6 Degrees of Freedeom Tracking Device: Glove I/O

Ahmed Malik, Luis Sanchez, Chase Muramoto, Sachin DeYoung
<amalik, lefsbe, muramoto_chase, sdeyoung>@berkeley.edu
University of California, Berkeley | Fall 2018

ABSTRACT

This paper discusses the development of an input/output (I/O) device in the form of a hand-worn glove, capable of interacting with a virtual environment in 6 degrees of freedom (6DoF) and enabling functionality for all five fingers. (Project video)

1 INTRODUCTION

This project addresses a fundamental deficiency in today's virtual and augmented reality technology: the inability to satisfactorily employ the user's hands as a control input. Commercial attempts to solve this include the Oculus Touch,[1] Samsung Gear VR Controller,[2] and HTC Vive Controllers.[3] All of these mimic traditional computer mice and controllers moreso than a human hand.

In order to more closely mimic the human hand, this project is implemented in the form of a hand-worn device, rather than a hand-held controller. The hand is tracked in physical, 3-dimensional space via the depth sensing capabilities of a Mircosoft Kinect v2,[4] in the form of an $[x, y, z]$ translation vector. A 3-axis gyroscope on a 9-axis inertial measurement unit (IMU) determines the orientation of the hand in rotational space, in the form of a $[\theta, \psi, \phi]$ vector. Flex sensors on each of the five fingers determine how much each finger is bent, and allow for grasping capabilities in a virtual environment.

From a systems perspective, Glove I/O is a collection of various different submodules. The underlying hardware is a combination of various ***Sensors & Actuators***, as the glove uses an Arduino Pro Mini to interface the I^2C IMU and 5 analog flex sensors, and it leverages the abilities of the Microsoft Kinect v2's IR depth sensor. There exists an ***Input & Output*** relationship between the inputs from the sensors and the resulting output in the 3D, Unity environment, where the user's control in the physical world corresponds to a similar action in the virtual world. Under the ***Networking*** submodule, a 2.4GHz radio connection is used to transmit rotational motion and finger actuation to Unity, and a UDP scheme is used to send translational motion data from the Kinect. Within the Unity data processing code, threads are used for ***Multitasking*** purposes, specifically to ensure no lag or latency build up in virtual hand movement.

2 DESIGN AND IMPLEMENTATION

Figure 1 depicts the architecture diagram for our system. The primary hardware designed exists on the glove itself, and is made up of the Arduino Pro Mini, IMU, flex sensors, and nRF24L01 2.4GHz radio. The software system is split up into 3 key components, across

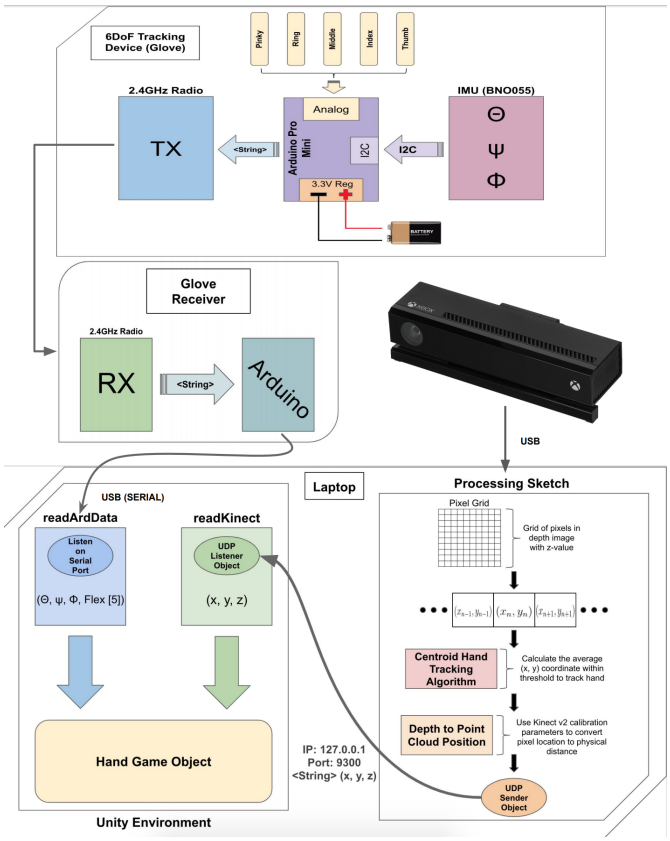

Figure 1: Hardware and software architecture diagram

the Arduino Pro Mini on the glove, the Processing sketch handling the video processing and hand tracking algorithm, and the Unity code interpreting the 11 different values from the sensors.

