

### 3.4.1. Procedure Steps

To perform this procedure, the following steps will be performed.

- System Preparation
    - Rehabilitation instructor enters patient and trials data
        - Patient's JND force for each finger
        - Number of trials, target force level, and distortion factor for each trial
    - Instructor assists patient in equipping exoskeleton and placing hand behind display screen
    - Patient is asked to exert the strongest and weakest force they feel comfortable exerting
    - Forces define the five target force levels for the trials
- Rehabilitation Trials
    - The trials commence according to the entered data and patient target force levels
        - Each trial shows the patient their performance, altered according to the entered distortion factor, and concludes by the patient indicating they feel they have reached the instructed grasp force

        - During the trial, data on the target force level, distortion factor, required force, and patient grasp force exerted is displayed to the instructor
- Post Trials
    - Patient is asked to again exert the strongest force they feel they are capable
    - Data on self-reported patient ability before and after the trial, along with trial data is generated for instructor analysis

### 3.4.2. Demo Requirements

In order to achieve this application, the systems of the haptic interaction system will be required to perform the following:

1. Simulation System
    - Take in and store system preparation data
    - Model virtual spring constraint, in sphere zone shape, locked to the frame of patients hand
    - Generate force commands for each finger based on sphere, finger pose, and distortion factor
    - Retrieve hand data from Force Exertion System and Hand Pose System, to report data for clinical analysis
    - Generate halting commands, to measure patient's minimum and maximum force exertion
    - Perform the sequence of data measuring and grasp trials
2. Force Exertion System
    - Exert forces and halts
    - Record patient force during force measuring halts
    - Compile and send force and pose data to Simulation System
    - Not pass limits of patient's hand mobility
3. Hand Pose System
    - Capture patient grasp pose and communicate it to Simulation System
4. Visualization System
    - Display patient hand grasp and virtual stress ball, according to distortion factor
    - Hide patient's physical hand from view

# 4. Final Design

## 4.1. System Implementation

A first iteration of the Haptic Interaction System was developed, based on the outlined design decision. Each sub-system was built and examined for any unforeseen problems or areas of needed improvement. The designs were refined for improved performance, based on this analysis,. Overall, the Force Exertion System executed forces on the user based on Simulation System commands and consisted of Exoskeleton Mechanics, Electrical System, and Control System. The Hand Pose Tracking System combines Force Exertion System with Simulation System by streaming user's hand pose, finger motion, and hand grasp to Simulation System. And the Simulation System provides virtual environment for user, in the meantime generates Haptic Interaction commands send back to Force exertion system.

## 4.2. Force Exertion System

The final design of Force Exertion System was a major sub-system in the Haptic Interaction System and would exert haptic feedback on the user's hand while measuring the force applied when met with user resistance. A diagram of Force Exertion System is below in Figure 16

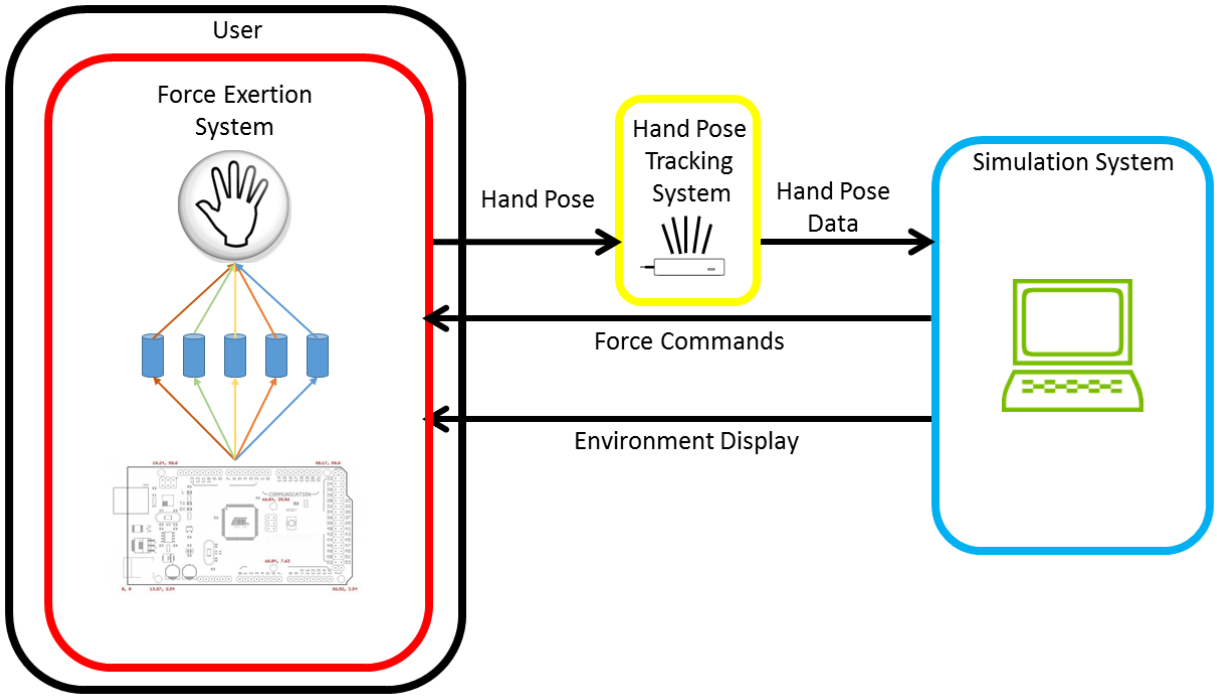



FIGURE 16: FORCE EXERTION SYSTEM FINAL DESIGN OVERVIEW DIAGRAM

To achieve these goals, the main exoskeleton mechanism had been implemented into the Forces Exertion System, which consisted of a Bowden Cable Exoskeleton with five Pololu 34:1 brushed gearmotors for each finger, while it was being controlled with an Arduino Mega, two Adafruit Motor control Shields, and current sensing modules. The completed Force Exertion System hardware can be seen in Figure 17 below.

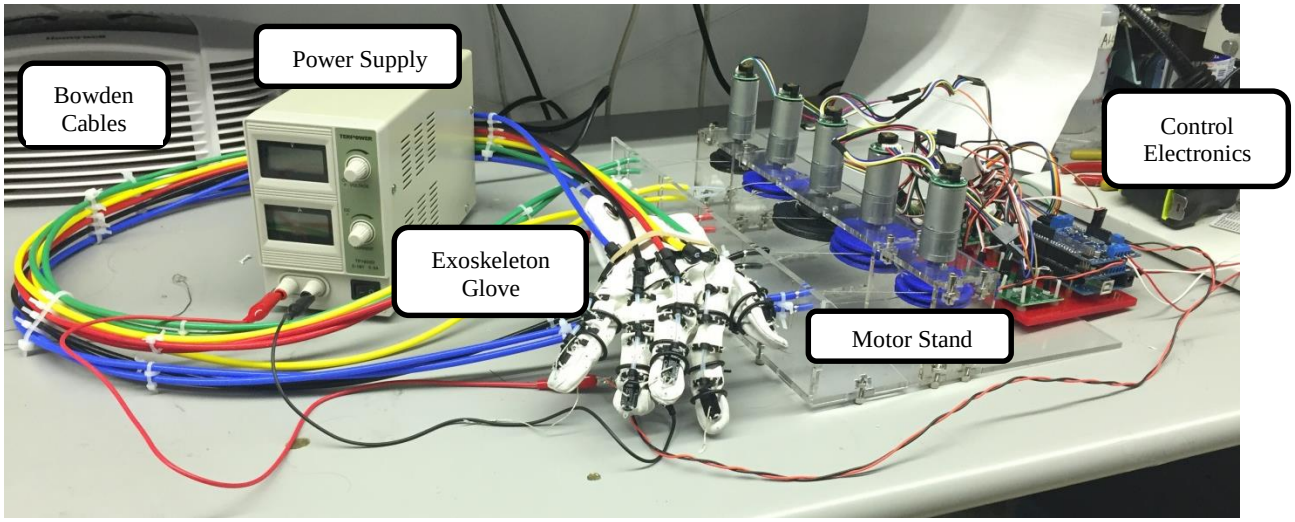



**FIGURE 17: FORCE EXERTION SYSTEM HARDWARE**

### 4.2.1. Bowden Cable Exoskeleton

The Bowden Cable Exoskeleton design was a modified design based on previous works developed by the Automation and Interventional Medicine (AIM) Lab at the Worcester Polytechnic Institute. Due to the previous exoskeleton tasks, those models were developed with potentiometers in order to perform position control on the exoskeleton. Thus, based on this design, certain parts had been redesigned in the Force Exertion System to match the needs for force control. Specifically, DC brushed Motors were used in the system, in place of potentiometers. This design tried to keep as much weight off of the user's hand, instead of implementing actuation mechanism on the user's hand. Hence, a separate motor stand had been used to remotely exert force in the direction perpendicular to the user's fingertip pads. The motor stand: five pairs of Bowden cables, five DC brushed motors and pulleys. This design allowed for users to have a large work space with an efficient, compact and modular solution for haptic interaction. The electronic components diagram is shown in Figure 18.

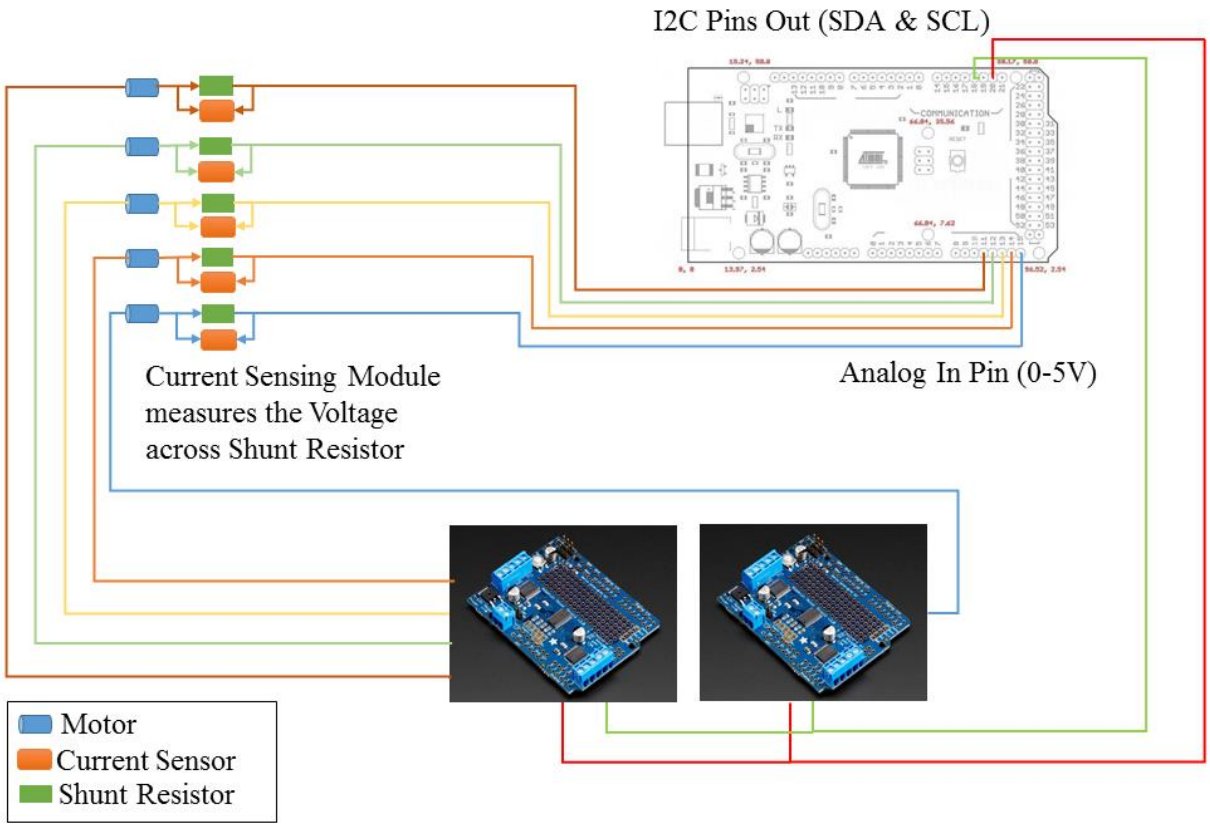



FIGURE 18: FORCE EXERTION SYSTEM ELECTRONIC COMPONENTS CONNECTION

### 4.2.2. DC Brushed Motors

DC Brushed Motors used in the Force Exertion System had been carefully selected based on which had adequate torque and could be easily back driven at when no current was applied. The final motor stand installation can be seen below in Figure 19. To achieve the 10N forces at the finger, to match normal user hand interaction forces as determined to by the PHANToM haptic interface development team, a torque of 0.143 NM was needed, based on the worst case defined by the largest diameter pulley of the motor stand.{Massie, 1994 #13} The back drive ability of the motors had to pass a manual test of being hooked up to a finger on the exoskeleton, and be back driven without tiring the user with extended use.

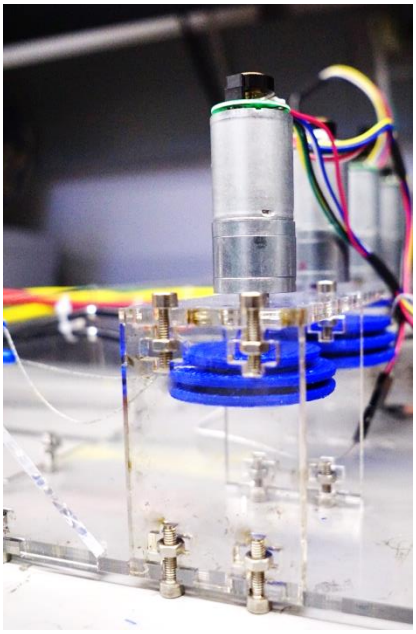


**FIGURE 19: POLOLU 34:1 GEARMOTOR WITH PULLEY ATTACHED ON MOTOR STAND**

To match these requirements, various actuators were evaluated during the preliminary design. Pololu 34:1 25D Metal Gearmotors had been chosen to mounted on the Motor Stand, as shown assembled in Figure 16, to drive the Bowden cable and pulley mechanism, while measuring force from user's fingers. At first equivalent Pololu motors with 75:1 gear ratios were analyzed, for their ability to match the 40N maximum human finger force, but these failed the back drive test due to being nearly immovable by the user. The 34:1 ratio motors passed the back drive test with little effort from the user and were still rated for a maximum of 20N finger force at their stall current, or a 0.282NM torque. The 10N finger forces required heat sink additions to the motor controller shield H bridge circuits, but could run at a maximum of 4N without these. The DC Brushed gearmotors selected could be used for both precise position and current control via attached encoders and separate current sensor module. However, the encoders were not implemented, due to not allowing for any increase in functionality based on the goals of this iteration of the system. The advantage of the selected DC motors were the compact size and cylinder shape which could be easily mounted on existed motor stand design. And the screw holes on the motor gearbox allowed two M3 screws to fasten to the motor unit. Another advantage that came along with DC motors was the range of motion. Instead of a restricted range actuator like hobby servos, DC motors provided unlimited bi-directional range which meant no need for calibration.

Finally Bowden cables were used to transfer the forces to the user, and could be adjusted to fit various hand sizes and different finger motion ranges. The Bowden cables worked as a rope and pulley system, where the motors actuated bi-grooved pulleys to apply an opening or closing hand force on the user's fingers. The addition of the plastic tube casings kept a constant offset between the pulley and exoskeleton, such that tension could be applied and maintained throughout the system's workspace. Nevertheless, DC brushed motors has overheat problem when exposed to long and repetitive running or stall times which could be solved by implement cooling devices such as fan or heat sink.

### 4.2.3. Microcontroller

The Microcontroller was also a cirtial components in the Force Exertion System, due to its many roles, including sensor reading, force control management, motor commanding, and serial communication


**FIGURE 20: ARDUINO MEGA 2560 MICROCONTROLLER**

**IMAGE: ROBOTSHOP.COM**

with the Simulation System. The Arduino Mega 2560 microcontroller chosen was one of most common efficient microcontrollers with open-source software and featured an ATmega2560 8-bit microcontroller chip. With all of the current sensing modules and motor control shields attached, the microcontroller could manage all their functions and still had digital IO, analog input, and serial communication spaces for additional hardware for use with future projects.

The management of all of the roles for the Arduino Mega gave a comprehensive list of tasks to perform. The system was implemented to run force control at a rate of 60Hz minimum, to match the standard interactive media processing rate of 60 frames of calculation per second. At this rate the Arduino Mega performed the following tasks: read any newly received serial force commands, sensed motor currents,

calculated motor commands based on an error adjusting control loop, and delegate these commands to the motor control shields.

### 4.2.4. Motor Controller Shield

The Motor Controller selected for the Force Exertion System were two Adafruit Motor Shields V2.3 as Figure 21 shows below which had met all requirements. They needed to provide at least 6V voltage on each motor, at least 2.2A to drive the motors at the highest torque, for 10N of finger force, and could use a separate power supply in order to protect the microcontroller. After the comparison with other motor controllers, the Adafruit Motor Shield had not only fulfilled all requirements, but also had dedicated PWM chips on board to provides more accurate control, and high expandability. Up to thirty two shields could be stacked for the control of one hundred and twenty eight DC motors, and two 5V hobbyservos. Another advantage of this motor drive over others was their easy-access function library and examples for quick learning and debugging.

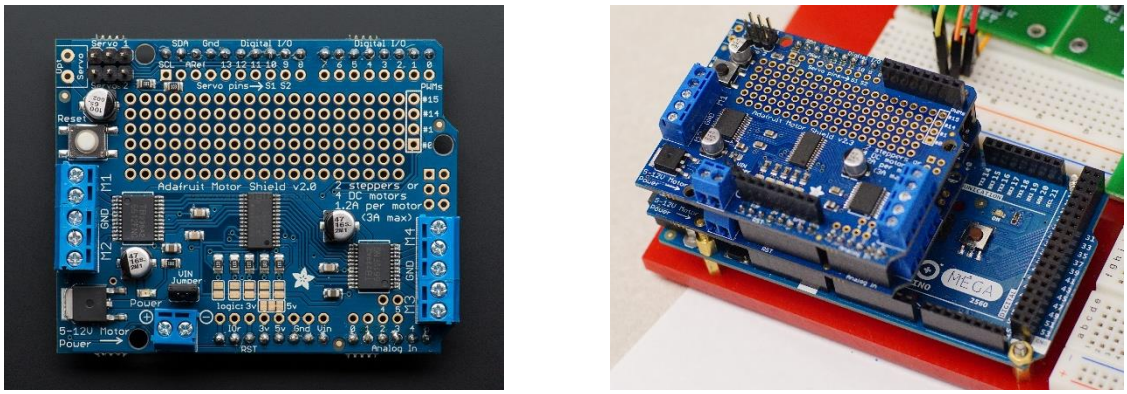



**FIGURE 21: ADAFRUIT MOTOR SHIELD V2 (LEFT) AND STACK WITH ARDUINO MEGA 2560 (RIGHT)**

**IMAGE: ARDUINO-ITALY.COM**

### 4.2.5. Current Sensing Module

The Current Sensing Module in the system was used to measure current being applied through each motor. By measuring current (I) and voltage (V) across each motor, the perpendicular force applied on each finger could be calculated. The module and its main part,the TI IMP 8601 Bidirectional Precision Current Sensing Amplifier, are detailed in Figure 22.

