Glovesy

BY

Alan Devine - 17412402

Sean Moloney - 17477122

A Technical Document

As a requirement for CA400

Last Revision: 06/05/2021

Dublin City University (DCU)

PLAGIARISM DECLARATION

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my/our work. I/We understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I/We engage in plagiarism, collusion, or copying. I/We have read and understood the Assignment Regulations. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me/us or any other person for assessment on this or any other course of study.

I/We have read and understood the referencing guidelines found at
`https://www.dcu.ie/info/regulations/plagiarism.shtml`,
`https://www4.dcu.ie/students/az/plagiarism`,
and/or recommended in the assignment guidelines.

Project Title	Glovesy
By	Alan Devine - 17412402
	Sean Moloney - 17477122
Field of Study	Computer Science
Project Advisor	David Sinclair
Academic Years	2020/2021

ABSTRACT

Glovesy is a wearable computer interfacing device in the form of a glove which will allow the user to interface with their computer by using custom macros or use the device for hand-tracking in VR or AR applications.
Keywords: : Wearables, human-computer interfacing, VR, AR, Arduino

GLOSSARY

1. VR Headset A head-mounted device for use in consuming Virtual Reality Content.
2. AR Headset Similarly to a VR Headset, an AR Headset is a head-mounted device which aims to provide a User with content that is built around their surroundings rather than replace them in the case of a VR Headset.
3. Arduino An Open-Source electronic prototyping platform.
4. ISR Interrupt Service Routine is a software process invoked by an interrupt request from a hardware device.

TABLE OF CONTENTS

LIST OF FIGURES

Introduction

1.1 Overview

Glovesy is an Arduino based wearable device which will allow the user to interface with their computer, either by using user-defined macros, which will be set up using our program, the Glovesy Configuration Suite, which will allow several gestures do be defined to certain actions within the PC or by allowing the user accurate hand and finger tracking for use in Virtual and Augmented Reality.

Gestures will be defined by a user and mapped to some action on their PC. Actions could include entering a combination of keystrokes, opening an application, raising/ lowering system volume, etc. They will be executed upon the user repeating their chosen gesture.

Motivation

The idea for glovesy came from our analysis of the input devices used in VR/ AR applications. Typically in the case of VR applications, standard input devices mostly come in the form of controllers held in each hand with an array of buttons and some mechanism to provide tracking on the top. AR headsets, generally rely on voice-activated services such, or, the use of a smartphone to interact with the device. In both applications, we agreed that using a glove based input device would provide a more immersive VR gaming experience, and more a more intuitive way of interacting with AR content.

Furthermore, we saw a great learning opportunity with this project. We would be exposed to programming microcontrollers, GUI development and game development to name a few. We also would have the opportunity to make use of some of the skills we developed while undergoing this degree, namely applying linear algebra to a real project.

Research

Overall a significant amount of research was carried out before beginning the development of Glovesy in order to see if our idea would be possible to implement given our current technical expertise and within a reasonable budget.

3.1 Finger Tracking

One of the main components that we carried out research on was how the fingers would be tracked. While we found a number of low-budget ways to track the user's fingers, such as using potentiometers attached to the fingers using fishing line or string, however using this method means that the glove will be much larger and bulkier than using flex sensors. This method also has the downside of the increased chance of snagging due to the fishing line attached to the fingers. In the end, we decided to use flex sensors, as while there are expensive versions of them such as the ones found on Adafruit, it is possible to buy the materials and construct them for much cheaper. Flex sensors are also very low profile and sit flush along the glove, making it feel more natural and reduces the chances of snagging.

3.2 Hand Tracking

When researching methods for tracking the user's hand, while there were a few other options, such as using a HTC Vive Tracker, however this is not only an expensive option, but it also requires at least one HTC Vive Lighthouse for tracker. On top of this, when attached, the tracker is bulky and can easily snag or catch on something due to it's design, especially when using a VR headset, since the user cannot see their sur-

roundings. Instead, the best option we could find was using an 9DoF IMU, specifically the Adafruit LSM6DS33 LIS3MDL, as it was both cheap and small, while also offering accurate accelerometer and gyroscope data.

3.3 Board

When looking for a board to use, there were a number of requirements which had to be satisfied for it to function adequately. Those requirements were:

- the board must have enough analog pins to read data from each of the flex sensors
- the board must have i2c compatibility in order to receive data from the IMU
- the board must be compact in terms of its form factor in order to reduce the bulk of the glove and make it more comfortable to wear

While researching said boards, there were a few boards that fit the requirements, such as the TinyCircuits TinyLily Mini, but we decided to use the Adafruit Feather M0 Bluefruit LE, since it not only meets all the requirements, but is also Bluetooth enable, which would allow us to create a wireless version of the device without needing to purchase new components.

3.4 Game Engines

During research into which game engine to use as the basis for the demonstrations, there were a number of valid options. One of said options being the Godot engine as it is a relatively easy engine to develop for, however, due to the limitations of the engine, it was decided that another engine would be better suited, especially due to difficulty when getting input from non-standard controllers. Another favourable option to use was Unreal Engine as it uses C++ which is a language we both have experience in. However, this option was ruled out due to compatibility issues when developing on linux systems. As such we decided to use Unity as it is a cross-platform game engine, as well as allowing non-standard input devices to be used with relative ease. This, though, was not an ideal

scenario as we have little experience using Unity as well as little experience programming in C#.

3.5 Language for implementing AR Suite

Choosing a language for this aspect of the project was rather straightforward. It had to meet certain requirements. The first requirement was for the language to be statically typed, this is mostly down to personal preference as I have found dynamically typed languages to be cumbersome when working with larger codebases. The next requirement was for the language to be platform agnostic. With all that considered, we settled on Java.

3.6 Build Tool

The choice in build tools came down to two candidates, Maven and Gradle. Having used both in the past, I settled on Gradle as it is typically quick to set up and modify as well as having great support in my Java ide of choice. It also provides some excellent features such as automatically generating a test result site on each build of the project.

3.7 GUI Library

Having settled on Java as the programming language to be used for the AR aspect of this project, we needed to choose a library for GUI development. Requirements for a GUI library mainly came down to the availability of documentation and the ability to create 3D models. While researching libraries, we discovered that Javafx satisfies both requirements.

3.8 Competitors

Upon researching to see if the idea had already been done by a company, or if the idea was even feasible to begin with, we discovered a number of different implementations of the

same idea. A very high fidelity solution to our idea can be found in the Manus VR Primus II, however this very high fidelity solution is very costly at a price of €2,499. Another solution which takes a different approach to the problem is the Senseglove, which is not only very bulky, being much larger than the user's hand, but is also very expensive, at €2,999. A final example is the HAPTX gloves which are once again very bulky, including a backback which must be worn during use.Another downside to all of the aforementioned competitors is their reliance on existing vr systems such as the HTC Vive Trackers, or the Occulus Controllers which not only require the user to have a vr environment set up, but also add bulk to the glove, particularly the Occulus Controllers, alongside being somewhat expensive.