2.1 Hardware

Figure 2 illustrates the circuit for all of the electronics on the glove device itself. All components are soldered onto a small protoboard and the board is sewed into the glove itself, as are the flex sensors. The entire setup is powered by a single 9V battery, velcroed to the wrist of the glove.

Besides learning to sew in order to attach the flex sensors, the flex sensors posed two other challenges. First, soldering any wire onto these leads required a lot of time and patience so as to ensure the solder connection wouldn't break during normal use. Second, though the flex sensors are constrained to the fingers in most directions, rigid wire soldered to the leads would bend the bottom of the sensor, causing erroneous voltage readings and finger actuation. To improve this, flexible, stranded wire was soldered to the sensors

[1] https://www.oculus.com/rift/accessories/
[2] https://www.samsung.com/global/galaxy/gear-vr/
[3] https://www.vive.com/us/accessory/controller/
[4] https://en.wikipedia.org/wiki/Kinect

EECS 149/249A, December 14, 2018, UC Berkeley
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

instead, and the wires were given extra length for strain relief to the leads.

The flex sensors are combined with a static, 10KΩ resistor to create a voltage divider and produce a variable voltage depending on how much the corresponding finger is bent.

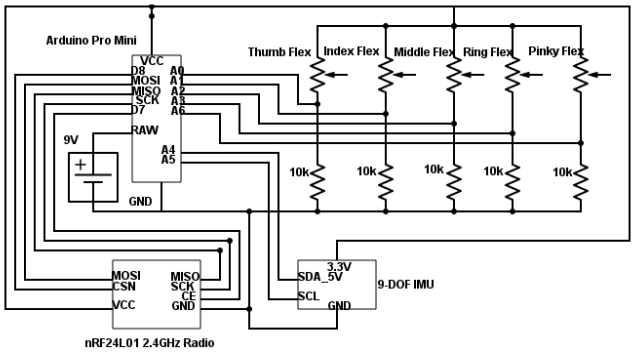

Figure 2: Circuit schematic of glove hardware

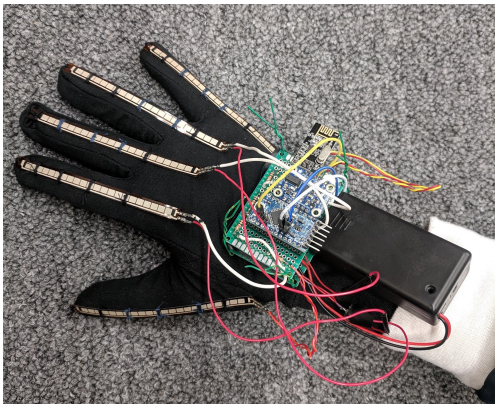

Figure 3: Final Glove I/O tracking device

Off the glove, the Microsoft Kinect v2 tracks the glove's translational motion in (x, y, z). The Kinect was selected for the purposes of this tracking device for its high-quality IR depth camera and wealth of knowledge, libraries, and drivers available online. Originally, the Kinect was planned to be mounted to the user, either on the chest or head, but it suffers from one fatal flaw not obvious on the datasheet. Specifically, the Kinect (v1 and v2) cannot discern any depth values closer than 509mm away from the sensor. A body mounted Kinect would force the user to always keep their hand stretched out maximally, and would be impossible for someone with short arms to use.

When creating the glove, the five flex sensors needed to be securely sewn onto the glove to follow the bend of the fingers. **Figure 4** shows the sensors in the process of being sewn, with several tie-down points along the sensor and a single, small point sewed at the tip. The pinky was originally supposed to employ a shorter, 3 inch sensor, but unlike the longer sensors, the shorter sensor didn't have enough material on the edges to sew it in, and so a longer sensor was used instead.

Figure 4: Flex sensors in process of being attached and secured onto the glove

2.2 Software

One of the biggest hurdles to overcome with this project was to find a software framework to work within that met the project's needs as best as possible. Initially, OpenCV with the OpenNI library on Python was considered, but proved too difficult to work with on macOS, ultimately breaking parts of the Python installation and requiring a reinstall. A similar story ensued when this was tested on Ubuntu. After several different libraries, tutorials, and environments, the Processing IDE with the OpenKinect and libfreenect libraries gave the best results, much in part thanks to Daniel Shiffman's great tutorials.[5]

2.2.1 ***Arduino Pro Mini.*** The `transmitData` code on the Arduino is very straightforward: it simply reads the IMU sensor data and the analog flex sensor data, and writes these values to the 2.4GHz radio as a string. The IMU data is accessed using the sensor's library – `Adafruit_BNO055`.