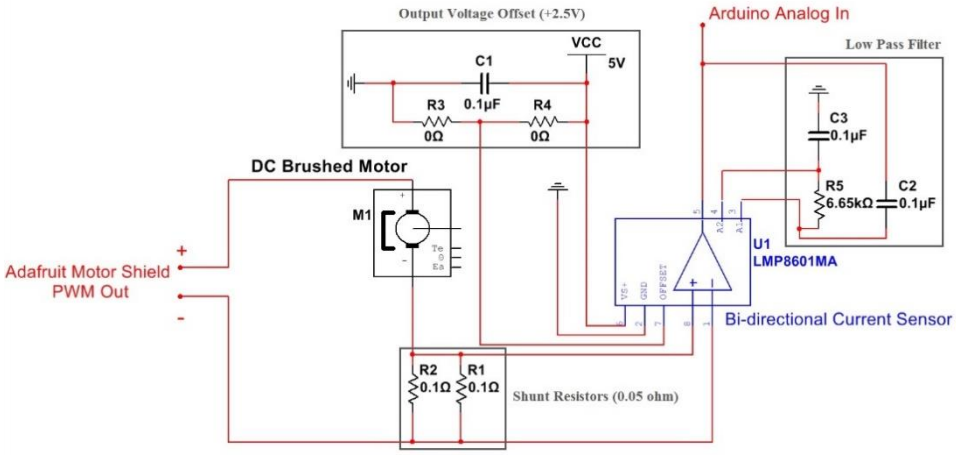



**FIGURE 22: CURRENT SENSING CIRCUIT**

This was done by connecting a shunt resistor and DC motor in series. Since the shunt resistor had relatively small resistance, in this case 0.05Ω, the voltage across the shunt resistor was extremely small and would not noticeably affect motor power. The 0.05Ωresistance value was chosen to scale the module output to the 0V to 5V range of the Arduino Mega ADC, to have the largest range of measurable forces, and maintain the most force resolution. The bidirectional current sensing amplifier was connected in parallel with the shunt resistor, which amplified the voltage across the shunt resistor for measurement, in this case the default gain being 20. Thus, the output voltage of the current sensor was calculated by equation (1) in which $V_{Sense}$ is the voltage across shunt resistor, g is the gain of amplifier and $V_{Offset}$ is the offset voltage determined by the voltage on offset pin (Pin 7) on amplifier.

$$V_{out} = V_{Sense} \times g + V_{Offset} \quad (1)$$

Because the offset in the circuit was defined based on voltage on module Pin 7, a voltage divider with two resistors having a value of 0Ω are introduced to raise all negative voltage value to positive. Base on equation (2), a voltage divider equation is used, substituting a small number, ε, to prevent error caused by division by zero.

$$V_{Offset} = \frac{R_2}{R_1 + R_2} \times V_{cc} = \frac{\varepsilon}{2\varepsilon} \times V_{cc} = \frac{1}{2} \times 5V = 2.5V \quad (2)$$

With help from previous equations, the current flow in the motor could be calculate based on output voltage of current sensor amplifier. The output voltage of current sensor amplifier in this case had been fed to the Arduino Mega analog input to a build-in ADC which scaled 0V to 5V into 0 to 1023 ADC counts. The soldered circuit of Motor Input/Output Board and Current Sensing Module are shown below in Figure 23 and 24.

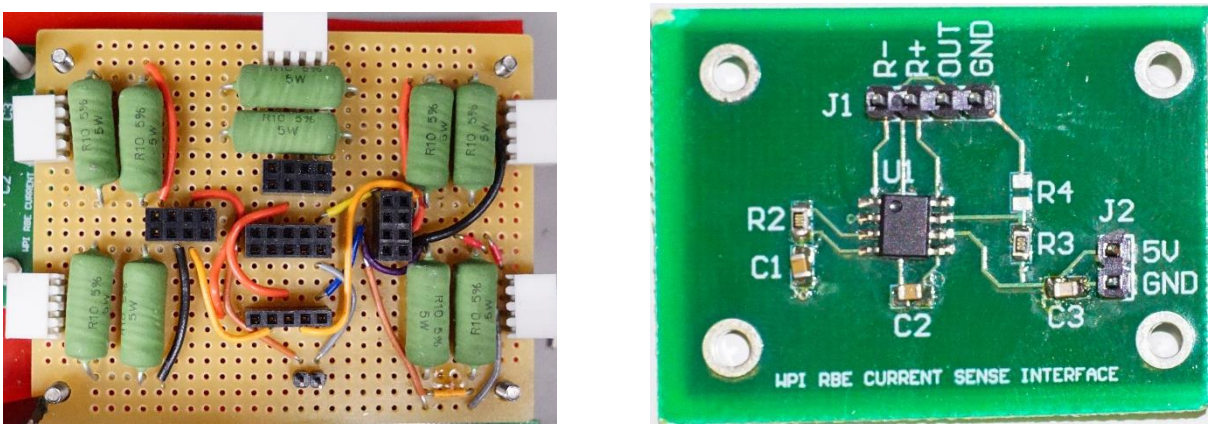



**FIGURE 11: MOTOR I/O BOARD (LEFT) AND CURRENT SENSING MODULE (RIGHT)**

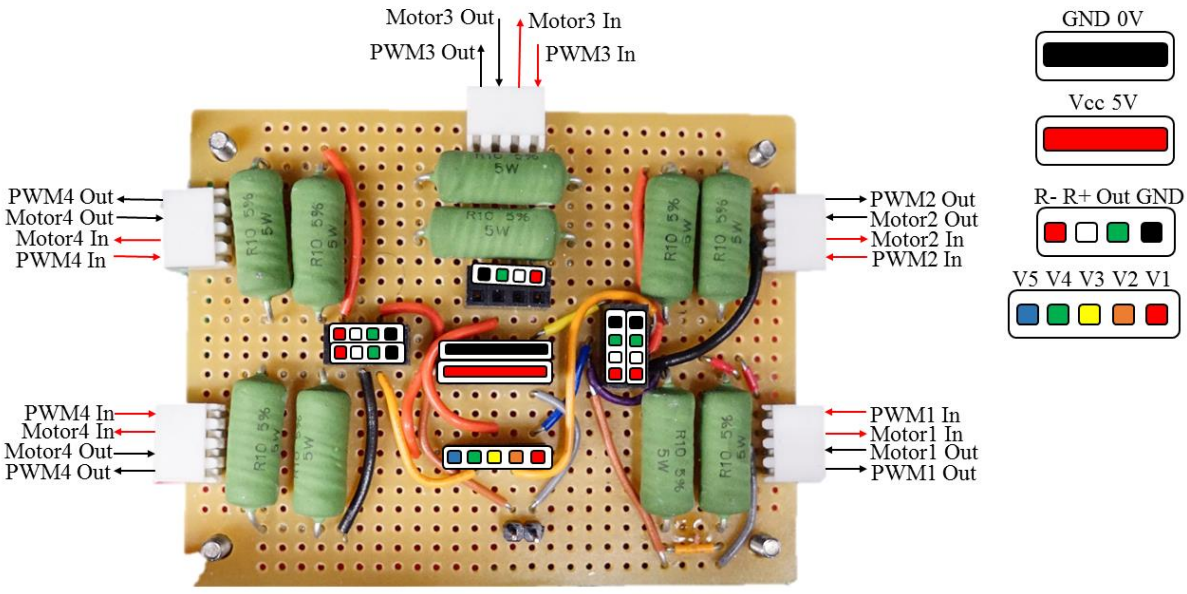
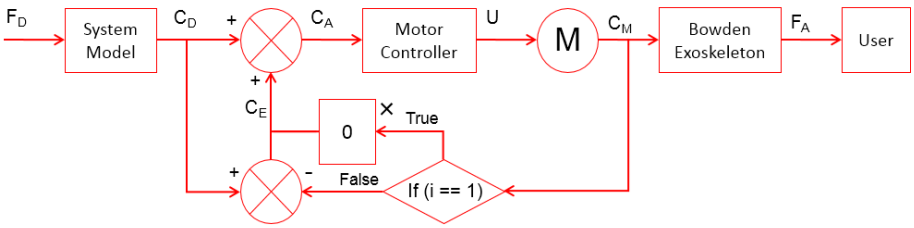



**FIGURE 24: SOLDER BOARD WIRING DIAGRAM**

### 4.2.6. Force Exertion Control

The force control in the final design was achieved by converting a desired force output to a current, based on a model of the system, adjusting for error, and then commanding the motors based on the calculated error adjusted value, at a rate of 60Hz.. The control loop handled error by first applying a current directly based on the desired force. Then, the output current was messured measured and compared to the desired force, to calculate an error. This error weaswas then used to increase the input current to the motor controllers, to attempt to close the force error gap. For an example, a desired current of 100 analog to digital convertor counts, that only outputted a measured current of 75 counts, would generate a 25 count error. The next loop of the control system, if still desiring the same force, would apply a current of 125 counts, to attempt a liner cancelation of the error in applied force. The. These current values were fed into the motor controllers shields, represented inside the motor controller block of the diagram below, and would generate PWM signals to run the motors at the commanded current. The final current control loop is shown below in Figure 25.




| Variables | Descriptions |
|---|---|
| $F_D$ | Desired Force Output |
| $C_D$ | Desired Current Output |
| $C_A$ | Error Adjusted Current Output |
| U | Motor PWM Signal Input |
| $C_M$ | Measured Current Output |
| $F_A$ | Actual Force Output |
| i | Control Loop Iterations Since Last $F_D$ |

**FIGURE 25: CURRENT CONTROL LOOP FOR FORCE EXERTION SYSTEM**

The current sensing could accurately measure the current on each motor and convert the value to forces in newtons being applied to the user through the Bowden cable exoskeleton.

## 4.3. Hand Pose Tracking System

To track the hand poses or the user, a Leap Motion sensor was implemented on a Microsoft Windows 7 machine, using the Leap Motion SDK version 2.2.0 and the core assets package version 2.2.2. The user's hand was tracked by wearing the infrared reflective glove, to verify functionality for the eventual exoskeleton glove. The Leap Motion sensor is connected using USB 2.0, with the user's hand, as show in Figure 26 below.

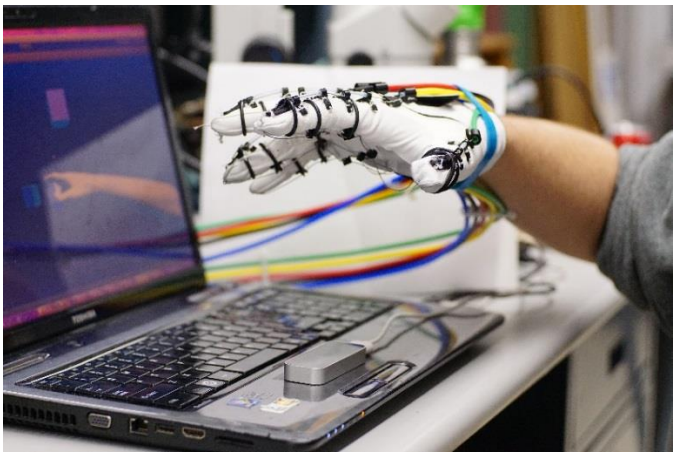


**FIGURE 26: HAND POSE TRACKING SYSTEM LEAP MOTION AND GLOVE**

This implementation of the Hand Pose System was analyzed based on the defined system requirements. The tracking of the user's hand pose was consistent over the sensor region and detailed, but still had some problematic sensing for some hand poses. Such hand poses included ones that either occluded portions of the hand from the sensor or had complex individual finger movements. However, the necessary poses for stress ball grasping of the demonstrative application were able to be performed consistently and accurately. The sensor and poses were tested by a user wearing the infrared glove while running the Leap Motion Visualization test environment, as shown in Figure 27 below.

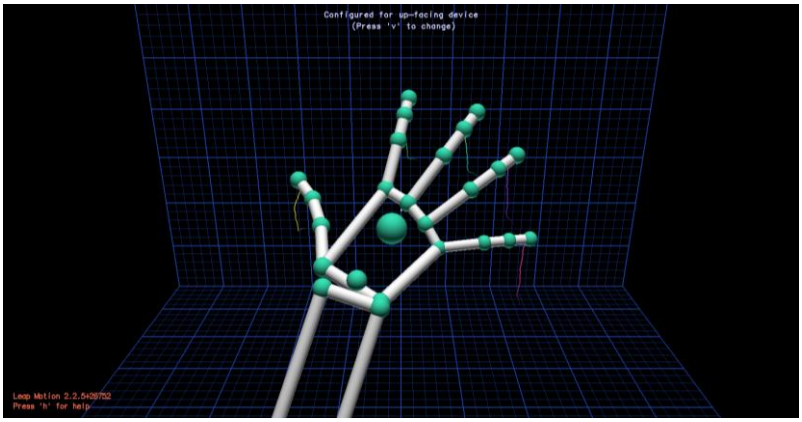



FIGURE 27: LEAP MOTION VISUALIZATION ENVIRONMENT

## 4.4. Simulation System

The simulation system was built based in Unity 3D version 4.6.0f3, with a Leap Motion tracking data driven user proxy, virtual objects with haptic behavior properties, a physics engine to handle interactions, and serial communication protocol to command the Force Exertion System. Unity contained development tools to create environments, objects, and activities, and could run an application to perform the desired task based on the user proxy and environment. As the user proxy interacted with the different types of objects in the environment, soft and hard, interaction forces were generated. These

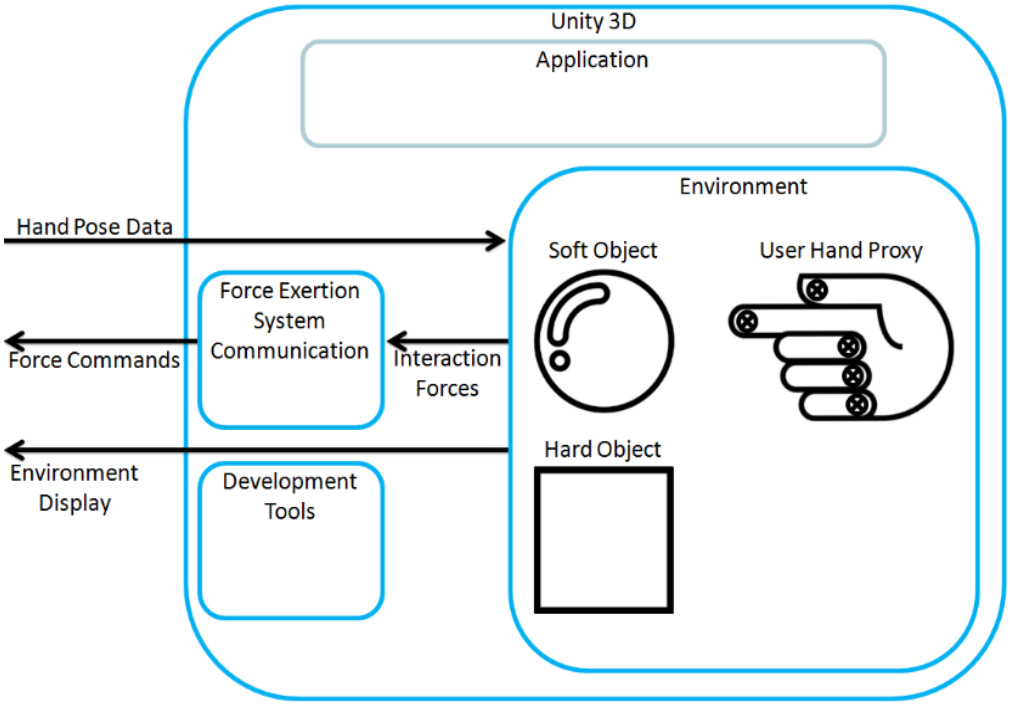



FIGURE 28: SIMULATION SYSTEM COMPONENTS

were processed into the component force that would be exerted on the user's fingertips and converted into protocol messages, including the force magnitude, direction, type, and what finger to exert it on. These messages were communicated to the Arduino Mega of the Force Exertion system using Microsoft's .NET 2.0 System IO serial port function. The Arduino Mega would then decipher the messages and update the goal forces of the Force Exertion System's motor control loops. A model of the Simulation System is in Figure 28.

### 4.4.1. Environment and Development Tools

The first feature of the Simulation System was the basic features of Unity, and how they allowed for holding a virtual environment and developing tools and applications. Unity had a workspace consisting of a 3-Dimensional space, where objects which contain components, such as behaviors and materials, could be added and manipulated. Development tools consisted of basic objects and components supplied by Unity, such as cubes and physics engines for collision handling, along with the ability to create custom components, in the form of scripts. These scripts, built in either C# or JavaScript, could be additional features to give more object abilities, such as artificial intelligence, or could contain the flow of logic and object management needed to make an application, such as a virtual museum tour. Unity ran using a component based development scheme, therefore it would run in a system of frames, each lasting about one sixtieth of a second, where the code for each component would be run once each frame. Therefore, with these features, the system requirements for holding a virtual environment and providing development tools was met. Future system developers will be able to build tools for more forms of haptic interactions along with applications such as medical rehabilitations and games.

### 4.4.2. User and Object Representations

The next feature of the Simulation System was the representation of the user and the virtual objects, and how they interacted to generate the different types of interactions. The hand of the user was represented by a proxy, an object driven by the hand pose data from the Hand Pose Tracking System, but was still partially affected back from other objects in the environment. The proxy contains two parts, a visible mesh and an invisible collider. The mesh was a 3D model of a hand, from the many available from the Leap Motion assets, which matched the pose data received from the Leap Motion. The mesh provided the user information on where their hand was in the virtual environment and established a presence in the environment. The collider was a less detailed set of shapes that follow the Leap Motion data, but react to physical collisions with other objects. This allowed for the user to have an effect on the

environment and therefore have interactions. For the proxy used in this iteration of the Simulation System, the mesh was a white clay looking hand, including the user's forearm, while the collider was a set of five sphere, one resting at each of the proxy's fingertips. The collider originally with the proxy, as set by the Leap Motion assets, utilized rectangular prisms for molding each bone. However these were not accurate to the controllability of the Force Exertion System and had inconsistent performance in interacting with virtual objects, due to the sharp angles causing unstable movements during collisions. Figure 29 below shows this proxy configuration in the Unity virtual environment.

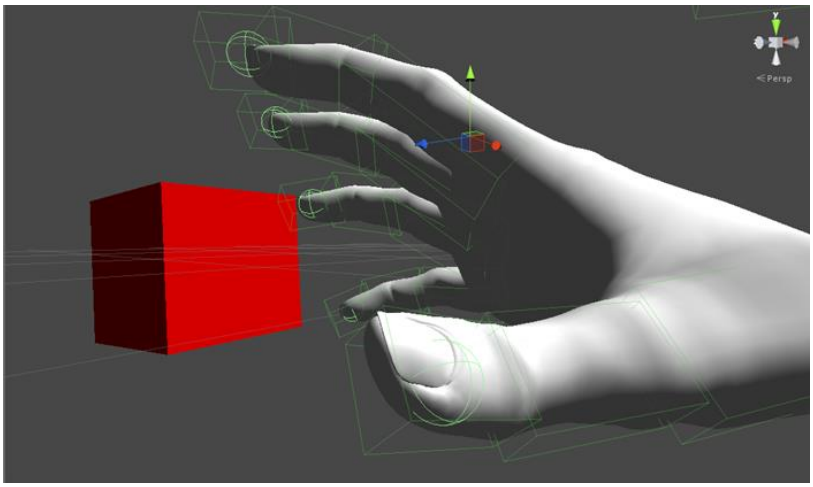


**FIGURE 29: USER'S HAND PROXY IN UNITY VIRTUAL ENVIRONMENT (MESH AND COLLIDER)**

This proxy configuration was chosen based on the types of forces that could be output by the Force Exertion System. By design, the Force Exertion System could only apply forces to the user's fingertips in this iteration. Therefore, to have all interactions in the virtual environment have a consistent response with exerted forces, the collider was built to only allow interactions through the user's fingertips with the environment.

Objects that allowed for haptic interactions also had a mesh and collider, along with a custom component script that dictated the type of interaction forces they responded with. The mesh and collider functioned the same way as the user's hand proxy and can be designed by a developer. The script handled interaction calculations.

