Integration of Variable Capacitors as Haptic Feedback in a Practical Data Glove Design

Brian Young
Honors Thesis in Computer Science

Abstract

Many haptic feedback systems that are commercially available are extremely expensive and have high performance requirements. During the initial research into the concept of haptic feedback, the application of a capacitor as an electrostatic break seemed a simple, light and clever application. The idea came from a conference proceeding paper called DextrES [2]. It seemed a little too simple and verification of the underlying concept seemed like a wise precaution. At the end of the paper there was a section on improvement ideas for continuing development. Some changes were made for simplicity and to bring the concept one step closer to a real product than DextrES [2]. Another key aspect was to have lower performance requirements and limited proprietary hardware than the commercially available haptic feedback gloves. The primary focus is on a practical application of the design proposed in the DextrES paper using consumer grade materials.

Introduction

Augmented Reality (AR) and Virtual Reality (VR) headsets and haptic feedback gloves are usually thought of as high-end videogame gear. Not only are the headsets and gloves expensive, so are the accessories to connect the glove to the headset and the computers with the necessary performance requirements are too. Haptic feedback gloves are a new addition to AR and VR systems for good reason. There are 27 degrees of freedom in the human hand. The task of making a glove to merely track the movements in reasonable time is difficult. Adding a method to provide the realistic sensation of gripping something that does not exist in the real world means that the entire glove either becomes too cumbersome or is not strong enough.

The common way to provide haptic feedback is by vibration motors and a force to provide resistance against contracting the hand further. There are problems with this approach. The tiny motors cannot supply enough torque to resist the force of the muscles in the hand. The resistance is contradictory to the sensation of how a person grips an object and some cases feels like something is gripping the hand. In contrast, using the concept of a capacitor allows for greater forces in a simple design that is presented in a form complimentary to the motion of a hand and mirrors how the action of gripping functions. In gripping an object, the sensation of holding something can be as simple as feeling the weight of it or closing the hand around until the materials in the simulated object resist with equal force. The method explored by DextrES was an inspired abstraction, using a capacitor to do physical work [2]. The way it worked was simple: when the plates are uncharged, there is no force and the two plates can slide, and when they are charged a large distributed force is exerted and the plates are locked together, preventing the hand from compressing any further. The force is dependent on the driving voltage. One issue

is that nobody makes capacitors like this currently and so a saga of building a capacitor began -- Which, as it turns out, is simple but getting one to work is the real trick.

The most common application for Augmented Reality and Virtual Reality is gaming, but during a Co-Op at Kronos a new application came to mind, a training system. With this new application it became less of a toy and more of a tool. There is something to be said for feeling the simulated controls in the real world.

Background

Due to the integration of custom hardware and custom software; some fundamental circuit theory and schematic convention background is required. Because this is a Computer Science honors thesis, the only assumption here is that the reader has basic electronics knowledge and things taught in logic design. However, for those attempting to recreate the work in this paper, most of the theoretical background and skill set have been omitted mainly because the DextrES source already exists and the Art of Electronics [1] is a gold standard for electrical engineers. A brief overview is given here for the reader.

First, circuit diagrams have a convention of positive side on the top and negative or ground down [1]; this is the consistent convention in the Art of Electronics. The reason is simple, as ground is down. The concept of positive and negative is easily shown on a typical AA battery. There is a side of the battery with a plus sign on it and the other with a negative sign on it. This is the convention that will be followed. There is also a concept of an open circuit and a closed circuit. A closed circuit is a circuit in which current flows from the source to ground. A light switch in the on position is a closed circuit. In contrast an open circuit is one where there is no path from the source to ground [1].

Second, there is a difference between power and voltage. Electrical power is measured in Watts, and one of the equations for deriving it is: $P = I^2R$ where P is power, I is electric current, and R is resistance [1]. Voltage is defined through Ohm's Law as: $V = IR$ where V is voltage, I is electric current and R is resistance [1]. When high voltage is discussed, that does not mean high power, nor should high power imply high voltage. One final note is that a circuit is a path along which electricity can flow. Lighting creates a circuit between a cloud and the ground and behaves the same as an electrical wall socket and a computer power cord.

One of the major concepts in this paper is the functionality of a capacitor. A capacitor in a typical electrical circuit is used to accumulate electrical charge and discharge it rapidly. The rapid discharge is ideal for use in a time critical system such as haptic feedback due to the instant release. There is a second aspect to a capacitor that makes it even better for this purpose: its construction. In the simplest form, a capacitor consists of two metal plates and a special intermediary called a dielectric. In short, all insulators are dielectric at a certain voltage and reduces the charge through a circuit and stores it. The capacitor has a constant called capacitance given in the SI unit Farads, though typical capacitors are in the pico-Farad and micro-Farad range. Capacitance is calculated by three factors, distance between the metal plates, the dielectric film used and the area of overlap between the metal plates this is typically calculated through relative permeability and the permeability in a vacuum [2].

$C = \frac{\epsilon_r \epsilon_0 A}{d}$, here C is capacitance, ϵ_r is the relative permeability, ϵ_0 is the permeability of a vacuum, A is the area of overlap and d is the distance between the two plates [2].

There are several types of capacitors: polarized and non-polarized -- and of those types there are variable capacitors [1]. Variable capacitors change their capacitance by altering one or both the distance between plates and the overlapping area of the plates. The most common type of variable capacitor is a rotary style using an airgap as shown in Figure 1.

Figure 1: Rotary Capacitor

While the capacitor used in the design is a variable capacitor, it is a linear sliding capacitor and is being used to exert a frictional force, not to store a charge. The only recourse was to construct a capacitor for this application. But the theory of how to build a capacitor and how to make it operate in a significant way are two different things. For unpolarized capacitors, it doesn't matter how the capacitor is placed into the circuit, but for a polarized capacitor the connection is critical. A polarized capacitor only charges when connected properly, like inserting a battery properly [1].

The DextrES glove paper [2] was the basis for this project. The construction required deconstructing and reconstructing the glove after each use. Despite the high voltage used in the design, there was little insulation, though the exceedingly low current, while reassuring, does not account for the spike in the capacitor discharging. The capacitor was tested from 250 V to 1500V in 250 volt increments to test the braking force, which from 500V onward ranged from 4 Newtons to 20 Newtons [2]. The haptic testing was done at 200, 400 and 800 volts. Their functional testing was done with feedback with the Electrostatic break, an array of sixteen cameras and motion capture balls on the glove. Tests were conducted with the break and with only the visual feedback. [2]

Improvements and Changes to the DextrES design

There are two parts to the design, hardware and software. The software relies on the hardware. The hardware comes first and then the overview of the software. The full code is presented in Appendix A. While the basic underlying concept of the DextrES [2] glove using the capacitor as haptic feedback is unchanged, a few changes were made for simplicity, safety and cost. In the original design, the Arduino was merely controlling piezo motors and acting as a digital switch for the high voltage supplied to the capacitors across the fingers. Spatial orientation was provided through an array of sixteen cameras and motion capture spheres on the glove, which was a bit too expensive for this project or for any commercial product. Instead, a simple positional sensor is feeding the information to the Arduino directly. An additional simplification that was also a cost and safety enhancement was switching from a separate

dielectric film that was held in by brackets to thin Kapton tape, which has the advantage of being thin with a dielectric insulator and a silicon-based adhesive that has the properties of a semi-conductor. The adhesive prevents the dielectric film from moving, but the semi-conductor properties do not block the dielectric film becoming charged in a substantial manner.

Now for some of the safety considerations. The original glove was using 1500 volts at a tiny current; it is not dangerous to people in terms of getting shocked. For a commercial product, on the other hand, 1500 volts is not going to sell.

The current glove operates at 450 volts. While concerning and worth additional precaution, it is a substantial improvement over 1500 volts [2]. In light of the high voltage, instead of the metal plates being exposed on the hand, the capacitor plates are contained within an insulating wooden box on the back of the hand. Movement of the plates is achieved through dental floss lines, with the wax providing an insulated and strong line to connect the fingers to the plates, but also to retract the plate back through a motor to the current hand position.

In addition to better safety and simplified design, fixes to the exposed parts are easy and the glove does not require reconstruction of the glove for every use with the DexterES [2] design. The position of the hand is calculated by a positional sensor instead of an array of 16 cameras, cutting the cost and data processing significantly.

Due to the changes and the data provided in the DexterES paper confirmation of the capacitor capabilities before constructing the full design was conducted to ensure the theory matched reality.

The build began with constructing a capacitor. In technical terms, the type of capacitor is a polarized variable capacitor. The polarized nature became clear in testing and is a result of the dielectric being polypropylene tape with a silicon adhesive. Silicon is a semi-conductor which is used in transistors, diodes and integrated circuits. The capacitor plates were made from shim stock (Figure 2).

Figure 2: Shim Stock and Kapton Tape

A length of shim stock was cut from the roll, then the length was cut into two lengths, cutting the width in have, like in Figure 3. For best results, cleaning the shim stock with dish soap and a cloth will remove the dirt, grease and oil that may be on the metal.