Design

4.1 System Architecture

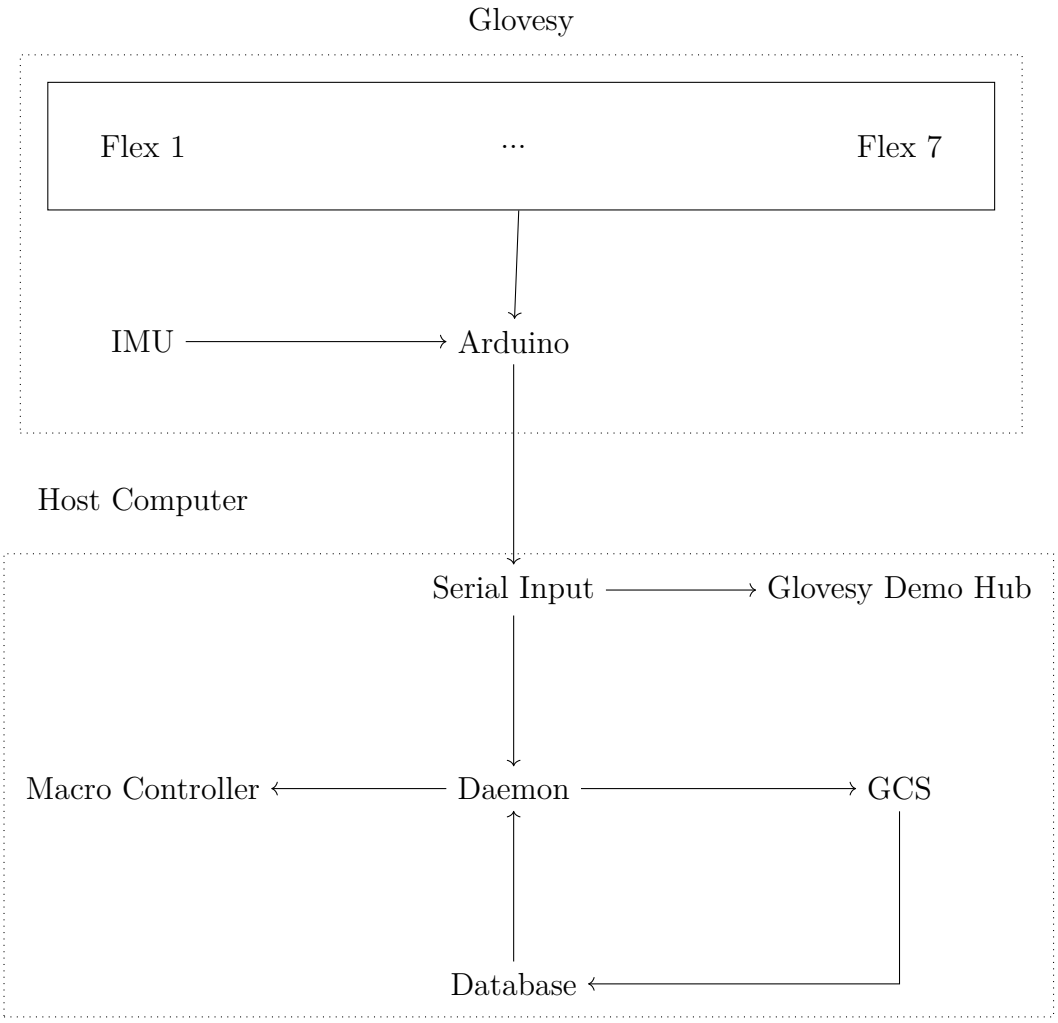

Figure 1: High-Level System Architecture Diagram

4.2 Flex Sensors

The flex sensors are constructing by using 3 layers of velostat, with conductive thread running up along both top and the underside, and is held together using electrical tape. They function by changing resistance according to how much the sensor is bent. Using this resistance, once the sensor is calibrated, the value can be normalized to be between 1.0 and 0.0 depending on how bent the finger is.