### 4.4.3. Interaction Forces

The most key feature of the Simulation System was the generation of interaction forces. The three types of interaction forces, forces, barriers, and unrestricted movement, were implemented between two types of objects and three stages of interaction. The two object types were soft and hard objects. Each of

these was manifested as the component script attached to objects to handle interaction calculations. Basically, for either object type the interaction had three stages, elastic collision, deformation, and

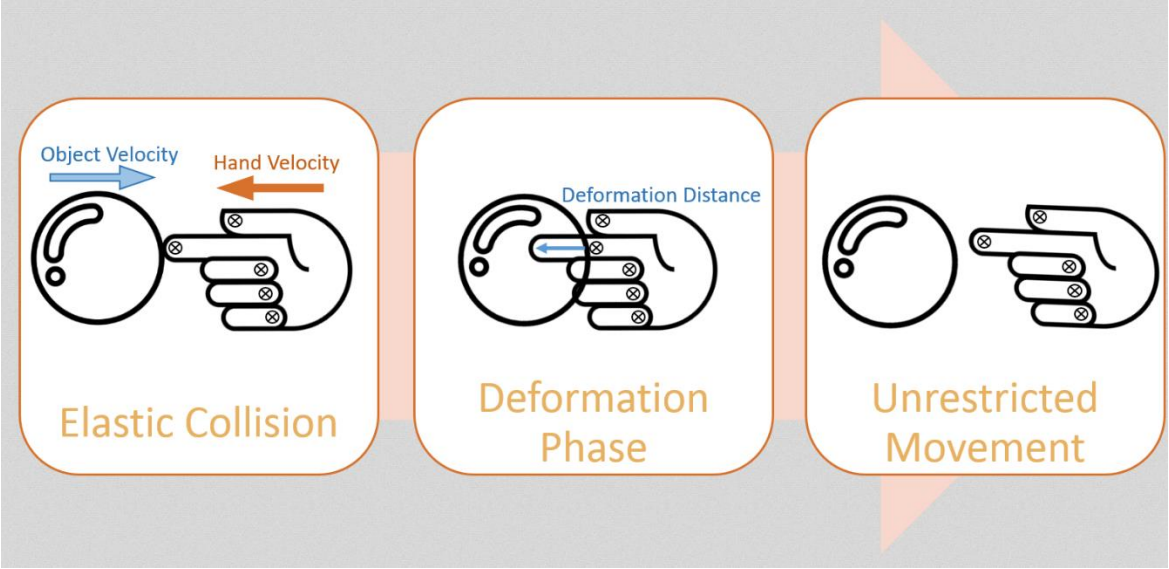



**FIGURE 30: OBJECT COLLISION PHASES FLOW CHART**

collision end. The only difference between the two object types was the stiffness factor used in the deformation stage calculation. A soft object's stiffness was variable and defined by the developer. A hard object had a maximum stiffness to represent solid non-deformable objects, such that the Force Exertion System applies a solid representative force. This force was approximately 4N and was applied once there was deformation equal to that of the user's fingertip pad maximum compression, about 3mm. This defined the ability for generating interactions with forces and barriers, but the distinction between forces and unrestricted movement was from the three stages of collision, as shown in Figure 30.

The three phases of interaction were broken down as follows. First was elastic collision, which calculated an interaction force based on the change in momentum of the object and fingertip in the collision. The force generated by this method matched the natural initial collision reaction generated by the Unity physics engine, the tool that managed physics based movement and collisions in Unity. The Force Exertion System was able to react to this force without user noticeable delay. However, the full magnitude of the force was often not felt due to a rise time to exerting the full force, that would be cut short when the next phase of collision occurred With this stage included, there are no discontinuities between the virtual environment and the user's experience.

The next phase was the deformation stage, which triggered if the finger and object colliders were still in contact after the first phase. In this phase, the magnitude of distance the fingertip of the user penetrated the collider of the object generated a proportional error force, based on a penalty feedback method. This interaction represented the object being deformed by the user and pushing back with a force related to the distance and object's stiffness. The calculation of this penalty force required some extra computation due to the directly accessible data given by Unity and the Unity physics engine.

The final phase was the unrestricted movement phase, and was triggered by the end of a collision between the colliders of an object and a user proxy finger. This simply ended the forces being applied on the user by triggering an unrestricted movement force type command to the Force Exertions System.

The forces calculated from each of these phases would be processed before being commanded to the Force Exertion System. Since the capable forces of the Force Exertion System were only in the direction perpendicular to the fingertip pads, each collision generated force would have its component along this direction taken to be sent to the Force Exertion System. The last step was to convert the force into two bytes, one signifying magnitude from 0 to 255, 0N to the maximum force of 4N, and one specifying direction, 1 for opening the hand and 2 for closing the hand. The code performing these equations for handling interactions with soft or hard objects can be seen in Appendix B.

### 4.4.4. Force Communication

The final component of the Simulation System was the communication between Unity of the Simulation System and the Arduino Mega of the Force Exertion System. The Unity side consisted of a script placed on an object just for background scripts and used the Microsoft .NET 2.0 System IO serial port library. The Arduino Mega side used the standard Arduino Serial functions. This system had very temperamental and inconsistent functionality and went through some changes from the initial design decision before reaching the final version.

In the initial version, the Unity side would send arrays of bytes representing force commands, following the command message protocols in Table 2 of Section 3.3.7, and would retrieve messages from the Arduino by reading the newest set of bytes according to the one message type the Arduino would send. On the Arduino side, each cycle of the 60Hz processing loop, the incoming serial buffer would be checked to see if the oldest byte was one of the defined message signs from Table 2. If so, the correct number of bytes would be read and processed depending on the message type. If not, then the byte would be discarded and the process would repeat on the next cycle. Handling of the incoming bytes was

performed with the Arduino library serial functions, the Arduino Mega serial buffer, and the knowledge of how long types of commands should be. The buffer contained all past messages until they were addressed and was set so that whenever messages were read from it, awareness of a byte still in transit would cause the system to wait until completion before reading the value and returning. The serial timeout was set 0.5 seconds to avoid waiting too long on problems and causing delayed force feedback on the user. In practice this setup allowed the Arduino Mega to grab complete commands and would only notably hang on byte reading when communication between the components was lost, such as the cable being disconnected. Timing of the systems were set by Unity and the Arduino Mega both running and processing commands at the same rate of 60 Hz, in order to keep up to date with current commands.

This method had the issue that the Force Exertion System could get out of sync with the Simulation System, since it had no awareness of when the commands originated, instead assuming only one command per frame. In order to improve this, the Unity side had a function added that would publish a byte at the end of each frame, signifying that the frame has ended to the Arduino. The Arduino side was updated to keep reading and processing input bytes until an end of frame message was found, therefore keeping in sync with the forces dictated by the actions in the Unity environment and application. Based on the 9600 baud rate of serial communication used for standard communication, the limit of messages were 20 bytes per frame, or 1200 bytes a second. Therefore, the limiting factor on command communication was the communication, for it would reach its limit of 5 single finger commands a frame, before Unity of the Arduino Mega reached its calculation limits based on their observed task processing speeds during full system task performance. This communication rate was chosen for being fast enough to provide the maximum amount of commands expected, for an optimized application, while still being slow enough not overload the system serial buffers during a period of not being addressed. A maximum of five commands was acceptable, for any additional force commands during the same update frame would be redundant and would have no result, except for delaying the Force Exertion System.

While this communication structure was able to be built and implemented, working with it for the full system was not implemented consistently. Over time, with the integration of the other features to a full system code on the Arduino Mega and Unity, serial communication performance experienced errors, and eventual lopsided communication where messages sent from Unity to the Arduino Mega no longer generated any responses. The final codes used for communication methods, based on the original implementation, can be seen in Appendix B.

## 4.5. Visualization System

The Visualization System consisted of a computer monitor and the output from virtual cameras held in the Simulation System. Figure 31 below shows the virtual environment with a virtual camera and its

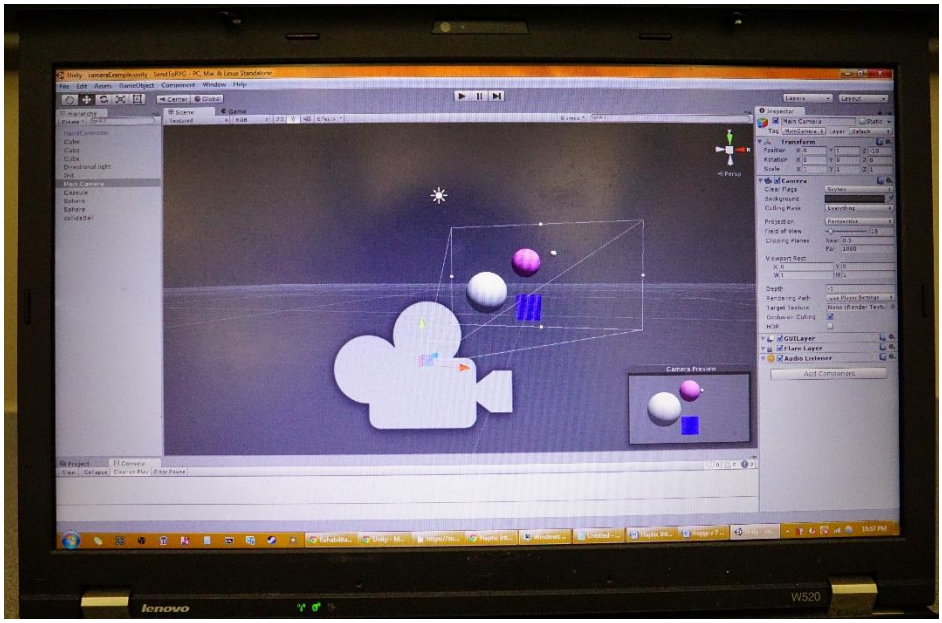


**FIGURE 31: VISUALIZATION SYSTEM MONITOR AND VIRTUAL CAMERA WITH OUTPUT**

output, displayed on a typical monitor screen.

Any monitor can be used for this method of displaying the virtual environment, as long as it is compatible with the host operating system, in this case Microsoft Windows 7. The virtual cameras work in the same function as all objects in Unity, behavior defined by components. The developer is capable of controlling the cameras' actions in space, and which ones have their output feed to the user's display. In addition Unity has implementation tools and guides for using the Oculus Rift as a display method. Therefore, the simulation system is capable of displaying the virtual environment with many controllable features and can be simply enhanced with Oculus Rift head mounted display functionality.

## 4.6. Demonstrative Application: Learned Non-Use Rehabilitation

The implementation of the learned non-use hand rehabilitation demonstrative application captured the main haptic interaction functionality of the full procedure process outlined in the design decisions of Section 3.4. The task of a patient grasping a simulated stress ball, and having the resultant force applied was implemented. The patient would place their hand on a rest, such that their wrist and hand were still free and could be tracked by the Leap Motion. Then, based on a simulated grasp of a virtual stress ball, the amount of closure of the patient's hand would be measured and used to generate a soft object deformation haptic interaction response. The magnitude of this force response was also adjustable by a scaling factor, which through subtle alteration over multiple grasps could generate the desired patient response of exerting forces past their learned limits.

This application used a simulated grasping method of interaction compared to the direct interaction of the soft and hard objects previously. Attempting a grasp to hold objects, when only able to manipulate with five frictionless points is not directly possible, so grasps would have to be detected and objects would have to be linked to the hand, in order to simulate the interaction, in use of low detail proxy colliders. In this case, the ball started in a grasp, and interactions were applied by the user, using the Leap Motion API's grasp closure detection function, which gives a value from 0 to 1, representing open to closed grasp of the user's hand as measured by the Leap Motion. This value dictated a force response based on the same logic as the deformation phase in the previous interaction methods. The grasp closure took the place of the explicit object penetration and generated a proportional force response to all the user's fingers, based on the stiffness of the ball, and now the additional scaling factor. An image of the demonstrative application running can be seen in Figure 32 and the code used to run it in Appendix B.

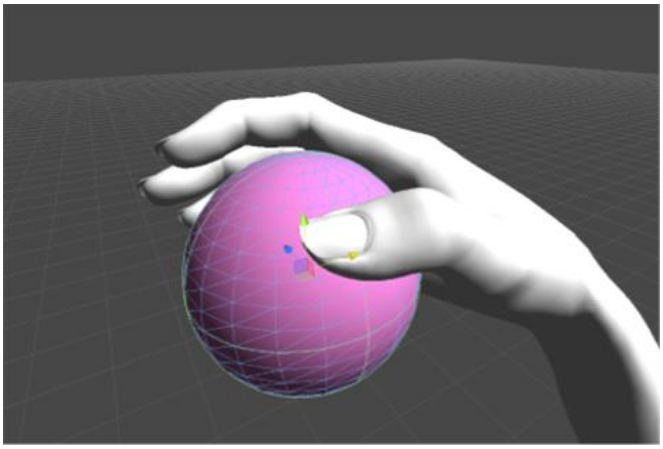


**FIGURE 32: DEMONSTRATIVE APPLICATION SIMULATED STRESS BALL GRASP**

# 5. Results and Discussion

## 5.1. Hand Pose Tracking System Functional Analysis

The Leap Motion was satisfactory over the expected region of the learned non-use rehabilitation demo. For the demo, since the objective was to measure and manipulate force based on the user's full hand grasp, only a static hand with slow all finger curl ranging from open to closed was needed to be measured for satisfactory use, at a minimum. The Leap Motion, even with gloved hands, was capable of this performance, with these actions being mirrored in the visualizer without noticeable delay or error in hand pose.

It was also desirable that the Leap Motion is capable of more radical movements and poses, such as fast pose changes and nonsynchronous finger movements. In this area the Leap Motion still performs well but some problems appear that detract from the overall system performance. First, were the responses for fast position changes of the hand, with fast being the speed of an average able bodied user waving their hand. When the finger poses were kept relatively static during the movements, the Leap Motion matched the user's speed and rarely incorrectly measured the finger poses. At faster speeds or changes in finger poses during the waving, the Leap Motion would have issues, such as incorrect measurements, however these would be corrected in the course of about a second. When the user would move in nonsynchronous finger movements, such as moving from a fist to an extended ring and index finger, the Leap Motion would often measure an incorrect pose, such as different fingers being the ones the user had moved. This happened approximately half of the time when such nonsynchronous movements were attempted, but would be corrected within a second once the user moved their whole hand position slightly.

Another primary limitation of the Leap Motion is sensory occlusion, where fingers cannot be detected by the sensor, due to other parts of the hand being in the way of sight. This limits the range of orientations capable to the user while still having desirable tracking performance. Some problematic poses included the user closing their hand with the back of their palm facing the Leap Motion, and closing less than all their fingers when oriented 90 degrees from front palm facing the sensor. In these situations the Leap Motion would lose sight of the user's fingers and its built in model based assumption method would activate to determine a pose. This model had a tendency to assume full

finger motions, so these example poses would get read as closing to a partial or full fist poses. These measurements would not be changed until the user exited the pose and removed the occlusion factor, where the measurement would jump to the correct pose.

The Leap motion had consistent and over sufficient performance in a typical computer office type station. The presence of other objects, such as computers, along with electromagnetic radiation did not have a noticeable effect on the sensor's performance. The only issue was in background color and lighting when the infrared reflective glove was used. When a white ceiling or a bright light was in the background of the Leap Motion's camera view, it would have a difficult time detecting the user's hand, often requiring multiple entrances into the sensor's work space or having a cover block the problematic background.

Overall, the performance of the Leap Motion met the needs of this iteration of the Haptic Interaction System and set a good and workable base for future iterations. The Leap Motion met the needs of the demo application while giving more functionality by allowing many normal types of hand movements to be used in interacting with the virtual environments, consistently throughout the work space. The issues in incorrect measurements, occlusions, and pose jumping hinder this ability, since such would supply the wrong interactions, limit certain types of movements, and cause sudden maximum forces due to near instantaneous speeds, respectively. These issues could be improved in future iterations by introducing more methods of sensing the user's hand pose, such as Leap Motions at different locations, to remove occlusion, and or data gloves, to have different types of error sources, then combining the information such as with a Kalman filter for better pose accuracy and consistency over more scenarios.

## 5.2. Force Exertion System Functional Analysis

The Force Exertion system was capable of applying forces, barriers, and unrestricted movement interaction types to the user. Based on the properties of the motors, the current limits of the Adafruit Motor Driver Shields, and diameters of the motor pulleys, the maximum force exertable on any finger is 4N, without heat sink additions. Through the combination of all the current control hardware, the system can exert 511 discrete forces from negative 4N to Positive 4N. While Unity was unlimited in specifying the precision of a force, and the ADC measuring the current sensor had 10 bits of resolution, the

Adafruit Motor Control shields could only specify 8 bits of current values, in a positive or negative direction. Therefore all of the resolution of the system is reduced to 511 different forces, the 512th being a redundant 0Nforce, and therefore gave a force resolution of 0.0157N, slightly over the weight of a USD quarter on Earth at sea level.

### 5.2.1. Force Control

In testing the force control loop, while distinct and consistent forces were achievable, there were issues with accuracy over the entire range of exertable forces. Additionally, due to the serial communication issues, the force control was tested by hardcoding different goal currents in the Arduino Mega, instead of defining them from Unity. As shown in in Figures 33 and Figure 34 below, are a measurements of the force control response to a step function in desired force goal, graphed as the values were read in iterations of the control loop running at 60Hz.