Figure 3: Cut and cleaned shim stock

The last step is the application of the Kapton tape to one of the cut lengths shown in Figure 3. It is recommended to wrap the Kapton Tape around one end, as shown in Figure 4, to avoid accidental arcing.

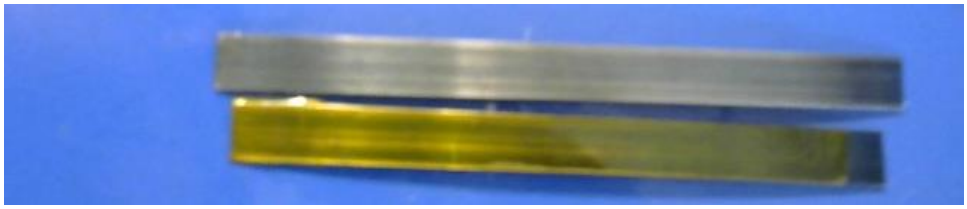


Figure 4: Dialectric applied to Shim Stock

The next step was to learn how the homemade capacitor worked. Since the heart of the design is the haptic feedback from the capacitor and the biggest unknown factor, the first task was to figure out the operating range of the capacitors and construct a holder that would insulate and avoid the capacitor from sliding sideways. The result was a channel cut into ¾" plywood length to hold the capacitor straight, but not add additional friction as shown in Figure 5.

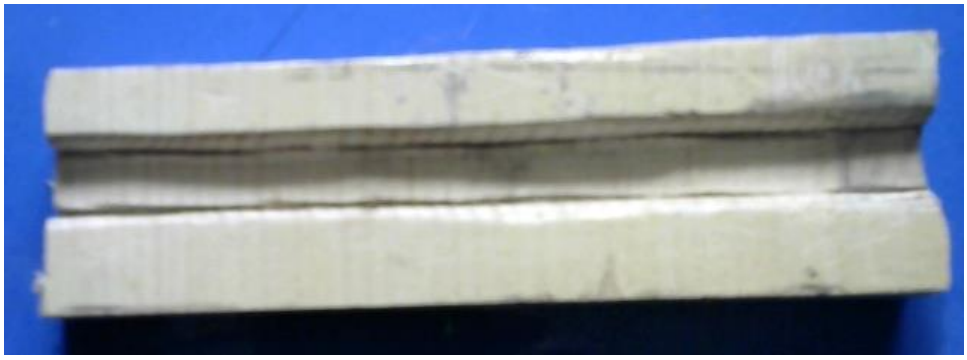


Figure 5: Wooden Channel

During the initial testing, we started with 5 volts and then increased to 25 volts on a DC power supply, but the capacitor didn't provide a significant force in a reasonable timeframe. High voltage power supplies were prohibitively expensive, so an attempt to convert AC from the wall began and with assistance from Albert Bradley, a retired Woods Hole Oceanographic Engineer. That avenue was quickly abandoned when after a blown fuse, some sparks, scorch marks and pops from the design on a breadboard had a double check of the math which was in excess of 200 Amperes. The amperage did not reach that value otherwise the rated fuse in a home fuse box would have blown. Since the circuit breaker box main fuse was not tripped, the actual amperage did not get close to the calculated value. When the necessary effort to even get a 1:1, much less a 2:1, voltage change seemed impractical and for an unknown result, AC was out. Albert Bradley had mentioned he had a high voltage adjustable power supply sitting in a closet [3]. The power supply was picked up and then the construction of what electrical engineers call a "Jesus Stick" which is a length of wood with a nail sticking out and hammered into one end. This was to safely discharge the capacitor during testing, which as an additional precaution testing a static band was worn.

A thought of stacking shorter plates on top of each other was proposed. This resulted in a lower the capacity. The lower capacity results in a lower applied force. Due to the polarized nature of the capacitor there was no noticeable force at any reasonable height for placing on a glove. The stack of small capacitor plates would have made the design more complex with the necessary linear force needing to be transferred to the design through some rotating axis.

When using a power supply for the first time after a long time sitting unused the first step is to check is the drift of the power supply. The drift is how the power supply's displayed value correlates to a measured value on a voltmeter to ensure accurate and operating parameters for the capacitor. It is enough to simply know the drift to perform a mental calibration. The measured range of the voltage supply measured by the Multi-meter was 36 to 450 Volts (See Appendix B, Table 1 and Figure B.1). The result of the high voltage testing was a capacitor that could hold

itself together and lift out of the channel concluding the proof-of-concept testing with a successful result.

There was one problem encountered with the high voltage power supply driving the capacitor. The power switch failed mechanically in that the switch was unable to stay in the on position. After several stop gaps, Albert Bradley came over for a socially distanced dinner on the deck. I mentioned the problem and agreed it was a mechanical failure of the switch; a replacement was delivered by Albert Bradley the next day. While the power supply is not an integrated part of the project construction, the academic opportunity to look at an electrical device's circuit for a repair to include it for the intellectually curious. The actual repair is just replacing a single switch and the photos of the circuit board before putting the power supply back together (see Appendix C).

When mixing High Voltage and people there is a natural fear response. When talking about AC or DC, there is some confusion. There is a voltage value next to the AC/DC tag. But AC and DC stand for Alternating Current and Direct Current, that is measured in Amperes. It is the current, not the voltage that cause electric shocks. Voltage is more like pressure in a water pipe, it pushes the electricity. Current is the flow of energy through a closed circuit. Think of current as water in a pipe. If the valve in the pipe is closed, no water flows. It is the same with electric current, but with some minor differences that rarely apply. What is considered potentially lethal depends on several conditions: the most important is if the Voltage is DC or AC. AC is more dangerous, a wall socket in a home is lethal, but it is not the voltage that kills. It does play a role, but it's the current that does the damage, and the damage is more like getting two signals crossed, one being the current from electricity and the current from neurons that control muscles and pain.

AC is dangerous at levels starting around 120 volts (common US household values) at 0.06 to 0.1 Amperes (a typical household light switch is rated for 10 Amperes). DC on the other hand needs 0.3 to 0.5 Amperes and a voltage of at least 500 Volts [6]. These values assume dry unbroken skin. Skin is an insulator, so bypassing the skin or with wet skin lower current can be lethal. The threshold for injury, muscle contractions and pain varies widely [7]. Since the maximum voltage from the High voltage power supply is 450 volts and is DC, the circuit is non-lethal. There is also a correlation with transformers that the higher the voltage, the lower the current. During tests in Current mode, the value was less than 0.05 Amperes and, when the capacitor was charged, the current dropped to zero.

This does not mean this design is perfectly safe, it just cannot kill a user taking reasonable precautions. There is still the potential for a noticeable shock for a short exposure, but nothing more [7]. Intentional testing of long exposure was not conducted and should never be conducted. The best safety precaution is to use common sense, respect the dangers of electricity and don't alter the circuit while there is power connected.

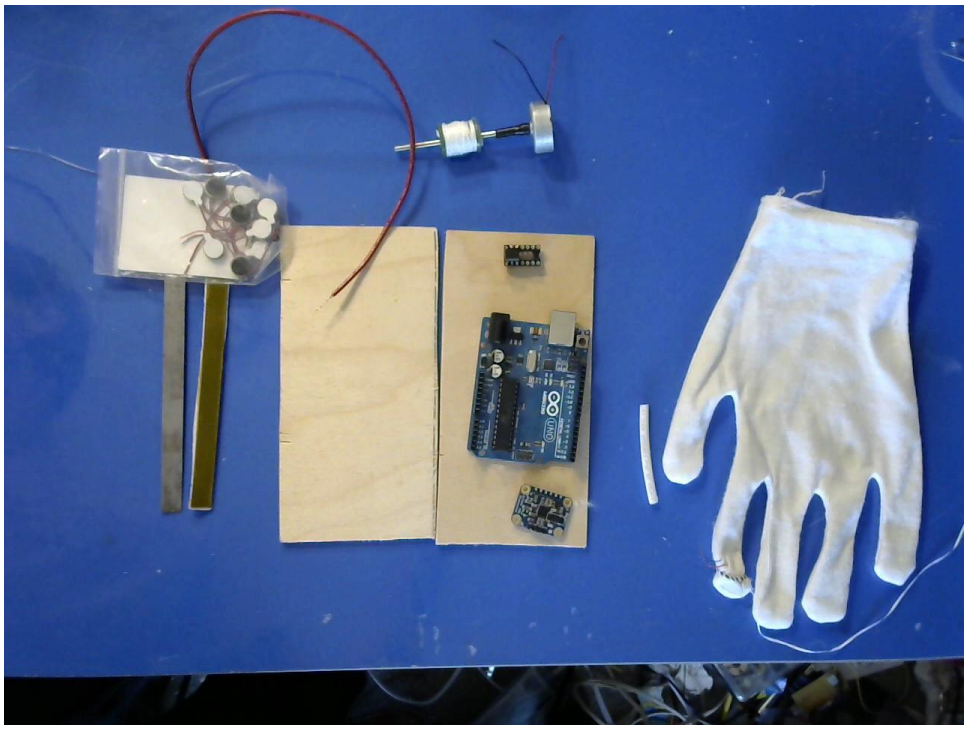


Figure 5: Major Parts prior to Assembly

Shown items (from top left to bottom right): Piezo Motors, Sliding Capacitor, DC gear motor(top center), Plywood, optocoupler, Arduino, Position Sensor,, dental floss, heat shrink tubing, and glove with O-ring sewn in.

