

UNIVERSITY OF OXFORD

Hand Gesture Recognition via Leap Motion Sensor

Authors:
Jahangir IQBAL

Supervisor:
Dr. Irina VOICULESCU

*A thesis submitted in fulfillment of the requirements
for the degree of Masters in Computer Science
in the*

Department of Computer Science

August 29, 2017

Declaration of Authorship

I, Jahangir IQBAL, declare that this thesis titled, “Hand Gesture Recognition via Leap Motion Sensor” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

University of Oxford

Abstract

Faculty Name
Department of Computer Science

Masters in Computer Science

Hand Gesture Recognition via Leap Motion Sensor

by Jahangir IQBAL

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

Declaration of Authorship	iii
Abstract	v
Acknowledgements	vii
1 Introduction	1
1.1 Motivation	1
1.1.1 Goals	1
2 Background	3
2.1 Leap Motion	3
2.1.1 Leap SDK v2 Java API	3
2.2 JavaFx	5
2.2.1 Java vs FXML	5
2.2.2 Scene Builder	8
2.2.3 Jfoenix Library	9
3 Constructing Hand UI Model	11
3.1 Basic 3D Modeling	11
3.2 Set Location Method	12
3.3 Concurrency	13
3.3.1 JavaFX Concurrent Package	14
3.3.2 Project Application	15
4 Rotation of the Hand UI Model	17
4.1 Description of Problem	17
4.2 JavaFx Coordinate System vs Leap Motion Coordinate System	19
4.3 Unexpected Leap Motion API Results	20
4.3.1 Pitch, Roll, Yaw in Leap API	20
4.3.2 Negative Zeros and Flipped Axes	21
4.4 Simplified Hand Model	23
4.5 Composite Rotational Transformations	24
5 Scoring of Gestures	29
5.1 Angle Based Comparison Function	29
5.2 Component Based Comparison Function	31
6 Application User Interface	37
6.1 Main Layout	37
6.1.1 Creating or Loading User	37
6.1.2 Saving User Data	38
6.1.3 Visual Rotation of Gesture	39
6.1.4 Adding a New Gesture	41

6.2	Tabular Display of Data	42
6.3	Writing and Reading from CSV	44
6.4	Artifacts and Distribution	44
7	Data Collection	47
7.1	The Approach Taken	47
7.2	User Feedback	47
8	Results and Analysis	49
9	Conclusion	51
	Bibliography	53

List of Figures

2.1	Leap Motion interaction area	3
2.2	Leap Motion Sample	4
2.3	JavaFX HelloWorld	5
2.4	UI Layout	6
2.5	JavaFX Scene Graph	6
2.6	FXML example	7
2.7	Controller for FXML	7
2.8	JavaFX Application Output	8
2.9	FXMLLoader	8
2.10	Scene Builder	9
2.11	External Libraries	10
3.1	Hand Bone Model	11
3.2	JavaFx Group Node	12
3.3	UIHandSimple Constructor	13
3.4	setRotationByVector Method	14
3.5	Task Class Code	15
3.6	Passing Task as Parameter	16
4.1	Hand in Flat Orientation	17
4.2	Hand with Yaw Rotation	18
4.3	Hand with Roll Rotation	18
4.4	Hand in Weird Handshake Position	18
4.5	Hand Fixed to Vertical Orientation	18
4.6	Leap Motion Coordinate System	19
4.7	Left vs Right Hand Coordinate System	19
4.8	JavaFX Coordinate System	20
4.9	Pitch	21
4.10	Yaw	21
4.11	Roll	21
4.12	Pitch() Debugging	22
4.13	Roll Debugging	22
4.14	getPitch() Function	23
4.15	UIHandSuperSimple Hand	24
4.16	fixRotations() Function	25
4.17	After First Yaw	25
4.18	After Pitch	26
4.19	After Final Yaw	26
4.20	matrixRotateNode() Function	27
4.21	Single Rotation Matrices	27
4.22	Composite Rotational Matrix	28
4.23	Rotational Angle and Axis Formulas	28

5.1	Bone Weights in Angular Comparison Function	29
5.2	Angular Comparison Function	30
5.3	angleIndexDistal() Function	31
5.4	compareAngles() Function	31
5.5	Gesture showing different finger poses.	32
5.6	Same gesture after a 90 degree rotation.	32
5.7	Finger Pose Mapping	32
5.8	Component Based Comparison Function	33
5.9	gradeFinger() Function	33
5.10	straightnessOfFinger() and curvednessOfFinger()	34
5.11	getThumbScore() Helper Function	35
6.1	Home Screen Layout	37
6.2	Pop-up window showing options to create or select user.	38
6.3	Pop-up window showing the previously created users.	38
6.4	Data Collection Scene	38
6.5	Save Gesture Dialog	39
6.6	Rotation Button in Action	40
6.7	Rotation Animation Timeline	40
6.8	Rotation Transform on Camera	41
6.9	Saving New Gesture	41
6.10	Analyze Data Scene	42
6.11	TreeTableView Editable Results Column	43
6.12	Editing Table Entries	44
6.13	Load Data From CSV	44
6.14	Build Project Artifact	45
6.15	Application Batch Script	45
6.16	JAR Application Folder Structure	46

For/Dedicated to/To my...

Chapter 1

Introduction

this is the introduction...

1.1 Motivation

apraxia

1.1.1 Goals

Chapter 2

Background

2.1 Leap Motion

Leap Motion controller, developed by Leap Motion Inc, US based company located in San Francisco, is a small infrared-enabled sensor that can be attached to one's computer via a USB cable. It comes with its own software that allows it to detect a user's hand movements in 3D space without any physical touch. It also can detect simple tools being held in the hand such as a pencil. The Leap Motion controller accomplishes this via two cameras and three embedded infrared LEDs which are able to track infrared light with wavelength outside that of the visible light spectrum. The sensor is able to detect motion in a wide space of around 8 cubic feet of area around it. This interaction area can be visualized as an inverted pyramid emanating from the Leap Motion sensor with a height, width and length of 2 cubic feet; as shown in Figure 2.1.

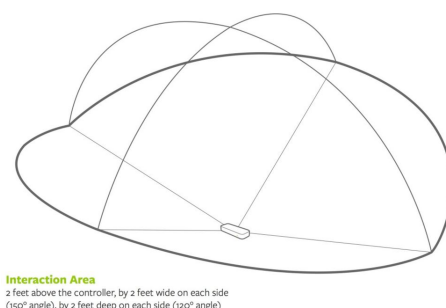


FIGURE 2.1: The area of user interaction for the Leap Motion Device.

2.1.1 Leap SDK v2 Java API

When the Leap Motion software is installed on one's computer, it does not include the libraries needed in order to develop applications utilizing the controller. In order to do that, the appropriate SDK version of the software must be downloaded from Leap Motion's website. The Java SDK version includes a JAR file and two native library files. The JAR file contains the Leap Motion API class definitions and the native libraries are OS platform specific files which allow a program using the Leap Motion API to communicate to the underlying service (Windows) or daemon (Linux or Mac) which is running the Leap Controller. Setting up a Java project requires that the distributed JAR file be set to the project's classpath. When running such a project, the JVM's library path parameter must be set to the location of the native libraries. A further discussion into the technicalities of setting up a Leap Motion Project in the IntelliJ IDE is carried out in Appendix A.

In this paragraph the data model that Leap Motion uses to represent the raw data received from device's cameras will be discussed. This data model consists of Frame objects that are continuously taken at a set rate of 60fps. These Frame objects contain all of the tracked data the Leap Motion sensor recorded in its field of view at a certain instance in time. The objects that the device keeps track include the two hands, their fingers and arm positions as estimated by normal human proportions. There are corresponding Java classes to represent these objects, such as the Hand, Finger, and Arm class. There is also a Bone class to represent specific types of bones in a hand. The Leap Motion Hand model is able to provide information about the identity, position, direction, rotation angles and other characteristics of the detected hand that it represents. This Hand model also contains methods which allow one to access other model objects contained within it; for example the fingers or the arm. Leap Motion software uses an internal hand model of the human hand to assist it in making predictions about the positions of certain parts of the hand even if they are not visible to the infrared cameras and thus not able to be calculated from the tracking data. The API also provides a Vector class that allows for useful math operations involving vectors such as finding the distance, dot product or cross product between them.

To understand the Leap Motion Java API more clearly, a very simple example will be presented that shows how the Frame data can be received from the device. Firstly, it is assumed that the project has been set up with the correct classpath for the Leap Motion Java SDK JAR file and run time parameters pointing to the appropriate native libraries. Figure 2.2 shows the basic set-up required to receive data from the controller using the Leap Java API.

```

1  import com.leapmotion.leap.*;
2
3  //Listener class which handles various events for the Controller
4  class SampleListener extends Listener {
5      //method to handle the event of a Frame received from Controller
6      public void onFrame(Controller controller) {
7          // Get the most recent frame and report some basic information
8          Frame frame = controller.frame();
9          System.out.println("Frame id: " + frame.id() + ", hands: " +
              frame.hands().count()
10     }
11 }
12
13 class Sample {
14     public static void main(String[] args) {
15         // Create leap motion controller instance
16         Controller controller = new Controller();
17
18         // Have the sample listener receive events from the controller
19         controller.addListener(new SampleListener());
20
21         // Remove the sample listener when done
22         controller.removeListener(listener);
23     }
24 }

```

FIGURE 2.2: This code sample shows how to connect to and receive data from the Leap Motion controller device.

2.2 JavaFx

JavaFX is a framework provided by Oracle Corporation that is intended to replace the Swing framework as the standard GUI library for developing desktop applications that can be run on any platform that supports Java. Since the JavaFX 2.0 release, JavaFX application can be written in pure Java code. Before that release, applications written using JavaFX libraries had to be written in JavaFX Script, a scripting language designed and used specifically for the purpose of creating GUI applications with JavaFX. This project uses the latest version of this framework, JavaFX 8, which also added support for 3D graphics and sensor support.

2.2.1 Java vs FXML

When writing a JavaFX application, there are two very different approaches that can be used to create the actual user interface (UI) for the application. These are pure Java code and FXML. A brief introduction both of these approaches will be given below.

The pure Java approach constructs the JavaFX application scene graph procedurally through code. Below is a simple “Hello World” program that shows a quick example of this approach in action. This small snippet of code contains the overall

```
1 public class HelloWorld extends Application {
2     public static void main(String[] args) {
3         launch(args);
4     }
5
6     @Override
7     public void start(Stage primaryStage) {
8         primaryStage.setTitle("Hello World!");
9         Button btn = new Button();
10        btn.setText("Say 'Hello World'");
11        btn.setOnAction(new EventHandler<ActionEvent>() {
12            @Override
13            public void handle(ActionEvent event) {
14                System.out.println("Hello World!");
15            }
16        });
17
18        StackPane root = new StackPane();
19        root.getChildren().add(btn);
20        primaryStage.setScene(new Scene(root, 300, 250));
21        primaryStage.show();
22    }
23 }
```

FIGURE 2.3: A simple hello world program using JavaFX written using just Java code.

structure and all of the basic components of a JavaFX application. The first point to note is that the main class which will run the JavaFX application must extend from the abstract base class called `javafx.application.Application` and implement its abstract `start()` method. The `start()` method serves as the main entry point for all JavaFX applications. In the application’s `main()` method, which is common to all

Java applications, a call must be made to the `launch()` method which is a method defined in the base `Application` JavaFX class that actually launches the application in doing so makes a call to the `start()` method.

The `start()` method receives a parameter of type `Stage` which serves as the primary `Stage` object for the application. `Stage` is the top-level user interface container object used by JavaFX to house the whole application. In colloquial terms it can be considered to be the “window” object of the whole application. The `Stage` object has a `setScene()` method which requires a `Scene` object to be passed in. `Scene` class is the container of current content being displayed by the application. An application can have multiple scenes which display different pages of the application. In JavaFX, the actual UI components, such as the `StackPane` layout `Node` shown in the simple `HelloWorld` example above, must be added to the `Scene` object in order for them to be displayed. This relationship between the `Stage`, `Scene` and “root” `Node` components of a JavaFX application is depicted in Figure 2.4.

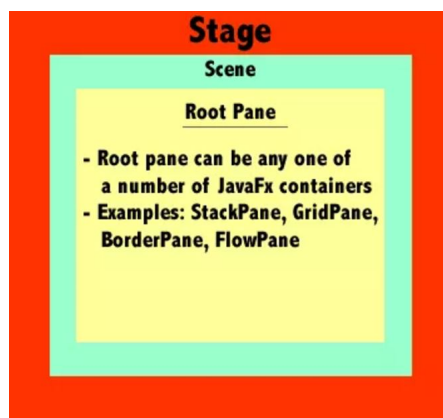


FIGURE 2.4: The layout components of every JavaFX application.