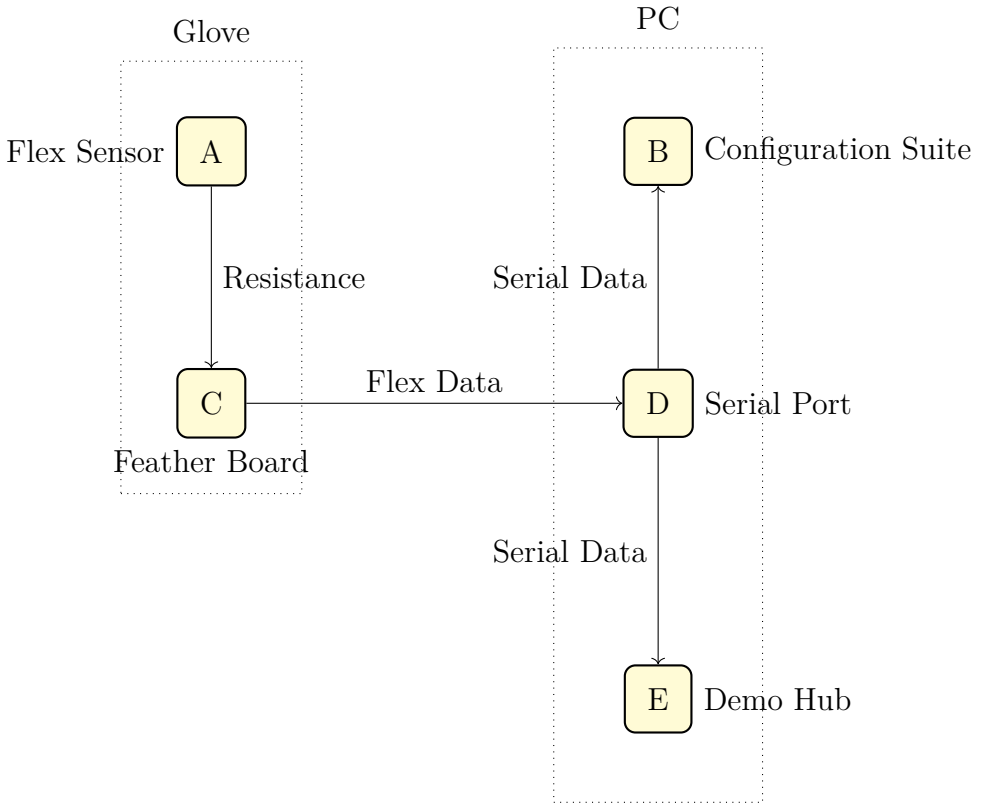

Figure 2: Flex Sensor Data Flow Diagram

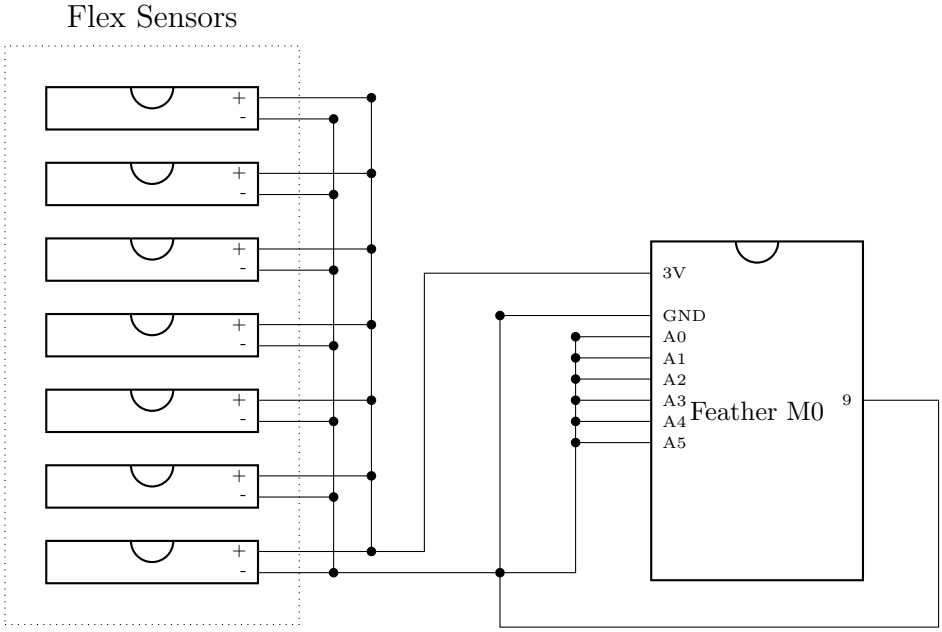

Figure 3: Flex Sensor Circuit Diagram

4.3 IMU

The IMU functions by sending Accelerometer, Gyroscope, and Magnetometer data to the board over I2C, thereby allowing the system to track hand movement and orientation.

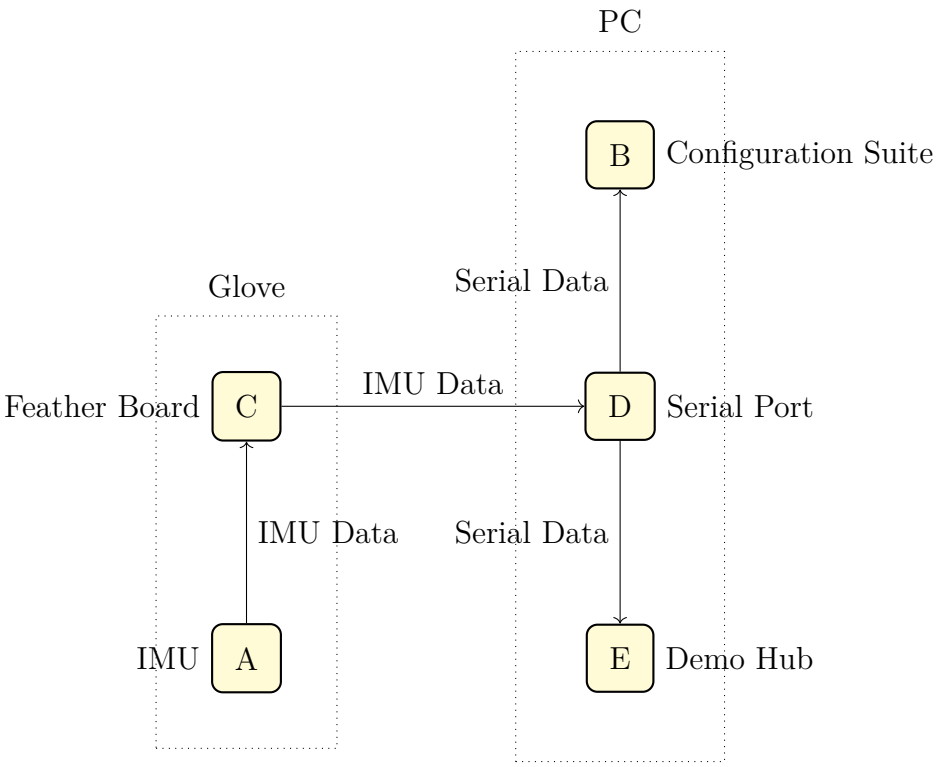

Figure 4: IMU Data Flow Diagram

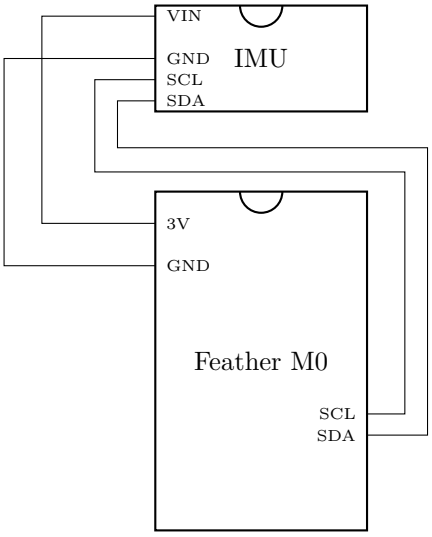

Figure 5: IMU Circuit Diagram

4.4 Board

The board used is the Adafruit Feather M0 Bluefruit LE, which transfers data from the flex sensors and IMU to the PC as comma seperated values over serial communications.