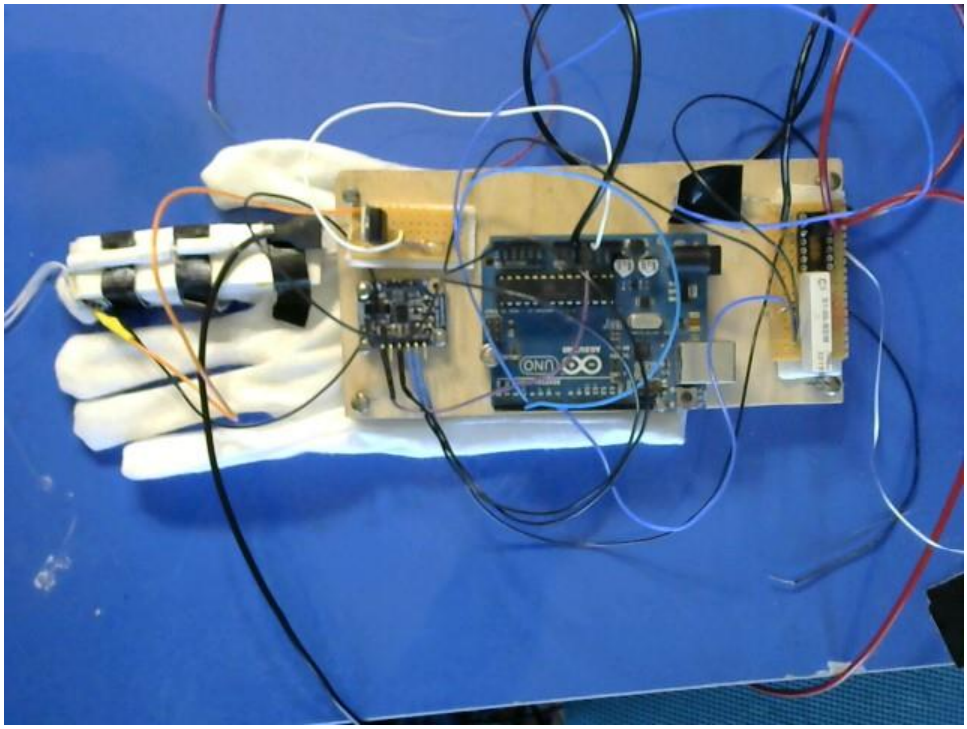


Figure 6: Final Physical Hardware Design

Due to the high voltage in the glove, certain precautions were taken in Figure 6. The box was made of thin plywood, a track to keep the metal strips was made of shrink wrap tubing. and all high voltage wires on the glove were also encased in shrink wrap tubing with Sugru molded over the soldered connection.

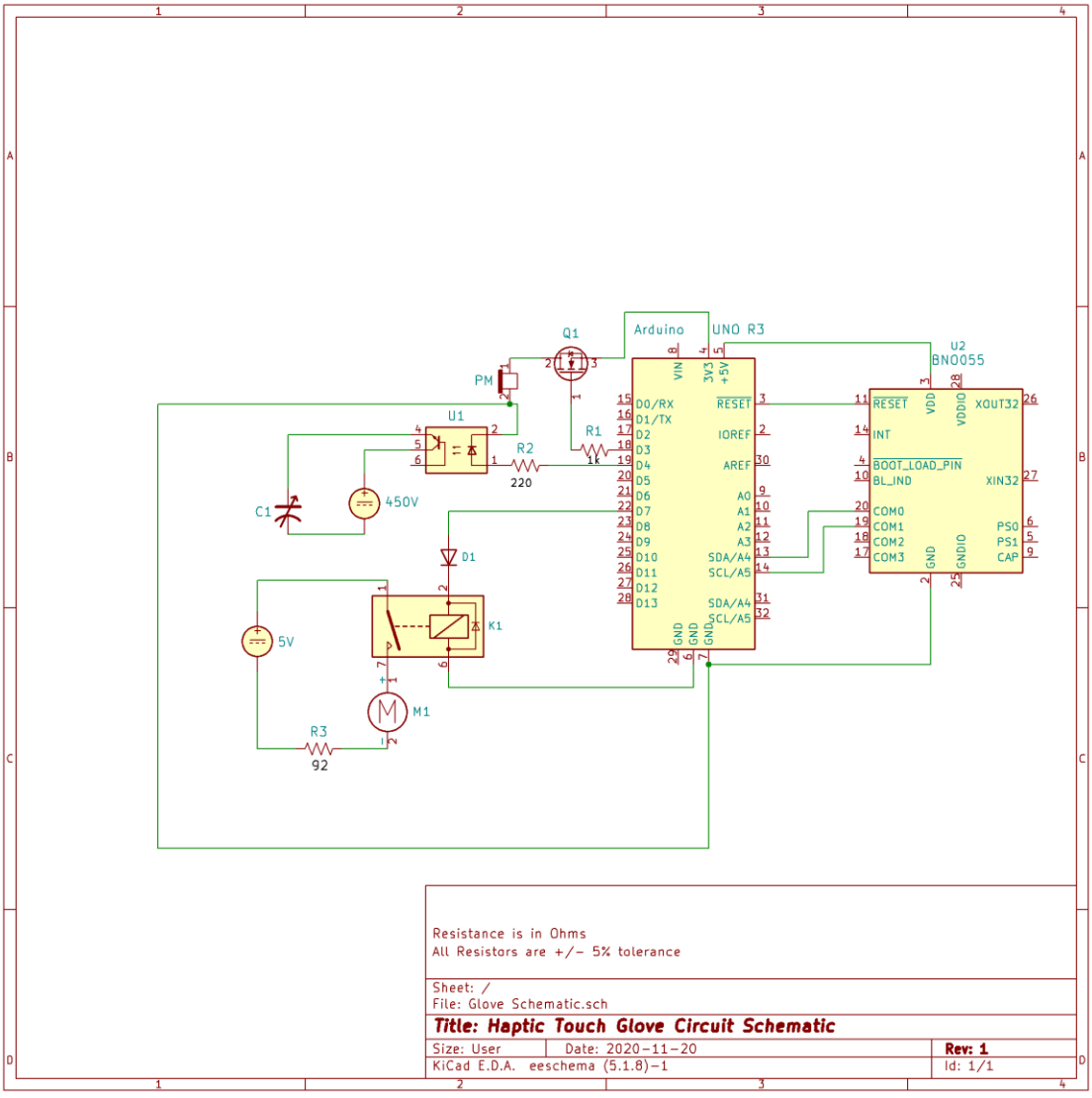


Figure 7: Full Circuit Schematic

A large portion of Figure 7 is protecting the Arduino (Arduino UNO R3 in Figure 7) from damage. The board BNO055 (U2 in Figure 7) is the position sensor. It is designed to interface with an Arduino and needs no protection. Q1 in Figure 7 is called N-Channel Power MOSFET [1]. What MOSFETs do is act like a digital switch, but they are triggered by an input signal that are supposed to be isolated but may become conductive due to oxide or electrostatic discharge. The resistors connected to the power MOSFETs are to protect the Arduino from electric currents

in excess of the output pin maximum current of 20mA [4]. Q1 is protecting the Pin 3 from a current of 50mA on the 3.3V pin [4] driving the vibration motor. A value of 1kΩ (R1 in Figure 7) was selected for a good safety margin. The relay activating the motor (M1 in Figure 7) due to the high current that exceeds the opto-isolator's ratings and the low voltage was unable to overcome the minimum threshold of the opto-isolator. There is a diode protecting against EMF backlash from the inductor in the relay (K1 in Figure 7). The resistor with 92Ω (R3 in Figure 7) is to prevent a short circuit between the motor and the power supply. The power supply for the 5V is a regular Lab DC power supply due to the lag of using the Arduino's 5V supply and the circuit would require the same protection either way, but there is a wider range at which the Lab Power supply can operate. The Voltage supply symbol for DC is a solid line and below it three dashed lines.

The custom variable capacitor (C1 in Figure 8) was the first and most important part to isolate from the Arduino and prevent a short from destroying the opto-isolator. There are two protection issues that have been addressed. The high voltage required to drive the capacitor was exceeded the range of an Arduino's safe operating range of 6 to 20 volts and the operating voltage of 5 volts [4]. An opto-isolator was selected to isolate the high voltage capacitor from the low voltage Arduino while allowing the Arduino to close the circuit by activating the LED in the opto-isolator. The opto-isolator selected was the Toshiba TLP748J Opto-isolator (U1 in Figure 7). The LED operated at 5 volts and the peak off voltage for the photo-sensitive switch is 600V [5]. An opto-isolator operates using an LED and to induce a charge in a photo-sensitive material that acts as a switch to connect a high voltage circuit while providing isolation within the circuit between the Arduino and the custom capacitor (C1 in Figure 7). The 220Ω resistor (R2 in Figure 7) is to prevent a short circuit and prevent the LED from burning out.