The UI components all extend from a parent `Node` class. One of the improvements that JavaFX brought in regards to its predecessor `Swing`, is that it represents the entire content of the scene as a tree. More specifically, all of the nodes displayed in a `Scene` container object must extend from a root level node and be part of the hierarchical scene graph of nodes. Figure 2.5 displays an example of the JavaFX scene graph.

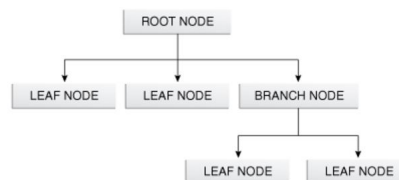


FIGURE 2.5: The Scene Graph architectural model for UI components in JavaFX applications.

Now that the pure Java code approach of writing JavaFX applications has been introduced, let us discuss the other approach to writing JavaFX applications; `FXML`. `FXML` is an XML-based language created by Oracle Corporation for the purpose of making it easier to define the UI of a JavaFX application. While the pure Java code approach is a much more imperative and procedural way to write JavaFX applications, `FXML` is a more declarative way. It resembles `HTML` and can also reference

a controller java class which can access and modify the UI elements defined in the FXML file. Figure 2.6 shows a very simple FXML file that creates a VBox layout and adds a button to it. This FXML file also has a Java controller class attached to it which is called MyController.java and shown in Figure 2.7

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <?language JavaScript?>
3 <?import javafx.scene.control.*?>
4 <?import javafx.scene.layout.*?>
5
6 <VBox fx:id="myView" layoutX="10.0" layoutY="10.0"
   xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/2.2"
   fx:controller="MyController">
7   <children>
8     <Button fx:id="okBtn" alignment="CENTER_RIGHT" contentDisplay="CENTER"
       mnemonicParsing="false" onAction="#printHelloWorld" text="Say Hello
       World" textAlignment="CENTER" />
9   </children>
10 </VBox>

```

FIGURE 2.6: A FXML file defining a simple layout and referencing a controller java class.

```

1 public class MyController
2 {
3     @FXML
4     private void initialize()
5     {
6         // this method runs first after all the UI components have been loaded and
6         bound.
7     }
8
9     @FXML
10    private void printHelloWorld()
11    {
12        System.out.println("hello world!");
13    }
14 }

```

FIGURE 2.7: A controller for the FXML file. The "FXML" annotation in is used to bind certain elements to Java objects in the class.

The result from both of these HelloWorld examples will be as showing in Figure 2.8. The large difference between these two approaches makes it difficult to combine them effectively, but it is possible to use them in conjunction with each other. This project takes such an approach. For the UI construction of the user's hand model, the pure Java code approach was taken. However, for the interface of the application the table construction showing all of the collected data, the FXML approach was taken.

One of the things that should be discussed is how communication can happen between different components of the application when FXML files are being used. There is a special class called FXMLLoader which is used to load FXML files and return the object graph of UI components these files contain. The FXMLLoader contains two {load()} methods one of which is a static class method and the other is

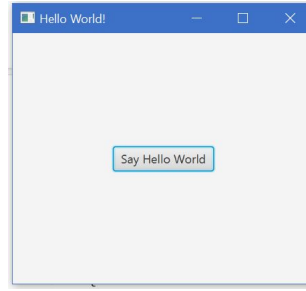


FIGURE 2.8: The output of the simple HelloWorld application.

an instance method. The important difference between these two methods is that that instance `{load()}` method can be used to gain access to the controller for the FXML class being loaded. Gaining access to the controller object for an FXML file in other other parts of an application is very important if one has to pass parameters into the view to change the interface. Figure 2.9 this key difference between the two methods.

```

1 // The approach uses the static "load" method. Not recommended
2 Parent root = FXMLLoader.load(getClass().getResource("fxml_example.fxml"));
3
4 //This approach allows one to access the controller.
5 //create instance of FXMLLoader
6 FXMLLoader loader = new FXMLLoader(getClass().getResource("fxml_example.fxml"));
7 //get the controller
8 MyController controller = loader.<MyController>getController();
9 //initialize controller with custom parameters
10 controller.initData(data);
11 //object graph root handle
12 Parent root = (Parent) loader.load();

```

FIGURE 2.9: Retrieving a reference to the controller object for a loaded FXML file.

2.2.2 Scene Builder

Scene Builder is a software that can be installed on one's computer to help design the UI and layout of a JavaFX application. It allows the user to drag and drop components from the library of available components to the central work area where they can be modified and their properties tweaked. This software is a free and open source tool that used to be developed by Oracle Corporation, but is now backed by Gluon. The easy to use drag and drop functionality of the Scene Builder allows for easy design and rapid iteration. The associated FXML code with UI created by Scene Builder can be incorporated into the JavaFX application. Using Scene Builder code bindings can also be placed on certain UI components to allow for them to execute specific logic that can be defined in the controller class of the FXML file. Figure 2.10 shows the main layout of the Scene Builder application. There are four main panels of information that have been labeled appropriately in the figure. These are: the Library panel, which contains all of the UI components available for use; the Document panel, which shows the object graph hierarchy of the current scene being built; the Content panel, which shows the work area for user interaction; and finally the

Inspector panel, which shows the various properties and layout parameters which can be modified for the currently selected UI component, as well as the various code bindings that can be placed on it.

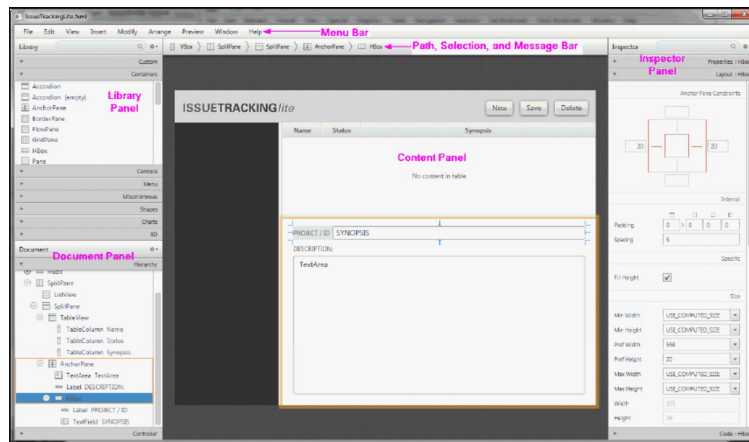


FIGURE 2.10: This shows the different areas of interest in the Scene Builder software.

2.2.3 Jfoenix Library

Jfoenix is a library that is used in this project. This is an open source Java library that implements Google Material Design for JavaFX components. This library can be included as a dependency via the Maven using the commands:

```

1 <dependency>
2   <groupId>com.jfoenix</groupId>
3   <artifactId>jfoenix</artifactId>
4   <version>1.4.0</version>
5 </dependency>

```

Maven is a build automation tool that can also serve as a package manager that makes finding and installing dependencies simple. This library will be downloaded from Maven 2 Central Repository. To include this Jfoenix library within Scene Builder, one has to find where the JAR file for the Jfoenix library was downloaded. Then, going to Scene Builder, clicking on the gear icon in the Library panel will show a drop-down menu which has the option for "Import JAR/FXML File" as depicted in Figure 2.11. Selecting this option will allow for the Jfoenix JAR file to be specified and the custom components to be loaded into the Scene Builder.

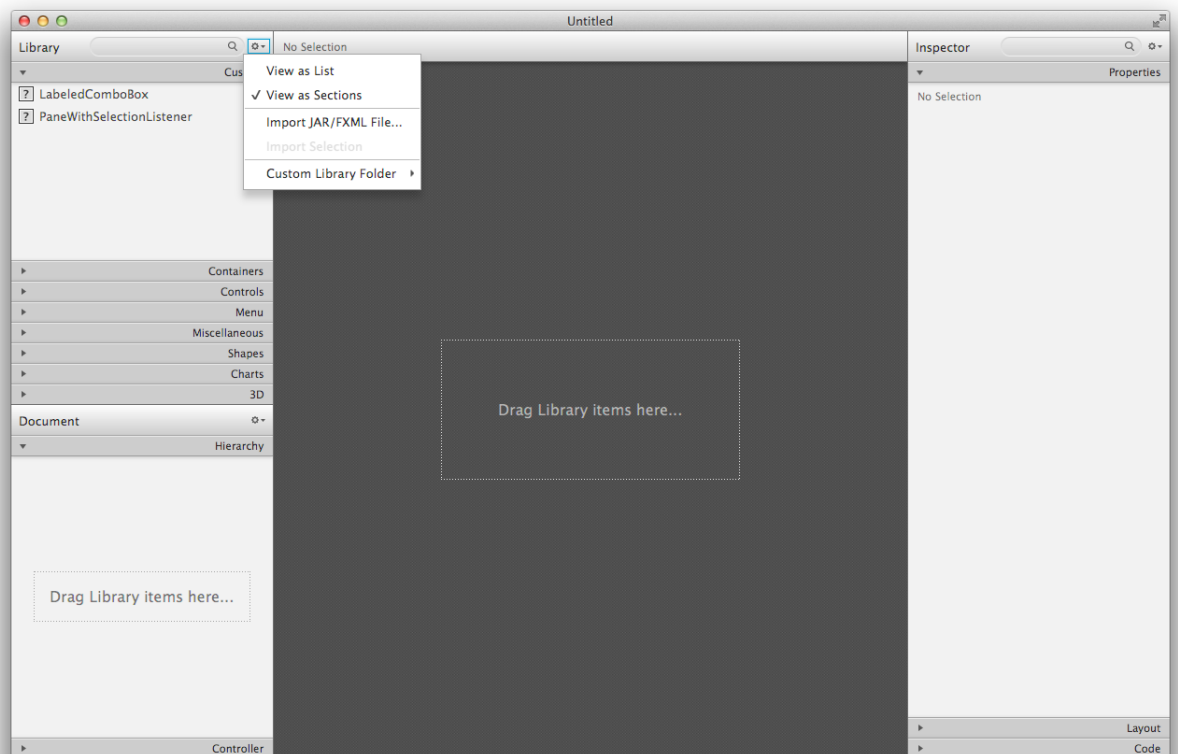


FIGURE 2.11: This is how to add a external library of custom components to the Scene Builder for UI prototyping.

Chapter 3

Constructing Hand UI Model

3.1 Basic 3D Modeling

The Leap Motion Java API's Hand class contains all the possible functions one might need to use when gaining more information about the hierarchical structure of this Java object. For example, given a Hand class object "hand", we can access the fingers objects for this hand via "hand.fingers()". Each Finger object contains four Bone objects which are indexed from 0-3. The Bone class does contain an Enum Type that allows one to easily access them via their anatomical names (distal, intermediate, proximal, metacarpal) rather than just using a numerical index. In abstract terms, the Bone object is a vector of sorts and the ends of this bone vector represent the joints at which the bone attaches to its neighboring bones. These "joints" can be accessed via the prevJoint() and nextJoint() methods which respectively return a vector position of the Bone closer to the wrist and of the Bone endpoint closer to the tip of the finger. The Figure 3.1 shows the bones and joints of the hand for which the Leap Motion sensor records data.

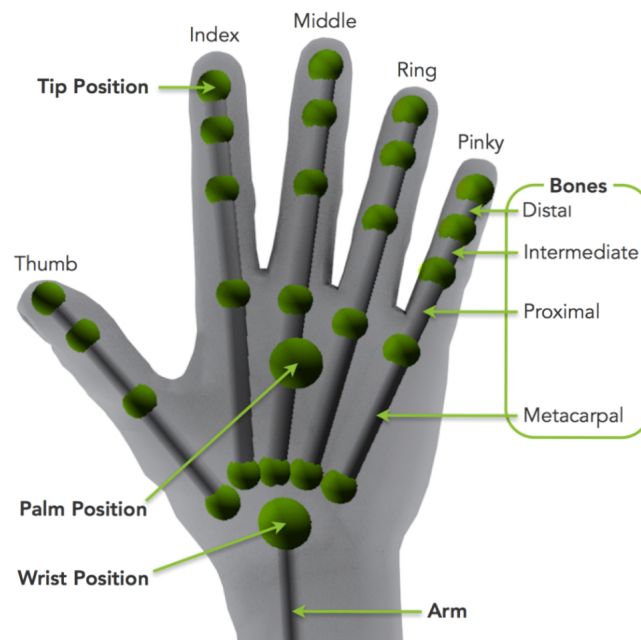


FIGURE 3.1: The Bones of the hand that Leap Motion device records data on.

A Hand object is only valid if it is detected by the Leap Motion device to be a physical object; if a Hand object is created via code using the Hand() constructor, that hand is considered "invalid" and will return true when the hand.invalid() method

is called it. The information contained within a valid Hand object read in from the device is Read Only and can not be changed or updated.

The Leap Motion device records numerical data about the hand and finger positions. Using the Hand class provided by the Leap Motion Java API and described above in the previous paragraph, a graphical model was constructed. For this GUI construction, a graphical representation of the hand was built using basic 3D geometric classes provided by the JavaFX framework. The bones of the hand model were represented by the Cylinder class and the joints were represented by the Sphere JavaFX class. This Hand model is contained inside the UIHand_SimpleJava class. This class extends a base abstract UIHand class which itself inherits from the JavaFX class called Group. Group is a type of Node in JavaFX that contains an ObservableList of children Nodes that will be added to the JavaFX Scene Graph in the order that they are added to the Group. An important point to note is that any transform, effect or property change applied to a Group will also be applied to all the children of that group. The Figure 3.2 shows an example of how a Group Node can contain multiple children nodes.

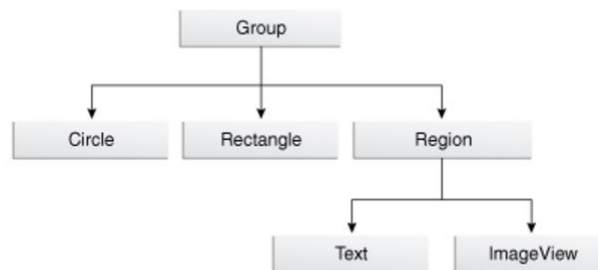


FIGURE 3.2: The Group Node structure in the JavaFX framework.

The UIHand_Simpleclass stores all the fingers bones in two dimensional array of Cylinder objects and all the respective joints in a different two dimensional array of Spheres. These arrays are of dimensions 5x3 to account for the five fingers and the three types of primary finger bones: distal, intermediate and proximal. In addition to these arrays, there is an array containing the four metacarpal bones of the hand; the thumb does not have a corresponding metacarpal bone like the other fingers. The also contains two more cylinders and a sphere to construct the palm section of the hand. To provide the hand with a uniform, well-blended color shading a PhongMaterial object is set as the hand's material property. Figure 3.3 shows the initialization of the UIHand_Simpleand part of its constructor to illustrate how the hand model was designed.

Each element of the hand, such as all the cylinders and spheres representing the various bones and joints, is added to the children of the encompassing parent group that represents the hand.

3.2 Set Location Method

The UIHand_Simpleclass also contains a method that allows for the graphical hand to be positioned according to the exact positions recorded in a Leap Motion Hand object. This method, which is called setLoc(Hand h), goes through each of the fingers and their respective bones and joints and sets the position and rotation of these these JavaFX nodes based upon the Hand object passed in. This method relies on a helper class called ViewMath which contains static methods that are called to position each