2.2.2 ***Centroid Hand Tracking Algorithm.*** Multiple algorithms were attempted to track the glove. Specifically, tracking based off of a visible color, tracking the highest object in view, tracking the closest object, and tracking the centroid of all pixels within some threshold. Out of all of these methods, centroid tracking worked the best for our purposes.

Though initially color tracking seemed the most viable in theory, the Processing and OpenKinect framework don't provide much out-of-the-box functionality to track based off of color, whereas

[5] https://shiffman.net/p5/kinect/

Figure 5: Centroid algorithm finding the center of the hand within the minimum and maximum depth thresholds set in the code

processing depth data is very simple. In our case, the Kinect camera simply could not be recognized as a normal camera to Processing and so we were unable to perform analysis on the RGB image; we tried to extend this to blob detection as well but that also failed.

The ideas behind highest point tracking and closest point tracking are very similar: they're simply "world record algorithms" that perform a search over the pixel-grid space to locate the highest (maximum y-pixel) and closest (minimum z-value) points respectively. Highest point tracking worked very well, and we considered adding a tall, tower-like structure onto the glove to be able to track it easier. Closest point tracking worked very poorly, particularly if the hand in view made any complex gestures – such as pointing – causing the algorithm to select wildly varying closest points. Looking back onto this, a low-pass filter likely could have removed most of the issues with closest point tracking, and consequently solve the biggest issue we face with the centroid algorithm.

Figure 6: Centroid algorithm finding the centroid between boths hands visible in the image

For the use case presented in this project – using the glove to control a virtual hand with a user's physical hand – the centroid algorithm makes a few assumptions. Namely, it assumes that only one hand is active within the visible range, and that *only* a hand is active within the range. For example, if you see **Figure 5**, the algorithm perfectly picks out the center of the hand just as we want. However, bringing two hands in as shown in **Figure 6**, causes the algorithm to do just what it's intended to and choose the center of both hands, rather than isolate one individual hand. Similarly, bringing in a leg or head would also throw off the algorithm.

A basic overview of the algorithm can be seen in **Figure 1**, in the Processing Sketch block. The point of the centroid algorithm is to locate all pixels that lie within the Kinect's minimum visible depth range (509mm) and maximum threshold set in the code by the user (usually 1000mm for our testing). Any objects within this range are colored, and all other objects outside are left black.

As the algorithm iterates across the pixel-grid space for a given frame (image), the algorithm records the sum of all the x-pixel locations, y-pixel locations, and total pixels within the depth range, and the average x-pixel and y-pixel locations are computed by dividing the sums over the total:

$$avgX = \frac{\sum_{i=min}^{max} x_i}{\sum_{i=0}^{n} p_i}, \quad avgY = \frac{\sum_{i=min}^{max} y_i}{\sum_{i=0}^{n} p_i}$$

Thus, the white dot in **Figures 5 and 6** is located at $(avgX, avgY)$. The z-value for the point is given by plugging these avgX and avgY values into the depth int-array, where the index offset is determined by the equation:

$$\text{offset} = avgX + (avgY \cdot \text{pixel-grid-width})$$

2.2.3 ***Unity Code.*** The two primary pieces of Unity code – written in C# – are `readArdData` and `readKinect`, as shown in **Figure 1**. As their name suggests, both read the incoming data from the Arduino on the glove or the Kinect, respectively, as a string, and process these strings into the euler angles (θ, ψ, ϕ), finger angles, and world position (x, y, z) for the Hand Game Object. We overcame a small hurdle to learn enough C# for scripting in Unity. Unity reads the data from the Arduino via a Serial port that the Arduino connects to the computer through, and it listens for the data from the Kinect via a UDP client listening on port 9300.

2.3 Integration

By far the most difficult and most rewarding challenges of this project came about during the integration phase, when trying to put together the several different aspects of the project, ranging from hardware to software to data types and data transfer protocols. This project required significant effort to first learn Unity, and then be able to extend it's capabilities to handle data from the glove system.