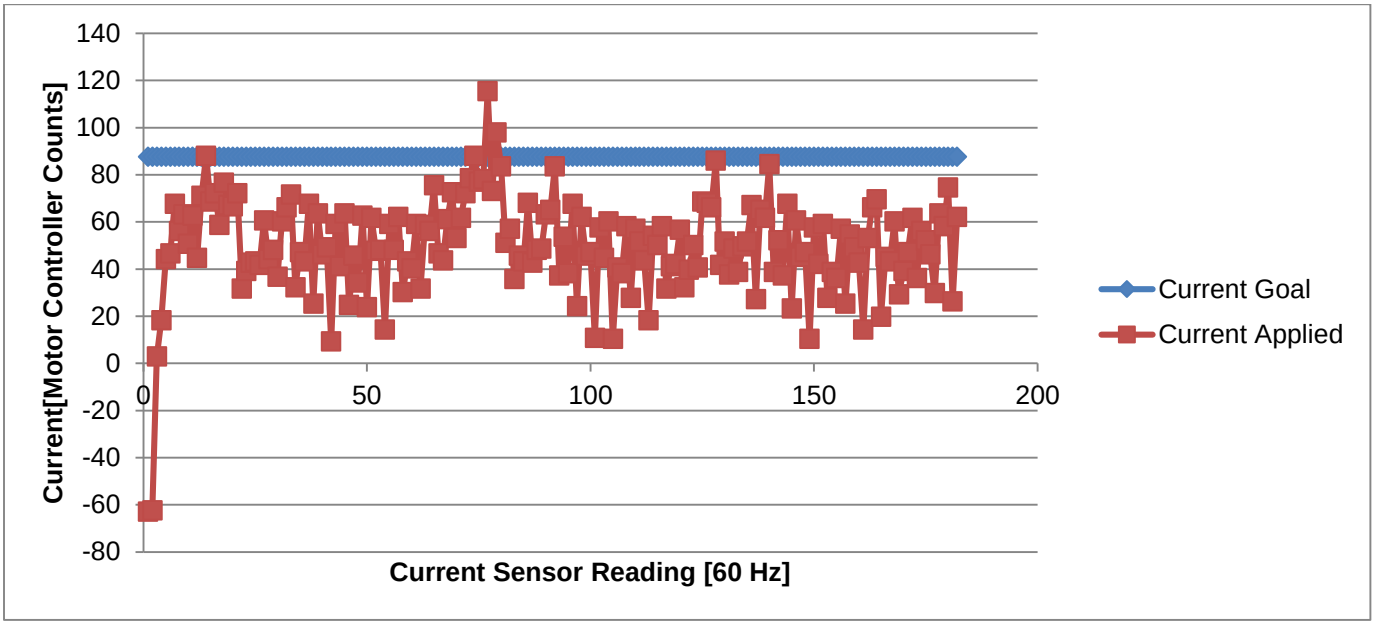



FIGURE 33: FORCE CONTROLLER STEP RESPONSE TO 150 MOTOR CONTROLLER COUNT TARGET

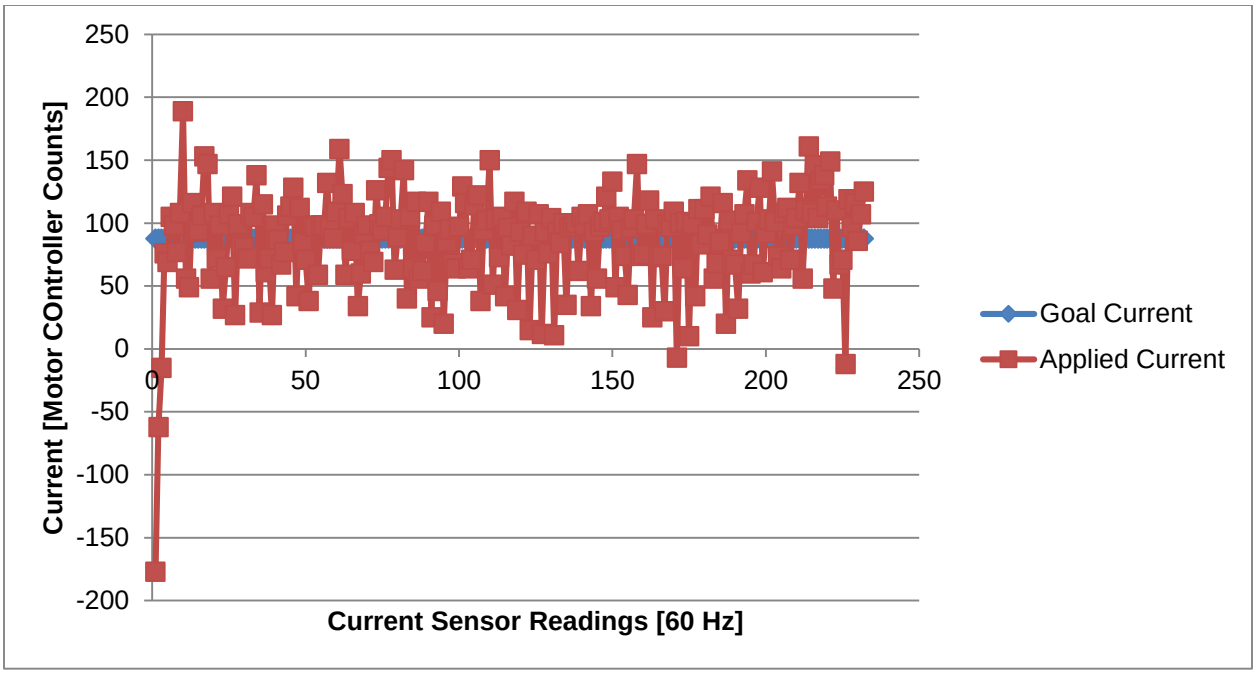



**FIGURE 34: FORCE CONTROLLER STEP RESPONSE TO 255 MOTOR CONTROLLER COUNT TARGET**

These graphs present the goal and applied forces of the Force Exertion System, as current, in the -512 to 511 count range of the current sensors, when a goal force is introduced at time count 1. In the first test, the force control was given a motor controller command of 150 counts out of a max 255, which would result in the 84 count value, which is therefore representing the goal for this command. The goal current was defined by a linear relationship given by max motor controller count and the resulting measured current count, or 255 and 84 counts respectively. In the second test, the target for the force controller was changed to 255 motor controller counts. In both of these tests, the testing finger of the exoskeleton was set to a joint limit, to remove the human reaction time effect from the recorded rise time to the goal currents.

The most immediate noticeable feature of the applied currents was the noise, being on the range of 25 counts above and below the average stable current, at about 45 counts. This error is approximately 0.1039 Amps and did not result in sensible oscillations' in forces applied to the user. The different average errors between the current goal and the current applied, for the different tests, was indicative of another issue with the system. The relationship between applied motor controller count and applied current was nonlinear. Therefore, with the same force control loop, the accuracy of the applied force varied depending on the desired force. The response time for these tests all showed a very quick and consistent response. In each test, the average response value of the applied current was reached by the

seventh data point, or approximately 0.117 seconds. In practice, these rise times were not noticeable to the user, with a seemingly constant and immediate force applied once they applied resistance.

The performance of the force controller could most benefit from improvements to the issue of accuracy over the range of forces. The other features of the system were sufficient for this interaction of the Haptic Interaction System, since no slack of lag in forces was added to the system, over that present in the mechanics of the exoskeleton. The force controller would need to be given awareness of the nonlinear input to output current relationship, or methods of coping with the errors through iterative error management. Due to the fact that the force controller had a limited space of possible conditions, desired applied currents, a linear approximation of the space could be used for more accurate results. While this would provide more accurate responses to desired currents, errors would still be present, and the process would have to be done regularly on each current sensor, to account for their different biases and nonlinearities. Additionally, a PID controller could be implemented. While such a force controller would still require tuning for each sensor, management of lingering errors could be performed through the integral component, along with noise handling through the derivative component, to result in a more robust solution than just a linear approximation.

### 5.2.2. Exoskeleton Stiffness

The dynamics of stiffness of the exoskeleton of the Force Exertion System throughout interactions had multiple sources, but was sufficient for this iteration of the Haptic Interaction System. These factors that affected the stiffness of the device being exoskeleton system slack, motor back drive ability, and the under constrained nature of the exoskeleton in respect to the degrees of freedom of the user's fingers. There was some slack in the exoskeleton, mostly due to the flexibility of the glove. While this was noticeable when the user would try to back drive the motors, when the motors actuated the fingers, the response was immediate without periods of loose finger cables. Additionally, the motors were capable of being back driven, however with some resistance, such that could have tier even an able body user with extended use. However, due to the under constrained nature of the exoskeleton fingers free movement was still easily possible. Since the exoskeleton could only control the overall curl of each finger, the user could still move in a subset of the state space where finger curl remains constant. While this was a side effect to only controlling forces in the perpendicular fingertip pad direction, it allowed for additional ease of movement, such that haptic interactions were easily possible in the limited directions of this iteration of the Haptic Interaction System. Therefore, while some low stiffness was felt by the user in interactions, this was not presence breaking and is improvable in future interactions of the Force

Exertion System. Increased stiffness of this exoskeleton could be achieved slightly by further tightening of the gloves' fixtures, but would benefit more significantly from additional methods of control, such as Bowden cables to more joints, or a ridged structure supporting the glove.

### 5.2.3. Exoskeleton Safety

Safety was a concern with the exoskeleton of the Force Exertion System, in the sense of adhering to the flexion limit of the user's fingers. While the maximum force of the Force Exertion System was under that of an average able bodied user, relying on that alone did not assure safe performance, especially with potential patients with lower strength capabilities. Flexion limit adherence was assured by the mechanical structure of the exoskeleton glove. At the maximum flexion, in the open or closed direction, for the glove, the physical cable guides would collide, therefore taking the force load off of the user, as shown in Figure 35.

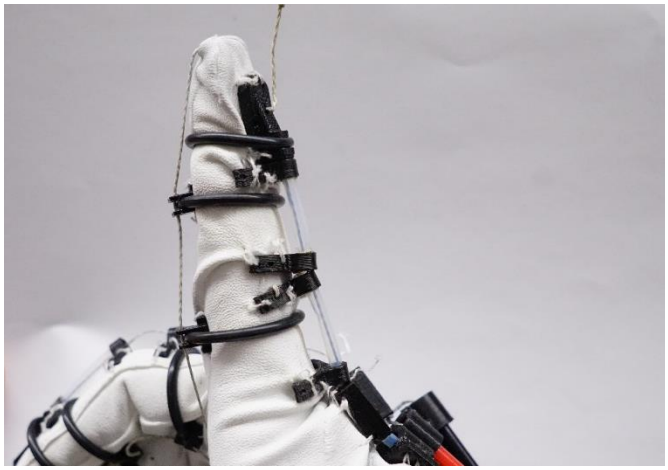


**FIGURE 35: EXOSKELETON GLOVE INDEX FINGER AT MAXIMUM FLEXION LIMIT**

## 5.3. Simulation System Functional Analysis

The Simulation System was sufficiently capable of representing environments, objects, and user proxies, along with performing haptic interactions and calculating forces, while force communication was less ideal. The development tools provided with Unity allowed for building environments and creating objects, haptic interaction procedures, and the demo application, and therefore can be used for additional development. Object and user representation was also sufficient. Objects could react in a physical nature similar to real objects, using the Unity physics engine and trigger haptic interactions. The user proxy could match the user's real space movements with high accuracy and low delay, and could trigger haptic interactions as well. An issue with the chosen fingertip based proxy is that it still allowed forces not perpendicular to the fingertip pads to interact in the virtual environment. This would cause a disconnect where the user would perform an action and only feel a portion of the resultant interaction forces. This could be fixed by an even further simplified user proxy collider or customizations to the

Unity physics engine. However, the implemented method still provides interaction functionality in enough interactions to be sufficient for this iteration of the Haptic Interaction System.

### 5.3.1. Haptic Interaction Generation

The haptic interaction processing components of the Simulation System had mixed to sufficient performance. For the three stages of soft and hard body haptic interactions, the elastic collision and unrestricted movement phases were sufficient while the deformation method was more inconsistent. Elastic collisions would calculate a realistic interaction force and ending the collisions would trigger a command to set the finger back to unrestricted movement. The deformation phase would often have spikes in force output, compared to the expected gradual increase with object penetration. The amount of calculations per each frame with a deformation collision would cause a slowdown in Unity performance. The spikes in forces could be improved by modeling the differences between the collider location data and the finger location data directly from the Leap Motion. The difference in this data and how the colliders triggered the collision while the Leap Motion data determined the force magnitude generated caused the generated force to be near saturated when the collision first started. The slowdown in performance could be improved by optimizing the collision code or Unity physics engine for this type of penetration measurement. The grasp interaction force generation, as used in the demo application, had much better performance. Due to only working on one source of information, the Leap Motion measured open to closed hand percentage value, the deformation style force response is continuous and realistic.

## 5.4. Serial Communication Issue and Sub-System Integration

Due to the eventual breakdown of communication of data from Unity of the Simulation System to the Arduino Mega of the Force Exertion System, the desired full implementation of the subs-systems of the Haptic Interaction System was not possible. Throughout the project, code was developed to run these systems independently and in combination. During the development phases of functional serial communication, along with individual sub-system performance, functional results were able to be gained about all the desired functionalities of the Haptic Interaction System, albeit not performing completely to the desired task. For example, the demonstrative application was able to be run with simplified serial communication for one finger interactions, from which performance was extrapolated and evaluated.

For the serial communication issue itself, a detailing of the varying functionality illustrates what could have caused the problems. The communication structure was first developed on an Arduino Uno

microcontroller, which ran the same Arduino software system as the Mega, the only difference being number of hardware ports and some on chip features. Here the system could communicate effectively, when the code only contained the processes for the communication. When this was implemented on the Arduino Mega, and the additional codes for force control and virtual objects were added to the overall system, communication had value errors. At first message data were getting converted to null, 0 binary, values, even when converted back to the Arduino Uno. This error was difficult to discern the source from, but was eventually solved. Through reworking the code that iterated through messages from Arduino to find the latest state data, and altering the rates of data reading on the Unity side, the loss of data was stopped. However it was left uncertain as to the exact cause of this issue. Even then, a persistent pattern of messages failing over time was found, when using the fully planned message protocol structure. This pattern resulted in in an average message data success rate of 93%, as determined by trials of repeated 5 byte messages back and forth between the sub-systems.

It was at this point, where the communication would be inconsistent and only run occasionally, with no discern relation to the alterations of the other system codes. The data recorded for performance of the various interactions between the sub-systems were taken at this point, including basic one finger haptic interactions with hard and soft virtual objects, and with the demo application. Even then, an issue began occurring regularly, where the Arduino Mega would crash during tasks between it and Unity, often less than a minute into operation. This error was not able to be reproduced during standalone performance of the Arduino Mega without the serial communication. Eventually, communication also became lopsided, with only messages from the Arduino Mega to Unity performing correctly, and therefore further testing of the combined Haptic Interaction System could not be performed.

While the inconsistency observed of these errors made it difficult for problem analysis, it has been narrowed down to the method of serial communication and possibly another component of the overall code. The hardware being an issue is unlikely due to the same full code performance on both microprocessors. Since hardware did not have an effect on the error, nor did simplifying the serial communication process. While still developing the overall system code, simplifying the serial communication to smaller messages or even just single byte transmissions back and forth, without command identification, improved the communication multiple times in the course of this project. However with the final code implemented for all the other sub-systems' functions, this simplification method no longer restored communication. A plausible reason for the noticed Arduino Mega crashes could have been due to serial buffer overflow due to messages not being processed soon enough. This

was one of the few differences between running the Arduino Mega in the full system versus isolating just the Force Exertion System. With the last implementation of the Force Exertion System code, it could not communicate with Unity and would crash often when connected through serial, but would perform all functions correctly, including targeted force control, when operated in isolation with its own in-editor serial input prompt.

Obviously, the communication between the virtual environment and user force interface device is vital to having haptic interactions with simulated environments. To achieve the full potential of the Haptic Interaction System more analysis into the cause of the communication failure, along with different methods of implementation communication, would need to be performed. A procedural rebuild up of the full system code could be performed, starting with a functional basic, one message type serial communication. With each feature added back in, the communication performance would be analyzed to try and find any potential unseen problematic connections. This analysis had been attempted in this project; however, once basic serial communication was achieved in isolation, it was added back to the full system code, replacing the old method, which could have left different lingering issues. Also, more research into the methods of serial peripherals to the Unity 3D game engine could be performed, for communication methods that are more robust and functional, as is normally seen in video game controller performance. These problems could have been related to the Microsoft .NET framework, and therefore, switching to the system compatible Mac OS X, and attempting use of its serial communication frameworks, could improve functionality or assist in identifying problems. It could also be beneficial to add to the experience pool of the development team of the Haptic Interaction System, to include members with knowledge of high level computer science and system communications, beyond that of the hardware level code experience by this iteration's robotic engineering background members.

## 5.5. Visualization System Functional Analysis

The Visualization System was sufficient for the basic requirements defined for the first iteration of the Haptic Interaction System. The virtual environment could be displayed easily and consistently. In addition Unity had implementation tools and guides for using the Oculus Rift as a display method. Therefore, the Simulation System was capable of displaying the virtual environment with many controllable features and could be enhanced with Oculus Rift head mounted display functionality.

## 5.6. Demo Application Functional Analysis

The demonstrative application for learned non-use rehabilitation, while only being part of the complete rehabilitation tool, and being the most susceptible to the serial communication issue, still demonstrated the functionality of the Haptic Interaction System. In phases of functional serial communication, the demo application was run on one and two fingers, to observe the response. The haptic interaction generated had similar feel to the soft virtual objects, as expected. However, the change in force was much more gradual and noticeably scaled down to 0N, without discontinuity, upon presenting a full open hand. The leap motion was able to accurately track the pose of the user's hand over the course of the grasps, without noticeable pose jumps causing sudden force changes. In these tests, the serial communication consisted of sending one to two single finger force commands, and were successful over the sort duration of the task. However, the Arduino Mega crash issue always limited these tasks to about less than a minute. Together, all aspects were included from displaying the environment, interacting with an object, generating and communicating force commands, and applying these forces to the user. With this the concept of the Haptic Interaction System was capable of being conveyed and communicated a conceptual application of how the system could be beneficial.

## 5.7. Development Guide

The development guide contained the necessary information for future users to set up and work with a Haptic Interaction System. Assuming the physical components for the system are had, users could have haptic interaction performance. Therefore futures users of this iteration will be able to install all that is needed to continue developing. Additional features could be sections relating to building a copy of the system, or how to enhance it.

## 5.8. Overall System Functional Analysis

Each of the sub-systems, with exception of serial communication in the final version, met the requirements on functionality laid out for the first iteration of the Haptic Interaction System. Together with these sub-systems it was possible to design virtual reality applications and perform haptic interactions with the objects held in their environments.

With this functionality however, the accessibility and expandability of the Haptic Interaction System was still achieved. Accessibility of the system was defined by the ease of access of its components and building of its sub-systems. Each of the components chosen are commercially available from well-established and large scale distribution companies, such as Arduino, Adafruit, Leap Motion, McMaster-Carr, and Texas Instruments. With the exception of the current sensing modules which require some circuit building, the components of the sub-systems are very simple to use to build the Haptic Interaction System. The total costs of the Haptic Interaction System components are shown below in Table 3.

| | Cost |
|---|---|
| Motor | $174.75 |
| Microcontroller | $45.95 |
| Motor Drive | $39.9 |
| Leap Motion Controller | $79.95 |
| 3D Printed Parts | $15 |
| Current Sensor | $7.7 |
| Acrylic Board | $21.19 |
| Glove | $20 |
| Tube | $5 |
| Sum | **$409.44** |

Table 3: Bill of Material

The expandability of the Haptic Interaction System was defined by how its components could be built off of for future functionalities and applications. On the virtual environment side, Unity 3D allows for development of custom applications and tools. On the physical side, the Arduino Mega allows for additional hardware and different control systems, to expand the capable haptic interactions users can feel. With the Development Guide, the setup and use of the Haptic Interaction System was explained, allowing for comprehension of how to build and expand the Haptic Interaction System.

# 6. Future Work

With the achievement of the creation of a platform Haptic Interaction System, the team's direct goal has been completed. However, this platform is a key to reaching the ultimate goal of the project, a system that allows for the full development of haptic interaction applications and hardware, to be able to get the development ability out to the people. To get to that point, there is much future work that can be done, which fall into three major categories.

## 6.1. Sub-System Enhancement

Each of the sub-systems of the Haptic Interaction System could be enhanced with extra features to provide for more available experiences and development tools. The Force Exertion System could be expanded to provide forces on more joints of users' hands or on additional body parts, such as the entire arm, for more detailed or larger scale haptic interactions. More types of haptic interactions for virtual objector or dedicated haptic interaction physics engines could be added to the Simulation system, for more varied, refined, and efficient virtual haptic interactions.

## 6.2. Virtual Reality Presence Sub-Systems

Additions to the Haptic Interaction System for granting users presence in virtual reality environments. These sub-systems could interact with more of the users' senses, such as visual, audial, or even olfactory and gustatory, to provide more types of simulated experiences. For visual as an example, a head mounted display could be added to the Visualization System, such as the Oculus Rift, through the built in Unity Oculus Rift tools. This addition would allow for strong presence applications where a user can nearly freely see an environment and interact with it, through the natural methods used in real space.