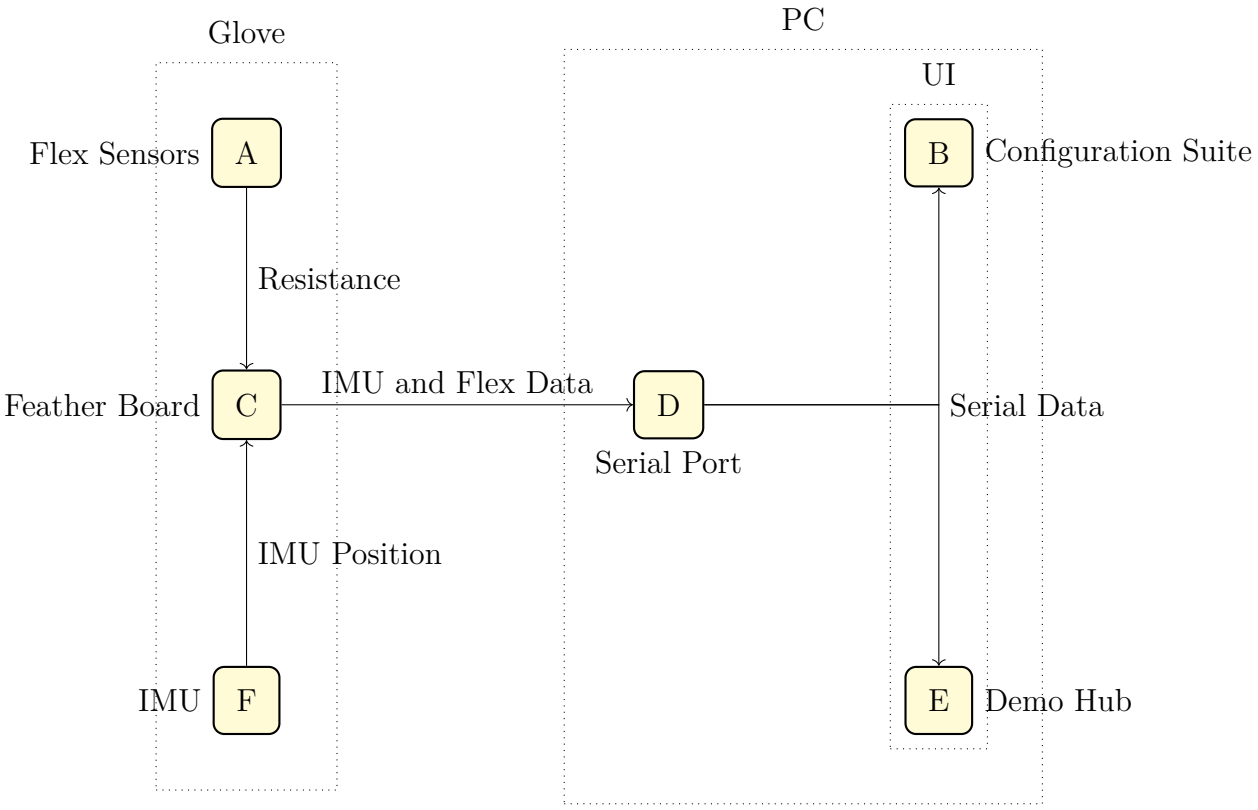

Figure 6: Data Flow Diagram

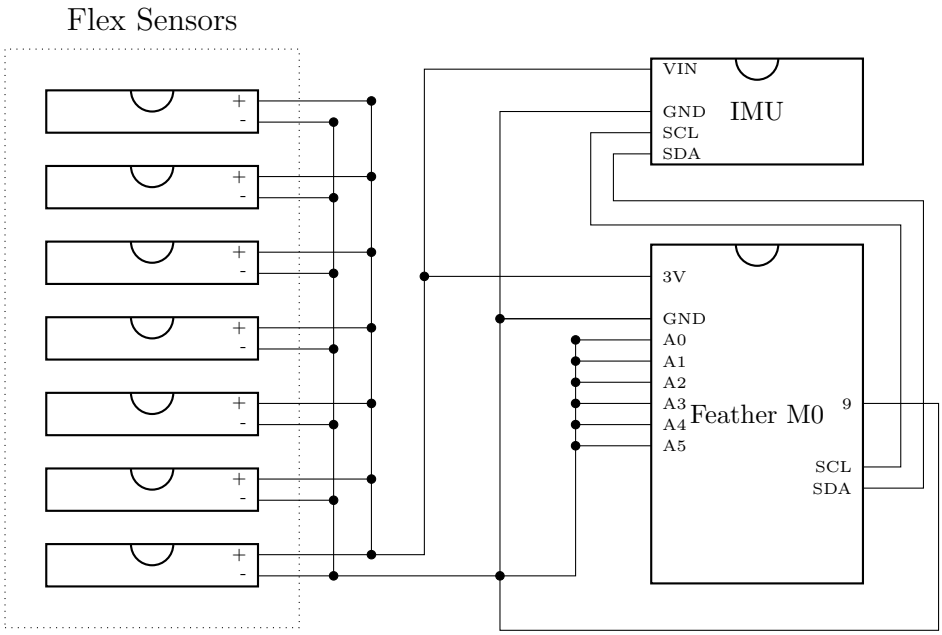

Figure 7: Board Circuit Diagram

4.5 Glovesy Configuration Suite

4.6 Glovesy Demo Hub

Glovesy Demo Hub is a game environment created using Unity which allows the user to use glovesy to interact with a virtual environment. It does this by reading the serial data received from glovesy, and assigning the values received to the in-game representation of the user's hand, allowing them to interact with physics objects.

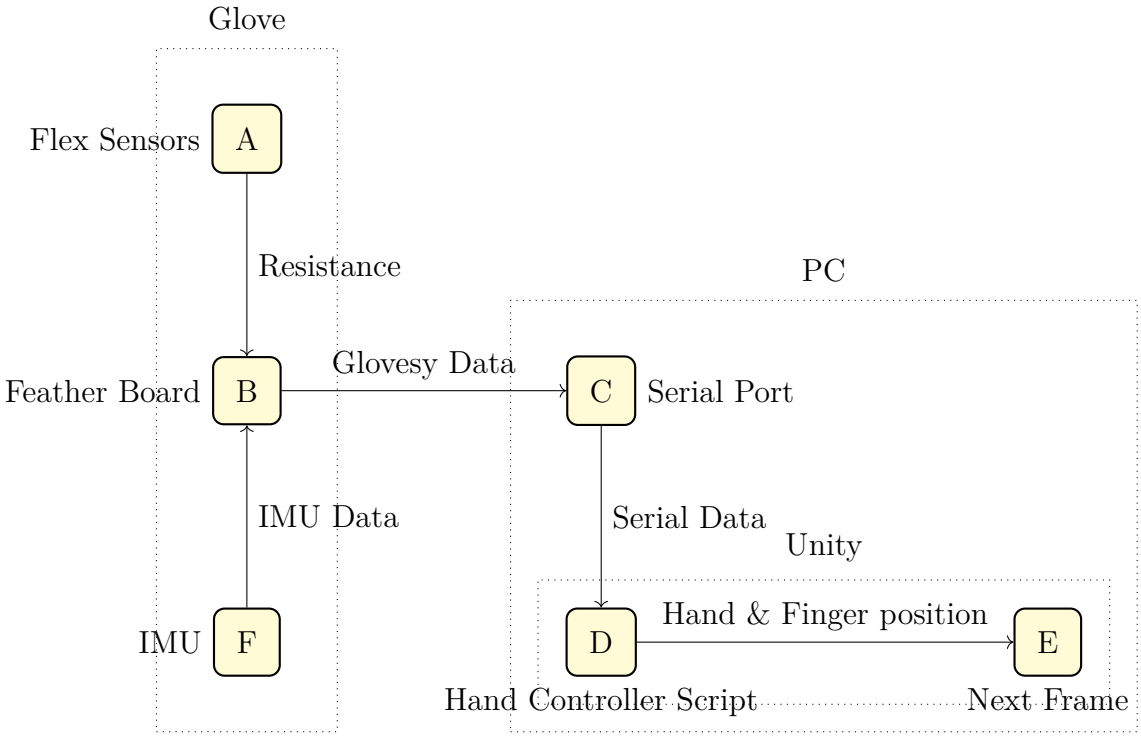

Figure 8: Demo Hub Data Flow Diagram

Implementation

5.1 Flex Sensors

The flex sensors are created by stacking 3 layers of velostat on top of each other and then attaching a conductive thread along the outside of it on both sides. Once this is done its routed through a 4.7kΩ resistor which is connected to ground, and goes to an analog sensor on the board. When this is done, the board will read the voltage by default.

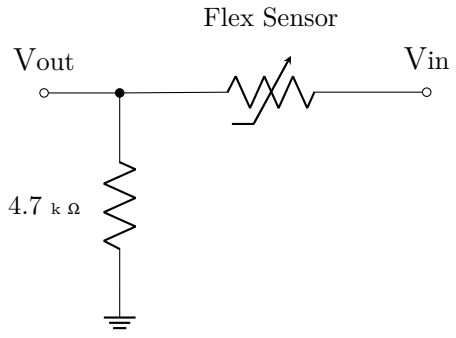

Figure 9: Flex Sensor Resistor Circuit

At this point the value from the flex sensor can be read as shown in Listing 1.

```
// Read thumb values
Serial.print(analogRead(A4));
Serial.print(",");

// Read Index values
Serial.print(analogRead(A0));
Serial.print(",");
Serial.print(analogRead(A1));
Serial.print(",");

// Read Middle finger values
Serial.print(analogRead(A2));
Serial.print(",");
Serial.print(analogRead(A3));
Serial.print(",");

// Read Ring finger value
Serial.print(analogRead(A5);
Serial.print(",");

// Read Pinky finger value
Serial.print(analogRead(9));
Serial.println();
```