Opto-isolators are a way to control a circuit from another circuit with complete isolation. These are ideal for low power circuits where a low voltage circuit is controlling a high voltage circuit. The tricky part is that most integrated circuits have an even number of pins; if the pin isn't connected to anything in the circuit it is marked as NC, those have been omitted. For the opto-isolator, there is no third pin on an LED -- there are only two. The way an opto-isolator works is analogously to a very tiny and very simple solar panel. The LED is turned on by the controller circuit, this induces an electric charge in a receptor, which appears to be a transistor, but a very special kind of transistor. This transistor's silicon chip is exposed to the LED's light. Old metal cap transistors are decent solar cells for what they are intended to be used for. An example is shown in Figure 8.

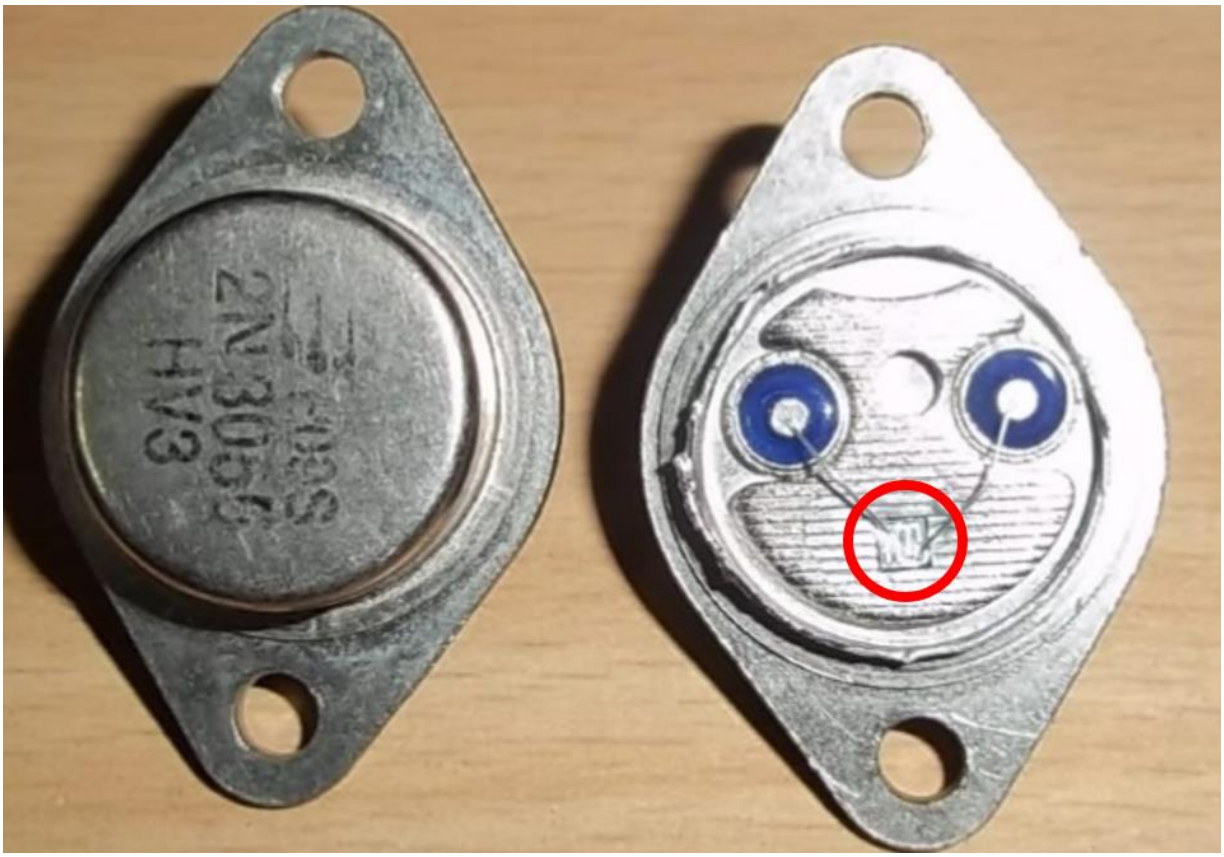


Figure 8: Inside an old metal cap transistor

The encircled square in Figure 8 is the photo-sensitive chip. The same kind of chip is in an opto-isolator, but with a filter to only conduct when the LED is on and closes the circuit.

Developing the Driver Code

Before getting into the driver details, there are some performance parameters and the tools used. The two development tools are the Arduino IDE and Unity. Unity, while bulky, is a mature industry standard game development engine. The Arduino IDE is specially designed to develop on the Arduino Boards. The Arduino communicates with the computer through a serial USB interface. Unity can pick up the Serial communication and write to it. The Arduino IDE can read and write as well. The key is to get two-way communication between the Arduino and the Unity Engine so the interactions in the game are registered on the glove in the form of charging the capacitor and to get the motion of the glove through the orientation sensor. This is merely the high-level overview of the communication, performance constraints and conversions needed to understand the code. See Appendix A for the full code details.

In terms of performance, there are tighter performance constraints in AR and VR games than normal games. This performance is gauged in frames per second. For traditional games anything between 30 and 60 frames per second is considered usable, most modern computer screens operate at around 60 frames per second [8]. For AR and VR, the absolute minimum is 60 frames per second and recommended 90 frames per second. The reason is human comfort: if the FPS is too low, the frames stay even if the head is moving. This results in a disagreement between the human body's spatial orientation and the visual information being presented. The

result of disagreement range from headaches, motion sickness and vertigo [8]. This issue is most profound in VR where the entire world is simulated. AR has a live camera feed with some virtual overlay which lessens the disorientation. While talking in frames per second is fine for a game, most hardware operates on milliseconds or a hertz frequency. Arduino can operate on milliseconds or microseconds. Thus converting a frame per second to milliseconds per frame was needed. $\frac{\text{ms}}{F} = \frac{1000\text{ms}}{FPS}$ where FPS is the target frames per second and there are a thousand milliseconds in one second.

In general, it is better performance and reliability wise to use an integer value. Taking the floor of the value ensures the calculated delay will result in at least the target FPS if not a little higher. The result is an integer. For 90 frames per second, the delay is 11 milliseconds. The communication involves sending and parsing messages. All Arduino boards have a single USB serial port, meaning all data from the glove must pass through the single serial line and then set to the property it is measuring. In this case, the properties being measured are acceleration, orientation, gyroscope and the finger bend sensor. The computer signals the clamping of the capacitor and the triggering of the piezo electric motor. The main issue is with the acceleration, orientation and gyroscope, these all have identical outputs in an x, y, z format. To process the signals correctly a message format was derived with a message prefix, two commas separating the values for x, y and z followed by a message suffix. Acceleration (_A[x],[y],[z]_), orientation (_O[x],[y],[z]_), and gyroscope (_G[x],[y],[z]_). In the position is calculated from the orientation and acceleration on the Arduino as _P[x],[y],[z]_ and sent over serial. In terms of precision the for the sensor data the resolution is limited to three decimal places to avoid sensor drift issues at higher precision that can cause issues in the simulation.

In sending a value over a serial connection, there is something called the Baud rate. In this application a Baud is the number of bits per second that can be processed. This includes some overhead in the form of stop, start and parity bits as well as the processor clock drift. The higher the baud rate, the more information can be sent, but the greater chance of errors due to the speed. Arduinos can send data over serial through certain baud rates. The most common baud rate used is 9600 baud [4], which can send up to 1,200 bytes per second. This is more than enough for the short messages sent through this application.

Processing the inputs in Unity is difficult due to interesting quirks; the biggest one is the coordinate system. To explain the coordinate systems two graphics are needed one for the mathematical and conventional version (9a) and what Unity has according to their UI(9b).