2.3.1 ***Multitasking.*** During the final testing with the system, it was observed that as time went on in the Unity simulation, the virtual hand's movements became more and more sluggish, eventually slowing down to a crawl where it would take more than a minute for it to react to the inputs in the physical world. Additionally, this lag only occurred on the orientation and flex sensor data coming from the Arduino via Serial port. This lag occurred due to the inherent property that Unity's `readLine()` function – being used to read the incoming data from the Arduino – was in fact a blocking

function, making it difficult for the processor to do anything other than read the data stream.[6]

To solve this, we implemented multitasking into the `readArdData` file, where one thread was created solely to read the incoming data from the Arduino on the Serial port, while the other thread performed all other computations and setting Hand Game Object positional values.

2.3.2 ***Reducing Unity's "Clock" Speed.*** Unity's `Update()` function runs extremely quickly, as it should since its goal is to update the frames in its current simulation as smoothly as possible. This poses some problems when trying to read and process the data stream from the Arduino, coming in much slower, as Unity would only partially record the stream of data, or completely miss it. To solve this problem, we took inspiration from the timer problem assigned in homework 2, where we were tasked to design a digital circuit to reduce the speed of the input clock by a factor of 20. We simply reduce how quickly the `Update()` function runs by only allowing data to be read and processed every $n = 8$ iterations, the number we found worked well for this scenario.

2.3.3 ***UDP Implementation.*** Originally, the translational motion data from the Kinect was planned to be written to a CSV file and then simultaneously read and processed by Unity. This proved nearly impossible to do, as the operating system locks the file during read/write, and synchronizing the two processes was the only recourse. Instead, our image processing sketch implements a UDP sender that sends the (x, y, z) vector as a string over port 9300, on localhost 127.0.0.1. Similarly, `readKinect` on Unity listens for the string of data on this port. **Figures 7 and 8** show the first successful transmission of data achieved.

```
-- UDP session started at Sun Dec 09 21:10:56 PST 2018 --
-- bound socket to host:null, port: 61974 --
[18-12-09 21:10:56.837 -0800] send packet -> address:/127.0.0.1, port:9300, length: 1
Sent number: 0
[18-12-09 21:10:59.357 -0800] send packet -> address:/127.0.0.1, port:9300, length: 1
Sent number: 1
[18-12-09 21:11:01.864 -0800] send packet -> address:/127.0.0.1, port:9300, length: 1
Sent number: 2
```

Figure 7: UDP sender on Processing sketch transmitting dummy data over port 9300, localhost 127.0.0.1

```
Ahmed — Listener.dll — dotnet
Waiting for broadcast
Received broadcast from 127.0.0.1:61974 :
 0
Waiting for broadcast
Received broadcast from 127.0.0.1:61974 :
 1
Waiting for broadcast
Received broadcast from 127.0.0.1:61974 :
 2
```

Figure 8: UDP client on Unity end listening for and receiving data on port 9300, localhost 127.0.0.1

[6] https://answers.unity.com/questions/400222/arduino-with-unity-bad-framerate.html

2.3.4 ***Corrective Factor On Depth Data.*** Two types of inaccuracies occur when using the raw positional and depth (x, y, z) data to determine object placement within a 3D virtual environment like Unity.

First, though the z-value (depth) is acquired in units of physical distance (in our case millimeters), x and y are simply pixel numbers. These are converted to values of physical distance using a depthToPointCloudPos function that uses the Kinect v2's camera hardware parameters to convert the pixels to a physical distance metric.[7]

Second, since the Kinect faces the user in our scenario, the depth values need to be inverted, such that moving closer to the Kinect moves the virtual hand into the screen, and moving away moves the virtual hand out of the screen. We achieve this with the following corrective factor, applied to the raw z-value:

$$\text{newZ} = |\text{oldZ} - \text{maxThresh}|$$

where maxThresh is the maximum depth the Kinect is able to visualize.

When the user's hand is the farthest away from the Kinect (maxThresh [mm]), the hand should placed at the 0 z-value in the 3D environment, and conversely, when the user's hand is at the minimum value the Kinect can discern (0 [mm]), the virtual hand's z-value should be maxThresh. This gives the virtual hand full range consistent with the physical world, and because the raw depth values are cut-off at the minimum and maximum thresholds, the newZ value is always within the linear regime of the absolute value function. Therefore, the system does not exhibit nonlinear motion.

3 EVALUATION

3.1 Hand Tracking

3.1.1 ***Translation Motion.*** The translation motion tracking worked very well given the assumptions that the centroid tracking algorithm needs to meet in order to operate perfectly. However, this type of hand tracking is limiting.