Listing 1: C Code to read voltage from flex sensor

While reading in this method is adequate, we decided to covert the voltage to the corresponding resistance using the code seen in Listing 2.

```
float get_resistance(int ADCflex) {
  // Convert value from flex sensor to resistance
```

```
  float VCC = 3.3;
  int R_DIV = 4700;
  float Vflex = ADCflex * VCC / 1023.0;
  float Rflex = R_DIV * (VCC / Vflex - 1.0);

  return Rflex;
}
```

Listing 2: C code for conversion from voltage to resistance

Once that has been implemented, both can be used to read the resistance of each flex sensor and print it to the serial port using the code in Listing 3.

```
float get_resistance(int ADCflex) {
  // Convert value from flex sensor to resistance
  float VCC = 3.3;
  int R_DIV = 4700;
  float Vflex = ADCflex * VCC / 1023.0;
  float Rflex = R_DIV * (VCC / Vflex - 1.0);
  return Rflex;
}
// Read thumb values
Serial.print(get_resistance(analogRead(A4)));
Serial.print(",");
// Read Index values
Serial.print(get_resistance(analogRead(A0)));
Serial.print(",");
Serial.print(get_resistance(analogRead(A1)));
Serial.print(",");
// Read Middle finger values
Serial.print(get_resistance(analogRead(A2)));
Serial.print(",");
Serial.print(get_resistance(analogRead(A3)));
```

```
21 Serial.print(",");
22 // Read Ring finger value
23 Serial.print(get_resistance(analogRead(A5)));
24 Serial.print(",");
25 // Read Pinky finger value
26 Serial.print(get_resistance(analogRead(9)));
27 Serial.println();
```

Listing 3: C code to print flex resistance to serial port

5.2 IMU

In order to use the IMU, we had to use the Adafruit_LSM6DS33 library which can be installed through the arduino IDE. Using this library we can initialise the IMU using the code as seen in Listing 4. Once the IMU has been initialised, gyroscope and accelerometer data can be read and printed to the serial port as shown in Listing 5.

```
1  #include <Adafruit_LSM6DS33.h>
2
3  // For SPI mode
4  #define LSM_CS 10
5  // For Software-SPI
6  #define LSM_SCK 13
7  #define LSM_MISO 12
8  #define LSM_MOSI 11
9
10 Adafruit_LSM6DS33 lsm6ds33;
11
12 void setup(void) {
13     // Accelerometer
14     lsm6ds33.configInt1(false,false,true);
15     // Gyro
```

```
16      lsm6ds33.configInt2(false,true,false);
17  }
```

Listing 4: Code to initialise the IMU

```
1  sensors_event_t accel;
2  sensors_event_t gyro;
3  sensors_event_t temp;
4
5  lsm6ds33.getEvent(&accel, &gyro, &temp);
6
7  // Accelerometer and Gyro data
8  Serial.print(accel.acceleration.x);
9  Serial.print(",");
10 Serial.print(accel.acceleration.y);
11 Serial.print(",");
12 Serial.print(accel.acceleration.z);
13 Serial.print(",");
14 Serial.print(gyro.gyro.x);
15 Serial.print(",");
16 Serial.print(gyro.gyro.y);
17 Serial.print(",");
18 Serial.print(gyro.gyro.z);
19 Serial.println();
```

Listing 5: Code to print IMU data to serial port

5.3 Board

To be able to program the board, we used the Arduino IDE which allowed for easy deployment of the code and easy reading of the serial port, as well as making it very simple and streamlined to install the libraries required to let our device function. The bulk of the coding for the board has already been mentioned in sections 5.1 and 5.2, as

all that was left to do was to combine the code for getting the flex data and the IMU data and set the baud rate as can be seen in Figures 6 and 7

```
#include <Adafruit_LSM6DS33.h>
// For SPI mode, we need a CS pin
#define LSM_CS 10
// For software-SPI mode we need SCK/MOSI/MISO pins
#define LSM_SCK 13
#define LSM_MISO 12
#define LSM_MOSI 11
Adafruit_LSM6DS33 lsm6ds33;
void setup(void) {
  Serial.begin(115200);
  while (!Serial)
    delay(10);
  if (!lsm6ds33.begin_I2C()) {
    Serial.println("Failed to find LSM6DS33 chip");
    while (1) {
      delay(10);
    }
  }
  // accelerometer DRDY on INT1
  lsm6ds33.configInt1(false, false, true);
  // gyro DRDY on INT2
  lsm6ds33.configInt2(false, true, false);
}
float get_resistance(int ADCflex) {
  // Convert value from flex sensor to resistance
  float VCC = 3.3;
  int R_DIV = 4700;
  float Vflex = ADCflex * VCC / 1023.0;
  float Rflex = R_DIV * (VCC / Vflex - 1.0);
  return Rflex;
```

```
}
```

Listing 6: Initialisation of the board

```
void loop() {
  sensors_event_t accel;
  sensors_event_t gyro;
  sensors_event_t temp;
  lsm6ds33.getEvent(&accel, &gyro, &temp);
  Serial.print(get_resistance(analogRead(A4)));
  Serial.print(",");
  Serial.print(get_resistance(analogRead(A0)));
  Serial.print(",");
  Serial.print(get_resistance(analogRead(A1)));
  Serial.print(",");
  Serial.print(get_resistance(analogRead(A2)));
  Serial.print(",");
  Serial.print(get_resistance(analogRead(A3)));
  Serial.print(",");
  Serial.print(get_resistance(analogRead(A5)));
  Serial.print(",");
  Serial.print(get_resistance(analogRead(9)));
  Serial.print(",");
  Serial.print(accel.acceleration.x);
  Serial.print(",");
  Serial.print(accel.acceleration.y);
  Serial.print(",");
  Serial.print(accel.acceleration.z);
  Serial.print(",");
  Serial.print(gyro.gyro.x);
  Serial.print(",");
  Serial.print(gyro.gyro.y);
  Serial.print(",");
```

```
30    Serial.print(gyro.gyro.z);
31    Serial.println();
32  }
```

Listing 7: Main loop for the board

5.4 Serial Monitoring

Serial monitoring was achieved using the JSerialComm library. This required two classes.

5.4.1 SerialPortListener

This object found in Listing 8 implements the SerialPortMessageListener which is an interface which allows for the capture of byte-delimited input streams. Below we define the delimiter as 0x0A which corresponds to the newline character. When a new line character is detected, the serialEvent() method is called, which splits the incoming data and updates the GloveState singleton.

```
1  package OSHandler;
2
3  import com.fazecast.jSerialComm.SerialPort;
4  import com.fazecast.jSerialComm.SerialPortEvent;
5  import com.fazecast.jSerialComm.SerialPortMessageListener;
6
7  public final class SerialPortListener implements
       SerialPortMessageListener {
8
9      private final GloveState gloveState;
10     private boolean captureEvents = false;
11
12     public SerialPortListener(GloveState gloveState) {
           this.gloveState = gloveState; }
```

```
13
14      @Override
15      public int getListeningEvents() { return
            SerialPort.LISTENING_EVENT_DATA_RECEIVED; }
16
17      @Override
18      public byte[] getMessageDelimiter() { return new byte[] {
            (byte) 0x0A }; }
19
20      @Override
21      public boolean delimiterIndicatesEndOfMessage() { return true; }
22
23      @Override
24      public void serialEvent(SerialPortEvent event) {
25          byte[] receivedData = event.getReceivedData();
26          String input = new String(receivedData).replace("\n", "");
27          this.gloveState.updateState(input.split(","));
28      }
29  }
```