Figure 9a: Normal 3D

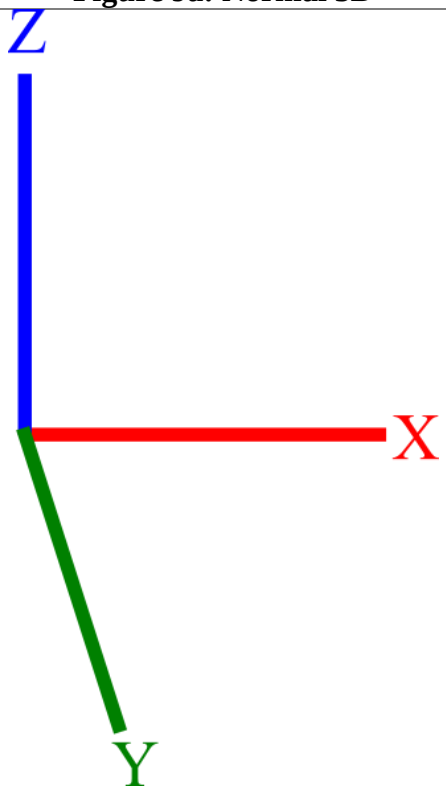
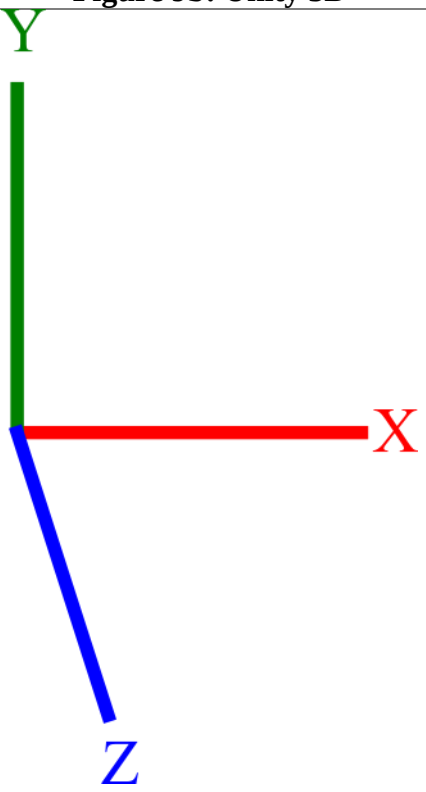


Figure 9b: Unity 3D



There is no good reason these should not be the same, but there is a simple theory on how Unity defined 3D the way it does. Take a 2D game engine with Y being vertical and X being horizontal. Now stick an axis on for depth, call it Z and call it done. The result is a headache for every Unity developer.

The virtual environment in Unity has a camera for the in-game view, a table, and a hand. The hand is the controlled object. The table just something for the haptic feedback trigger. Unity has two systems for handling object collisions a trigger and a collider system for handling object collisions [9]. The two systems are setup differently and provide different functionality. The collider system uses physics and individual collision points of contact to produce a reaction based on physics [9]. The trigger collision is just two objects collided [9]. For this application, two objects colliding is enough, there is no need for any physics, just something to trigger the capacitor. While the two objects start colliding, Unity sends a “1” over the serial connection to the Arduino, and when the objects start stop colliding a “0” is sent over the serial connection to release the capacitor. That signal is processed by the Arduino to activate and deactivate the haptic feedback accordingly.

The communication of the information is sent by string over the serial connection and after some checking the format and values the information is passed to update the orientation. Angles are passed directly, but the position must be calculated as a relative change from the starting positions as the change in coordinates.

Conclusion

While the design requirements of the glove and cost of it have been reduced significantly, more work is needed to bring the idea to a practical application. The current design with the power supply in terms of a battery pack would require 38 typical lead-acid car batteries. The alternative is to make a thinner dielectric film, which is something that requires a lab tech, a lot of money and a clean room. Even more design improvements can be made, such as adding a bend sensor -- either a resistive bend sensor or a piezo-electric bend sensor -- to the design with inverse kinematics of the finger. Though improving this design is not something anyone could do just by reading this paper or understanding the theoretical background, there is an applied physical design to this project that requires hands on experience, tools for diagnostics, soldering and prototyping. There is also a mechanical design component.

Works Cited

[1] P. Horowitz and W. Hil, The Art Of Electronics, Cambrige University Press, 1989.

[2] R. Hinchet, V. Vechev, H. Shea and O. Hilliges, "DextrES: Wearable Haptic Feedback for Grasping in VR via a Thin Form-Factor Electrostatic Brake," in UIST '18: Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology, Berlin, 2018.

[3] A. Bradley, Phone Conversations with Albert Bradley., Falmouth, 2020.

[4] Arduino, "Ardunio Uno Rev3 Tech Specs," 2019. [Online]. Available: https://store.arduino.cc/usa/arduino-uno-rev3. [Accessed 25 January 2019].

[5] Toshiba, Toshiba TLP748J Opto-isolator, Toshiba Electronic Devices and Storage Corporation, 2019.

[6] Merck, The Merck Manual of Medical Information: Home Edition., Merck, 1997.

[7] P. W. N. R. F. Zitzewitz, Merrill Physics, Principles and Problems, Glencoe : McGraw-Hill, 1995.

[8] J. Linietsky and A. Manzur, "VR Primer," 2014-2019. [Online]. Available: https://docs.godotengine.org/en/stable/tutorials/vr/vr_primer.html. [Accessed 10 February 2019].

[9] Unity Game Engine, "Scripting API," 2020. [Online]. Available: https://docs.unity3d.com/ScriptReference/index.html. [Accessed 9 November 2020].

Appendix A: Code

The code for this project is Arduino code and C# code. Arduino code is closest to C++, but it has some other aspects from similar languages. C# code is for Unity.
This code is current as of 12/11/2020 for current code see the author's personal GitHub: https://github.com/FreelanceJavaDev/ArduinoDataGloveHV

Arduino Code:

```
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>

/*
  BNO055 Connections
  ==================
  Connect SCL to analog 5
  Connect SDA to analog 4
  Connect VDD to 3.3-5V DC
  Connect GROUND to common ground
*/

/* Set the delay between fresh samples */
uint16_t BNO055_SAMPLERATE_DELAY_MS = 100; //shortened to 11ms for actual
uint16_t PRINT_DELAY_MS = 500; //testing, shortened to 55 for actual
const uint8_t PRECISION = 3;
uint8_t printCount = 0;
bool activateHaptics = false; //activate capacitor
double dx, dy, dz;


//velocity = accel*dt (dt in seconds)
//position = 0.5*accel*dt^2
const double ACCEL_VEL_TRANSLATION = (double)(BNO055_SAMPLERATE_DELAY_MS) /
1000.0; // dt
const double ACCEL_POS_TRANSLATION = 0.5 * ACCEL_VEL_TRANSLATION *
ACCEL_VEL_TRANSLATION; //0.5*dt^2
const double DEG_2_RAD = 0.01745329251; //trig functions require radians,
BNO055 outputs degrees

// Check I2C device address and correct line below (by default address is
0x29 or 0x28)
//                   id, address
Adafruit_BNO055 bno = Adafruit_BNO055(55, 0x28);


int capacitor_pin = 4; //pin 4
```

```
int piezo_pin = 8; //vibration motor pin signal driver
int motor_pin = 7; //motor activation signal pin

void reelBack() {
 unsigned long t_m = micros();
 digitalWrite(motor_pin, LOW); //ON
 while((micros() - t_m) < 5000); // 5ms
 digitalWrite(motor_pin, HIGH); //OFF
}

void capacitorClamp(){
 digitalWrite(capacitor_pin, HIGH); //ON
}

void capacitorRelease() {
 digitalWrite(capacitor_pin, LOW); // OFF
}

//setup hardware, runs once
void setup() {
 Serial.begin(9600);
 pinMode(capacitor_pin, OUTPUT);
 pinMode(piezo_pin, OUTPUT);
 digitalWrite(piezo_pin, LOW);
 digitalWrite(capacitor_pin, LOW); //OFF
 pinMode(motor_pin, OUTPUT);
 digitalWrite(motor_pin, HIGH);

 while(!Serial);
 /* Initialise the sensor */
 if (!bno.begin())
 {
  /* There was a problem detecting the BNO055 ... check your connections */
  Serial.println("Ooops, no BNO055 detected ... Check your wiring or I2C
ADDR!");
  while (1);
 }

 //bno.setSensorOffsets(calibrationData);
 dx = 0;
 dy = 0;
 dz = 0;
 delay(1000);

}
```

```
//runs repeatedly after setup()
void loop() {

 unsigned long t_0 = micros();
 sensors_event_t orientationData , angVelocityData , linearAccelData;
 //imu::Vector<3> euler = bno.getQuat().toEuler();
 bno.getEvent(&orientationData, Adafruit_BNO055::VECTOR_EULER);
 bno.getEvent(&angVelocityData, Adafruit_BNO055::VECTOR_GYROSCOPE);
 bno.getEvent(&linearAccelData, Adafruit_BNO055::VECTOR_LINEARACCEL);

 //absolute postion determined in Unity
 dx = dx + ACCEL_POS_TRANSLATION * linearAccelData.acceleration.x;
 dy = dy + ACCEL_POS_TRANSLATION * linearAccelData.acceleration.y;
 dz = dz + ACCEL_POS_TRANSLATION * linearAccelData.acceleration.z;

 if(printCount * BNO055_SAMPLERATE_DELAY_MS >= PRINT_DELAY_MS) {
  printSensor(&orientationData, "_O");
  Serial.flush();
  //String orientationPrint = "_O" + String(euler[0], PRECISION)+ "," +
String(euler[1], PRECISION) + "," + String(euler[2], PRECISION) + "_";
  printSensor(&angVelocityData, "_G");
  Serial.flush();
  printSensor(&linearAccelData, "_A");
  Serial.flush();
  String posPrint = "_P" + String(dx, PRECISION) + "," + String(dy,
PRECISION) + "," + String(dz, PRECISION) + "_";
  Serial.println(posPrint);
  Serial.flush();
  printCount = 0;
 }
 else { ++printCount; }

  while ((micros() - t_0) < (BNO055_SAMPLERATE_DELAY_MS * 1000))
 {
  if(Serial.available() < 0) { //somethig to read
   bool rd_value = Serial.parseInt() == 1 ? true : false;
   hapticFeedback(rd_value);
  }
 }

}

//haptic feedback Unity will send 0 for release or 1 for charge
void hapticFeedback(bool unity_sig) {
 if(!activateHaptics && unity_sig) {
  digitalWrite(piezo_pin, HIGH);
```

```
  delay(5); //5ms
  digitalWrite(piezo_pin, LOW);
  activateHaptics = unity_sig;
 }
 else if(activateHaptics && !unity_sig) {
  activateHaptics = unity_sig;
 }
 if(activateHaptics){ capacitorClamp(); }
 else {
   capacitorRelease();
   reelBack(); //activate capactor reel motor
 }
}

//outputs sensor data to string then string is printed, note print and
println can only print one varaible OR one string literal per call.
//Unity may take individual prints as individual messages ensures 1 message
per sensor
void printSensor(sensors_event_t* event, const String prefix) {
 String output = prefix;
  switch(event->type) {
   case SENSOR_TYPE_ACCELEROMETER:
    output += String(event->acceleration.x, PRECISION) + "," + String(event-
>acceleration.y, PRECISION) + "," + String(event->acceleration.z, PRECISION);
     break;
   case SENSOR_TYPE_ORIENTATION:
    output += String(event->orientation.x, PRECISION) + "," + String(event-
>orientation.y, PRECISION) + "," + String(event->orientation.z, PRECISION);
    break;
   case SENSOR_TYPE_GYROSCOPE:
   case SENSOR_TYPE_ROTATION_VECTOR: //gyro & rotation are the same thing
    output += String(event->gyro.x, PRECISION) + "," + String(event->gyro.y,
PRECISION) + "," + String(event->gyro.z, PRECISION);
    break;
   default: return; //something's gone wrong
  }
  output += "_";
  Serial.println(output);
}
```