## 6.3. Application Development

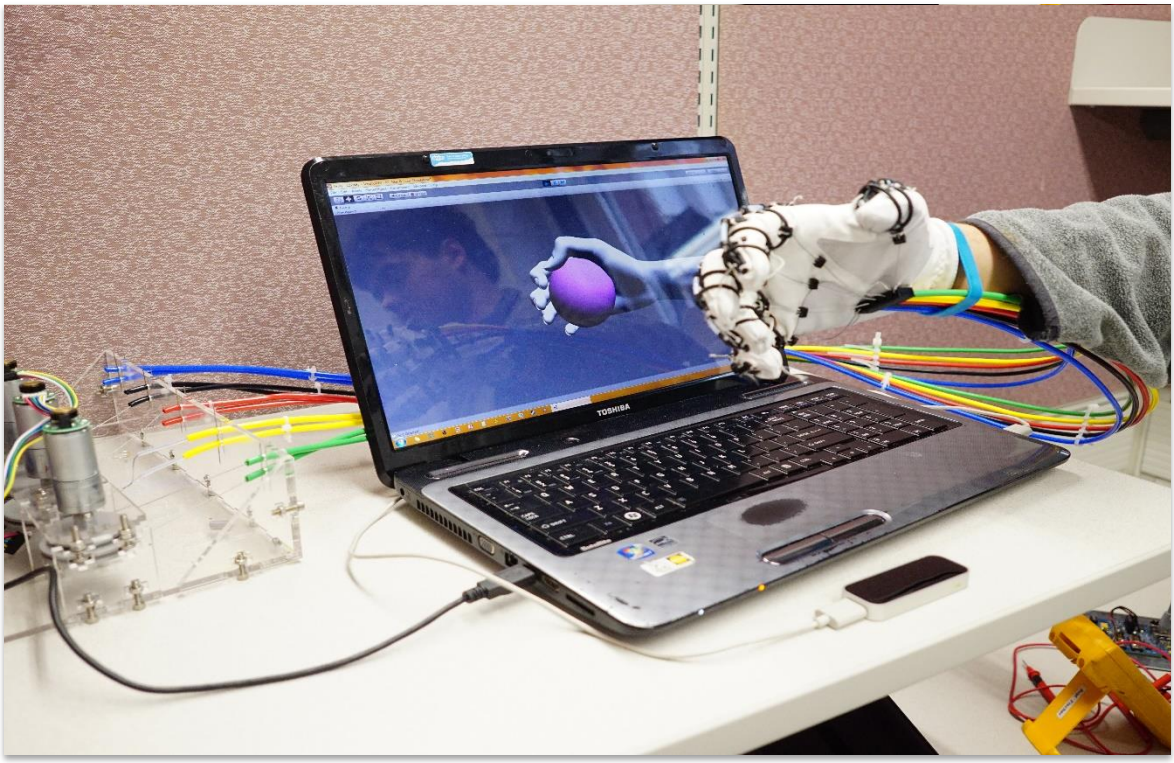
The Haptic Interaction System could be benefited by developing applications for it. These could be further demonstrative applications, to show additional features and application concepts. The applications could also be completed ideas, based on the functional state of a certain iteration of the Haptic Interaction System. Applications could be further rehabilitation tools, games, and experience simulations.

## 6.4. System Marketing and Distribution

To directly work on reaching the ultimate goal of the Haptic Interaction System, information about it and methods of getting it to developers could be greatly helpful. With a capable enough system, methods of marking the device to developers, explaining its function and purpose, could be created. In addition, methods of distributing the plans and or even parts of the system could be worked on. Together this would work towards bring the Haptic Interaction System to the market of developers for innovation.

# 7. Conclusion

The Haptic Interaction System created by this project team has been able to demonstrate the functionalities set forth by its system requirements, to provide haptic interactions with virtual environments. In addition it contributed to further understanding the intricacies of implementing simulated haptic interactions. The system created is capable of bringing access to new kinds of haptic experiences and resources. With further development, this system has the potential to provide a new way to harness creativity to solve problems and help improve society.



# 8. Bibliography


1. Preusche, C., & Hirzinger, G. (2007). Haptics in telerobotics. The Visual Computer, 23(4), 273-284.
2. Grifantini, K. (2009). Open-Source Data Glove | MIT Technology Review.
3. *Haptic Devices*. (2003). 4033 Aurora Ave N. Suite 201 Seattle, WA 98103: Mimic Technologies Inc.
4. Sutherland, I. E. (1964, January). Sketch pad a man-machine graphical communication system. In Proceedings of the SHARE design automation workshop (pp. 6-329). ACM.
5. Sutherland, I. E. (1965). The ultimate display. Multimedia: From Wagner to virtual reality.
6. Zimmerman, T. G., Lanier, J., Blanchard, C., Bryson, S., & Harvill, Y. (1987, May). A hand gesture interface device. In ACM SIGCHI Bulletin (Vol. 18, No. 4, pp. 189-192). ACM.
7. Booton, J. (2015). GoPro buys virtual reality video producer Kolor.
8. Consumer Virtual Reality market worth $5.2bn by 2018 » KZero Worldswide. (2015). from http://www.kzero.co.uk/blog/consumer-virtual-reality-market-worth-13bn-2018/
9. Lerner, M. A., Ayalew, M., Peine, W. J., & Sundaram, C. P. (2010). Does training on a virtual reality robotic simulator improve performance on the da Vinci surgical system? *J Endourol, 24*(3), 467-472. doi: 10.1089/end.2009.0190
10. Homepage | cyberglovesystems.com. (2015). from http://cyberglovesystems.com/
11. Burdea, G. C. (2003). Virtual Reality Technology.
12. Novint - Home. (2015). from http://home.novint.com/
13. Massie, T. H., & Salisbury, J. K. (1994). *The phantom haptic interface: A device for probing virtual objects.* Paper presented at the Proceedings of the ASME winter annual meeting, symposium on haptic interfaces for virtual environment and teleoperator systems.
14. Srinivasan, M. A. (1995). What is haptics. Laboratory for Human and Machine Haptics: The Touch Lab, Massachusetts Institute of Technology.
15. Bowyer, S. A., Davies, B. L., & Rodriguez y Baena, F. (2014). Active constraints/virtual fixtures: A survey. Robotics, IEEE Transactions on, 30(1), 138-157.
16. Newton, C. (2015). The Rise and Fall and Rise of Virtual Reality - Digital Natives.
17. P, S., & Hog, N. (2010). Novel Actuation Methods for High Force Haptics. doi: 10.5772/8702

18. Prange, G. B., Jannink, M. J., Groothuis-Oudshoorn, C. G., Hermens, H. J., & IJzerman, M. J. (2006). Systematic review of the effect of robot-aided therapy on recovery of the hemiparetic arm after stroke. Journal of rehabilitation research and development, 43(2), 171.
19. Seymour, N. E., Gallagher, A. G., Roman, S. A., O'Brien, M. K., Bansal, V. K., Andersen, D. K., & Satava, R. M. (2002). Virtual Reality Training Improves Operating Room Performance: Results of a Randomized, Double-Blinded Study *Ann Surg* (Vol. 236, pp. 458-464).
20. Stroke Facts | cdc.gov. (2015).
21. Sale, P., Lombardi, V., & Franceschini, M. (2012). Hand robotics rehabilitation: feasibility and preliminary results of a robotic treatment in patients with hemiparesis. Stroke research and treatment, 2012.
22. Sutter, P. H., Iatridis, J. C., & Thankor, N. V. (1989). *Response to Reflected -Force Feedback to Fingers in Teleoperations.* Paper presented at the Proc. Of the NASA Conf. On Space Telerobotics, NASA JPL.
23. What is physiotherapy? | The Chartered Society of Physiotherapy. (2015).
24. Delph, M. A., Fischer, S. A., Gauthier, P. W., Luna, C. H. M., Clancy, E. A., & Fischer, G. S. (2013, June). A soft robotic exomusculature glove with integrated sEMG sensing for hand rehabilitation. In Rehabilitation Robotics (ICORR), 2013 IEEE International Conference on (pp. 1-7). IEEE.
25. Geomagic. Haptic Devices from Geomagic. from http://www.geomagic.com/en/products-landing-pages/haptic

# 9. Appendix

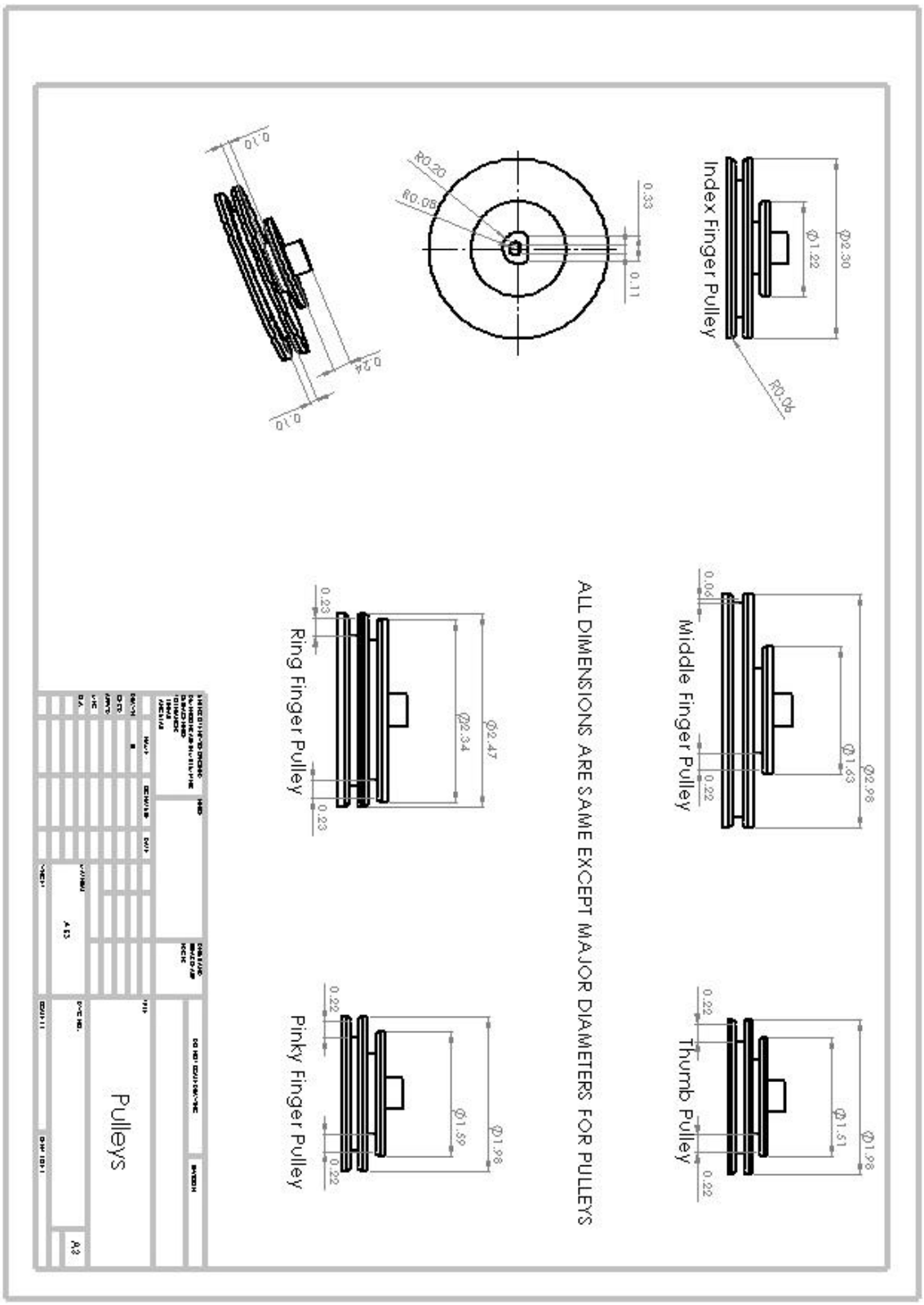
## Appendix A: Bill of Material

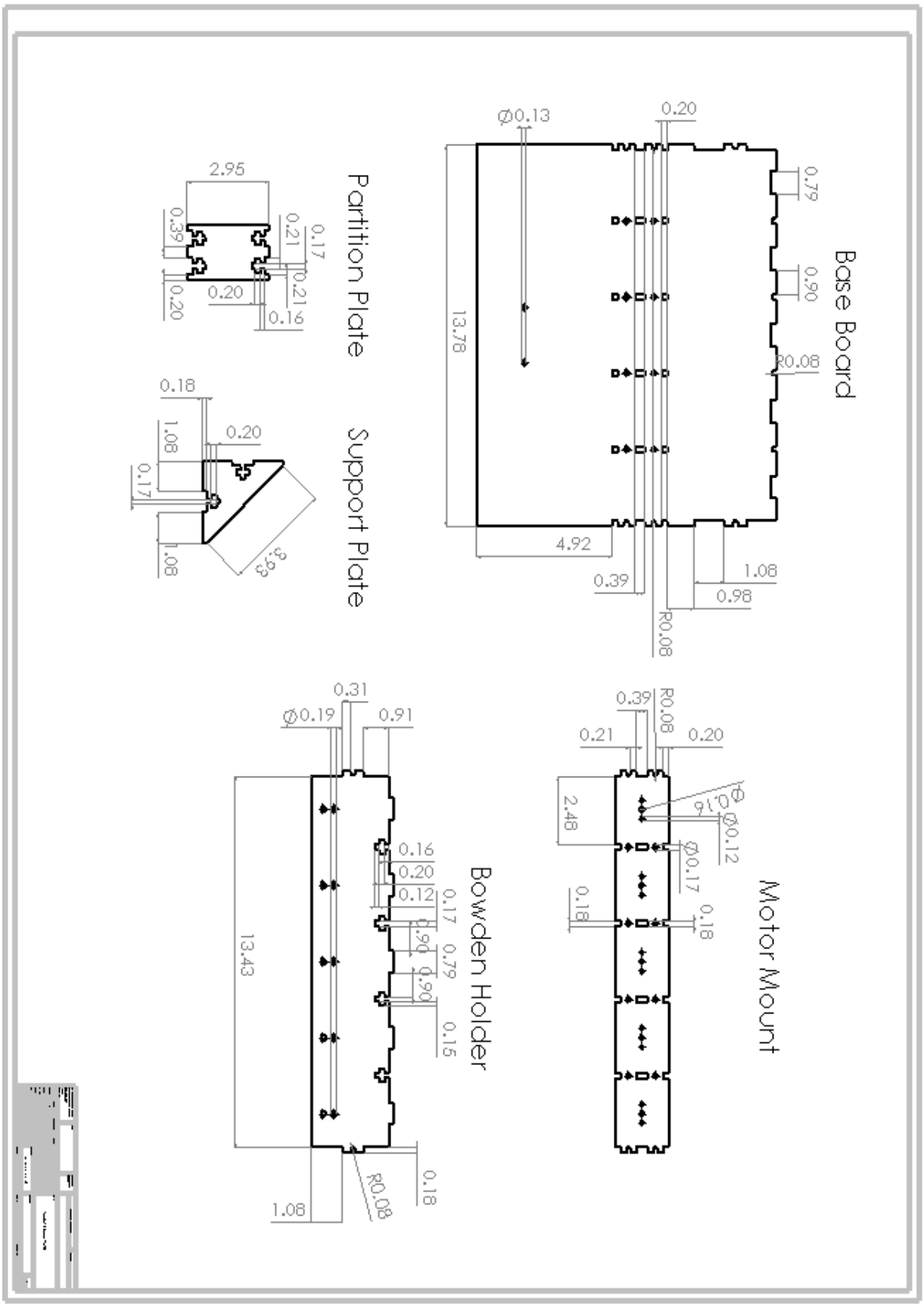
### [Haptic Interaction in Virtual Reality MQP]

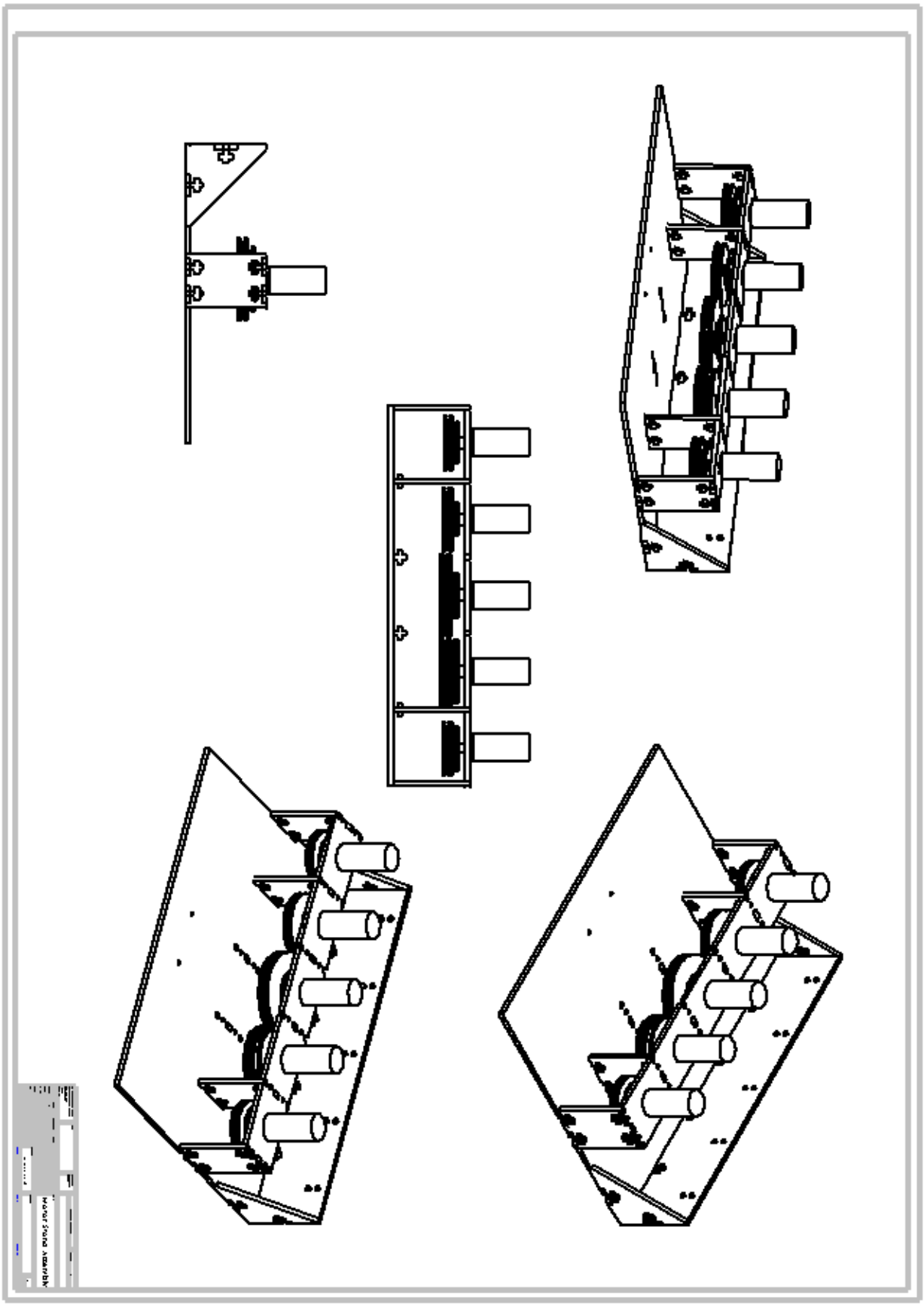
| | |
|---|---|
| Part Count : | 23 |
| Total Cost : | $410.31 |

| Part # | Part Name | Description | Qty | Units | Unit Cost | Cost |
|---|---|---|---|---|---|---|
| 1 | Pololu 34:1 Metal Gearmotor 25Dx52L mm with 48 CPR Encoder | This cylindrical, 2.54" x 0.98" x 0.98" brushed DC gearmotor with a 34.014:1 metal gearbox | 5 | 1 | $ 34.95 | $ 174.75 |
| 2 | Adafruit Motor/Stepper/Servo Shield for Arduino v2 Kit - v2.3 | drive up to 4 DC motors or 2 stepper motors | 2 | 1 | $ 19.95 | $ 39.90 |
| 3 | Leap Motion Controller | The Leap Motion Controller is 3" long. It takes up hardly any space on your desk, but you use the space above it. | 1 | 1 | $ 79.99 | $ 79.99 |
| 4 | 3D Printed Parts | 3D printed pulleys by FlashForge 3D Printer with IC3D 1.75mm ABS Filament | 5 | 1 | $ 3.00 | $ 15.00 |
| 5 | LMP8601, 60-V, Bi-Directional, Low- or High-Side, Voltage Output Current Sensing Amplifier | The part will amplify and filter small differential signals in the presence of high common mode voltages. The input common mode voltage range is –22V to +60V when operating from a single 5V supply. | 5 | 1 | $ 2.70 | $ 13.50 |
| 6 | 4.5mm Extruded Arcrylic Sheet | Used to build Motor Stand by Laser cutting | 2 | 1 | $10.10 | $ 20.20 |
| 7 | FootJoy Men's WeatherSof Golf Glove | Made with a cabretta leather palm patch and thumb, this glove offers a softer feel and added durability. | 1 | 1 | $ 11.99 | $ 11.99 |
| 8 | Polyethylene Tubing | Used to build Bowden Cable Mechanism | 1 | 1 | $ 4.99 | $ 4.99 |
| 9 | Tekpower TP1803D Linear Digital Variable DC Power Supply | This digital DC power supply has maximum output voltage of up to 18 volts and current up to 3 amps. It comes with rotary switches for setting up the voltage and current. | 1 | 1 | $ 49.99 | $ 49.99 |
| | **Total** | | **23** | | | **$ 410.31** |