Listing 8: Serial Port Listener Code

5.4.2 SerialComms

SerialComms is a runnable object, visible in Listing 9 which manages the connection to Arduino. If found, the SerialPortListener object will be added to the arduinoPort , otherwise an exception will be raised.

```
1  String deviceName = "Feather M0";
2
3  SerialPort arduinoPort = null;
4  for (SerialPort commPort : SerialPort.getCommPorts()) {
5      if (commPort.getDescriptivePortName().contains(deviceName)) {
```

```
6           arduinoPort = commPort;
7           commPort.openPort();
8           break;
9       }
10  }
11
12  if (arduinoPort == null)
13      throw new FileNotFoundException(String.format("Could not find
            device \"%s\".", deviceName));
14
15  arduinoPort.addDataListener(serialPortListener);
```

Listing 9: SerialComms Code

5.5 Main GUI

Since we are making use of JavaFX, each window is broken down into two components, a controller and a corresponding FXML file. Similarly to HTML, the FXML file outlines the layout of the window with all functionality being defined in the controller. For example, in Listing 10 is the definition of a button used to launch the configuration window. We can several fields such as the position of the button which is denoted by the layoutX and layoutY parameters. Additionally in we can see fx:id and onAction parameters.

```
1  <Button fx:id="configurationButton"
2      layoutX="622.0"
3      layoutY="575.0"
4      mnemonicParsing="false"
5      onAction="#reconfigureGlove"
6      text="Configure Glove" />
```

Listing 10: FXML button

In Figures 11 and 12 is the corresponding code in the Controller class. As you can see

both of the aforementioned snippets are referenced in the FXML from Listing 10, hence they are attributed with the @FXML decorator.

```
@FXML public Button configurationButton;
```

Listing 11: Creating a button

```
@FXML
public void reconfigureGlove() throws IOException {
    this.serialComms.startCapture();

    URL url =  new
       File("src/main/resources/configurationWindow.fxml").toURI().toURL();
    Parent root = FXMLLoader.load(url);
    Stage stage = new Stage();
    stage.setScene(new Scene(root, 500, 500));
    stage.setAlwaysOnTop(true);
    stage.showAndWait();

    this.serialComms.stopCapture();

    Document maxValues = stateHandler.findMax();
    Document minValues = stateHandler.findMin();

    GloveConfiguration gloveConfiguration = new
       GloveConfiguration();
    gloveConfiguration.setMinValues(minValues);
    gloveConfiguration.setMaxValues(maxValues);

    gloveConfigurationHandler.updateAll(gloveConfiguration);
    populateGloveConfig();
}
```

Listing 12: Glove reconfiguration code

5.6 Database

While MongoDB is a flexible database, we wanted it to enforce some standard when adding data to it. This was achieved by breaking up each type of data into two separate classes, a data object and its associated handler. Each handler is an implementation of the DBHandler interface.

```
1  interface DBHandler {
2
3      MongoCollection<Document> collection = null;
4
5      void addEntry(Document doc);
6
7      List<Document> findAllEntries();
8      Document findEntry(String query) throws NoSuchElementException,
           AccessException;
9      Boolean containsEntry(Document query);
10     void deleteEntry(String query);
11 }
```

Listing 13: Example database handler

Take the example of the Application data object and the Application Handler in Listing 14. Below we have the implementation of the Application class. It is a simple class which merely acts to store data and provide some assurances regarding if the supplied paths are valid.

```
1  public class Application {
2
3      private String name;
4      private String path;
5
6      public Application(Document doc) throws FileNotFoundException,
           AccessException {
7          this.name = (String) doc.get("name");
8          setPath((String) doc.get("path"));
9      }
10
11     public Application(String name, String path) throws
           FileNotFoundException, AccessException {
12         this.name = name;
13         setPath(path);
14     }
15
16     public Application(String name) { this.name = name; }
17
18     public String getName() { return this.name; }
19
20     public String getPath() { return this.path; }
21
22     public void setName(String name) { this.name = name; }
23
24     public void setPath(String path) throws FileNotFoundException,
           AccessException
25     {
26         if (!verifyPath(path))
27             throw new FileNotFoundException(String.format("The file
                   \"%s\" does not exist", path));
28         if (!verifyExecutable(path))
```

```
29              throw new AccessException(String.format("The file
                    \"%s\" is not an executable.", path));
30          this.path = path;
31      }
32      public static boolean verifyPath(String path) {
33          File file = new File(path);
34          return file.exists();
35      }
36
37      public static boolean verifyExecutable(String path) {
38          File file = new File(path);
39          return file.canExecute();
40      }
41  }
```

Listing 14: Implementation of the Application class

In Listing 15 we have the corresponding interface with the MongoDB database. The benefits of this implementation is that we can be enforce rules on what data is entered into the database, as well as provide methods for commonly used operations.

```
1  public class ApplicationHandler implements DBHandler {
2      private final MongoCollection<Document> collection;
3
4      public ApplicationHandler(String url, String collectionName) {
5          MongoClient mongoClient = new MongoClient(new
               MongoClientURI(url));
6          MongoDatabase database =
               mongoClient.getDatabase(collectionName);
7          collection = database.getCollection("applications");
8      }
9
10     @Override
11     public void addEntry(Document query) {
12         if (!isValidFile(query))
13             throw new IllegalArgumentException("");
14
15         collection.insertOne(query);
16     }
17
18     @Override
19     public List<Document> findAllEntries() {
20         List<Document> apps = new ArrayList<>();
21         FindIterable<Document> iterable = collection.find();
22         for (Document document : iterable) apps.add(document);
23         return apps;
24     }
25
26     @Override
```

```
27      public Document findEntry(String query) throws
           NoSuchElementException, AccessException {
28          FindIterable<Document> docs =
               collection.find(Filters.eq("name", query));
29          Iterator<Document> doc = docs.iterator();
30          return doc.next();
31      }
32
33      @Override
34      public Boolean containsEntry(Document query) {
35          try {
36              this.findEntry(query.getString("name"));
37              return true;
38          } catch (NoSuchElementException | AccessException ex) {
39              return false;
40          }
41      }
42
43      @Override
44      public void deleteEntry(String name) {
45          collection.deleteOne(Filters.eq("name", name));
46      }
47      ...
48  }
```

Listing 15: MongoDB interface

5.7 Macro Execution

Macro Execution was achieved using the Robot object in the AWT package which came as part of the standard library. Keyboard Macros defined by the user are stored with as a squence of the state of the key, i.e. presses or released, followed by the keycode defined by Java AWT. In Listing 16 is a macro which when executed, changes window using alt + tab.

```
press,18
press,9
release,9
release,18
```

Listing 16: Keycodes for alt+tabbing

This is achieved by the code in Listing 17. We start off by parsing the key sequence. If the key sequence is valid, each step is executed using the aforementioned Robot object. A key sequence is considered invalid if an action is not equal to ”press” or ”release”, or if a pressed key is not released.

```
1  public void executeKeySequence(String[] keySequence) {
2      String key;
3      String action;
4
5      if (!parseKeySequence(keySequence))
6          throw new InvalidParameterException("The provided key
               sequence failed to parse successfully");
7
8      for (String line : keySequence) {
9          action = line.split(",")[0];
10         key = line.split(",")[1];
11
12         if (action.equals("press"))
```

```
13              robot.keyPress(Integer.parseInt(key));
14          else
15              robot.keyRelease(Integer.parseInt(key));
16      }
17  }
18
19  public boolean parseKeySequence(String[] keySequence) {
20      ArrayList<String> keys = new ArrayList<>();
21      String key;
22      String action;
23
24      for (String line : keySequence) {
25          action = line.split(",")[0];
26          key = line.split(",")[1];
27
28          if (!action.equals("press") && !action.equals("release"))
29              return false;
30
31          if (action.equals("press")) {
32              keys.add(key);
33              continue;
34          }
35
36          if (keys.contains(key)) {
37              keys.remove(key);
38          }
39      }
40
41      return keys.size() == 0;
42  }
```

Listing 17: Macro execution

5.8 Glovesy Demo Hub

Glovesy Demo Hub is a application built using unity which allows the user to use glovesy to interact with a virtual environment.