Firstly, the algorithm has difficulty determining the centroid of the pixels in the image if the hand is at the border of maxThresh. This is because the pixel density is much lower, and so the average value changes sporadically when calculating an accurate centroid. At one moment, the palm of the hand may be fully visible, and at another it may not: this causes jittery motion in the Unity environment. This sort of "frame-skipping" could be fixed with a simple low-pass filter to smooth out the motion of the hand near maxThresh.

Second, the centroid tracking algorithm doesn't necessarily track a hand itself. Ideally, this project would implement a schema that could detect the user's hand given any image, via a method such as color tracking or localizing around some key, visible feature on the glove itself – perhaps an IR emitter.

3.1.2 ***Rotational Motion.*** The rotational motion of the glove performed exceptionally well, and the virtual hand was able to track the physical glove very well. However, one downside to the system is that it required a calibration period when starting the Unity simulation, where the glove had to be pointing straight at the screen in order to acquire the zero location in (θ, ψ, ϕ).

[7] https://shiffman.net/p5/kinect/

Figure 9 shows the (θ, ψ, ϕ) data received from the sensor during testing. Notice how yaw spikes up to 360°. At the zero value in yaw, sweeping in a counter-clockwise fashion will immediately spike the yaw value to 360, will sweeping in a clockwise fashion will follow normally. We correct for this wrapping behavior when processing the data and determining the Hand Game Object's yaw angle.

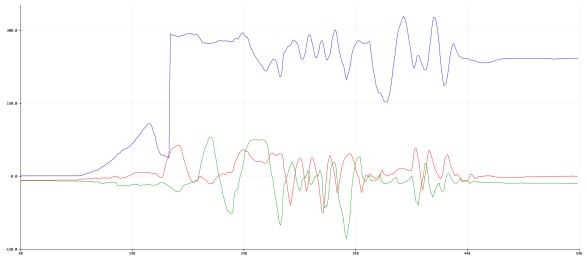

Figure 9: Rotational motion data with yaw in blue, pitch in red, and roll in green. x-axis [ms], y-axis [degrees]

3.1.3 ***Finger Actuation.*** **Figure 10** illustrates the flex sensor data from the fingers during normal use. Notice the 5 peaks at the end of the sample that represent all 5 fingers, from the thumb in yellow to the pinky in purple. As shown, this data is very clean, easy to process, and Unity has no problems updating its Hand Game Object's finger parameters. Using the finger actuation ability, the glove device is able to accurately pick up and grasp objects in the virtual environment.

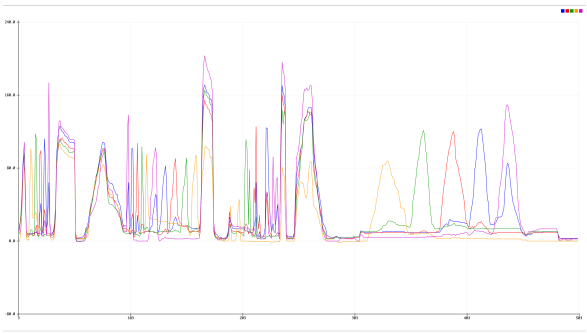

Figure 10: Flex sensor data where each finger is mapped to its own graph with clearly distinct values. Thumb in orange, index finger in green, middle finger in red, ring finger in blue, and pinky finger in purple. x-axis [ms], y-axis [degrees]

3.2 Latency

According to the Human Benchmark Reaction Time Test results, the average human reaction time is 273ms **(Figure 11)**, across a sample size of 75,258,761 individual tests at the time of writing this paper.[8]. To determine whether Glove I/O is quick enough to provide a smooth experience, the latency for every frame in a 40 second long simulation was calculated. **Figures 12 and 13** showcase the results

[8]https://www.humanbenchmark.com/tests/reactiontime/statistics

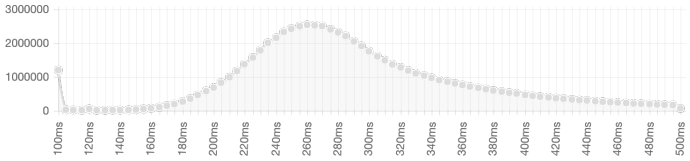

Figure 11: Average reaction time for all time since the start of the test. Plot taken from Human Benchmark.

of this experiment, where the average latency across the simulation was only **123.16ms**, far below the 273ms average. The first peak in