Unity Code: (3 files)

Arduino Communication Threads:

```
using System;
using System.Collections;
using System.IO;
using System.IO.Ports;
using System.Threading;
using UnityEngine;

public class ArduinoCommThread
{
  private string portName; //COM5
  private int baudRate; //9600
  private int reconnectDelay;
  private int messageQueueSize; //1 for serial
  private SerialPort comPort;
  private const int readTimeout = 11; //8 for 120fps, 11 for 90 fps
  private const int writeTimeout = 11; //8 for 120fps, 11 for 90 fps
  private Queue readQueue, writeQueue;
  private bool stopRequest = false;
  private bool sendStatusMessages = false;
  private bool dropOldestMessage = true;
  //create Communication thread for reading and writing
  public ArduinoCommThread(string portName, int baudRate, int reconnectDelay,
int messageQueueSize) {
    this.portName = portName;
    this.baudRate = baudRate;
    this.reconnectDelay = reconnectDelay;
    this.messageQueueSize = messageQueueSize;
    readQueue = Queue.Synchronized(new Queue());
    writeQueue = Queue.Synchronized(new Queue());
  }

  //Read serial messsages recived from Queue.
  public string ReadSerialMessage() {
    if (readQueue.Count == 0) { return null; }
    return (string)readQueue.Dequeue();
  }

  //Write serial message to queue
  public void WriteSerialMessage(string message) {
    writeQueue.Enqueue(message);
  }

  //makes a stop request, thread locked
  public void RequestStop() {
```

```
    lock(this) { stopRequest = true; }
  }

  //Runs forever, attemps to get connection then calls run once
  public void RunForever() {
    try { //debug Log for unexpected error
       while(!IsStopRequested()) {
        try {
          AttemptConnection();
          while(!IsStopRequested()) { RunOnce(); }
        }
        catch(Exception ioe) {
          Debug.LogWarning("Exception: " + ioe.Message + " in " +
ioe.StackTrace);
          readQueue.Enqueue(ArduinoController.ARDUINO_DEVICE_DISCONNECTED);
          ClosePort();
          Thread.Sleep(reconnectDelay);
        }
      }
      while(writeQueue.Count != 0) {
        string writeMessage = (string)(writeQueue.Dequeue());
        comPort.WriteLine(writeMessage);
      }

    } catch(Exception e) { Debug.LogError("Unknown exception: " + e.Message +
" " + e.StackTrace); }
  }


  //try and open connection
  private void AttemptConnection() {
    if (comPort == null)
    {
      comPort = new SerialPort(portName, baudRate);
      comPort.ReadTimeout = readTimeout;
      comPort.WriteTimeout = writeTimeout;
      comPort.Open();
      if (sendStatusMessages)
      { readQueue.Enqueue(ArduinoController.ARDUINO_DEVICE_CONNECTED); }
    }
  }

  //close connection
  private void ClosePort() {
    if(comPort == null) { return; }
    try { comPort.Close(); } catch (IOException) {}
```

```
    comPort = null;
  }

  //check if stop has been requested
  private bool IsStopRequested() {
    lock(this) { return stopRequest; }
  }

  //check if anything to read or write
  //used inside of RunForever
  private void RunOnce() {
    try {
      if(writeQueue.Count > 0) {
        comPort.WriteLine((string)(writeQueue.Dequeue()));
      }
      string recvMessage = comPort.ReadLine();
      if(recvMessage != null){
        if (readQueue.Count < messageQueueSize)
{ readQueue.Enqueue(recvMessage); }
        else
        {
          object droppedMsg;
          if (dropOldestMessage)
          {
            droppedMsg = readQueue.Dequeue();
            readQueue.Enqueue(recvMessage);
          }
          else { droppedMsg = recvMessage; }
        }
      }
    } catch (TimeoutException) {
      //just means nothing to read
    }
  }
}
```

Arduino Controller Handler:

```
using System.Threading;
using UnityEngine;

public class ArduinoController : MonoBehaviour
{
  [Tooltip("Port name of the Arduino's Serial Port. Format is: COM#")]
  public string portName = "COM5"; //this will differ on different machines
  [Tooltip("Baud rate the serial device is using to send data in bits per
second. See arduino serial for compatible baudrates.")]
  public int baudRate = 9600; //Must Match what Arduino is sending

  [Tooltip("The object that interprets messages.")]
  public GameObject messageListener;
  [Tooltip("Reconnect delay in miliseconds")]
  public int reconnectDelay = 1000;

  [Tooltip("Maximum number of unread messages.")]
  public int maxUnreadMessages = 1;
  [Tooltip("Connection Message signal.")]
  public const string ARDUINO_DEVICE_CONNECTED = "__Conn__";
  [Tooltip("Disconnected Message signal,")]
  public const string ARDUINO_DEVICE_DISCONNECTED = "__Dconn__";

  private Thread thread;
  private ArduinoCommThread arduinoComm;

  //When enabled start connection, start thread.
  void OnEnable() {
    arduinoComm = new ArduinoCommThread(portName, baudRate, reconnectDelay,
maxUnreadMessages);
    thread = new Thread(new ThreadStart(arduinoComm.RunForever));
    thread.Start();
  }

  //When disabled request stop, have thread rejoin main thread.
  void OnDisable() {
    if(arduinoComm != null) {
      arduinoComm.RequestStop();
      arduinoComm = null;
    }

    if(thread != null) {
      thread.Join();
      thread = null;
    }
```

```
  }

  //Trigger haptics on collision with object
  void OnTriggerEnter(Collider info)
  {
    Debug.Log("Hand colliding");
    arduinoComm.WriteSerialMessage("1"); //Activate haptics
  }

  void OnTriggerStay(Collider info) { } //don't do anything

  //Release haptics on end with collision of object.
  void OnTriggerExit(Collider other)
  {
    Debug.Log("Hand done colliding");
    arduinoComm.WriteSerialMessage("0"); //deactivate haptics
  }
  // Update is called once per frame
  void Update()
  {
   if(messageListener == null) { return; } //do nothing with no message
listener.

   string recvMsg = arduinoComm.ReadSerialMessage();

   if(recvMsg == null) { return; } //no message

   //pass message to listener, in this case HandModel.cs
   if(ReferenceEquals(recvMsg, ARDUINO_DEVICE_CONNECTED))
   { messageListener.SendMessage("OnConnectionEvent", true); }
   else if (ReferenceEquals(recvMsg, ARDUINO_DEVICE_DISCONNECTED))
   { messageListener.SendMessage("OnConnectionEvent", false); }
   else { messageListener.SendMessage("OnMessageArrived", recvMsg); }
  }

  public void writeMessage(string msg)
  { arduinoComm.WriteSerialMessage(msg); }

  public string readMessage()
  { return arduinoComm.ReadSerialMessage(); }


}
```