Upon creating the environment, the first step was to create a virtual hand within said environment as is show in Figure 10. Empty game objects named pivots are used in order to act as a point for the finger segments to rotate around, essentially simulating a knuckle.

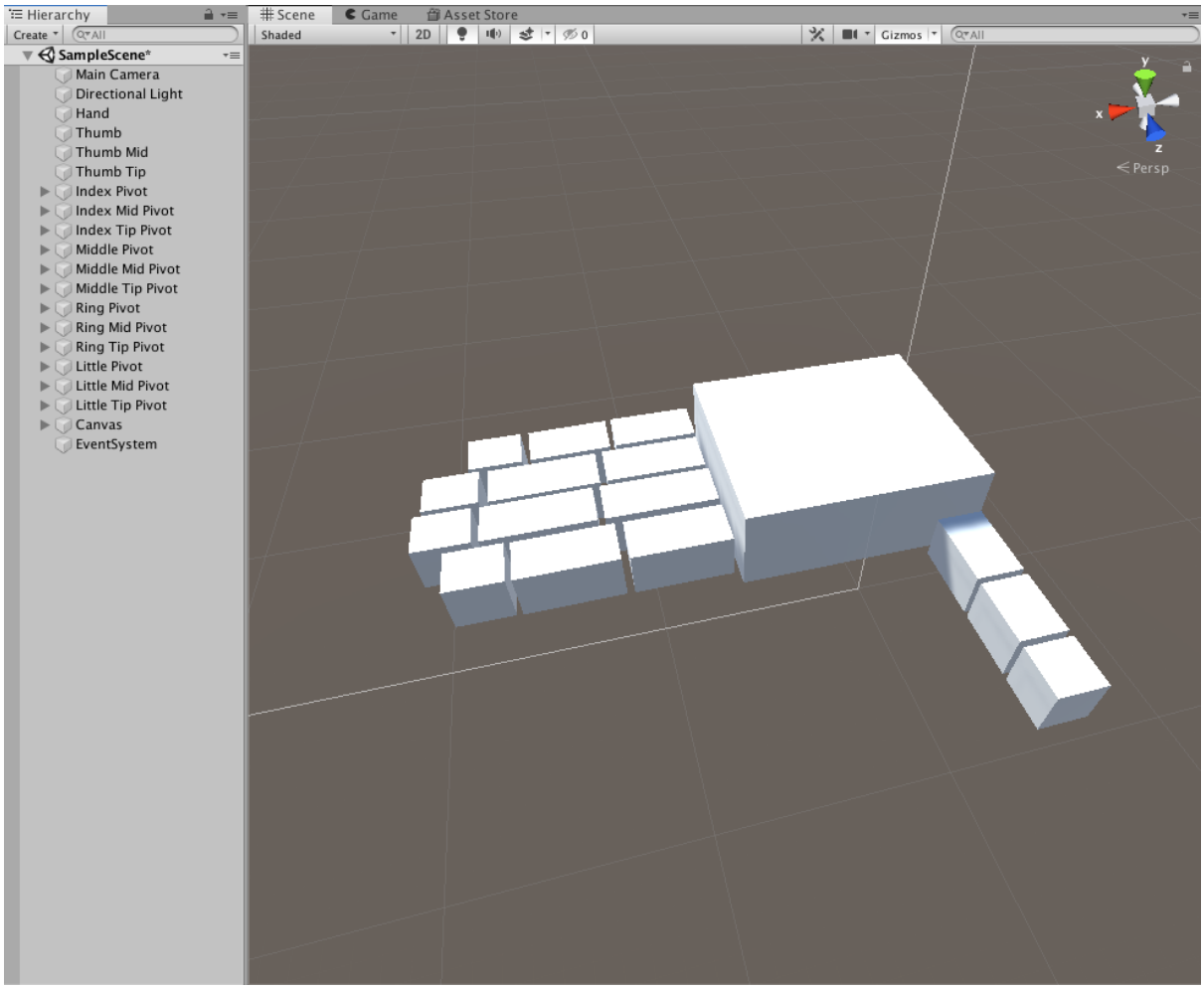

Figure 10: Unity environment with constructed hand

Once the hand was constructed within the scene, a script had to be written to be able to read data from the serial port. This code can be found in Listings 18 and 19. The serial reader works by creating a StreamReader which is pointed to the device file path, and constantly reads the string of comma seperated values sent by glovesy, and converting them to an array of floats.

```
1  StreamReader reader;
2  string path = "/dev/ttyAMC0";
3  string GlovesyData;
4  float[] ParsedData = {0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,0.0f,
5                        0.0f,0.0f,0.0f,0.0f,0.0f,0.0f};
6  // Start is called before the first frame update
7  void Start() {
8      init_serial(path);
9      // Read Serial Port for glovesy data
10     InvokeRepeating("ReadSerial",0f,.001f);
11 }
12 void init_serial(string path) {
13     try {
14         reader = new StreamReader(path);
15         return;
16     } catch (FileNotFoundException e) {
17         Debug.Log("Error: " + e);
18         return;
19     }
20 }
```

Listing 18: Code to initialize reading from serial port

```
1  void ReadSerial() {
2      try {
3          GlovesyData = reader.ReadLine();
```

```
4           if (GlovesyData.Length > 0) {
5               string[] tmp = GlovesyData.Split(',');
6               for (int i=0; i<13; i++) {
7                   float toFloat = float.Parse(tmp[i]);
8                   ParsedData[i] = toFloat;
9               }
10          }
11          return;
12      } catch {
13          return;
14      }
15  }
```

Listing 19: Code to read from serial port

When Serial data is being received, the flex sensor data can start being used to control the fingers on the hand, however, before using the values directly, they must be normalised using the calibration function found in Listing 20 as otherwise there is no reference point for full extension or retraction of each finger joint. The type returned by the calibration function is IEnumerator so that it can be run as a coroutine within unity. It calibrates by getting the minimum and maximum resistance, which are then used in Listing ?? to normalise the values to be between 1.0 and 0.0. The calibration process can be seen in Figure 11. Finally, after having been normalised, the data can be used to control the fingers within the environment. This is done by creating an invisible object which will move in an arc according to the flex sensor value associated with that object, and each section of the hand rotates around it's pivot point in order to point towards the invisible target object, causing the finger to bend. The code for this can be seen in Listing 22

```
1   IEnumerator GetMinMax() {
2       Text instruction =
            GameObject.Find("Instruction").GetComponent<Text>();
3       if (timer < 5.0f) {
4           instruction.text = "Please hold your hand open";
```

```
5           for (int i=0; i<7; i++) {
6               if (Max[i] == -1.0f || ParsedData[i] > Max[i]) {
7                   Max[i] = ParsedData[i];
8               }
9           }
10      } else if (timer < 10.0f) {
11          instruction.text = "Now make a fist as tightly as possible";
12          for (int i=0; i<7; i++) {
13              if (Min[i] == -1.0f || ParsedData[i] < Min[i]) {
14                  Min[i] = ParsedData[i];
15              }
16          }
17      } else {
18          instruction.text = "";
19          CALIBRATED = 1;
20      }
21      yield return null;
22  }
```