```

1 public class UIHand_Simple extends UIHand {
2     private static float fingerRadius = 14f;
3     //Hand elements
4     private Cylinder[][] fingerBones;
5     private Sphere[][] fingerJoints;
6     private Cylinder[] knuckleSpans;
7     ...
8     public UIHand_Simple(Color color, boolean wireframe) {
9         //Execute the Group constructor
10        super();
11        //Create the materials
12        PhongMaterial dark = new PhongMaterial(color);
13        PhongMaterial light = new PhongMaterial(color.brighter());
14        //Initialize Finger bones
15        fingerBones = new Cylinder[5][];
16        for (int i = 0; i < 5; ++i) {
17            fingerBones[i] = new Cylinder[3];
18            for (int j = 0; j < 3; ++j) {
19                fingerBones[i][j] = new Cylinder();
20                //set material and radius and drawMode
21                fingerBones[i][j].setMaterial(dark);
22                fingerBones[i][j].setRadius(fingerRadius /
23                    ViewMath.radiusScaleFactor);
24                if (wireframe) fingerBones[i][j].setDrawMode(DrawMode.LINE);
25            }
26        }
27        ...

```

FIGURE 3.3: A snippet of code showing how a the UIHandSimple class, representing the graphical hand, is constructed.

individual cylinder representing a bone. Two of the important methods in ViewMath are `setPositionByVector(Node n, Vector v)` and `setRotationByVector(Node n, Vector v)`. The method `setPositionByVector` sets the translate properties of the JavaFX Node passed in to the XYZ position recorded in the vector. The `setRotationByVector` method rotates the JavaFX Node passed into it by the direction which is represented by the second argument vector. This method first takes the direction and “corrects” it by flipping the z-value. This is done because JavaFX’s coordinate system has the Z-axis increasing outward from the computer screen, while Leap Motion has the Z-axis increasing into the screen. The `setRotationByVector` finds the angle of rotation finding the the angle of the passed in direction to the Y-axis. In addition to the angle of rotation, the axis upon which the rotation will occur also needs to be defined. The axis of the rotation is found by taking the cross-product between the Y-axis and the “corrected” direction. Figure 3.4 shows how rotation is set for nodes in the hand model.

3.3 Concurrency

One of the key concepts that is used in writing this project’s application was that of concurrency. Concurrency in a JavaFX application is very important as it allows for the UI of to be responsive to user interactions despite the fact that the application might also be executing other tasks in the background. In other to achieve this requirement, it is necessary to employ multi-threading so that the main application

```

1 //This method rotates a given JavaFx node to point in the direction passed in
2 public static void setRotationByVector(Node node, Vector direction) {
3     //Correct the direction to correspond to JavaFx Coordinate system
4     Vector correctedDirection = new Vector(direction.getX(), direction.getY(),
5         -direction.getZ());
6     //Find the angle of the direction to the y-axis; in degrees
7     double angle = correctedDirection.angleTo(Vector.yAxis()) * 180 / Math.PI;
8     //Find the axis of rotation by taking the cross product of the corrected
9     //direction with the y-axis
10    Point3D axis = vectorToPoint(correctedDirection.cross(Vector.yAxis()));
11    //Set the axis and angle of rotation on the Node object
12    node.setRotate(angle);
13    node.setRotationAxis(axis);
14 }

```

FIGURE 3.4: A snippet of code showing how the rotation is set for an arbitrary Node object of the JavaFX Hand Model.

thread can focus on responding to user interactions and other time-consuming tasks can be delegated to background threads. The UI in a JavaFX application is represented by the Scene Graph, which has been discussed earlier. The Scene Graph is not thread-safe and it should only be accessed and updated via the main running application thread, which is called the JavaFX Application thread. Implementing long running tasks on the JavaFX Application thread will invariably make the UI of the application unresponsive. The best practice is to avoid this problem by letting the JavaFX Application thread focus on just processing user events.

3.3.1 JavaFX Concurrent Package

One might consider implementing the Runnable interface and creating their own thread objects from scratch to employ in the multi-threaded environment required for building JavaFX applications. However, such an approach is not recommended; it can lead to unnecessary complexity and hard to debug problems such as deadlock, which is when competing threads are stuck waiting forever, and race conditions where critical data can be modified relatively simultaneously by two competing threads. Instead, it is much better to use the `javafx.concurrent` package and the classes contained within it to achieve the multi-threading required for a responsive application. The APIs provided by this package encode the best concurrent design implementations which allow for easy interaction with the UI and also ensure that such interactions happen on the appropriate thread of the program.

To somebody who is familiar with Java technologies, he/she will know that Java already provides a complete set of concurrency related libraries in the `java.util.concurrent` package. However, these APIs are designed for traditional Java Abstract Data Types (ADTs) such as Lists, Maps etc. JavaFX applications usually are dealing with observable ADTs such as `ObservableList` and `ObservableMap`. The main difference between the observable ADTs and the traditional ADTs is that the observable ADTs allow for automatic synchronization between themselves and the view components of the UI. In web development lingo this is sometimes referred to as "two-way" data binding. It means means that if the data in the view changes changes, this change is automatically propagated to the underlying data structure without requiring any work on the programmer's part. Likewise if the observable model is updated with

new data the view components will be updated to reflect that change as well. Because of this convenience observable ADTs are very much suited for use in building JavaFX applications. The `javafx.concurrent` package uses the existing APIs found in `java.util.concurrent` package and repurposes them to also take into account the observable ADTs. It also considers other constraints faced by GUI application developers such as the JavaFX Application thread and its primary role in handling UI interaction.

Broadly speaking, the `javafx.concurrent` package consists of a `Worker` interface, which provides the APIs for communication between the background worker to the UI thread, and two classes called `Task` and `Service`, both of which implement the `Worker` Interface. `Task` is a fully observable implementation of the corresponding `java.util.concurrent.FutureTask` class. Therefore, this task class is very much suited for implementing asynchronous tasks in JavaFX that can handle user interaction and respond to events executed on the UI. This ability to handle user events is further displayed by the fact that the task class implements the `EventTarget` interface.

3.3.2 Project Application

Creating a custom task requires extending the `Task` class and implementing the `call()` method. The `call()` method should contain code that only changes states which are safe to be modified from the background thread. Therefore the `call()` method cannot change the active scene graph nodes displayed on the screen as that may cause runtime exceptions. Nevertheless, since `Task` is designed to be used in GUI applications, it does have the ability to update observable data properties, change notifications for errors and cancellation of tasks, and respond to event being fired. Figure 3.5 gives an example of one of the instances from the project which used a `Task` class to perform some work in the background thread as the UI was being refreshed to the main screen again.

```
1 private static class StaticEndTask extends Task<Void> {  
2     @Override  
3     protected Void call() throws Exception {  
4         targetHand.setVisible(false);  
5         scene2Button.setVisible(true);  
6         ...  
7     }  
8 }
```

FIGURE 3.5: This shows part of the implementation of the `Task` class with `call()` method defined.

It is important to note that the `Task` class, since it implements the `java.util.concurrent.FutureTask` class, fits into the traditional Java concurrency model also. The `FutureTask` class implements the `Runnable` interface, one of the key requirements for being able to be executed as a thread. Therefore, the `Tasks` can also be used within the Java concurrency `Executor` API and also can be passed to a thread as a parameter. To see this fact in action, consider the Figure 3.6 which shows a method that gets called when the user presses a button to go back to the main screen of the application. This snippet of code shows the `Platform.runLater()` method being passed various `Tasks` as parameters. The `Platform.runLater()` method can be called from any background thread and will add the passed tasks to a queue to be executed in order at a later time on the JavaFX Application thread and then this method returns immediately to

the caller. Because of the concurrent behavior of the application, some of the methods in the application that deal with setting parameters and changing object data were initialized with the keyword "synchronized" which prevents multiple threads interfering with the same variables at the same time and generally preventing data consistency errors.