## Appendix B: Part Drawing




Base Board
Partition Plate
Support Plate
Motor Mount
Bowden Holder
0.20
Ø0.13
0.79
0.90
R0.08
13.78
4.92
0.39
1.08
0.98
2.95
0.39
0.17
0.21
0.21
0.20
0.20
0.16
0.18
1.08
0.20
0.17
1.08
3.53
0.39
R0.08
0.21
0.20
2.48
Ø0.12
Ø0.17
0.18
0.18
0.31
Ø0.19
0.91
0.16
0.20
0.12
0.17
0.79
0.90
0.90
0.15
13.43
0.18
1.08
R0.08



# Appendix C: Code

## *B.1. Force Exertion System Code*

```
//Force Exertion System Program with Simulation System Comunication for the Haptic Interaction System
//By   Matt Rafferty and Jiaqi Ren
//Haptic Interaction in Virtual Reality MQP 2014-2015
//Worcester Polytechnic Institute


//Updated 04/30/2015 , Version 6


//Contains:
//Motor control
//communication with simulation system


//Libray Include:
#include <TimerOne.h> //contains functions for using board Timer 1 for time tracking and interups


#include <Wire.h> //ITC and TWI communication Library
#include <Adafruit_MotorShield.h> //Motor Shield Functions
#include "utility/Adafruit_PWMServoDriver.h" // Motor Shield Utilities


//state flags
boolean handStateSendFlag = false; //flag identifying the need to send the system state to the simulation system
boolean updateControlLoopFlag =false; //flag that signals time to run main loop control and serial read again.


//Serial
int serialRate = 9600; //rate for serial comunications
int serialTimeOut = 1000; //time to wait for serial values when reading serial buffer
//Timer
unsigned int timerPeriod = 100000; //set timer period to 1 thousandth of a second
volatile long int time = 0; //count of timer counts passed since system activation
volatile double runLoopTime=0.0;//Timer that counts until time to run the main control loop again
volatile double loopTimeOffset=0.0;//offset that accounts for extra 0.00006666666 seconds lost from the rounding of 1/60 seconds to 0.016 secconds
long unsigned int count = 0; //main loop counter to run ceratin functions after so many control iterations


//Command Values
//These bytes represent the available command types from the Unity Simulation System
byte updateFingerForce = 1;// B0000001; //49;//1 //B0000001; // assigns  the corresponding message figer the corresponding force
byte updateAllFingerForces = 2; //50;//2 //B00000010; // assignes the corisponding message force to all fingers
byte updateAllFingerForcesSeperate =4;//52;//4 // B00000100;  // assigns the corisponding forces to all fingers
byte sendHandState =3;//51;//3 //B0000011; // trigers the colection of hand curl and force values, and sneds them to the Simulation System
byte frameEnd = 5;//53;//5//B00000101 Informs of the end of a frame in the simulation system, and stops the processing of commands until the next loop
cycle
byte noSerialAvalible = 255;//  Indicates that a serial read function has failed due to the lack of bytes in the serial buffer
byte lastMessage = B0; // holds the value of the last command recived


//Define Moptor Shield Bit adresses
Adafruit_MotorShield AFMSbot(0x61); // Rightmost jumper closed
Adafruit_MotorShield AFMStop(0x60);  // Default address, no jumpers


//Define Motor Objects
//defined by which motor port each motor is connected to
Adafruit_DCMotor *ThumbMotor = AFMSbot.getMotor(1);
Adafruit_DCMotor *IndexMotor = AFMStop.getMotor(2);
Adafruit_DCMotor *MiddleMotor = AFMStop.getMotor(1);
Adafruit_DCMotor *RingMotor = AFMStop.getMotor(3);
```

```
Adafruit_DCMotor *PinkyMotor = AFMStop.getMotor(4);


//Define Sensor Pins


//Analog Input
int f1CurrentPin = A11;
int f2CurrentPin= A12;
int f3CurrentPin= A13;
int f4CurrentPin = A14;
int f5CurrentPin = A15;


//Hand State Variables


//Force Control Values
//setpoints for the force managing current control
//Fingers (f) 1-5 corispond to right hand thumb to pinky finger
//TargetForces are the forces the FES is to exert on the user's fingers
//Types are the type of collision being applied to the finger. 0 is unrestricted motion, 1 is force, and 2 is restrict movement (solid object)
byte f1TargetForce = 0;
byte f1Type = 0;
byte f2TargetForce = 0;
byte f2Type = 0;
byte f3TargetForce = 0;
byte f3Type = 0;
byte f4TargetForce = 0;
byte f4Type = 0;
byte f5TargetForce = 0;
byte f5Type = 0;


//Direction of the Target forces given
//1 signifies oppening the hand and 2 signifies closing the hand
byte f1Direction = 1;
byte f2Direction = 1;
byte f3Direction = 1;
byte f4Direction = 1;
byte f5Direction = 1;


//Sensor Current Read Values
//sensor readings taken from the current sensing circuits
int f1ReadCurrent =0;
int f2ReadCurrent =0;
int f3ReadCurrent =0;
int f4ReadCurrent =0;
int f5ReadCurrent =0;


//Goal Current Value
//calculated from Simulation System Force Commands
int f1GoalCurrent =0;
int f2GoalCurrent =0;
int f3GoalCurrent =0;
int f4GoalCurrent =0;
int f5GoalCurrent =0;

//Flags
//Various flag variables to trigger responses to differnt conditions


//Flag to indicate a new force command has been recived and the Control for that finger
//Must start a new control loop
```

```
boolean f1NewForceFlag = false;
boolean f2NewForceFlag = false;
boolean f3NewForceFlag = false;
boolean f4NewForceFlag = false;
boolean f5NewForceFlag = false;

//Error Values for FInger PID control options
signed int f1IError =0;
signed int f1LastError = 0;
signed int f2IError =0;
signed int f2LastError = 0;
signed int f3IError =0;
signed int f3LastError = 0;
signed int f4IError =0;
signed int f4LastError = 0;
signed int f5IError =0;
signed int f5LastError = 0;
//-------------------------------------------------------------------------------------------------------
//Build needed elements on Force Exertion System activation
void setup() {

 Serial.begin (serialRate);
 Serial.setTimeout(serialTimeOut); //set serial time out rate
 //Initialize Timer
 Timer1.initialize(timerPeriod); //count 1 per timerPeriod //1000000 = 1 second
 Timer1.attachInterrupt(timerISR); //sets function timerISR to run each time the timer increases count


 //Set Up Motors for use
 AFMStop.begin();  // set PWM motor control with the default frequency 1.6KHz

 // Enable All Motors
 ThumbMotor->setSpeed(0);
 IndexMotor->setSpeed(0);
 MiddleMotor->setSpeed(0);
 RingMotor->setSpeed(0);
 PinkyMotor->setSpeed(0);

 //Set all motors to Forward (close position)
 ThumbMotor->run(FORWARD);
 IndexMotor->run(FORWARD);
 MiddleMotor->run(FORWARD);
 RingMotor->run(FORWARD);
 PinkyMotor->run(FORWARD);

 //Set all motors to release mode
 ThumbMotor->run(RELEASE);
 IndexMotor->run(RELEASE);
 MiddleMotor->run(RELEASE);
 RingMotor->run(RELEASE);
 PinkyMotor->run(RELEASE);

 //initialize Current Sense Pin
 f1ReadCurrent = analogRead(f1CurrentPin);
 f2ReadCurrent = analogRead(f2CurrentPin);
 f3ReadCurrent = analogRead(f3CurrentPin);
 f4ReadCurrent = analogRead(f4CurrentPin);
 f5ReadCurrent = analogRead(f5CurrentPin);

 //Serial Empty to remove extra frame end signs from buffer
 Serial.flush();
}


//Core of the Foce Exertion System program, handles serial communications and force and or position control
void loop() {
```

```
//check for force comand message and updates force types and targets
updateForceCommand();


 //if time to run control loop
if(updateControlLoopFlag  == true){



//finger 1 update-------------------------------------------------------------------------
int f1MotorSpeed =0;
//if force on
if(f1Type == 1){
//If new force
  if (f1NewForceFlag == true){
      f1NewForceFlag = false;
    //set motor direction by direction sign
    if (f1Direction == 1){//positive direction
     ThumbMotor->run(BACKWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f1MotorSpeed= constrain(f1GoalCurrent,0,255);
    }
    else if (f1Direction == 2){//negative direction
      IndexMotor->run(FORWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f1MotorSpeed= constrain(map(f1GoalCurrent,-255,0,255,0),0,255);
    }
    IndexMotor->run(f1MotorSpeed);

  }

  //if not a new force
  else{
    //read current
    readFingerCurrent(1);
    signed int roataionSign= directionSign2RotationDirection(f1Direction);// get rotation direction value

    //Get the error in current
    signed int differenceCurrent = (roataionSign*f1GoalCurrent) - f1ReadCurrent;
    signed int alteredCurrent = (roataionSign*f1GoalCurrent) - differenceCurrent*0.2;//Alter current based on error //ID terms for alternate use //+
f1IError*0.08;// + f1LastError*0.05;
    //Update PID errors
    f1IError += differenceCurrent;
    f1LastError = differenceCurrent;

    //If correct control value is forward force
    if (alteredCurrent >= 0){
     ThumbMotor->run(BACKWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f1MotorSpeed= constrain(alteredCurrent,0,255);
    }
    else if (alteredCurrent < 0){//negative direction
      IndexMotor->run(FORWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f1MotorSpeed= constrain(map(alteredCurrent,-255,0,255,0),0,255);
    }
    //Set motor current
    ThumbMotor->setSpeed(f1MotorSpeed);
  }
}
//Motor is set to unrestricted movement mode
else{
  f1MotorSpeed=0;
  ThumbMotor->setSpeed(f1MotorSpeed);
```

```
    }//-------------------------------------------------------------------------------



  //finger 2 update-----------------------------------------------------------------
 int f2MotorSpeed =0;
//if force on
if(f2Type == 1){
//If new force
  if (f2NewForceFlag == true){
     f2NewForceFlag = false;
     //set motor direction by direction sign
   //Serial.print(6);
    if (f2Direction == 1){//positive direction
     IndexMotor->run(BACKWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f2MotorSpeed= constrain(f2GoalCurrent,0,255);
    }
    else if (f1Direction == 2){//negative direction
      IndexMotor->run(FORWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f2MotorSpeed= constrain(map(f2GoalCurrent,-255,0,255,0),0,255);
    }
    IndexMotor->run(f2MotorSpeed);

  }
  //if not a new force
  else{
    //read current
    readFingerCurrent(2);
    signed int roataionSign= directionSign2RotationDirection(f2Direction); // get direction
    //Get the error in current
    signed int differenceCurrent = (roataionSign*f2GoalCurrent) - f2ReadCurrent;
    signed int alteredCurrent =(roataionSign*f2GoalCurrent) - differenceCurrent*0.2; //Alter current based on error //ID terms for alternate use +
f2IError*0.08;// + f2LastError*0.05;
    f2IError += differenceCurrent;
    f2LastError = differenceCurrent;

    //If correct control value is forward force
    if (alteredCurrent >= 0){
     IndexMotor->run(FORWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f2MotorSpeed= constrain(alteredCurrent,0,255);
    }
    else if (alteredCurrent < 0){//negative direction
      IndexMotor->run(BACKWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f2MotorSpeed= constrain(map(alteredCurrent,-255,0,255,0),0,255);
    }
    Serial.println(f2MotorSpeed);
    IndexMotor->setSpeed(f2MotorSpeed);
  }
}
//Motor is set to unrestricted movement mode
else{
  f2MotorSpeed=0;
  IndexMotor->setSpeed(f2MotorSpeed);
    }//-------------------------------------------------------------------------------



       //finger 3 update-----------------------------------------------------------------
 int f3MotorSpeed =0;
//if force on
```

```
 if(f3Type == 1){
 //If new force
  if (f3NewForceFlag == true){
     f3NewForceFlag = false;
     //set motor direction by direction sign
   //Serial.print(6);
    if (f3Direction == 1){
     MiddleMotor->run(BACKWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f3MotorSpeed= constrain(f3GoalCurrent,0,255);
    }
    else if (f3Direction == 2){//negative direction
      MiddleMotor->run(FORWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f3MotorSpeed= constrain(map(f3GoalCurrent,-255,0,255,0),0,255);
    }
    MiddleMotor->run(f3MotorSpeed);

  }
  //if not a new force
  else{


    //read current
    readFingerCurrent(3);
    signed int roataionSign= directionSign2RotationDirection(f3Direction);
    //Get the error in current
    signed int differenceCurrent = (roataionSign*f3GoalCurrent) - f3ReadCurrent;
    signed int alteredCurrent =(roataionSign*f3GoalCurrent) - differenceCurrent*0.2; //Alter current based on error //ID terms for alternate use +
f3IError*0.08;// + f3LastError*0.05;
    f3IError += differenceCurrent;
    f3LastError = differenceCurrent;

    //If correct control value is forward force
    if (alteredCurrent >= 0){
     MiddleMotor->run(BACKWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f3MotorSpeed= constrain(alteredCurrent,0,255);
    }
    else if (alteredCurrent < 0){//negative direction
      MiddleMotor->run(FORWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f3MotorSpeed= constrain(map(alteredCurrent,-255,0,255,0),0,255);
    }
    Serial.println(f3MotorSpeed);
    MiddleMotor->setSpeed(f3MotorSpeed);
  }
 }
 //Motor is set to unrestricted movement mode
 else{
  f3MotorSpeed=0;
  MiddleMotor->setSpeed(f3MotorSpeed);
    }//--------------------------------------------------------------------------------



       //finger 4 update-------------------------------------------------------------------
  int f4MotorSpeed =0;
 //if force on
 if(f4Type == 1){
 //If new force
  if (f4NewForceFlag == true){
     f4NewForceFlag = false;
     //set motor direction by direction sign
   //Serial.print(6);
    if (f4Direction == 1){
```

```
      RingMotor->run(BACKWARD);
       //set motor speed to run at magnitude of flGoalCurrent
       f4MotorSpeed= constrain(f4GoalCurrent,0,255);
      }
      else if (f4Direction == 2){//negative direction
       RingMotor->run(FORWARD);
       //set motor speed to run at magnitude of flGoalCurrent
       f4MotorSpeed= constrain(map(f4GoalCurrent,-255,0,255,0),0,255);
      }
      RingMotor->run(f4MotorSpeed);

    }
    //if not a new force
    else{


      //read current
      readFingerCurrent(4);
      signed int roataionSign= directionSign2RotationDirection(f4Direction);
      //Get the error in current
      signed int differenceCurrent = (roataionSign*f4GoalCurrent) - f4ReadCurrent;
      signed int alteredCurrent = (roataionSign*f4GoalCurrent) - differenceCurrent*0.2; //Alter current based on error //ID terms for alternate use +
f4IError*0.08;// + f4LastError*0.05;
      f4IError += differenceCurrent;
      f4LastError = differenceCurrent;

      //If correct control value is forward force
      if (alteredCurrent >= 0){
       RingMotor->run(BACKWARD);
        //set motor speed to run at magnitude of flGoalCurrent
        f4MotorSpeed= constrain(alteredCurrent,0,255);
      }
      else if (alteredCurrent < 0){//negative direction
        RingMotor->run(FORWARD);
        //set motor speed to run at magnitude of flGoalCurrent
        f4MotorSpeed= constrain(map(alteredCurrent,-255,0,255,0),0,255);
      }
      Serial.println(f4MotorSpeed);
      RingMotor->setSpeed(f4MotorSpeed);
    }
   }
   //Motor is set to unrestricted movement mode
   else{
    f4MotorSpeed=0;
    RingMotor->setSpeed(f4MotorSpeed);
     }//-------------------------------------------------------------------------------




         //finger 5 update-----------------------------------------------------------------
   int f5MotorSpeed =0;
  //if force on
  if(f5Type == 1){
  //If new force
    if (f5NewForceFlag == true){
       f5NewForceFlag = false;
       //set motor direction by direction sign
     //Serial.print(6);
      if (f5Direction == 1){
       PinkyMotor->run(BACKWARD);
        //set motor speed to run at magnitude of flGoalCurrent
        f5MotorSpeed= constrain(f5GoalCurrent,0,255);
      }
```

```
    else if (f5Direction == 2){//negative direction
      PinkyMotor->run(FORWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f5MotorSpeed= constrain(map(f5GoalCurrent,-255,0,255,0),0,255);
    }
    PinkyMotor->run(f5MotorSpeed);

   }
   //if not a new force
   else{


    //read current
    readFingerCurrent(5);
    signed int roataionSign= directionSign2RotationDirection(f5Direction);
    //Get the error in current
    signed int differenceCurrent = (roataionSign*f5GoalCurrent) - f5ReadCurrent;
    signed int alteredCurrent = (roataionSign*f5GoalCurrent) - differenceCurrent*0.2;//Alter current based on error //ID terms for alternate use +
f5IError*0.08;// + f5LastError*0.05;
    f5IError += differenceCurrent;
    f5LastError = differenceCurrent;

    //If correct control value is forward force
    if (alteredCurrent >= 0){
     PinkyMotor->run(BACKWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f5MotorSpeed= constrain(alteredCurrent,0,255);
    }
    else if (alteredCurrent < 0){//negative direction
      PinkyMotor->run(FORWARD);
      //set motor speed to run at magnitude of flGoalCurrent
      f5MotorSpeed= constrain(map(alteredCurrent,-255,0,255,0),0,255);
    }
    Serial.println(f5MotorSpeed);
    PinkyMotor->setSpeed(f5MotorSpeed);
   }
  }
  //Motor is set to unrestricted movement mode
  else{
    f5MotorSpeed=0;
    PinkyMotor->setSpeed(f5MotorSpeed);
     }//-------------------------------------------------------------------------------------


    }

  //check for a send state update message and sends message if needed
  checkForStateRequest();
} // end loop




//Functions-----------------------------------------------------------------------------------------------------

//-----------------------------------------------
//Time interupt to update Time passed count
void timerISR(){
  time++;
  runLoopTime++;

  //trigger flag to send hand state data every second
```

```
 if (time%100 == 0){
  handStateSendFlag = true;
 }
 //check if time, accounting for lost time due to 1/60th truncation, adds to time to update main loop again
 if( (runLoopTime+loopTimeOffset)/16.0 >=1.0){
  loopTimeOffset+= runLoopTime-16.0;//capture lost time due to trucation of 1/60
  updateControlLoopFlag  = true;
  runLoopTime=0.0;
 }
}




//--------------------------------------------------------------------------------
//function for read force commands
void updateForceCommand(){

 while( (Serial.available() > 0) && (Serial.peek() != frameEnd) ){ //runs message reading while serial buffer has messages and it has found a frame end sign

   byte commandType = Serial.peek();//view first byte to identify type of command

   if(commandType != noSerialAvalible){// check that command is not the no serial avalible byte
    lastMessage = commandType;
    }


  if (commandType == updateFingerForce){ //Updates a single finger's target force and type
  //message format: {command sign, finger number, force type, force target}
  byte message[4];
  Serial.readBytesUntil('\n',message,4); //get message

   //identify finger to be updated and update its target force and type
   if (message[1] == 1){ //update finger 1
    f1NewForceFlag = true;
    f1Type = message[2];
    f1TargetForce = message[3];
    f1Direction = message[4];
    //Convert Target Forces to Goal currents
    f1GoalCurrent = f1TargetForce;
   }

   else if ((message[1] == 2)){//update finger 2
    Serial.println(8);
    Serial.println(message[0]);
    Serial.println(message[1]);
    Serial.println(message[2]);
    Serial.println(message[3]);

    f2NewForceFlag = true;
    f2Type = message[2];
    f2TargetForce = message[3];
    f2Direction = message[4];
    //Convert Target Forces to Goal currents
    f2GoalCurrent = f2TargetForce;
   }



   else if (message[1] == 3){ //update finger 3


    f3NewForceFlag = true;
    f3Type = message[2];
    f3TargetForce = message[3];
    f3Direction = message[4];
```

```
  //Convert Target Forces to Goal currents
  f3GoalCurrent = f3TargetForce;


 }


 else if (message[1] == 4){ //update finger 4


  f2NewForceFlag = true;
  f4Type = message[2];
  f4TargetForce = message[3];
  f4Direction = message[4];
  //Convert Target Forces to Goal currents
  f4GoalCurrent = f4TargetForce;

 }

 else if (message[1] == 5){ //update finger 5


  f5NewForceFlag = true;
  f5Type = message[2];
  f5TargetForce = message[3];
  f5Direction = message[4];
   //Convert Target Forces to Goal currents
   f5GoalCurrent = f5TargetForce;
 }

}

else if(commandType == updateAllFingerForces){//if the command is to update  the same force target and type to all fingers and ends updating process
 //message format: {command sign, finger all code, force type, force target,direction}


 byte message[5];
Serial.readBytesUntil('\n',message,5); //get message


 //Update Flags to show new forces have been recieved
 f1NewForceFlag = true;
 f2NewForceFlag = true;
 f3NewForceFlag = true;
 f4NewForceFlag = true;
 f5NewForceFlag = true;

 //assign the same force target and type to all fingers
 f1Type = message[2];
 f1TargetForce = message[3];
 f1Direction = message[4];
 f2Type = message[2];
 f2TargetForce = message[3];
 f2Direction = message[4];
 f3Type = message[2];
 f3TargetForce = message[3];
 f3Direction = message[4];
 f4Type = message[2];
 f4TargetForce = message[3];
 f4Direction = message[4];
 f5Type = message[2];
 f5TargetForce = message[3];
 f5Direction = message[4];

 //Convert Target Forces to Goal currents
 f1GoalCurrent = f1TargetForce;
```

```
   f2GoalCurrent = f2TargetForce;
   f3GoalCurrent = f3TargetForce;
   f4GoalCurrent = f4TargetForce;
   f5GoalCurrent = f5TargetForce;
   return; //end updating process
  }

  else if (commandType == updateAllFingerForcesSeperate){//if the comand is to update all the finger foces with differnt force target and type values
   //message format: {command sign, finger all code, force type1,force type2,force type3,force type4,force type5,
                        //  force target 1,force target 2,force target 3,force target 4,force target 5, force direction 1,force direction 2,force direction 3,force
direction 4,force direction 5}
   byte message[17];
  Serial.readBytesUntil('\n',message,17); //get message
  //Update Flags to show new forces have been recieved
   f1NewForceFlag = true;
   f2NewForceFlag = true;
   f3NewForceFlag = true;
   f4NewForceFlag = true;
   f5NewForceFlag = true;


   f1Type = message[2];
   f1TargetForce = message[7];
   f1Direction = message[12];
   f2Type = message[3];
   f2TargetForce = message[8];
   f2Direction = message[13];
   f3Type = message[4];
   f3TargetForce = message[9];
   f3Direction = message[14];
   f4Type = message[5];
   f4TargetForce = message[10];
   f4Direction = message[15];
   f5Type = message[6];
   f5TargetForce = message[11];
   f5Direction = message[16];

   //Convert Target Forces to Goal currents
   f1GoalCurrent = f1TargetForce;
   f2GoalCurrent = f2TargetForce;
   f3GoalCurrent = f3TargetForce;
   f4GoalCurrent = f4TargetForce;
   f5GoalCurrent = f5TargetForce;
   return;//end updating prcess
  }

  else{
    //value is not a command remove for next command read
    if (commandType != sendHandState){
      Serial.read(); //remove useless byte from serial buffer

    }
  }
 }
return;
}


//-----------------------------------------------
//Checks if it is time to send status update for force targets, and sends them
void checkForStateRequest(){
 byte commandType = Serial.peek(); //get comand type from serial recive buffer without removal
     if(commandType!= 0B11111111){
     lastMessage = commandType;
     }
```

```
 if (commandType == sendHandState){ // update hand state if message from Simulation System dictates
   byte message[1];
   Serial.readBytesUntil('\n',message,1); //get message
   byte commandSign = B0000001;
   byte messageSend[21] =
{commandSign,f1Type,f2Type,f3Type,f4Type,f5Type,f1TargetForce,f2TargetForce,f3TargetForce,f4TargetForce,f5TargetForce,f1Force,f2Force,f3Force,f4Fo
rce,f5Force,f1Direction,f2Direction,f3Direction,f4Direction,f5Direction};
   char i =0;
   for (i =0; i<21; i++){
     Serial.write(messageSend[i]); // sends the message buffer of 21 bytes
   }
   Serial.println();//end message
   handStateSendFlag = false;
 }

 else if(handStateSendFlag == true){ //update hand state if timer for update rate dictates

   byte commandSign = B0000001;
   byte messageSend[21] =
{commandSign,f1Type,f2Type,f3Type,f4Type,f5Type,f1TargetForce,f2TargetForce,f3TargetForce,f4TargetForce,f5TargetForce,f1Force,f2Force,f3Force,f4Fo
rce,f5Force,f1Direction,f2Direction,f3Direction,f4Direction,f5Direction};
   char i =0;
   for (i =0; i<21; i++){
     Serial.print(messageSend[i]); // sends the message buffer of 21 bytes
   }
   Serial.println();//end message
   handStateSendFlag = false;
 }
}


//read current for input current finger and map it to a -256 to 255 range
//Mapping is due to the limitations of the forcesw exertable by the Adafruit motor shields.  They can only output a byte in either direction
//Therfore the extra reslution from the ADC is unused.
void readFingerCurrent(int i){
  if (i== 1){
    f1ReadCurrent = map(analogRead(f1CurrentPin),0,1023,-256,255);
  }

  else if (i== 2){
    f2ReadCurrent = map(analogRead(f2CurrentPin),0,1023,-256,255);
  }

  else if (i== 3){
    f3ReadCurrent = map(analogRead(f3CurrentPin),0,1023,-256,255);
  }

  else if (i== 4){
    f4ReadCurrent = map(analogRead(f4CurrentPin),0,1023,-256,255);
  }


  else if (i== 5){
    f5ReadCurrent = map(analogRead(f5CurrentPin),0,1023,-256,255);
  }
}


//Converts a message direction sign to a positive or negative motor rotation direction
signed int directionSign2RotationDirection(byte motorDirection){
 if(motorDirection == 1){return 1;}
else if(motorDirection== 2){return -1;}
return 0;

}
//Example Print string to test Simulation System message retreval
```

```
void printTest(){
  byte a = B11;
  byte b = B11111;
  byte messageSend[21] = {b,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a,a};

  char i =0;
  for (i =0; i<21; i++){
   Serial.write(messageSend[i]); // sends the message buffer of 21 bytes
  }
  Serial.println();//end message
}
```