Hand Model behavior:

```
using System.Text.RegularExpressions;
using UnityEngine;

public class HandModel : MonoBehaviour
{
  private string[] lastOrientation; //check if orientation has changed
  private string[] lastGyro; //check if gyro has changed
  private string[] lastAccel; //check if acceleration has changed
  private string[] lastPos; //check if position has changed.
  private bool updatePos, updateGyro, updateOrientation;
  private static float AXIS_CORRECTION = -90.0F; //because of unity's 3D axis
  private Vector3 INITAL_POS = new Vector3(0.2F, 0.9F, -9.5F);

  //Handles Messages sent from Arduino Controller.
  void OnMessageArrived(string msg) { parseMessage(msg); }

  //handles Connection event and disconnection events from Arduino Controller
  void OnConnectionEvent(bool success) {
    if(success)
    { Debug.Log("Connection Established."); }
     else { Debug.Log("Connection Terminated or failed to connect."); }
  }

  // Start is called before the first frame update
  //ensure string arrays are valid and set to defaults. Defualts could be
null or empty
  void Start()
  {
    lastOrientation = new string[3];
    lastAccel = new string[3];
    lastGyro = new string[3];
    lastPos = new string[3];
    updateOrientation = false;
    updateGyro = false;
    updatePos = false;
  }

  // Update is called once per frame
  void Update()
  {
    updateGameObject(updatePos, "_P");
    //updateGameObject(updateGyro, "_G"); //curse you unity. I'll deal with
this later.
  }
```

```
  //format is x,y,z from sensors, translate to x,z,y
  //x->x, z->y, y->z for no particular reason other than it's Unity
  private void parseMessage(string msg) {
    string updateType = msg.Substring(0,2);
    string temp = msg.Remove(msg.LastIndexOf('_'));
    string data = temp.Remove(0, 2);
    string[] vector = new string[3];
    switch(updateType) {
      case "_O": //orientation
        vector = Regex.Split(data, ",");
        Debug.Log("Orientation update: x:" + vector[0] + ", z:" + vector[1] +
", y:" + vector[2]);
        updateOrientation = updateLastOrientation(vector);
        break;
      case "_G": //gyroscope
        vector = Regex.Split(data, ",");
        Debug.Log("Gyro Update: x:" + vector[0] + ", z:" + vector[1] + ", y:"
+ vector[2]);
        updateGyro = updateLastGyro(vector);

        break;
      case "_A": //accelerometer
        vector = Regex.Split(data, ",");
        Debug.Log("Accelerometer Update: x:" + vector[0] + ", z:" + vector[1]
+ ", y:" + vector[2]);
        updateLastAcceleration(vector);
        break;
      case "_P": //position
        vector = Regex.Split(data, ",");
        Debug.Log("Pos Update: dx:" + vector[0] + ", dz:" + vector[1] + ",
dy:" + vector[2]);
        updatePos = updateLastPosition(vector);
        break;
      default: break;
    }
  }

  //Update game object parameters, more may be added.
  //Position and orientation are for the transform object.
  //position is a delta offset, must be addes to object position.
  private void updateGameObject(bool needsUpdate, string updateContext) {
    if(needsUpdate) {
      Vector3 temp;
      switch (updateContext) {
        case "_O":
          temp.x = float.Parse(lastOrientation[0]);
```

```
        temp.z = float.Parse(lastOrientation[1]);
        temp.y = float.Parse(lastOrientation[2]);
        temp.x = (temp.x < 355.0F) ? temp.x : 0;
        temp.y = (temp.y < 355.0F) ? temp.y : 0;
        temp.z = (temp.z < 355.0F) ? temp.z : 0;

        temp.y += AXIS_CORRECTION;
        temp.z += AXIS_CORRECTION;
        this.transform.rotation.eulerAngles.Set(temp.x, temp.y, temp.z);
        updatePos = false;
        break;
      case "_G":
        temp.x = float.Parse(lastGyro[0]);
        temp.z = float.Parse(lastGyro[1]);
        temp.y = float.Parse(lastGyro[2]);
        temp.x = (temp.x > 1.0F || temp.x < -1.0F) ? temp.x : 0;
        temp.y = (temp.y > 1.0F || temp.y < -1.0F) ? temp.y : 0;
        temp.z = (temp.z > 1.0F || temp.z < -1.0F) ? temp.z : 0;
        this.transform.Rotate(temp);
        updateGyro = false;
        break;
      case "_P":
        temp.x = (float.Parse(lastPos[0]))+ INITAL_POS.x;
        temp.z = (float.Parse(lastPos[1])) + INITAL_POS.z;
        temp.y = (float.Parse(lastPos[2])) + INITAL_POS.y;

        this.transform.position = temp;
        updatePos = false;
        break;
      default: break;
    }

  }
}

//if current orientation is different update lastOrientation
//using strings so there are no floating point comparison issues.
//returns true if any of the componets differ.
//At Start() the beginning the lastOrientation is empty/null
private bool updateLastOrientation(string[] msgData) {
  bool update = false;
  if(lastOrientation[0] == null || lastOrientation[0] == "") {
    lastOrientation[0] = msgData[0];
    lastOrientation[1] = msgData[1];
    lastOrientation[2] = msgData[2];
    return true;
```

```
  }
  else {
    if(!lastOrientation[0].Equals(msgData[0])) {
      lastOrientation[0] = msgData[0];
      update = true;
    }
    if(!lastOrientation[1].Equals(msgData[1])) {
      lastOrientation[1] = msgData[1];
      update = true;
    }
    if(!lastOrientation[2].Equals(msgData[2]))
    {
      lastOrientation[2] = msgData[2];
      update = true;
    }
    return update;

  }
}

//if current gyorscope reading is different update lastGyro
//using strings so there are no floating point comparison issues.
//returns true if any of the componets differ.
//At Start() the beginning the lastGyro is empty/null
private bool updateLastGyro(string[] msgData) {
  bool update = false;
  if(lastGyro[0] == null || lastGyro[0] == "") {
    lastGyro[0] = msgData[0];
    lastGyro[1] = msgData[1];
    lastGyro[2] = msgData[2];
    return true;
  }
  else {
    if(!lastGyro[0].Equals(msgData[0])) {
      lastGyro[0] = msgData[0];
      update = true;
    }
    if(!lastGyro[1].Equals(msgData[1])) {
      lastGyro[1] = msgData[1];
      update = true;
    }
    if(!lastGyro[2].Equals(msgData[2]))
    {
      lastGyro[2] = msgData[2];
      update = true;
    }
```

```
      return update;

    }
  }

  //if current acceleration is different update lastAcceleration
  //using strings so there are no floating point comparison issues.
  //returns true if any of the componets differ.
  //At Start() the beginning the lastAccleration is empty/null
  //this is a critical update, changes in accleration are part of the physics
engine.
  private bool updateLastAcceleration(string[] msgData) {
    bool update = false;
    if(lastAccel[0] == null || lastAccel[0] == "") {
      lastAccel[0] = msgData[0];
      lastAccel[1] = msgData[1];
      lastAccel[2] = msgData[2];
      return true;
    }
    else {
      if(!lastAccel[0].Equals(msgData[0])) {
        lastAccel[0] = msgData[0];
        update = true;
      }
      if(!lastAccel[1].Equals(msgData[1])) {
        lastAccel[1] = msgData[1];
        update = true;
      }
      if(!lastAccel[2].Equals(msgData[2]))
      {
        lastAccel[2] = msgData[2];
        update = true;
      }
      return update;

    }
  }

  //update lastPos, delta_x, delta_y and delta_z are all reset after being
sent.
  //using strings so there are no floating point comparison issues.
  //returns true if any of the componets differ.
  //At Start() the beginning the lastPos is empty/null
  //Since lastPos is the delta_x, delta_y, and delta_z updates should be
added reguardless of value. Even if 0, 0, 0
  private bool updateLastPosition(string[] msgData) {
```

```
    lastPos[0] = msgData[0];
    lastPos[1] = msgData[1];
    lastPos[2] = msgData[2];
    return true;
  }
}
```

Appendix B: Data Tables

Figure B.1: The High Voltage Power Supply

Table B.1: Power Supply Displayed to Actual Output

Positions from "min."	Displayed Voltage (V) LOW	Reading from Multi-meter (V) LOW	Displayed Voltage (V) HI	Reading from Multi-meter (V) HI
0	32	36	237	234.9
1	50	54.7	256	272.5
2	66	72.8	273	289.5
3	85	92.4	291	308
4	104	113	311	328
5	119	129.7	329	345
6	136	148	346	361
7	152	164.5	365	380
8	171	184	385	399
9	189	203.5	406	419
10	204	291.4	421	433
11	222	237.9	438	449