```
1 public static void endStaticTest(double finalscore, long finaltime, boolean
    success) {
2     Platform.runLater(new StaticScoreTask(finalscore, finaltime, success));
3     try {
4         Thread.sleep(5000);
5     } catch (InterruptedException e) {
6         // TODO Auto-generated catch block
7         e.printStackTrace();
8     }
9     Platform.runLater(new StaticEndTask());
10 }
```

FIGURE 3.6: This code sample from the project shows Task objects being passed successfully to a method which expects to receive Runnable objects.

Chapter 4

Rotation of the Hand UI Model

In this chapter the rotation of the Hand UI model will be discussed.

4.1 Description of Problem

The UI model of the hand is built from the Leap Motion sensor data as discussed in the chapter. This data represents the hand as it is displayed in reality above the Leap Motion controller device. Originally the representation of the hand was built from this hand without any further modifications. However, to demonstrate why this is sometimes not the ideal situation, see Figure 4.1. It shows the hand as it is displayed in reality; it is in parallel plane above the device.

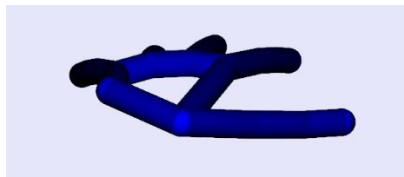


FIGURE 4.1: This shows the user's hand in a flat orientation; as determined by the Leap Motion sensor.

It should be noted how difficult it is to see the fingers and thumb. They are only barely visible. Therefore the user would be forced to force his or her hand into a vertical position by straining their wrist just so they can see the gesture they are performing on the screen. Figures 4.2 and 4.3 show other orientations the hand can take; these figures show the hand after a yaw rotation and after a roll rotation respectively. Again, note the slight difficulty in figuring out the thumb and fingers positions and orientations when the hand is rolled around the z-axis. The hand that is shown in the yaw rotation in Figure 4.2 is able to be seen a little clearly because the wrist was strained upwards.

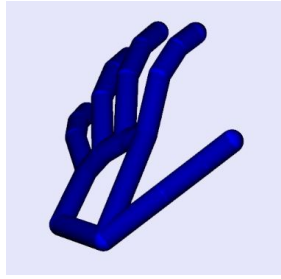


FIGURE 4.2: The user's hand after the certain yaw rotation around the y-axis.

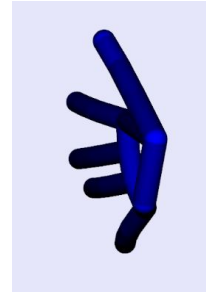


FIGURE 4.3: The user's hand after the certain roll rotation around the z-axis.

Finally all of these different orientation can of course overlap with each other to create a complex orientation of the hand as shown in Figure 4.4, which shows the user's left hand in a weird handshake sort of position while being rotated to the right and tilted upwards.

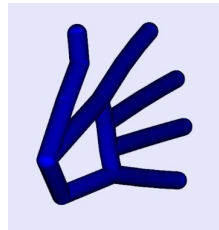


FIGURE 4.4: This shows the user's hand in a flat orientation; as determined by the Leap Motion sensor.

This chapter will explain how these different possible orientations the user's hand might be in while they are performing gestures shown on the screen are accounted for and "undone". Doing this results in the user's hand to be displayed in a set vertical orientation on the screen despite the different ways the user may have oriented their hand above the device in reality. The final resulting hand for all of the figures seen previously will be as is shown in Figure 4.5 after the composite rotation have transformed it to a vertical position.



FIGURE 4.5: This shows what the user's hand will look like after all of the possible rotational transforms have been undone and the hand has been fixed to a vertical orientation.

This feature of the application makes it easier for the user to see what his or her hand is doing without requiring them to always contain their hand in a specific orientation. The goal of this application is to measure the accuracy with which a

user is able to complete certain gestures, not the orientation of their hand. In fact, the algorithms that are used to grade the correctness of the user's attempted gesture do not take the user's hand orientation into account. They focus on the fingers bones and their relative orientations to each other. At the beginning of this project, one of the ideas discussed was to build some sort of rig which would be used to place the user's hand in a set orientation. However because of what will be explained in this chapter such a rig is no longer necessary.

4.2 JavaFx Coordinate System vs Leap Motion Coordinate System

The Leap Motion controller has a different coordinate system from JavaFX. This distinction is very important to get out of the way as the rest of the chapter will rely on such an understanding. The coordinate system for the Leap Motion device is shown in Figure 4.6. Note the placement of the green LED light in the Leap Motion device as shows the orientation of the otherwise symmetrical picture.

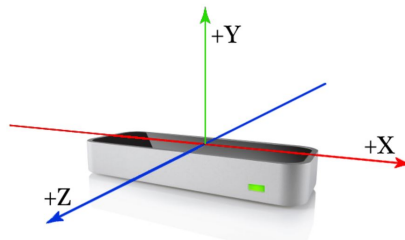


FIGURE 4.6: The coordinate system that is used by the Leap Motion sensor. Data collected will based upon these axes.

This coordinate system is referred to as a right-handed coordinate system because of the easy way in which it can be represented by the first three fingers of the right hand as shown in Figure 4.7.

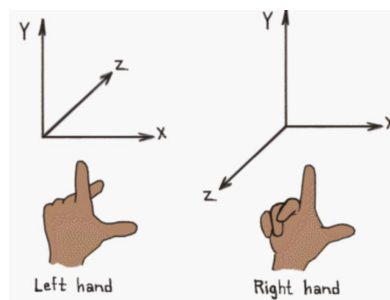


FIGURE 4.7: Left and right handed coordinate systems and how they can be shown via fingers.

In contrast, the traditional coordinate system which is used in computer science and which is what JavaFX also follows is shown in Figure 4.8. The JavaFx coordinate system is not left handed or right handed as far as we can tell from the Figures shown.

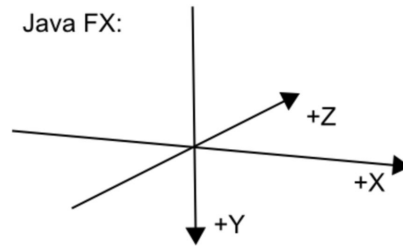


FIGURE 4.8: The coordinate system used in JavaFX. Setting transforms on objects will be based on this coordinate system.

Understanding these somewhat subtle difference and being able to switch from one context to the other was very important when considering the various rotational transforms that needed to be applied to the Hand UI model in order to straighten it to a vertical position. For example, one of the things that caused me some headaches sometimes when I was fixing and debugging my code was the fact that 3D rotations are measured in a very specific way. When it is said that some object is rotated around the y-axis by 90 degrees, this means that the if we imagine ourselves to be an observer standing on the y-axis and looking down the negative direction of this y-axis, then we would observe the object to rotate counter-clockwise about the y-axis. Therefore, it becomes very clear why it is significant to have clear understanding of the coordinate system that is being used for a particular section of code. Leap Motion data is returned with the default settings of the Leap Motion coordinate system. This data must be appropriately modified when the code using it is based upon the JavaFX coordinate system. Likewise composite rotations around certain axis might become jumbled up if one is not careful.

4.3 Unexpected Leap Motion API Results

In the course of working on this feature, which was explained in first section of this chapter, I realized that I was being returned results from the Leap Motion API that I was not expecting. I tested and debugged further into these issues until I realized some interesting things about the Leap Motion API which helped me to implement the vertical rotation of the user's hand successfully. In this section, I will go over some of the unexpected results that I obtained when I was learning more about the Leap Motion API's. I created a special Java class to help me in this endeavor which contains code samples with some simple yet illustrative scenarios which show how the Leap Motion API works.

I first noticed these problems when I was working user hand data, however, I soon realized these issues were more so with how the Vector is defined in the Leap Motion API. Therefore, the examples shown in this section will primarily be using vectors to illustrate the issues encountered, but of course these issues then extend to the collected hand data because of the nature of how Hand objects are defined in Leap Motion.

4.3.1 Pitch, Roll, Yaw in Leap API

Leap Motion's Java API includes a Vector class that contains several useful operations one might need to perform. For example, it include three functions that return the pitch, roll and yaw angles for the vector. However, right away we need to understand that these angles which are returned are with respect to certain axes. The

`pitch()` is an instance method for a vector object that will return the angle between this vector's projection onto the y-z plane and the negative z-axis. Figure 4.9 shows this. Yaw is defined to be the angle between the negative z-axis and the projection of the vector onto the x-z plane (Figure 4.10).

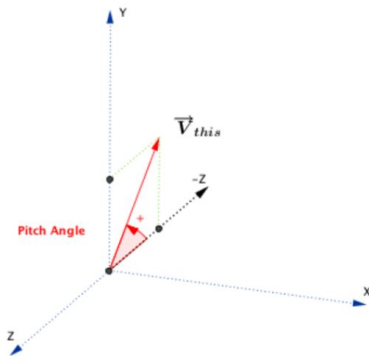


FIGURE 4.9: Pitch angle.

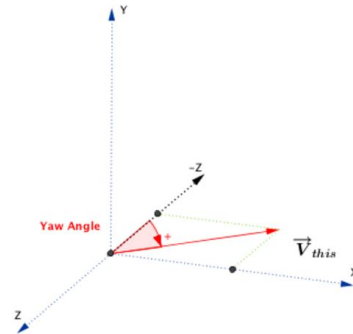


FIGURE 4.10: Yaw angle.

`roll()` function is defined to be gotten by projecting the vector in question onto the x-y plane and taking the angle between this projection and the positive y-axis (which as we know is defined to be pointing upwards in the Leap Motion coordinate system and downwards in JavaFX). See Figure 4.11 to get a visual understanding of how the roll angle is defined for a given vector.

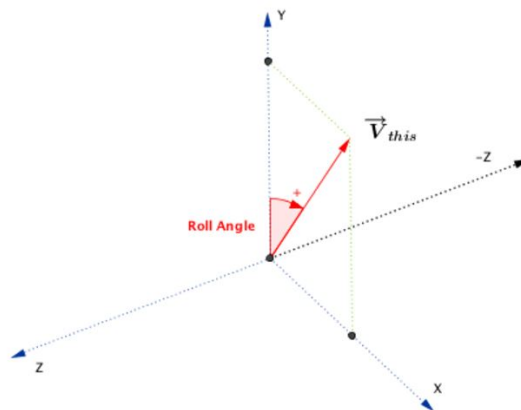
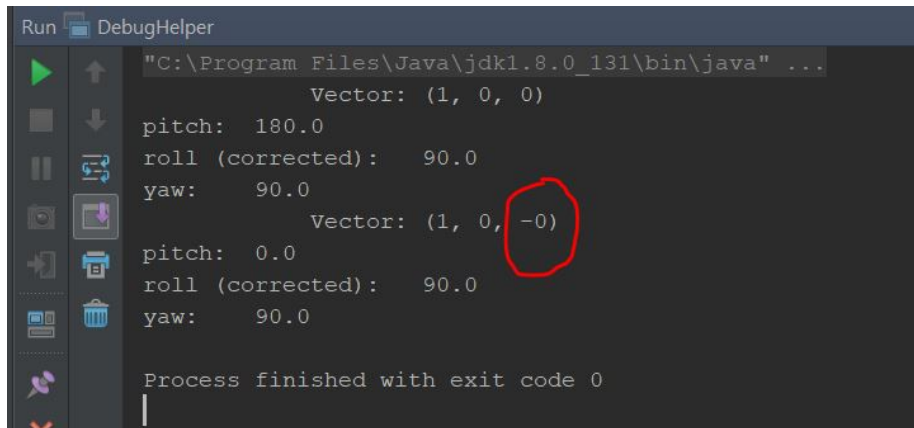


FIGURE 4.11: Roll angle as defined in Leap Motion documentation.

4.3.2 Negative Zeros and Flipped Axes

During my testing, I realized that the Vector class returns some strange results when we test the `pitch()` function with the simple x-axis unit vector $\langle 1, 0, 0 \rangle$. One would expect that the pitch would be zero, since the projection of the x-axis unit vector onto the y-z plane is zero. However, as the debugging result in Figure 4.12 shows, that was not the case.



```

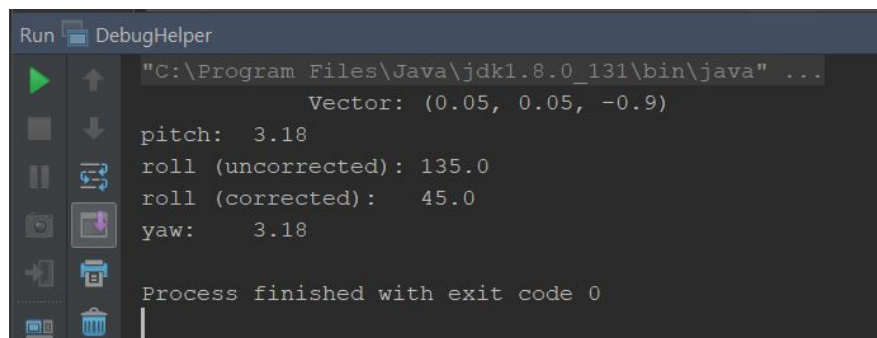
Run DebugHelper
"C:\Program Files\Java\jdk1.8.0_131\bin\java" ...
Vector: (1, 0, 0)
pitch: 180.0
roll (corrected): 90.0
yaw: 90.0
Vector: (1, 0, -0)
pitch: 0.0
roll (corrected): 90.0
yaw: 90.0
Process finished with exit code 0

```

FIGURE 4.12: A simple example's run output that demonstrates the negative zeros and their importance in Leap Motion API.

It appears to be that a negative zero must be supplied in the z-component of the vector to get a zero pitch angle. When I found this out, I was very surprised as it was the first time I realized that there was such a thing as a negative floating point zero. However, another thing to note is that in the debug output, I have the word "corrected" next to the roll angle.

Like the pitch() angle, I noticed something that was a bit strange about the results the roll() function was returning during my debugging. It was returning 135 degrees when I was expecting it to return 45 degrees (see Figure 4.15) because the vector it is being tested on makes such a projection onto the x-y plane that the roll should be 45. I think the documentation page either has a mistake the roll() method or maybe the Vector class has a error in the way the roll method is defined.