*B.2. Communication with Force Exertion System Code*

```
//Simulation System: Communication with Force Exertion System
//By   Matt Rafferty and Jiaqi Ren
//Haptic Interaction in Virtual Reality MQP 2014-2015
//Worcester Polytechnic Institute

//Updated 04/30/2015 , Version 4

//Contains:
//foce sending function
//state reading functions

//Runs on: Background object

//Libraries used:
using UnityEngine;
using System.Collections;
using System.IO.Ports; //Serial Communication Library
using System.Threading;

//communication class
public class commune_FES : MonoBehaviour {
	public static SerialPort sp = new SerialPort("COM9", 9600); //declare serial port at COM used by arduino of FES, at given baud rate
	int timeSinceSerialClear = 0; //tracks time past to update the hand pose and clear the serial buffer again.
	byte[] handState = new byte[18]; //hold the hand State comand byte and the 15 bytes according to finger force type, force, and pose angle
[fingerStateCode,ForceTypres,Forces,Curls]
	int cou = 0;
	bool messageEmptyFlag = false;
	//public string inValue;

	// Use this for initialization
	void Start () {
		OpenConnection(); // open the serial port for communication

	}

	// Update is called once per frame
	void Update () {
		timeSinceSerialClear = timeSinceSerialClear + 1; //increase frames pasted since hand state update


		if (timeSinceSerialClear >= 30){  //Check if it is time to update the Hand State information and clear the serial buffer
			timeSinceSerialClear=0;


			byte[] message = updateFESHandState(); //// get serial message of latest hand state
			handState = message;  //update hand state variable array

			//print recived message to consl
			if(message.Length>0){
				for (int i =0;i<message.Length;i++){
					print(message[i]);
				}
			}


		}

	}


	//opens the serial port connection
	//orignating from SendToRYG Unity serial demo project
	public void OpenConnection()
	{
		if (sp != null) {
```

```
                if (sp.IsOpen) {
                        sp.Close();
                        print("Closing port, because it was already open!");
                }
                else {
                        sp.Open();  // opens the connection
                        sp.ReadTimeout = 40; // sets the timeout value before reporting serial read exception
                        print("Port Opened!");
                        //                      message = "Port Opened!";
                }
        }
        else {
                if (sp.IsOpen){
                        print("Port is already open");
                }
                else {
                        print("Port == null");
                }
        }
}
//Peform Functions when application closes
void OnApplicationQuit() {
        sp.Close(); //close the serial port
}

//Update the hand state data with the latest serial hand state message (at the end of the serial buffer)
public static byte[] updateFESHandState(){
        byte[] message = serialTryReadLine(); //byte array to hold first message
        byte[] message2; // byte array to hold second message
        byte emptyMessage = 0; // empty byte for comparison

        if (message[0] != emptyMessage){ //if a message is recived
                while (true){ //check for more messages until none remaine in serial recive buffer, keeping last known message
                        message2 = serialTryReadLine();

                        if (message2[0] == emptyMessage){
                                //print ("latest Found");
                                break;
                        }
                        else{
                                message = message2;
                                //print ("old update message");
                        }
                }
        }
        if (message.Length > 0) { //if a more current message is avalible
                return message; //return message
        }
        else{
                return new byte[0]; //return empty message to indicate falure
        }
}

//attempts to read the serial buffer, and retuns a 0 byte[] if no data is avalible (resulting from an exception)
public static byte[] serialTryReadLine(){
        try
        {
                byte[] inBytes = new byte[18];
                sp.Read(inBytes,0,18);


                return inBytes;
        }
        catch(System.Exception) // if no message is avalible
        {
                byte[] fail = new byte[1]; //build byte array with 1 0 value
```

```
                        byte failData = 0;
                        fail[0] = failData;
                        print ("grab fail");
                        return fail; //feturn failure indicator
                }
        }

        //sends the given message comand to the Force Exertion System
        public static void commandFES (byte[] message){
                //sp.WriteLine(System.Text.Encoding.Unicode.GetString(message));//convert byte to string for sending
                //print (message.Length);
                sp.Write(message,0, message.Length);
        }

}
```

*B.3. Hard Haptic Body Code*

```
//Simulation System: Hard Haptic Body Component Code
//By   Matt Rafferty and Jiaqi Ren
//Haptic Interaction in Virtual Reality MQP 2014-2015
//Worcester Polytechnic Institute

//Updated 04/30/2015 , Version 5

//Contains:
//three phases of interaction foce generation
//static stiffness to simulate hard objects

//Runs on: Object with ridgid body and collider

using UnityEngine;
using System.Collections;

public class hardHapricBody : MonoBehaviour {
	float  maxForce = 4.0438f;  // max force magnitude capable by the Force Exertion System
	float stiffness = 4.0438f/0.0015875f;//define stiffness such that maximum interaction force (10 Newtons) is exerted at the distance of finger pad compression when pressed on solid object
	public float realism = 1f; //Scales Elastic On collision forces, to incresse effectivness of lightweight hand interactions


	// Use this for initialization
	void Start ()
	{

	}

	//functions----------------------------------
	//oncollision start
	//trigger a foce based on

	void OnCollisionEnter(Collision collisionInfo) {
		float force = 0; //force generated from collision of objects

		//check that collided object is the bone of the physics Leap Motion Proxy of the user's hand and that selfhas a ridgid body
		if ((string.Equals (collisionInfo.gameObject.name, "bone3")) && (gameObject.GetComponent<Rigidbody> () != null)) {

			//Get Masses of finger and self
			float mFinger = collisionInfo.gameObject.rigidbody.mass;
			float mObject = rigidbody.mass;

			//Get Velocities of finger and self
			float vFinger = collisionInfo.relativeVelocity.magnitude;
			float vObject = 0;

			//Define Collision Type
			float collType = 1; //1:collision type is an elastic collision, 0: collision type is fully inelastic

			//calculate collision force based on elastic collision change in momentum
			force = Time.deltaTime * (mFinger * (vFinger - (((collType * mObject * (vObject - vFinger)) + mFinger * vFinger+ mObject * vObject) / (mFinger + mObject))));

			//Expressforce in vector form
			Vector3 forceVector = force * collisionInfo.relativeVelocity.normalized;

			//calculate Component of Force that is parallel to the normal of the user's finger tip pads
			if (realism < 0) {realism = 0;}
			float forcePerpendiculatFingerTip = realism * getFingerPadForce(forceVector,collisionInfo.gameObject);

			byte forceDirection = 0;
			//convert force direction to finger curl direction
			if (Mathf.Sign (forcePerpendiculatFingerTip) == 1){
				forceDirection = 0; //set finger curl to open direction (backwards)
```

```
                                        }
                                        else if (Mathf.Sign (forcePerpendiculatFingerTip) == -1){
                                                forceDirection = 1; //set finger curl to close direction (fowards)
                                        }

                                        byte forceByte = (byte)Mathf.Round(mapValue(forcePerpendiculatFingerTip/Mathf.Sign (forcePerpendiculatFingerTip), 0,
maxForce, 0, 255));
                                        if (forceByte > 254) {
                                                forceByte = 254;
                                        }
                                        else if(forceByte <0){
                                                forceByte = 0;
                                        }
                                        print("force byte");
                                        print (forceByte);

                                        //get finger name
                                        string finger = collisionInfo.gameObject.transform.parent.gameObject.name;

                                        //get finger number
                                        byte fingerNumber = fingerName2Number(finger);

                                        //build and send force value to arduino as an update finger force command
                                        byte[] FESCommand = {1,fingerNumber,1,forceByte,forceDirection};
                                        commune_FES.commandFES(FESCommand);

                        }
                        //collided object is not a Haptic Object
                        else{
                                        force = 0;
                        }
        }


        //oncollosion still
        //trigger PD controller, with p from defined constant and e possibly from body wall intrusion
        void OnCollisionStay (Collision collisionInfo)
        {
                        float force = 0; //force generated from collision of objects
                        if ( ( (string.Equals (collisionInfo.gameObject.name, "bone3")) || (string.Equals (collisionInfo.gameObject.name, "collideBall") ) ) &&
(gameObject.GetComponent<Rigidbody> () != null)) {//Get Averaqe vector point of collision points

                                        //Build an average vector from all interaction's collision points
                                        Vector3 avgCollisionPoint = new Vector3 (0, 0, 0);
                                        foreach (ContactPoint contact in collisionInfo.contacts) {
                                                avgCollisionPoint += contact.point;
                                        }
                                        avgCollisionPoint = avgCollisionPoint / collisionInfo.contacts.Length;

                                        //Get Radius of Collider Body attched to Finger Tip of User Proxy
                                        float fingerTipProxyRadius = 0f;
                                        SphereCollider fingerTipProxy = collisionInfo.gameObject.GetComponent<SphereCollider> ();
                                        if (fingerTipProxy != null) {
                                                fingerTipProxyRadius = fingerTipProxy.radius;
                                        } else {
                                                print ("User Proxy has no collision objects, therfore no reaction forces are decernable");
                                                return;
                                        }

                                        //Get Global location of collision's finger tip object, using raw Leap Motion Frames
                                        Vector3     currentFingerTipLocation =
leapMotionUtility.getRightHandFingerTipPositionGlobalFrame(fingerName2Number(collisionInfo.gameObject.transform.parent.gameObject.name));

                                        //Define position difference between the average collision point and the center reference transforrmation of self
                                        Vector3 Oc = avgCollisionPoint - gameObject.transform.position;
```

```
                        //Define position difference between self center and finger center
                        Vector3 Of = currentFingerTipLocation - gameObject.transform.position;

                        //Define the offset from finger tip position to colliding edge of finger tip collider proxy
                        Vector3 Frneg = -1*fingerTipProxyRadius* Oc.normalized;

                        //Define magnitude of distance from FInger tip colliding edge and center of self
                        float diffDist = Vector3.Distance(currentFingerTipLocation +(fingerTipProxyRadius*Oc.normalized*-1f),
                                        gameObject.transform.position);
                        //Define vector of distance finger tip colliding edge to outside of self collision boundry
                        Vector3 intrudeVector = (Oc.magnitude - diffDist)*Oc.normalized;

                        //Define force vector based on intrude vector and object stiffness
                        Vector3 forceVector = intrudeVector * stiffness;

                        //convert force vector to magnitude allong parralle finger tip pad normals
                        float forcePerpendiculatFingerTip = getFingerPadForce (forceVector, collisionInfo.gameObject);

                        //convert force direction to finger curl direction
                        byte forceDirection = 0;
                        if (Mathf.Sign (forcePerpendiculatFingerTip) == 1) {
                                forceDirection = 0; //set finger curl to open direction (backwards)
                        } else if (Mathf.Sign (forcePerpendiculatFingerTip) == -1) {
                                forceDirection = 1; //set finger curl to close direction (fowards)
                        }


                        //Convert Finger Pad Force into a integer magnitude of force from 0 to 254
                        byte intrudeForceByte = (byte)Mathf.Round (mapValue (forcePerpendiculatFingerTip, 0, maxForce, 0, 255));
                        if (intrudeForceByte > 254) {
                                intrudeForceByte = 254;
                        } else if (intrudeForceByte < 0) {
                                intrudeForceByte = 0;
                        }

                        //build and send force value to arduino as an update finger force command

                        //get finger name
                        string finger = collisionInfo.gameObject.transform.parent.gameObject.name;

                        //get finger number
                        byte fingerNumber = fingerName2Number (finger);

                        byte[] FESCommand = {1,fingerNumber,1,intrudeForceByte,forceDirection};
                        commune_FES.commandFES (FESCommand);
                } else {
                        force = 0;
                }
        }




        //Oncollision end
        //trigger freespace for finger
        void OnCollisionExit (Collision collisionInfo)
        {
                float force = 0; //force generated from collision of objects
                if ((string.Equals (collisionInfo.gameObject.name, "bone3")) && (gameObject.GetComponent<Rigidbody> () != null)) {//Get Averaqe
vector point of collision points
                        //get finger name
                        string finger = collisionInfo.gameObject.transform.parent.gameObject.name;
```

```
                        //get finger number
                        byte fingerNumber = fingerName2Number (finger);
                        //send set finger to free space command
                        byte[] FESCommand = {1,fingerNumber,0,0,0};
                        commune_FES.commandFES (FESCommand);
                } else {
                        force = 0;
                }
        }

        //get finger number from name
        //Return:
        //1-2-3-4-5 = thumb-index-middle-ring-pinky
        //0 = invalids finger name
        byte fingerName2Number (string fingerName)
        {
                byte number = 0; //default value of 0, error condition retunrs 0 which is ignored by FES
                if (fingerName != null) {
                        //determine which finger is given and return FES number code for that finger
                        if (fingerName.Equals ("thumb")) {
                                number = 1;
                        } else if (fingerName.Equals ("index")) {
                                number = 2;
                        } else if (fingerName.Equals ("middle")) {
                                number = 3;
                        } else if (fingerName.Equals ("ring")) {
                                number = 4;
                        } else if (fingerName.Equals ("pink")) {
                                number = 5;
                        }
                }
                return number;
        }



        //Maps a given value from the first range to the second range
        public static float mapValue (float value, float from1, float to1, float from2, float to2)
        {
                return (value - from1) / (to1 - from1) * (to2 - from2) + from2;
        }

        //Calculate the force parralle to the normal of the user's finger pad
        //Takes force vector and finger tip bone and retrns the magnitude of the vector in the direction of the finger pad
        float getFingerPadForce (Vector3 forceVectorGlobal, GameObject fingerTipBone)
        {
                //Rotate the force vector from the global frame to the frame defined by the finger tip bone's rotation
                Vector3 forceVectorFingerTip = fingerTipBone.transform.rotation * forceVectorGlobal;
                ;
                //Take component in y direction, along finger pad normal
                float padPerpendicularForce = forceVectorFingerTip.y;

                return padPerpendicularForce;

        }
}
```