Appendix C: Inside a High Voltage Power Supply

Figure C.1: Front View

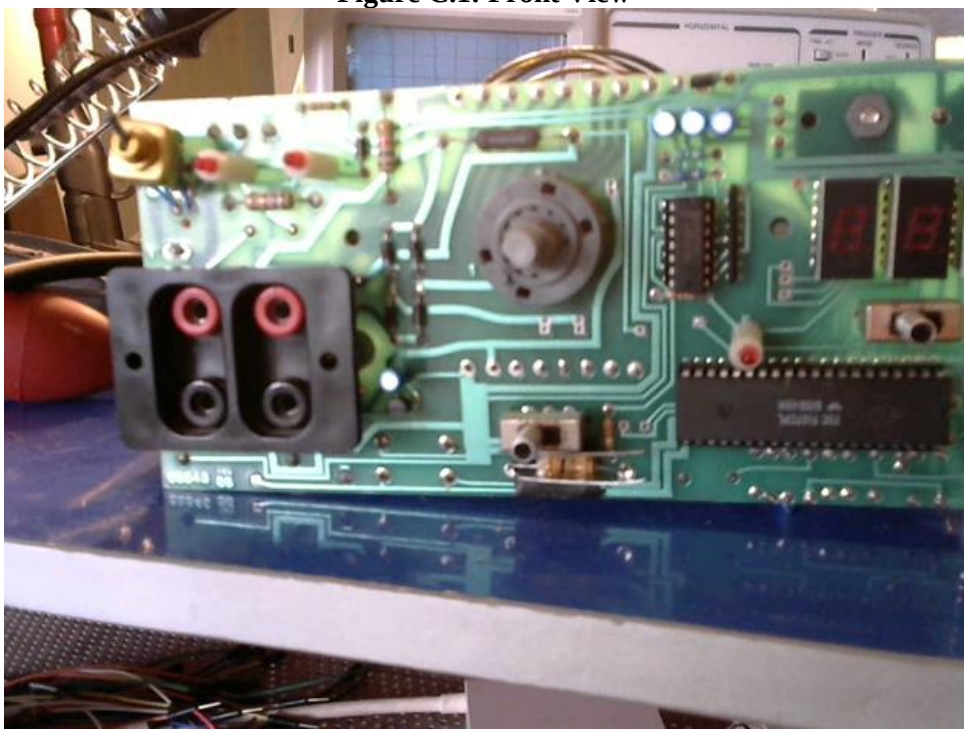


Figure C.2: Top view, electrical tape for scale

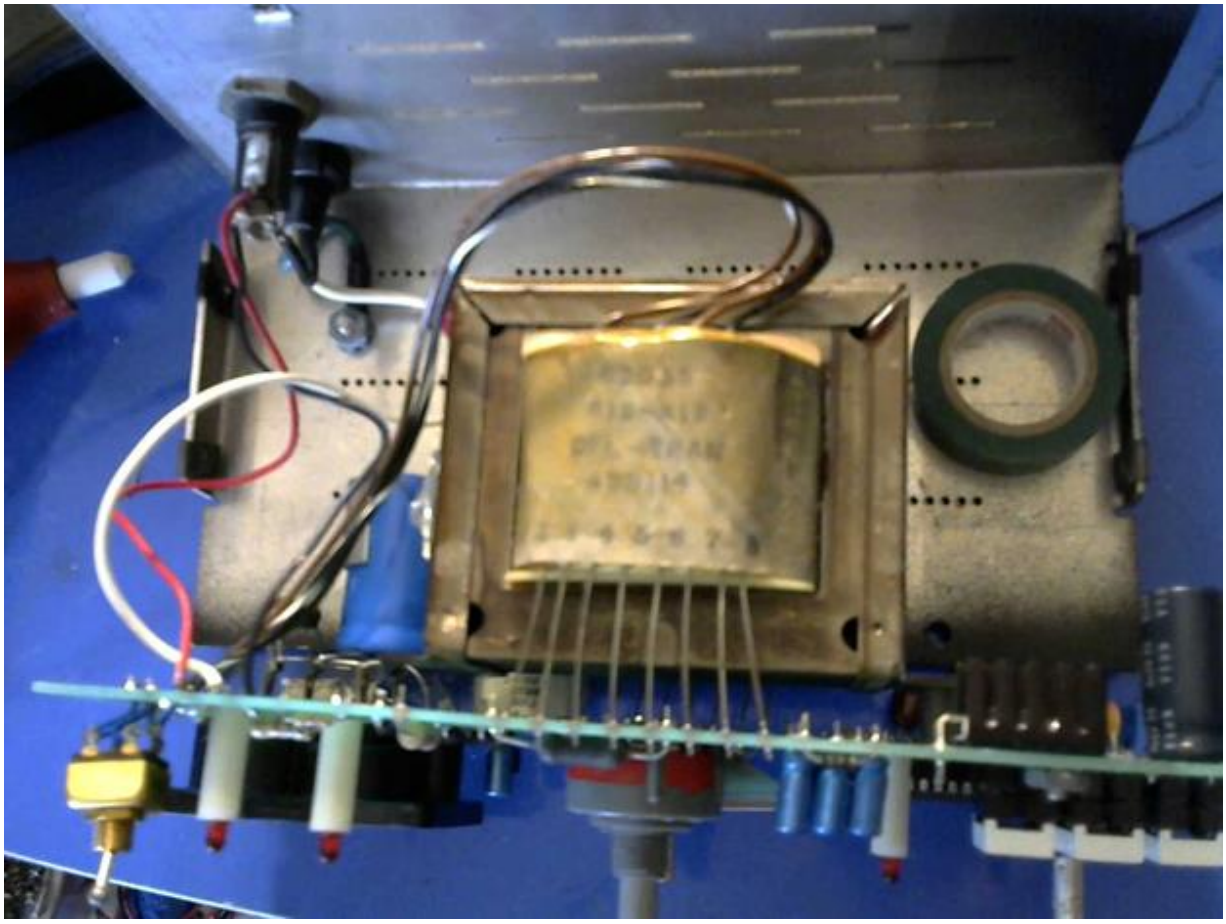


Figure C.3: Rear View

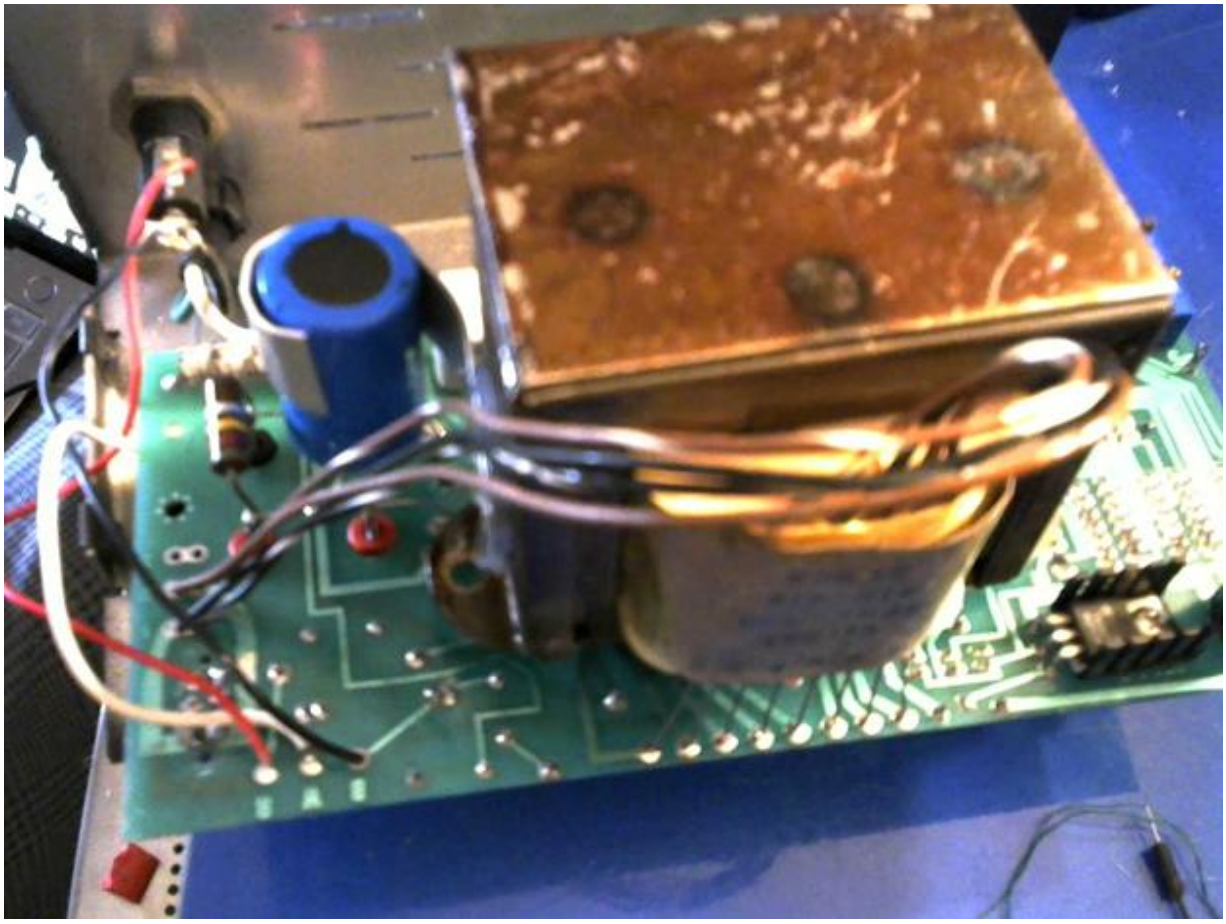


Figure C.4: Bottom View