```

Run DebugHelper
"C:\Program Files\Java\jdk1.8.0_131\bin\java" ...
Vector: (0.05, 0.05, -0.9)
pitch: 3.18
roll (uncorrected): 135.0
roll (corrected): 45.0
yaw: 3.18
Process finished with exit code 0

```

FIGURE 4.13: The roll() angle returned for a vector needs to be "corrected".

The angle being returned is actually between the negative y-axis (Leap Motion coordinate system) and the projection. Therefore I wrote my own correctedRoll() which basically takes a given roll angle and subtracts it from 180 degrees to return a roll angle that is in accordance to how roll() is defined in Leap Motion documentation. In response to finding out about these problems, I wrote my own methods to find pitch, roll and yaw when I was dealing with hand objects. The code for one of these methods is shown in Figure 4.14.

```

1 private float getPitch(Hand h) {
2     //angle of hand direction to y-axis
3     float angleAmount = (float)
4         Math.toDegrees(h.direction().angleTo(Vector.yAxis()));
5     //switch direction of angle
6     if (h.direction().getZ() > 0.0f) {
7         return (-1.0f * angleAmount); //returning a -negative angle to "undo" the
8             positive pitch noticed in hand
9     } else {
10         //99 % of the time, pitch will be a positive angle, so this case will run
11             most of the time.
12         return angleAmount;
13     }
14 }

```

FIGURE 4.14: This function returns the pitch of the hand object as the angle the hand makes to the x-z plane.

This method does not rely on projections to the y-z plane to find out the pitch of the hand. Intuitively, when I imagined the pitch of the hand model to be to angle it is making to the x-z plane; regardless of which direction the wrist or the fingers of the hand are pointing in. The Leap pitch() function expects the direction of the hand to be relatively inline with the z-axis in order to give a valid pitch measurement. The method I defined does not have this restriction and is able to return an informative pitch angle no matter if the user has rotated his/hand around the y-axis. I calculate the yaw for a particular hand in the same fashion, except for that function I take the angle between the direction of the hand and the negative z-axis.

4.4 Simplified Hand Model

One of the lessons that this project has highlighted for me is how important it is to simplify a complex problem when finding a solution to it. Of course this sounds like a relatively obvious lesson, but I feel this project forced me to practice it many times. I would like to give a short example of this by talking a little bit about the UIHandSuperSimple class that I wrote when I was working on the problem this chapter addresses. The aptly named super simple hand model was a model of the user's hand that has just three components, a rectangular box to represent the palm, a cylinder to represent the thumb and a sphere to represent the fingers. See Figure ?? to view this masterpiece.

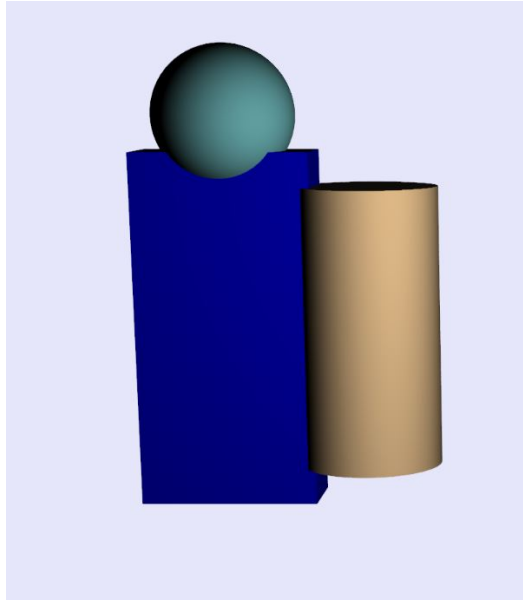


FIGURE 4.15: A very basic model of the hand.

The reason this hand was important during my journey to solving this problem was because I could easily represent different orientations in it by using just simple vectors. This allowed me to quickly debug and figure out errors in the way I was approaching things. The other Hand UI model could not be set only by a single vector in such a way that the every single bone of the hand would also be appropriately set. This was because that model relies on a lot more data from the Leap Device to build the model more accurately. The simplicity of dealing with this hand model eventually also allowed me to figure out how to undo the unwanted hand rotations for the UI display that were part of the Leap Motion data.

4.5 Composite Rotational Transformations

To understand how the UI hand model is rotated to a vertical position, we will consider the starting hand position to be the "weird handshake position" as is shown in Figure 4.4, which represents the left hand in a handshake position that is slightly pitched upwards along the x-axis and with a 45 degree yaw to the right around the y-axis. The method which will reset this user's hand to a vertical orientation automatically (without requiring the user to change the position of their hand at all) is shown in Figure 4.16.

```

1 private void fixRotations(Hand h) {
2     //add a yaw to perform before the matrixRotateNode sets the axis and angle.
3     float firstYaw = getFirstYaw(h); // dont need to convert to radians for
        Rotate transform java class.
4     if (this.getTransforms().size() == 0) {
5         this.getTransforms().add(new Rotate(firstYaw, new Point3D(0, 1, 0)));
6     } else {
7         this.getTransforms().set(0, new Rotate(firstYaw, new Point3D(0, 1, 0)));
8     }
9
10    //correct to fit Javafx coordinate system and convert to radians.
11    float pitch = (float) Math.toRadians(getPitch(h) * (-1.0f));
12    float finalYaw = (float) Math.toRadians(getFinalYaw(h) * (-1.0f));
13    float roll = 0; //roll is never needed
14
15    //order of operations: pitch, yaw, roll.
16    ViewMath.matrixRotateNode(this, roll, pitch, finalYaw);
17 }

```

FIGURE 4.16: This function fixes the rotations of the UI Hand model so that it will be always displayed in a vertical orientation, no matter what orientation the user's hand is in.

This method takes in a Leap Motion Hand object and basically performs three transformations on it. Firstly, it performs a yaw rotation of the amount determined by the `getFirstYaw()` method. The `getFirstYaw()` method determines what the angle of the hand's direction is to the negative z-axis; and so the first transform that will get added to the UI hand model is will rotate the hand about the y-axis. Figure 4.17 shows what the hand model will look like after this first rotational transform has been performed.

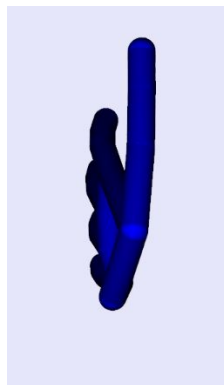


FIGURE 4.17: The hand model of Figure 4.4 after first yaw.

Secondly, the pitch angle is determined the `getPitch()` to see how much to rotate the hand upward to be inline with the y-axis. Also the finalYaw angle is determined using the `getFinalYaw()` method which will return an angle representing how much the hand model has to be rotated around the y-axis again when the hand is pointing upwards in the positive y-direction in the JavaFX coordinate system. There is no roll angle that is needed for putting the hand in a vertical position. The pitch and finalYaw angle (along with a 0 value for the roll angle) are passed into a helper function called `matrixRotateNode()`, shown in Figure 4.20. This method computes

the composite rotational matrix based upon the angles that are passed into it and performs the correct composite rotation around the calculated axis of rotation on the JavaFX Node object that is passed into as its first argument. If for a moment we consider the hand model originally started in the weird handshake position as stated above, then after the first yaw, and pitch have been performed, the hand model will be resting with its fingers pointing straight up and its palm facing the x-axis. See Figure 4.18

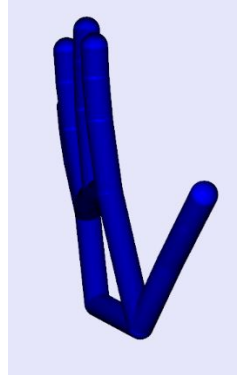


FIGURE 4.18: The hand model of Figure 4.4 after first yaw and pitch have been performed.

Finally, after the final yaw has also been performed, the hand model will be resting in the final vertical orientation with its fingers pointing up and the palm facing the negative z-axis as shown in Figure 4.19



FIGURE 4.19: The hand model after it's orientation has been fixed.

The final yaw angle is determined by which direction the palm normal of the leap motion hand is facing. This palm normal can be pointing along the x-axis or the z-axis. A weighted average of the two directions is taken to determine the angle to be computed for the final yaw.

```

1 public static void matrixRotateNode(Node n, double roll, double pitch, double
   yaw) {
2     //build rotational matrix
3     double A11 = Math.cos(roll) * Math.cos(yaw);
4     ...
5     double A32 = -Math.cos(yaw) * Math.sin(pitch);
6     double A33 = Math.cos(pitch) * Math.cos(yaw);
7
8     //angle of rotation
9     double angleOfRotation = Math.acos((A11 + A22 + A33 - 1d) / 2d);
10    if (angleOfRotation != 0d) {
11        double den = 2d * Math.sin(angleOfRotation);
12        Point3D axisOfRotation = new Point3D((A32 - A23) / den, (A13 - A31) / den,
            (A21 - A12) / den);
13        //rotate node
14        n.setRotationAxis(axisOfRotation);
15        n.setRotate(Math.toDegrees(angleOfRotation));
16    }
17 }

```

FIGURE 4.20: This function takes in a JavaFX node object and three angles representing the roll, pitch and yaw angles and returns that object after it has been rotated by appropriate final angle around the computed axis of rotation.

The code shown in the `matrixRotateNode()` in the Figure 4.20 above looks very complicated. It is the result of mathematical formulas that determine how to rotate an object in 3D space. Given three angles representing the pitch, yaw and roll we can compute a final matrix of rotation from the three smaller matrices that would represent rotations about the specific x-axis, y-axis, z-axis that the pitch, yaw, and roll angles would represent respectively. The matrices that would represent these three individual rotations are shown in Figure 4.21.

$$\begin{aligned}
 \mathbf{A}_X &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \phi & \sin \phi \\ 0 & -\sin \phi & \cos \phi \end{bmatrix} \\
 \mathbf{A}_Y &= \begin{bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{bmatrix} \\
 \mathbf{A}_Z &= \begin{bmatrix} \cos \psi & \sin \psi & 0 \\ -\sin \psi & \cos \psi & 0 \\ 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

FIGURE 4.21: The three matrices representing individual rotations about the x, y, and z axis.

Their composite matrix would be computed by $A =$

$$A_z * A_y * A_x$$

and the result of this matrix multiplication operation is shown in Figure 4.22.

$$\mathbf{A} = \begin{bmatrix} \cos \theta \cos \psi & \cos \phi \sin \psi + \sin \phi \sin \theta \cos \psi & \sin \phi \sin \psi - \cos \phi \sin \theta \cos \psi \\ -\cos \theta \sin \psi & \cos \phi \cos \psi - \sin \phi \sin \theta \sin \psi & \sin \phi \cos \psi + \cos \phi \sin \theta \sin \psi \\ \sin \theta & -\sin \phi \cos \theta & \cos \phi \cos \theta \end{bmatrix}$$

FIGURE 4.22: The result of the matrix multiplication.

The method uses this final matrix to determine the angle and axis of rotation by the formulas shown in Figure 4.23.

$$\begin{aligned} \theta &= \arccos \left(\frac{1}{2} [A_{11} + A_{22} + A_{33} - 1] \right) \\ e_1 &= \frac{A_{32} - A_{23}}{2 \sin \theta} \\ e_2 &= \frac{A_{13} - A_{31}}{2 \sin \theta} \\ e_3 &= \frac{A_{21} - A_{12}}{2 \sin \theta} \end{aligned}$$

FIGURE 4.23: The formulas which determine the angle of rotation and the axis of rotation $\langle e_1, e_2, e_3 \rangle$.

Chapter 5

Scoring of Gestures

5.1 Angle Based Comparison Function

This is the first way by which a hand gesture is scored. This scoring method compares one hand against another; namely, it is usually comparing the user's hand versus the target hand shown on the screen that the user was trying to imitate. It uses the angles between the three foremost bones, (distal, intermediate, proximal), of the five fingers as the primary means of determining how close a user's hand is to the target hand. It also uses the angle the wrist makes to the arm in its calculations to determine the final score for the hand. Figure 5.2 shows some important parts of the `compare()` function which scores the angular similarities between two hands. This function first makes sure that the two hands that are being compared are of the same kind; ie the hands must both be left or both must be right, otherwise the result of the `compare()` function will be 0. It then finds angles between adjacent finger bones in both hands and compares the two angles for the amount of similarity between them. This similarity between the two angles, which is the result of the `compareAngles()` method call, will be a number between 0 and 1. A weight applied to this similarity measure and then the result added to the summation variable `x`. This function relies on some weight parameters, see Figure 5.1, that are set higher for the longer bones that are closer to the knuckles. For example the proximal bones have a weight of 4; the intermediate bones have a weight of 2 and the distal bones have a weight of 1. This is because the bigger bones closer to the palm of the hand are a bit more limited in their mobility. Therefore, determining correlation between these corresponding bigger bones such as proximal has a higher influence on the overall value of the `compare()` function.