Listing 20: Calibration code

```
1   void Normalize() {
2       for (int i=0; i<7; i++) {
3           float Normalized = (ParsedData[i] - Min[i]) / (Max[i] -
                Min[i]);
4           // Suppress outliers
5           if (Normalized > 1.0f) {
6               Normalized = 1.0f;
7           } else if (Normalized < 0.0f) {
8               Normalized = 0.0f;
9           }
10          NormalizedValues[i] = Normalized;
11      }
```

```
12      return;
13  }
```

Listing 21: Normalisation of flex sensor data

```
1  // Update is called every frame
2  void Update() {
3      Normalize();
4
5      // Set Target Position based on data from flex sensors
6      IndexTarget.transfrom.position = IndexTargetPos + new Vector3(
7                                           -1f+NormalizedValues[2],
8                                           -1f+NormalizedValues[2],
9                                           0f);
10
11     // Point Finger towards target object
12     IndexPivot.transform.LookAt(IndexTarget.transform.position,
13                                 new Vector3(0f,1f,0f));
14 }
```

Listing 22: Code to move fingers

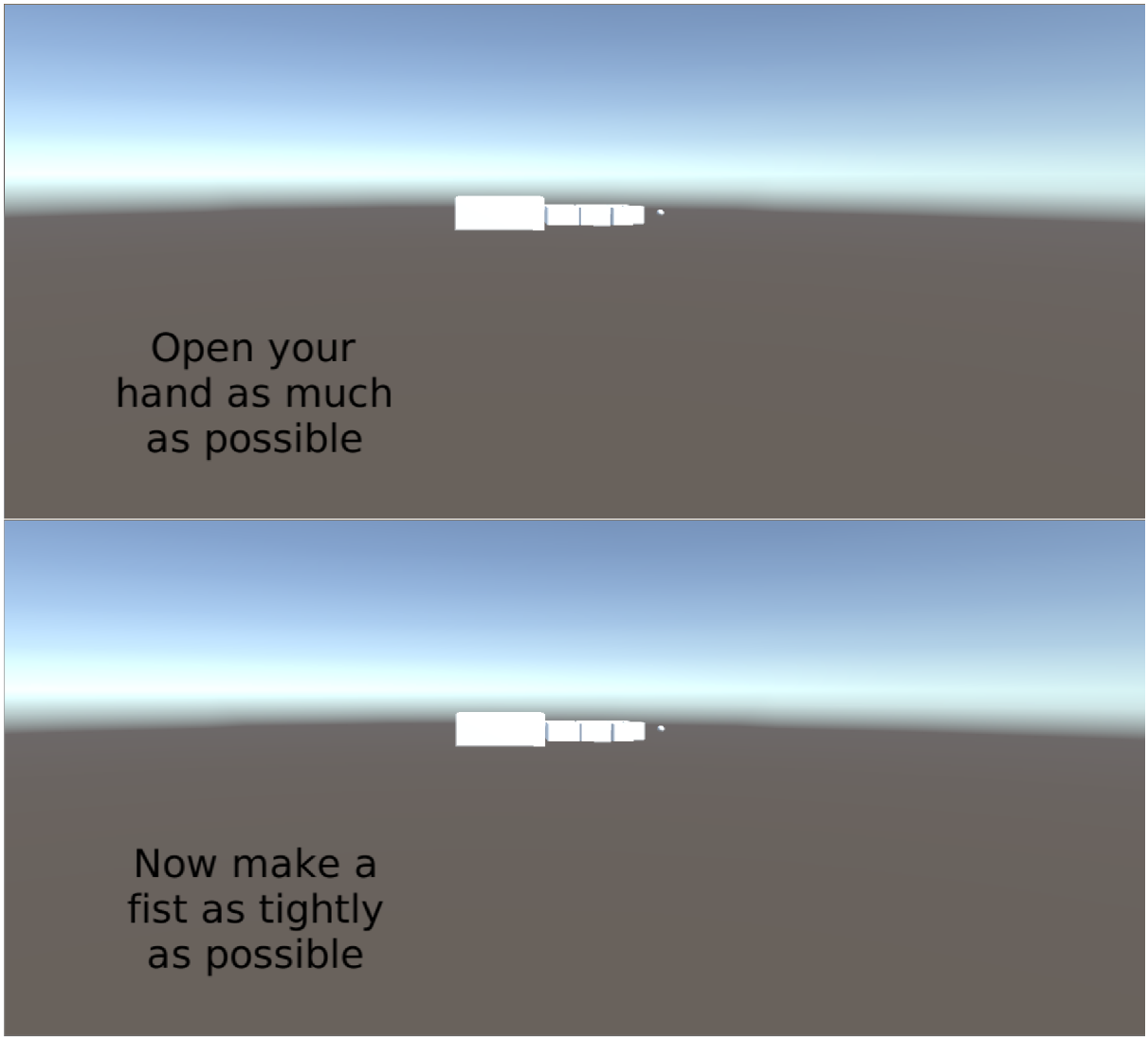

Figure 11: Calibration process

Problems and Solutions

6.1 Reading Serial Data worked in a main function, but not when integrated

When implementing the serial data, I worked in a main function within SerialComms.java. Once it was working standalone, I tried to integrate it with the rest of the system only to find that it would read once, then stop altogether. After a bit of research it came to my attention while the library we used for reading serial data, JSerialComm, did create and start a thread, the thread would join after one serial event. The solution to this was to make SerialComms.java an implementation of the Runnable interface.

6.2 Z-Axis joint contraction

When Implementing joint contraction, initially, there was an issue were fingers would not rotate on the Z-Axis. This is obviously a huge issue as for the most part fingers only Z-Axis, or, the axis normal to palm of the hand. The issue stemmed from JavaFX3D having two ways of moving objects on screen, setLayout and setTranslate. Layout is derived from the strictly GUI side of the package. This was fixed by changing Layout to Translate.

6.3 Not enough analog pins

While initially constructing the glove and its flex sensors, we both noticed that the flex sensor being used for measuring thumb movement wasn't returning any data. After

making a number of new flex sensors, thinking it was a problem with its construction, Alan realised that due to our inexperience in reading pinout diagrams, we failed to notice that the extra analog pins that we thought were usable were actually reserved for use by the Bluetooth module of the board. This meant that rather than having 10 usable analog pins as we thought, we only had 7. This was a problem since we designed the glove to use 8 flex sensors. In order to address this we removed one of the flex sensors from the ring finger, rather than removing the pinky finger. We chose the ring finger since it's less motile that the middle or index finger, and is also less predominant when gripping objects.