*B.4. Leap Motion Utility Code*

```
//Simulation System: Leap Motion Utilities
//By   Matt Rafferty and Jiaqi Ren
//Haptic Interaction in Virtual Reality MQP 2014-2015
//Worcester Polytechnic Institute

//Updated 04/30/2015 , Version 2

//Contains:
//tfunctions for locating right hands in the virtual environment, with null hand handling
using UnityEngine;
using System.Collections;
using Leap;

public class leapMotionUtility: MonoBehaviour
{
	//Controller controller;
	static Controller controller = new Controller ();
	static Frame frame;
	// Use this for initialization
	void Start ()
	{

	}

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

		//Finger f2 = h1.Fingers[1];
		//print (f2.TipPosition);
		//Bone p = Bone();
		//Bone b3 = f2.Bone(Leap.Bone());
		//Vector b3Pos=b3.Center;
		//print (b3Pos);



	}







	//FUNctions----------------------------------\\

	//return the globl position of the given right hand finger (index 1 to 5)
	public static Vector3 getRightHandFingerTipPositionGlobalFrame (int fingerNumber)
	{
		int i = 1;// counter to read old hand frames if current is invalid
		int frameLimit = 5;//number of frames to check before retun error
		bool controllerAvailible = controller.IsConnected;
		if (controllerAvailible) {
			frame = controller.Frame ();
		} else {
			while (i <= frameLimit) {
				frame = controller.Frame (i);
				if (frame.IsValid == true) {
					break;
				} else {
					i++;
				}
			}
		}
```

```
		//Ger right Hand
		Hand rightHand = new Hand ();
		foreach (Hand hand in frame.Hands) {
			if (hand.IsRight) {
				rightHand = hand;
				break;
			}
		}

		if (rightHand.IsValid) {
			Finger f = rightHand.Fingers [fingerNumber];
			//convert f tip position to a vector3, and to units of meters from millimeters
			Vector3 fTipPosVec3 = new Vector3 (f.TipPosition.x, f.TipPosition.y, f.TipPosition.z) / 100;
			//print ("frame finger position global");
			//print (fTipPosVec3 + GameObject.FindGameObjectWithTag ("HandController").transform.position);
			return fTipPosVec3 + GameObject.FindGameObjectWithTag ("HandController").transform.position;
		} else {
			//print ("invalid hand");
			return new Vector3 (0, 0, 0);
		}
	}

	public static Vector3 getRightHandPositionGlobalFrame ()
	{
		int i = 1;// counter to read old hand frames if current is invalid
		int frameLimit = 5;//number of frames to check before retun error
		bool controllerAvailible = controller.IsConnected;
		if (controllerAvailible) {
			frame = controller.Frame ();
		} else {
			while (i <= frameLimit) {
				frame = controller.Frame (i);
				if (frame.IsValid == true) {
					break;
				} else {
					i++;
				}
			}
		}

		//Ger right Hand
		Hand rightHand = new Hand ();
		foreach (Hand hand in frame.Hands) {
			if (hand.IsRight) {
				rightHand = hand;
				break;
			}
		}

		if (rightHand.IsValid) {
			Vector3 handPosVec3 = new Vector3 (rightHand.PalmPosition.x, rightHand.PalmPosition.y, rightHand.PalmPosition.z) /
100;
			return handPosVec3 ;//+ GameObject.FindGameObjectWithTag ("HandController").transform.position;
		}else {
			//print ("invalid hand");
			return new Vector3 (0, 0, 0);
		}
	}



	//retrun the int value of the sphere radius of the effective hand radius of curvature
	public static float getRightHandSphereRadiusGlobalFrame ()
	{
		int i = 1;// counter to read old hand frames if current is invalid
```

```
		int frameLimit = 5;//number of frames to check before retun error
		bool controllerAvailible = controller.IsConnected;
		if (controllerAvailible) {
			frame = controller.Frame ();
		} else {
			while (i <= frameLimit) {
				frame = controller.Frame (i);
				if (frame.IsValid == true) {
					break;
				} else {
					i++;
				}
			}
		}

		//Ger right Hand
		Hand rightHand = new Hand ();
		foreach (Hand hand in frame.Hands) {
			if (hand.IsRight) {
				rightHand = hand;
				break;
			}
		}

		if (rightHand.IsValid) {

			return rightHand.SphereRadius;
		}else {
			//print ("invalid hand");
			return 0f;
		}
}

//retrun the int value of the sphere radius of the effective hand radius of curvature
public static float getRightHandGrabStrengthGlobalFrame()
{
		int i = 1;// counter to read old hand frames if current is invalid
		int frameLimit = 5;//number of frames to check before retun error
		bool controllerAvailible = controller.IsConnected;
		if (controllerAvailible) {
			frame = controller.Frame ();
		} else {
			while (i <= frameLimit) {
				frame = controller.Frame (i);
				if (frame.IsValid == true) {
					break;
				} else {
					i++;
				}
			}
		}

		//Ger right Hand
		Hand rightHand = new Hand ();
		foreach (Hand hand in frame.Hands) {
			if (hand.IsRight) {
				rightHand = hand;
				break;
			}
		}

		if (rightHand.IsValid) {

			return rightHand.GrabStrength;
		}else {
			//print ("invalid hand");
```

```
                return 0f;
            }
        }
    }
```

*B.5. End of Frame Code*

```
//Simulation System: End of frame indicator code
		//By   Matt Rafferty and Jiaqi Ren
		//Haptic Interaction in Virtual Reality MQP 2014-2015
		//Worcester Polytechnic Institute

		//Updated 04/30/2015 , Version 1

		//Contains:
		//out put of end of frame sign

using UnityEngine;
using System.Collections;
//Handles the saending of
public class endOfFrame : MonoBehaviour {

		void Start() {
			StartCoroutine(waitEndOfFrame());
		}
		void Update(){
			StartCoroutine(waitEndOfFrame());
			}
		//coroutine to putput a end of message sign once the event end of frame occurs
		IEnumerator waitEndOfFrame() {
			//print ("in frame");
			yield return new WaitForEndOfFrame();
			//print ("frame end");
			byte[] endFrameSign = {5};
			commune_FES.commandFES(endFrameSign);

		}
}
```

*B.6. Soft Haptic Body Code*

```
using UnityEngine;
using System.Collections;

public class softHapticBody : MonoBehaviour
{
	float maxForce = 4.0438f;  // max force magnitude capable by the Force Exertion System
	public float stiffness = 1f;
	public float realism = 1f; //Scales Elastic On collision forces, to incresse effectivness of lightweight hand interactions


	// Use this for initialization
	void Start ()
	{

	}

	//functions----------------------------------
	//oncollision start
	//trigger a foce based on

	void OnCollisionEnter(Collision collisionInfo) {
		/*byte[] command = new byte[4];
		command [0] = 1;
		command [1] = 4;
		command [2] = 50;
		command [3] = 1;
		commune_FES.commandFES(command);*/
		float force = 0; //force generated from collision of objects

		//check that collided object is the bone of the physics Leap Motion Proxy of the user's hand and that selfhas a ridgid body
		if ((string.Equals (collisionInfo.gameObject.name, "bone3")) && (gameObject.GetComponent<Rigidbody> () != null)) {

			//Get Masses of finger and self
			float mFinger = collisionInfo.gameObject.rigidbody.mass;
			float mObject = rigidbody.mass;

			//Get Velocities of finger and self
			float vFinger = collisionInfo.relativeVelocity.magnitude;
			float vObject = 0;

			//Define Collision Type
			float collType = 1; //1:collision type is an elastic collision, 0: collision type is fully inelastic

			//calculate collision force based on elastic collision change in momentum
			force = Time.deltaTime * (mFinger * (vFinger - (((collType * mObject * (vObject - vFinger)) + mFinger * vFinger+ mObject
* vObject) / (mFinger + mObject))));

			//Expressforce in vector form
			Vector3 forceVector = force * collisionInfo.relativeVelocity.normalized;

			//calculate Component of Force that is parallel to the normal of the user's finger tip pads
			if (realism < 0) {realism = 0;}
			float forcePerpendiculatFingerTip = realism * getFingerPadForce(forceVector,collisionInfo.gameObject);

			byte forceDirection = 0;
			//convert force direction to finger curl direction
			if (Mathf.Sign (forcePerpendiculatFingerTip) == 1){
				forceDirection = 0; //set finger curl to open direction (backwards)
			}
			else if (Mathf.Sign (forcePerpendiculatFingerTip) == -1){
				forceDirection = 1; //set finger curl to close direction (fowards)
			}

			byte forceByte = (byte)Mathf.Round(mapValue(forcePerpendiculatFingerTip/Mathf.Sign (forcePerpendiculatFingerTip), 0,
maxForce, 0, 255));
```

```
                        if (forceByte > 254) {
                                forceByte = 254;
                        }
                        else if(forceByte <0){
                                forceByte = 0;
                        }
                        Debug.Log("FORCE BYTE IMPACT: " + forceByte);

                        //get finger name
                        string finger = collisionInfo.gameObject.transform.parent.gameObject.name;

                        //get finger number
                        byte fingerNumber = fingerName2Number(finger);

                        //build and send force value to arduino as an update finger force command
                        byte[] FESCommand = {1,fingerNumber,1,forceByte,forceDirection};
                        commune_FES.commandFES(FESCommand);

                }
                //collided object is not a Haptic Object
                else{
                        force = 0;
                }
        }


        //oncollosion still
        //trigger PD controller, with p from defined constant and e possibly from body wall intrusion
        void OnCollisionStay (Collision collisionInfo)
        {
                float force = 0; //force generated from collision of objects
                if ( ( (string.Equals (collisionInfo.gameObject.name, "bone3")) || (string.Equals (collisionInfo.gameObject.name, "collideBall") ) ) &&
(gameObject.GetComponent<Rigidbody> () != null)) {//Get Averaqe vector point of collision points

                        //Build an average vector from all interaction's collision points
                        Vector3 avgCollisionPoint = new Vector3 (0, 0, 0);
                        foreach (ContactPoint contact in collisionInfo.contacts) {
                                avgCollisionPoint += contact.point;
                        }
                        avgCollisionPoint = avgCollisionPoint / collisionInfo.contacts.Length;

                        //Get Radius of Collider Body attched to Finger Tip of User Proxy
                        float fingerTipProxyRadius = 0f;
                        SphereCollider fingerTipProxy = collisionInfo.gameObject.GetComponent<SphereCollider> ();
                        if (fingerTipProxy != null) {
                                fingerTipProxyRadius = fingerTipProxy.radius;
                        } else {
                                print ("User Proxy has no collision objects, therfore no reaction forces are decernable");
                                return;
                        }

                        //Get Global location of collision's finger tip object, using raw Leap Motion Frames
                        Vector3     currentFingerTipLocation =
leapMotionUtility.getRightHandFingerTipPositionGlobalFrame(fingerName2Number(collisionInfo.gameObject.transform.parent.gameObject.name));

                        //Define position difference between the average collision point and the center reference transforrmation of self
                        Vector3 Oc = avgCollisionPoint - gameObject.transform.position;

                        //Define position difference between self center and finger center
                        Vector3 Of = currentFingerTipLocation - gameObject.transform.position;

                        //Define the offset from finger tip position to colliding edge of finger tip collider proxy
                        Vector3 Frneg = -1*fingerTipProxyRadius* Oc.normalized;

                        //Define magnitude of distance from FInger tip colliding edge and center of self
                        float diffDist = Vector3.Distance(currentFingerTipLocation +(fingerTipProxyRadius*Oc.normalized*-1f),
```

```
                                        gameObject.transform.position);
                        //Define vector of distance finger tip colliding edge to outside of self collision boundry
                        Vector3 intrudeVector = (Oc.magnitude - diffDist)*Oc.normalized;

                        //Define force vector based on intrude vector and object stiffness
                        Vector3 forceVector = intrudeVector * stiffness;

                        //convert force vector to magnitude allong parralle finger tip pad normals
                        float forcePerpendiculatFingerTip = getFingerPadForce (forceVector, collisionInfo.gameObject);

                        //convert force direction to finger curl direction
                        byte forceDirection = 0;
                        if (Mathf.Sign (forcePerpendiculatFingerTip) == 1) {
                                forceDirection = 0; //set finger curl to open direction (backwards)
                        } else if (Mathf.Sign (forcePerpendiculatFingerTip) == -1) {
                                forceDirection = 1; //set finger curl to close direction (fowards)
                        }


                        //Convert Finger Pad Force into a integer magnitude of force from 0 to 254
                        byte intrudeForceByte = (byte)Mathf.Round (mapValue (forcePerpendiculatFingerTip, 0, maxForce, 0, 255));
                        if (intrudeForceByte > 254) {
                                intrudeForceByte = 254;
                        } else if (intrudeForceByte < 0) {
                                intrudeForceByte = 0;
                        }
                        Debug.Log ("intrude Force Byte: " + intrudeForceByte);
                        //build and send force value to arduino as an update finger force command

                        //get finger name
                        string finger = collisionInfo.gameObject.transform.parent.gameObject.name;

                        //get finger number
                        byte fingerNumber = fingerName2Number (finger);

                        byte[] FESCommand = {1,fingerNumber,1,intrudeForceByte,forceDirection};
                        commune_FES.commandFES (FESCommand);
                } else {
                        force = 0;
                }
        }

        //Oncollision end
        //trigger freespace for finger
        void OnCollisionExit (Collision collisionInfo)
        {
                float force = 0; //force generated from collision of objects
                if ((string.Equals (collisionInfo.gameObject.name, "bone3")) && (gameObject.GetComponent<Rigidbody> () != null)) {//Get Averaqe
vector point of collision points
                        //get finger name
                        string finger = collisionInfo.gameObject.transform.parent.gameObject.name;
                        //get finger number
                        byte fingerNumber = fingerName2Number (finger);
                        //send set finger to free space command
                        byte[] FESCommand = {1,fingerNumber,0,0,0};
                        commune_FES.commandFES (FESCommand);
                } else {
                        force = 0;
                }
        }

        //get finger number from name
        //Return:
        //1-2-3-4-5 = thumb-index-middle-ring-pinky
        //0 = invalids finger name
        byte fingerName2Number (string fingerName)
```

```
	{
		byte number = 0; //default value of 0, error condition retunrs 0 which is ignored by FES
		if (fingerName != null) {
			//determine which finger is given and return FES number code for that finger
			if (fingerName.Equals ("thumb")) {
				number = 1;
			} else if (fingerName.Equals ("index")) {
				number = 2;
			} else if (fingerName.Equals ("middle")) {
				number = 3;
			} else if (fingerName.Equals ("ring")) {
				number = 4;
			} else if (fingerName.Equals ("pink")) {
				number = 5;
			}
		}
		return number;
	}



	//Maps a given value from the first range to the second range
	public static float mapValue (float value, float from1, float to1, float from2, float to2)
	{
		return (value - from1) / (to1 - from1) * (to2 - from2) + from2;
	}

	//Calculate the force parralle to the normal of the user's finger pad
	//Takes force vector and finger tip bone and retrns the magnitude of the vector in the direction of the finger pad
	float getFingerPadForce (Vector3 forceVectorGlobal, GameObject fingerTipBone)
	{
		//Rotate the force vector from the global frame to the frame defined by the finger tip bone's rotation
		Vector3 forceVectorFingerTip = fingerTipBone.transform.rotation * forceVectorGlobal;
		;
		//Take component in y direction, along finger pad normal
		float padPerpendicularForce = forceVectorFingerTip.y;

		return padPerpendicularForce;

	}
}
```

*B.7. Force Distort Demo Code*

```
//Simulation System: Ball Grasp Demo Code
//By   Matt Rafferty and Jiaqi Ren
//Haptic Interaction in Virtual Reality MQP 2014-2015
//Worcester Polytechnic Institute

//Updated 04/30/2015 , Version 2

//Contains:
//stress ball size and loction setting
//force output

//Runs on: Background object, in scene with stress ball taged object(s)
using UnityEngine;
using System.Collections;


public class forceDistortDemo : MonoBehaviour {
	public static float ballStiffness = 0.5f; // ball stiffnes for force generation.
	// Use this for initialization
	void Start () {

	}

	// Update is called once per frame
	void Update () {
		//get ball object, based on stress ball tag
		GameObject ball = GameObject.FindGameObjectWithTag ("StressBall");
		//get grasp of right hand in system
		float grasp = leapMotionUtility.getRightHandGrabStrengthGlobalFrame ();
		//define ball radius based on grasp
		float ballRadius = (1 - grasp);
		//set ball scale to match radius given by grasp
		ball.transform.localScale = new Vector3(ballRadius,ballRadius,ballRadius);
		//Command force to FES, with an updateAllFingerForces type message
		byte force = (byte)(grasp * ballStiffness);
		byte[] FESCommand = {2,0,1,force,1};
		commune_FES.commandFES(FESCommand);
	}
}
```

Appendix D: User Guide

# Haptic Interaction System Use Guide

By Matt Rafferty and Jiaqi Ren
rmrafferty@wpi.edu, jren@wpi.edu
Haptic Interaction in Virtual Reality MQP 2014-2015
Worcester Polytechnic Institute

This guide contains basic information for setting up and running the Haptic Interaction System.

**Unity Project Installation**

Download Unity 3D Free edition, version 4.6.0f3. Later distributions are expected to be functional with the Haptic Interaction System, however some changes to the first iteration project code and asset installation process may have to be changed.

Create a project, and under Edit> Project Setting> Player, set Api Compatibility Level to .NET 2.0. This allows for the use of .NET 2.0 For serial port communication with the Force Exertion System, on Windows computers.

Download and install Leap Motion SDK version 2.2.5.26752, for Windows. (Available here: https://developer.leapmotion.com/) Download the standard Unity Assets for leap motion. (Available here: https://developer.leapmotion.com/downloads/unity/4.6)

Follow the instructions for using the Unity 4.6 assets for Unity Free. (available here: https://developer.leapmotion.com/getting-started/unity). This includes where to place key files in your Unity project, how to manually import the assets, such as hand models and tools, and how to build Leap Motion applications in Unity.

Build the provided codes into C# scripts, matching their defined class names. Apply them as they instruct, either to a background object, one with no mesh, physics or other components, to have the code run once a frame in the background, or to an object to have haptic interactions with. Import a hand

controller prefab object from the assets.  Then, disable all colliders for the default hand controller script's physics models and add sphere colliders to the fingertip bone objects.

Finally, upload the Arduino provided code to the Arduino Mega of the Force Exertion System, and the Haptic Interaction System will be ready to run!