```
1 //weights for various bone types
2 static double weight_pinky_proximal = 4;
3 static double weight_pinky_intermediate = 2;
4 static double weight_pinky_distal = 1;
5 ...
```

FIGURE 5.1: An example of the weights set for different bone types in the pinky finger.

```

1 public double compare(Hand h1, Hand h2) {
2     //check if both hands are of the same type.
3     if (h1.isLeft() == h2.isLeft()) {
4         double x = 0;
5         //wrist
6         x += compareAngles(angleWristArm(h1), angleWristArm(h2))* weight_wrist;
7         //five fingers "proximal". compareAngles always returns between 0-1
8         x += compareAngles(anglePinkyProximal(h1),
9             anglePinkyProximal(h2))*weight_pinky_proximal;
10        x += compareAngles(angleRingProximal(h1), angleRingProximal(h2))*
11            weight_ring_proximal;
12        x += compareAngles(angleMiddleProximal(h1), angleMiddleProximal(h2))*
13            weight_middle_proximal;
14        x += compareAngles(angleIndexProximal(h1), angleIndexProximal(h2))*
15            weight_index_proximal;
16        x += compareAngles(angleThumbProximal(h1), angleThumbProximal(h2))*
17            weight_thumb_proximal;
18        //five fingers "intermediate"
19        x += compareAngles(anglePinkyIntermediate(h1),
20            anglePinkyIntermediate(h2))* weight_pinky_intermediate;
21        ...
22        //five fingers "distal"
23        x += compareAngles(anglePinkyDistal(h1), anglePinkyDistal(h2))*
24            weight_pinky_distal;
25        ...
26        x /= totalWeight();
27        return x;
28    } else{
29        //if comparing left hand to right hand (or vice versa), return 0
30        return 0;
31    }
32 }

```

FIGURE 5.2: This snippet of code shows the main skeleton of the function that determines the similarity between two hands by comparing angles between various bones in the hands.

It is also worth looking into how the `compareAngles()` function is defined as this function determines what percentage of the weights get applied. It takes in two angles as its parameters. These angles are determined via various functions which find angles between consecutive bones of a specific finger. An example of one such function is shown in Figure 5.3 which shows how the angle between the distal bone and the intermediate bone in the index finger is determined. The angle returned by these functions will always be less than 180 degrees because of the way the `angleTo()` function is defined in the Leap Motion API.

```

1 private float angleIndexDistal(Hand h) {
2     Vector direction1 =
3         h.fingers().get(1).bone(Bone.Type.TYPE_DISTAL).direction();
4     Vector direction2 =
5         h.fingers().get(1).bone(Bone.Type.TYPE_INTERMEDIATE).direction();
6     float rawAngle = direction1.angleTo(direction2); //always less than 180
7     return normalize(rawAngle, h); //flips angle on xAxis if palm facing upwards
8 }

```

FIGURE 5.3: This function is one example of how the angles between adjacent bones are determined.

The `compareAngles()` is a mathematical function that determines the similarity between two angles passed into it by using the cosine trigonometry function. It will return a number between 0 and 1. It first finds the difference between the two angles. If the angles are so far apart and the distance is greater than 45 degrees, then the function will return a zero to indicate that there is not any meaningful closeness between the two angles being compared. Figure 5.4 shows this function's code.

```

1 private double compareAngles(float angle1, float angle2){
2     double differenceBtwAngles = Math.abs(angle1-angle2);
3     //tmp can be at most pi/4 = 45
4     double tmp = Math.min(differenceBtwAngles, Math.PI/4);
5     //if tmp is exactly 45, will return 0. cos(90) = 0.
6     return Math.cos(2*tmp);
7 }

```

FIGURE 5.4: This function determines how similar (or close together) two angles using cosine.

5.2 Component Based Comparison Function

The second way by which a score is assigned to a hand representing an attempted gesture will be discussed in this section.

The idea behind this method is to take a given hand and decompose it into smaller component that can be scored individually. Then these components scores will be combined to arrive at the cumulative score for the entire hand. Each finger is seen as a component. After considering all of the gestures being tested in this project, I realized that each finger can be in one of three main kinds of poses. All of the fingers except for the thumb are only seen in some variations of being straight, or being curved. The thumb, however, has its own special kind of pose, which deals with connecting to other fingers. For example in some of the gestures the thumb is touching the pinky; in some gestures it is touching the middle finger. Therefore, the third possible pose is represented by a finger name, such as "pinky" or "index" etc., and it represents the finger the thumb is touching in the gesture being analyzed. The way the algorithm is designed, it makes sense to assign this dynamic third pose to the thumb only. Figure 5.5 and Figure 5.6 shows one of the gestures used in this project, namely `gesture9Left`, to illustrate what is meant by the different poses different components of the hand can take. As we can see, all of the fingers are straight; the ring finger is curved; and the thumb is touching the ring finger. This "pose signature" of this gesture as it is used in code is shown in Figure 5.7. The pose signature

for a certain gesture is the same regardless of whether left or right hand is being used. That is why the code sample shows two case statements for `gesture9Left` and `gesture9Right`.

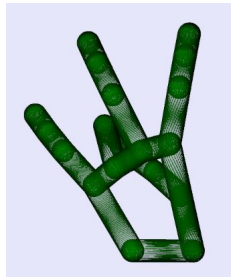


FIGURE 5.5: Gesture showing different finger poses.

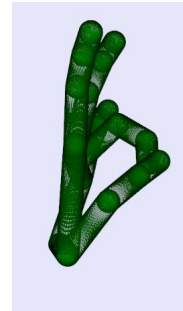


FIGURE 5.6: Same gesture after a 90 degree rotation.

```

1 case "gesture9Left":
2 case "gesture9Right":
3     fingerPoseMap.put("index", "straight");
4     fingerPoseMap.put("middle", "straight");
5     fingerPoseMap.put("ring", "curved");
6     fingerPoseMap.put("pinky", "straight");
7     fingerPoseMap.put("thumb", "ring");
8     break;

```

FIGURE 5.7: In the component based scoring, each gesture type gets a certain mapping for the kinds of poses fingers are expected to be in for that gesture.

The comparison function for the component based scoring of hand gestures is shown in Figure 5.8. It returns a number 0-100 just like the angle based comparison function to indicate the score for the hand being graded. This function gets the fingers for the hand and goes through and finds the individual grades for each finger. Then it combines the into a cumulative grade by weighing the fingers equally.

```

1 public static int compare(Hand h, String gestureType) {
2     FingerList fingerList = h.fingers();
3     //make sure you have five fingers
4     if (fingerList.count() == 5) {
5         //calculate grades for each finger
6         HashMap<String, Double> grades =
            getFingersGradedMap(getFingerHashMap(fingerList),
            getFingerPoseMap(gestureType));
7         //grade for whole hand
8         double totalGrade = cumulativeGrade(grades);
9         //score 0-100
10        return (int) (totalGrade * 100.0);
11    }
12    return -1;
13 }

```

FIGURE 5.8

To give a clearer idea about how the fingers actually get graded, the `gradeFinger()` function is shown in Figure 5.9. This function relies on three helper functions which calculate the straightness and curvedness of fingers and a function which returns the score for the thumb.

```

1 private static double gradeFinger(HashMap<String, Finger> fingerMap, Finger f,
2     String pose) {
3     if (pose.equals("straight")) {
4         return straightnessOfFinger(f);
5     } else if (pose.equals("curved")) {
6         return curvednessOfFinger(f);
7     }
8     //thumb is not touching any finger
9     else if (pose.equals("thumb")) {
10        return straightnessOfFinger(f);
11    }
12    //thumb touching other fingers
13    else {
14        Finger theFingerThumbTouches = fingerMap.get(pose);
15        return getThumbScore(f, theFingerThumbTouches);
16    }
17 }

```

FIGURE 5.9: Given a finger a certain pose, this function returns a grade (0-1) for that finger. It uses helper functions to calculate grades for a finger in one the three main kinds of poses.

Two of these helper functions, the `straightnessOfFinger()` and `curvednessOfFinger()` are shown in Figure 5.10. These functions first find the sum of the angle between consecutive bones in the finger that is being graded. For a perfectly straight finger, the sum of these angles should be around 0 degrees. However, to allow for some leniency in the grading 30 degrees are subtracted from the sum of the angles. This allows for a buffer for the user that we intuitively as humans might gauge as being relatively straight. For measuring the curvedness of a finger, the sum of the angles between the bones of the fingers should be as close to 270 as possible. However, again a buffer was provided to allow for not perfectly curled fingers to still

be valid enough to return a good score. Of course these parameters can be adjusted if this application was used in the real world. These were what I felt were good parameters when I wrote these grading functions.

```

1 private static double straightnessOfFinger(Finger f) {
2     //best case = 0; worst case is: 90+90+90 = 270.
3     double sumOfAngles = getSumOfThreeAnglesBetweenFingerBones(f);
4     sumOfAngles = sumOfAngles - 30; //offset by 30 degrees
5     double score = sumOfAngles / 270; //closer to 0 means a better score
6     score = 1 - score; //conventional scale: 0 = bad, 1 = good.
7     return snapScore0to1(score);
8 }
9 private static double curvednessOfFinger(Finger f) {
10    //best case is: 90+90+90 = 270; adjusted bestcase = 210; worst case = 0;
11    double sumOfAngles = getSumOfThreeAnglesBetweenFingerBones(f);
12    double score = sumOfAngles / 210; //closer to 1 means a better score
13    return snapScore0to1(score);
14 }

```

FIGURE 5.10: These helper functions are similar to each other. They are used in grading the four fingers.

The helper function `getThumbScore()`, shown in Figure 5.11 is the more complicated of the three. The way a score is calculated for a thumb is by finding the distance between the tip bone of the thumb and any of the three outermost bones on the finger the thumb is supposed to be touching. The smallest distance is chosen as the tip of the thumb might be closer to any three of the distal, intermediate or proximal bones. This is because some people rest their thumb on the tip of the distal bone, others rest on top of the distal or the intermediate. This distance is scaled down by the smallest bone length multiplied by a scaling factor. Like the other two functions, `straightnessOfFinger()` and `curvednessOfFinger()`, the score that is returned is snapped to be between 0-1.