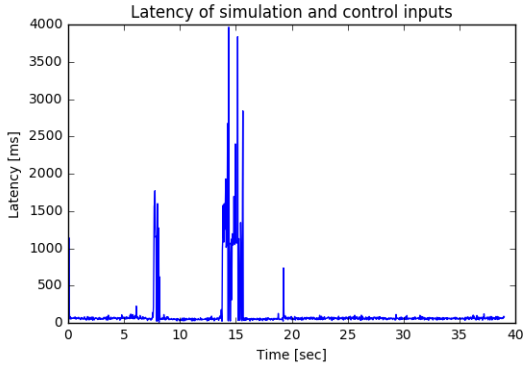

Figure 12: Latency per frame across a 40 second interval

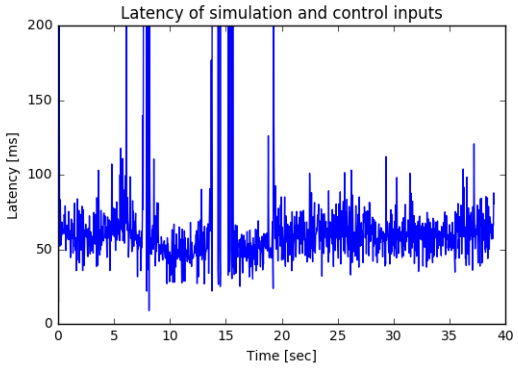

Figure 13: Same plot as Figure 12, but zoomed in to show a better view of the average latency

this scenario corresponds to the 2.4GHz radio connection timing out for a moment, and requiring the sensors on the hand to be power-cycled. Glove I/O developed some troubles keeping a consistent radio connection if it moved too far away – most notable outside in the Cory courtyard. The second peak corresponds to the Processing Kinect script timing out – as it usually does since USB3 is unable to provide sustained power to the Kinect if other electronics like an Arduino is plugged in – which was promptly reset. With more fine tuning, the radio connection and Processing sketch could be

improved to limit these time-outs, further reducing the latency. Even still, the current average latency is very promising.

4 DISCUSSION

Given more time, Glove I/O could be more refined, both as a physical, usable product and in the underlying software. The protoboard would be converted to a professionally printed PCB, and wherever possible, surface mounted components would be used. A proper, hand tracking algorithm would replace the centroid tracking algorithm. The Kinect would also be replaced by something much smaller, such as either a small, high-quality camera or IR blaster and receiver pair for depth sensing; this setup would be mounted to the glove user with a professional mounting harness.

The high-level implications of Glove I/O are very widespread, because fundamentally the device is an input/output device, much like a mouse and keyboard, and could be used for various types of control in either a virtual or physical environment. We've presented the use-case for a virtual environment in this project, but we can see this project being repurposed to control a robot like Kobuki or even a drone. This type of control relies on some sort of gesture recognition, and Glove I/O is a perfect platform given its 6DoF motion data and additional, finger actuation control.

The data from the glove could be very powerful to learn humans' behavior in regards to how we use our hands for grasping or general, everyday tasks. This type of data could be used to train a machine learning model that could lead to breakthroughs in prosthetic design, humanoid robots, and more life-like virtual environments.

One impetus for this project was an augmented reality headset paired with a 6DoF device like Glove I/O, enabling a CAD engineer to view their solid-part designs in real space, allowing them to walk around the object and scrutinize every little detail, rather than do this on a 2D computer screen. This could give engineers a better idea of how their designs would really fit in the physical world.

5 SPECIAL THANKS

Thank you to the professors and GSIs for all your help and a wonderful semester in EECS149. We've thoroughly enjoyed the class and this chance to work on a large developmental project.

Thank you to Bala Kumaravel for his advice and guidance through tricky portions of this project, as well as access to hardware components.

Thank you to Oscar Dorado for his help in teaching us how to use Unity to best meet the needs of our project.

6 HARDWARE DATASHEETS

Flex Sensor's Datasheet https://cdn.sparkfun.com/datasheets/Sensors/ForceFlex/FLEXSENSORREVA1.pdf

BNO055 IMU's Datasheet https://cdn-shop.adafruit.com/datasheets/BST_BNO055_DS000_12.pdf

nRF24L01 Transceiver's Datasheet https://www.sparkfun.com/datasheets/Components/nRF24L01_prelim_prod_spec_1_2.pdf

Arduino Pro-Mini Schematics https://www.arduino.cc/en/uploads/Main/Arduino-Pro-Mini-schematic.pdf

Arduino Uno Schematics https://www.arduino.cc/en/uploads/Main/Arduino_Uno_Rev3-schematic.pdf