```
1 private static double getThumbScore(Finger thumb, Finger otherFinger) {
2     //bones in thumb and finger
3     HashMap<String, Bone> thumbMap = getHashMapOfBonesFromFinger(thumb);
4     HashMap<String, Bone> fingerMap = getHashMapOfBonesFromFinger(otherFinger);
5     //get center point of thumb's tip bone
6     Vector thumbTip = thumbMap.get("distal").center();
7     //finger bones
8     Bone d = fingerMap.get("distal");
9     Bone i = fingerMap.get("intermediate");
10    Bone p = fingerMap.get("proximal");
11    //length of bones
12    float smallestBoneLength = (Math.min(Math.min(d.length(), i.length()),
13        p.length()));
14    //distances from thumb tip to finger bones
15    float d1 = thumbTip.distanceTo(d.center());
16    float d2 = thumbTip.distanceTo(i.center());
17    float d3 = thumbTip.distanceTo(p.center());
18    double minDistance = (double) (Math.min(Math.min(d1, d2), d3));
19    //scale and score
20    double distanceScaledByBoneLength = minDistance / (smallestBoneLength * 3);
21    double score = 1 - distanceScaledByBoneLength;
22    return snapScore0to1(score);
23 }
```

FIGURE 5.11: This function calculates the grade for the thumb that is supposed to be touching one of the four fingers. It calculates this score by distances rather than using angles.

One final thing to note about this comparison function is that it does not rely on a target hand for the comparison. Instead it relies on set gesture poses that are expected for the different gestures to arrive at its score. Therefore, it is more stable form of comparison. If the target hands are themselves not very good examples of the gestures being displayed, the angle based comparison method's performance could be unnecessarily affected.

Chapter 6

Application User Interface

In this chapter some of the useful UI features of the application will be discussed.

6.1 Main Layout

The application has three main scenes. The first one is the home screen which shows buttons to take the user to the other two scenes; "Enter Test Mode" button takes the user to the scene which is used for collecting data, and "Analyze Data" button takes the user to a scene that allows him/her to view the collected data. The home screen also contains two radio buttons to allow the user to select which hand (left or right) he/she will be testing with the gestures. Figure 6.1 shows the layout of the home screen.

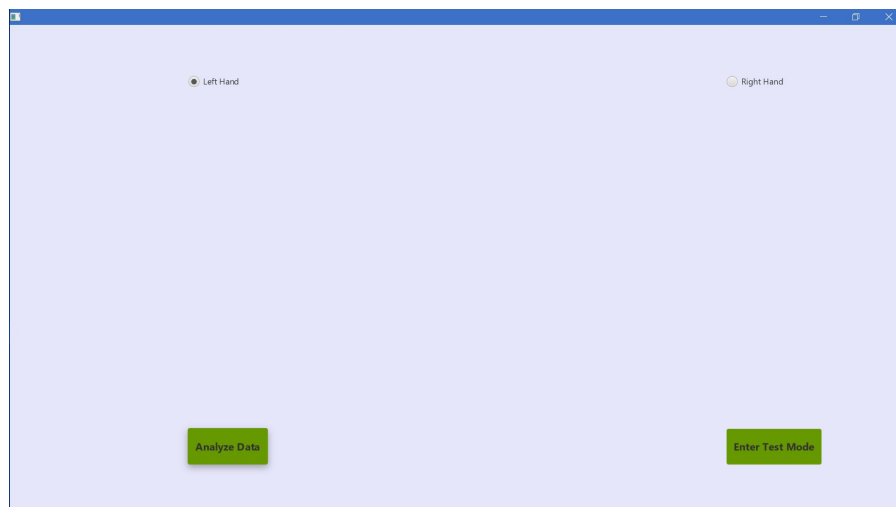


FIGURE 6.1: The scene that the user sees initially when they load the application.

6.1.1 Creating or Loading User

The user can click on the "Enter Test Mode" to go to the scene where data will be collected. However, before the user can go to the data collection scene, they must first select a previously saved user or create a new user. All of the hand gesture data collected will be stored in an appropriately named folder for the user. Therefore, when the user clicks on the "Enter Test Mode" the first thing that comes up is a small pop-up screen that asks whether the user would like to create a new user or select an older user; see Figure 6.2 and Figure 6.3.

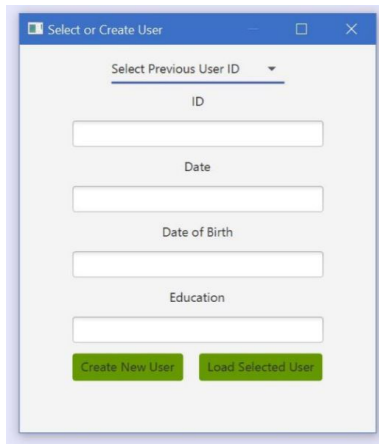


FIGURE 6.2: Pop-up window showing options to create or select user.

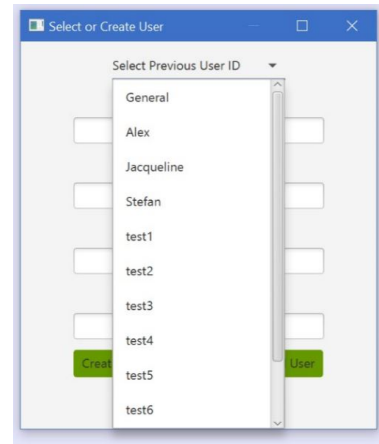


FIGURE 6.3: Pop-up window showing the previously created users.

The users shown in the dropdown menu are loaded from a CSV file which is used to record them. Whenever a user is created, a new entry is added to the CSV file for that user. As can be expected these operations are handled by using a convenient User class which encapsulates all the data associated with such an object.

6.1.2 Saving User Data

After having created or selected a user, the user is taken to the screen where he/she can start to proceed to practicing the ten gestures shown for whichever left/right hand was selected on the home screen; see Figure 6.4. On this data collection screen there are five buttons: the Next and Previous buttons cycle through the gestures, the Save button saves the currently being displayed user hand, the "End Testing" button goes back to the home screen, and the Rotate button which rotates the user and target hand a full 360 degrees slowly.

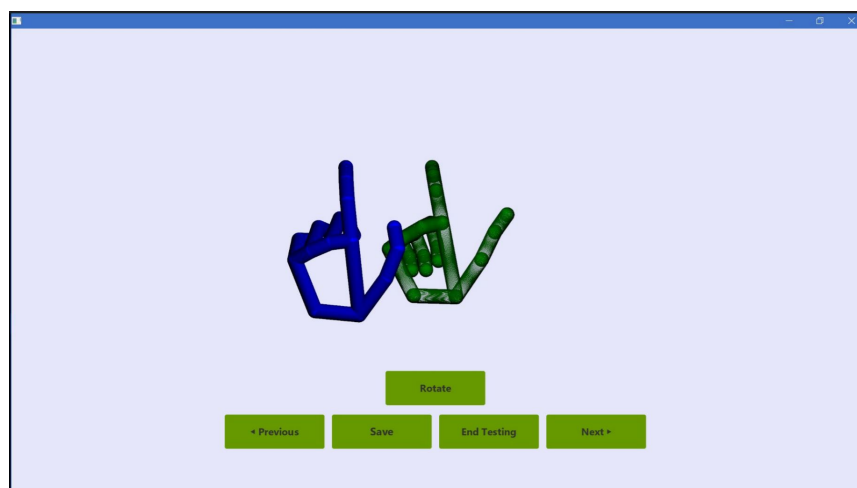


FIGURE 6.4: This scene is where the user can complete the shown gestures on the screen and the clinicians can collect the user's gesture data.

There are also some keyboard shortcuts that were coded that function in place of some of the buttons. For example, left and right arrows are mapped to the Previous and Next buttons, while the Enter key is mapped to causing the Save Gesture dialog window to pop up just as the Save button does. The user's hand will appear in full blue color on the screen, whereas the target hand the user is trying to imitate will appear in a dark green mesh material. The user's hand will be saved correctly regardless of whether it is exactly covering the target hand. The target hand is just there to give the user indication of the gesture he/she is doing. Figure 6.5 shows the Save Gesture dialog which allows the clinicians to type some comments and mark the result of the user's attempt in replicating the shown gesture.

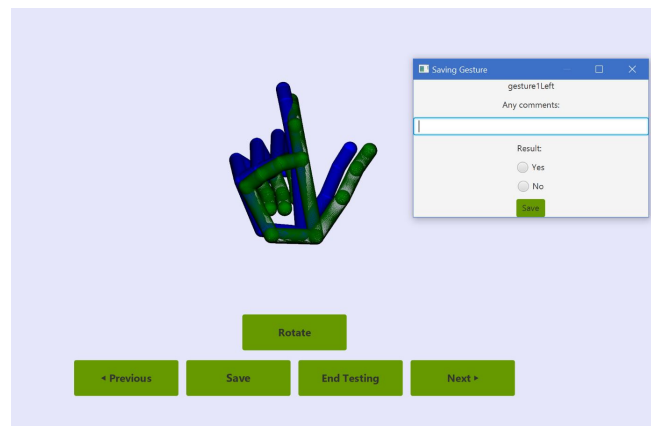


FIGURE 6.5: This dialog lets the clinicians grade and comment on the user's gesture before saving it.

By default the result is saved to be "Yes" which indicates the user successfully completed the gesture. In the code, there is an object called `HandInfo` which represents the data being saved including the full file path of where the Leap Motion Hand object will be serialized to, and the comments and the result of the gesture attempt.

6.1.3 Visual Rotation of Gesture

The Rotate button causes the 3D camera that is being used in the application to spin around. The user and target hands themselves do not move at all, it is the camera that orbits around them while being focused on them. A picture taken while the rotation was happening is shown in Figure 6.6.

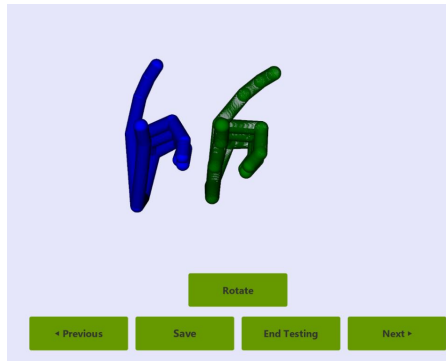


FIGURE 6.6: This figure shows instance of the rotation in action. The hands are shown at 90 degree angle because of the camera rotating around them.

The way this is achieved via code is shown in Figure 6.7. In order for the user to observe the rotation happen in real time on the screen, a Timeline object was used to set up an animation which can update the observable `angleProperty()` of a rotation transform object attached to the camera in advance. The timeline was set to last for seven seconds and the starting and ending angles were 0 and 360 degrees respectively. The code which actually initializes and adds the Rotate transform to the 3D perspective camera of the application is shown in Figure 6.8.

```

1 //set up rotation timeline
2 Timeline timeline = new Timeline(
3     new KeyFrame(Duration.seconds(0), new KeyValue(rotateAroundY.angleProperty(),
4         0)),
5     new KeyFrame(Duration.seconds(7), new KeyValue(rotateAroundY.angleProperty(),
6         -360)));
7 timeline.setCycleCount(1);
8 ...
9 //rotate button plays animation
10 rotateButton = new Button("Rotate") {
11     @Override
12     public void fire() {
13         timeline.play();
14     }
15 };

```

FIGURE 6.7: This code shows the Timeline object that was used to animate the movement of the 3D camera around the y-axis.

Since the "rotateAround" transform object is added first to the camera's transforms, it will be the last transform to be executed. This is exactly what we want so the camera will remain focused on the hands as it rotates.

```

1 // The 3D camera
2 PerspectiveCamera camera = new PerspectiveCamera(true);
3 // The rotation transform to be updated later
4 rotateAroundY = new Rotate(0, Rotate.Y_AXIS);
5 // rotation transform is added first. It will be executed last
6 camera.getTransforms().addAll(rotateAroundY, new Translate(0, -5, -50), new
    Rotate(-10, Rotate.X_AXIS));

```

FIGURE 6.8: This code shows the rotate transform that's added to the camera. This transform gets updated by the animation timeline object.

The hands themselves are not affected at all by the rotation and in fact during the seven seconds in which the camera is being rotated around the y-axis, the user can continue trying to imitate the gesture. However, during testing of the application it was found out that most users prefer to have the rotation happen in separately initially so they can just observe. Only after the rotation finishes is when they would start trying to perform the gesture.

6.1.4 Adding a New Gesture

From the data collection scene page, it is also possible to create a new target gesture and add it to the stock of target gestures being considered. This functionality was not really required by the clinicians, however it did come in useful when I was developing the application and it does have the potential of becoming a useful feature if the application is ever deployed into a real life environment. Currently the application interface does not have a button the user or clinicians can press to bring up the Save Gesture dialog window. Instead, there is a keyboard command that is listened for in the code and which is triggered by pressing the letter "G". This brings up the Save Gesture dialog, a picture of which is shown in Figure 6.9.

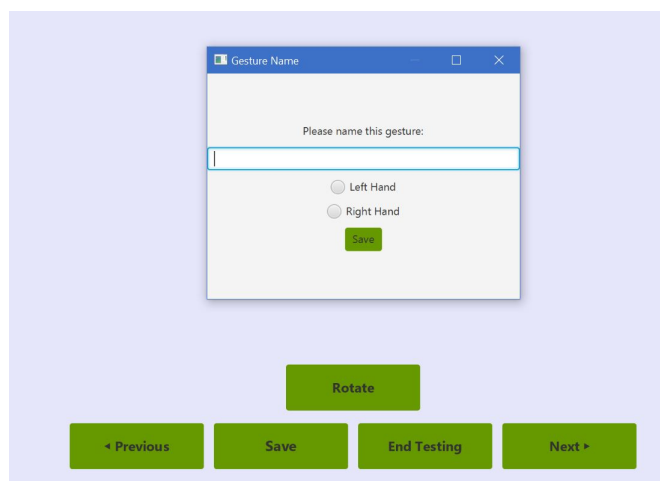


FIGURE 6.9: This dialog window will save the current user hand as a new gesture, in the list of target gestures.

Saving a new gesture will add it to the end of the list of currently used gestures.

6.2 Tabular Display of Data

If the user clicks on the "Analyze Data" button on the home screen, he/she will be taken to a scene shown in Figure 6.10. This scene shows all of the collected data for the currently selected user in a table on the left. The user is able to select rows in the table by pressing the Up or Down keys on the keyboard or by clicking on a row with their mouse.

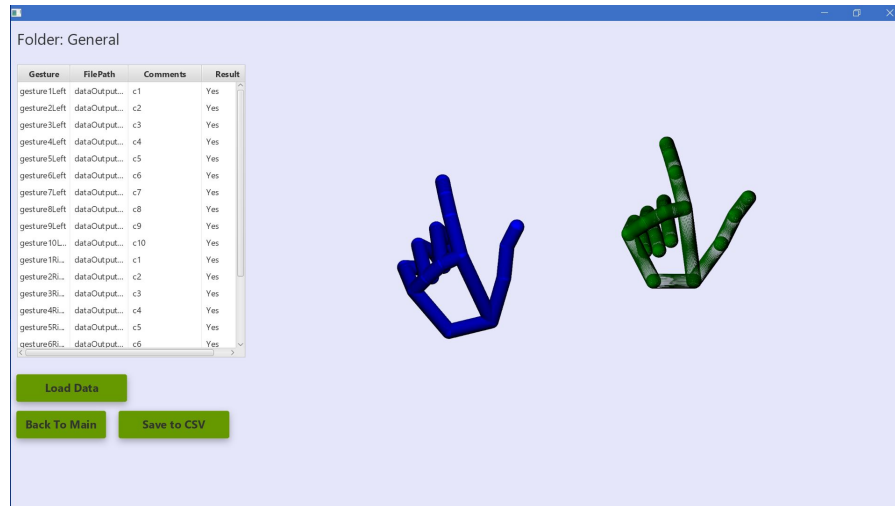


FIGURE 6.10: The scene which allows user to view and edit some of the meta-data on the hand objects saved for the currently loaded user.

This table is sortable by the columns and the columns can also be rearranged to be in a different order. In addition, some of the columns are actually editable and allow the user to specify changes to the data recorded. These changes will be updated for the user when the application is closed or the user navigates to a different scene. The way in which this table is constructed and made to be editable is shown in the snippets of code in Figure 6.11. This code is all from the controller Java class that is set for this scene. A part of this scene was built using Scene Builder and part of it was procedurally created using just Java code. The table section was prototyped in Scene Builder and the section of the scene to the right showing the two hands was created in Scene Builder.

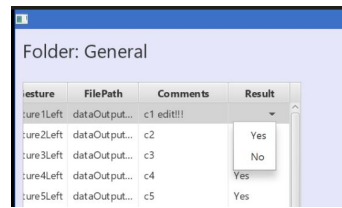
```

1 //bind references with fxml annotations
2 @FXML
3 private TreeTableView<HandInfo2> treeTableView;
4 @FXML
5 private TreeTableColumn<HandInfo2, String> col3;
6 ...
7 //set up results column; using lambda function
8 col3.setCellValueFactory((TreeTableColumn.CellDataFeatures<HandInfo2, String>
    param) -> param.getValue().getValue().result2Property());
9 //set up results column to be editable
10 ObservableList<String> list = FXCollections.observableArrayList();
11 list.add("Yes");
12 list.add("No");
13 col3.setCellFactory(ChoiceBoxTreeTableCell.forTreeTableColumn(list));
14 col3.setOnEditCommit(new EventHandler<TreeTableColumn.CellEditEvent<HandInfo2,
    String>>() {
15     @Override
16     public void handle(TreeTableColumn.CellEditEvent<HandInfo2, String> event) {
17         TreeItem<HandInfo2> h =
18             treeTableView.getTreeItem(event.getTreeTablePosition().getRow());
19         h.getValue().setResult(event.getNewValue());
20     }
21 });
22 //make table editable
23 treeTableView.setEditable(true);

```

FIGURE 6.11: These sections of code show how the table and a specific column were accessed from the FXML file. The code also shows how the column was set up to be editable by the user.

The code sample shows how the Result column of the table is setup using a lambda function syntax. Its `setCellValueFactory()` function takes in a parameter from which the `result2Property()` is returned. This property represents an observable data type that automatically responds to data changes, therefore when the Results column is updated in the view, the changes will automatically get reflected in the underlying model without the programmer having to do anything else. Similarly, the `setOnEditCommit()` function is set up the same column to allow user to edit it. The Filepath, Comments, and Result column were all made to be editable. An example of the table being edited is shown in Figure ?? This was done to help facilitate faster data collection and allow clinicians to come back and edit or correct the data if they need to. An instance of how this would be useful is if the patient being tested does not pass some of the gestures, then the clinicians can continue with the default settings of saving the gestures as passing and at a later time come back and edit the table with the real results. This might be more polite than selecting the Result of a gesture attempt to be "No" in from the data collection scene. The comments column was included to allow the clinicians to write any special comments they might need to in regards to a certain gesture.



Gesture	FilePath	Comments	Result
Gesture1Left	dataOutput...	c1 edit!!!	
Gesture2Left	dataOutput...	c2	Yes
Gesture3Left	dataOutput...	c3	No
Gesture4Left	dataOutput...	c4	Yes
Gesture5Left	dataOutput...	c5	Yes

FIGURE 6.12: This demonstrates the editable feature of the table.

6.3 Writing and Reading from CSV

Another useful feature of the application is the ability to easily read data in from a CSV file. To do so, the user clicks on the "Load Data" button; this opens up a Java FileChooser dialog which allows the user to specify the CSV file (see Figure ??). The directory path for the dialog window that opens up will be set the current folder location from which the application is running.

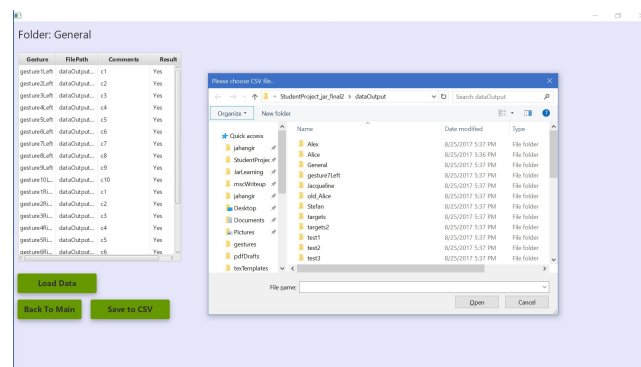


FIGURE 6.13: Load data from CSV file.

Similarly, the option to save the current data shown in the table to a CSV file is also given. It is accessed by clicking on the "Save to CSV" button which opens up a dialog very similar to the one above, except that dialog allows the user to name the file the data will be saved as. To facilitate these features, the HandInfo class has methods defined that allow for its data properties to be output to a comma separated string.

6.4 Artifacts and Distribution

After the application was completed, it had to be packaged in such a way that it should be very easy to distribute to the clinicians. In addition, it should be very easy to run. In other words, no complicated command line options must be required to be typed by the clinicians in order to use the application. Of course the Leap Controller and its necessary software should be installed on the computer in order for this application to be used. In 2013, Leap Motion its own version of an App Store where developers could upload and publish the applications they made with the Leap Motion software. It also represented the central place for people who had bought the Leap Motion controller to be able to find quality applications they could purchase or download onto their computer. However, the company decided to close down this online App Store on June 30, 2017. It is focusing now more on VR technology and

how it can be incorporated with a newer version of the Leap Motion company's software [1]. I had been originally planning on uploading my application to the Leap Motion's App Store but because of this news, I had to work on the packaging and distribution by myself.

For my development environment, I used the IntelliJ IDE. This IDE provides tools to help create build artifacts, such as a standalone JAR file, for a Java project. In order to create the artifact, one needs to open the Project Structure window and navigate to the Artifacts option in the left hand panel. The various build properties that were set up for the distribution of this project are shown in Figure 6.14.

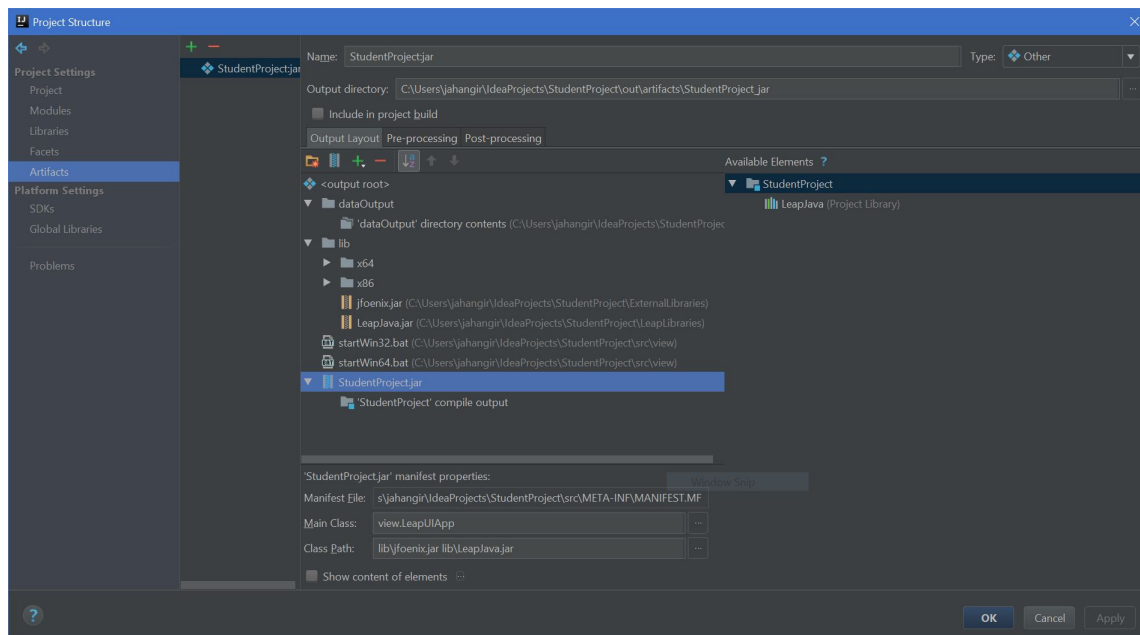


FIGURE 6.14: Project Structure Artifact settings. In the "Output Layout" tab it also shows where the manifest file containing these settings will be saved.

In the "Output Layout" section of the settings, two folders were set up also. One of these folders, dataOutput, will take the contents of the project's dataOutput folder at the time of the artifact build process. The other folder, lib, is set up to contain all of the additional libraries the project relies on such as Jfoenix and LeapJava. This lib folder also contains the native libraries needed to allow the application to talk to the system process that will be running the Leap Motion controller device. Lastly, I also needed to create two batch files, startWin64.bat and startWin32.bat, to allow the clinicians to run the application by just double clicking on one of these files depending on the system architecture of their computer. The batch script that resides in the startWin64.bat file is shown in Figure 6.15.

```

1 @echo off
2 java -Djava.library.path=%cd%\lib\x64 -Dfile.encoding=windows-1252 -jar
   %cd%\StudentProject.jar

```

FIGURE 6.15: This script contains the Java command that will run the prepared Jar file.

The result of the build artifact process will result in the application being packaged up into folder with its content shown in Figure 6.16. The JAR file can be run by double clicking on the appropriate batch file. The batch file's Java command will be executed with the runtime parameters to help it locate the native libraries within the folder structure. Because of the way the batch script is defined with the

%cd%

the application can be saved to or moved any location on one's computer and the application will be able to run just fine.



FIGURE 6.16: The older structure of the application containing all of its dependencies.

Therefore, the application is packaged and distributed in a self-contained fashion. As long as the user's computer (currently only for Windows OS) has Java v1.8 and Leap Motion software v1.2 installed the application can be run easily.

Chapter 7

Data Collection

Data collection for this project happened in two main ways. One was via the clinicians, who represented the clients who would be using the software this project was building on patients in John Radcliffe Hospital. The other way was through myself reaching out to students and other people around my college to ask them if they would like to participate in the data collection part of my MSc project.

7.1 The Approach Taken

Throughout the course of this entire project, I met with the clinicians, Dr Samrah Ahmed, Dr Christopher Butler, and Nikolas Drummond, several times to discuss the application as it was being designed and built. The features they requested were fully tried to be implemented in this software and in the final visit to the hospital, the final version of the application was delivered for the clinicians for testing and collecting some control data. The feedback received from them was very positive and all of their major UI enhancements and usability features were implemented satisfactorily.

7.2 User Feedback

Chapter 8

Results and Analysis

this is the results chapter.

Chapter 9

Conclusion

This is the conclusion ...

Bibliography

- [1] MultiMedia LLC. MS Windows NT kernel description. 1999. URL: <http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm> (visited on 09/30/2010).