UNIVERSITY NAME

DOCTORAL THESIS

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 Doctor of Philosophy

in the

Research Group Name
Department or School Name

August 21, 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:

"Thanks to my solid academic training, today I can write hundreds of words on virtually any topic without possessing a shred of information, which is how I got a good job in journalism."

Dave Barry

University Name

Abstract

Faculty Name
Department or School Name

Doctor of Philosophy

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 vii

Acknowledgements ix

1 Background 1
- 1.1 Leap Motion . . . 1
  - 1.1.1 Leap SDK v2 Java API . . . 1
- 1.2 JavaFx . . . 3
  - 1.2.1 Java vs FXML . . . 3
  - 1.2.2 Scene Builder . . . 6
  - 1.2.3 Jfoenix Library . . . 7

2 Constructing Hand UI Model 9
- 2.1 Basic 3D Modeling . . . 9
- 2.2 Set Location Method . . . 10
- 2.3 Concurrency . . . 11

A Frequently Asked Questions 13
- A.1 How do I change the colors of links? . . . 13

List of Figures

1.1 Leap Motion interaction area 1
1.2 Leap Motion Sample 2
1.3 JavaFX HelloWorld 3
1.4 UI Layout 4
1.5 JavaFX Scene Graph 4
1.6 FXML example 5
1.7 Controller for FXML 5
1.8 JavaFX Application Output 6
1.9 FXMLLoader 6
1.10 Scene Builder 7
1.11 External Libraries 8

2.1 Hand Bone Model 9
2.2 JavaFx Group Node 10
2.3 UIHandSimple Constructor 11
2.4 setRotationByVector 11

List of Tables

List of Abbreviations

LAH	**L**ist **A**bbreviations **H**ere
WSF	**W**hat (it) **S**tands **F**or

Physical Constants

Speed of Light $\quad c_0 = 2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ (exact)

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

For/Dedicated to/To my…

Chapter 1

Background

1.1 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 1.1.

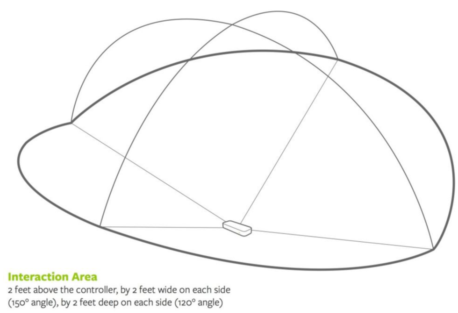

FIGURE 1.1: The area of user interaction for the Leap Motion Device.

1.1.1 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 1.2 shows the basic set-up required to receive data from the controller using the Leap Java API.

```java
import com.leapmotion.leap.*;

//Listener class which handles various events for the Controller
class SampleListener extends Listener {
    //method to handle the event of a Frame received from Controller
    public void onFrame(Controller controller) {
        // Get the most recent frame and report some basic information
        Frame frame = controller.frame();
        System.out.println("Frame id: " + frame.id() + ", hands: " +
            frame.hands().count()
    }
}

class Sample {
    public static void main(String[] args) {
        // Create leap motion controller instance
        Controller controller = new Controller();

        // Have the sample listener receive events from the controller
        controller.addListener(new SampleListener(););

        // Remove the sample listener when done
        controller.removeListener(listener);
    }
}
```

FIGURE 1.2: This code sample shows how to connect to and receive data from the Leap Motion controller device.

1.2 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.

1.2.1 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

```
public class HelloWorld extends Application {
public static void main(String[] args) {
   launch(args);
}

@Override
public void start(Stage primaryStage) {
   primaryStage.setTitle("Hello World!");
   Button btn = new Button();
   btn.setText("Say 'Hello World'");
   btn.setOnAction(new EventHandler<ActionEvent>() {
      @Override
      public void handle(ActionEvent event) {
         System.out.println("Hello World!");
      }
   });

   StackPane root = new StackPane();
   root.getChildren().add(btn);
   primaryStage.setScene(new Scene(root, 300, 250));
   primaryStage.show();
}
}
```

FIGURE 1.3: 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 is 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 is 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 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 1.4.

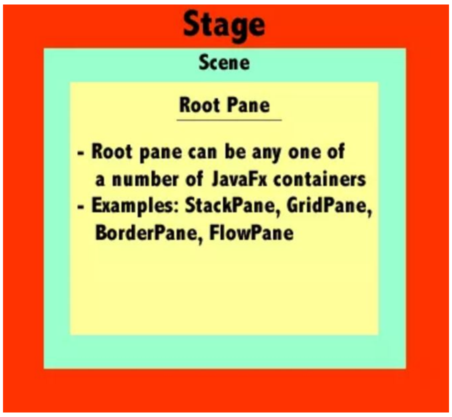

FIGURE 1.4: The layout components of every JavaFX application.

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 1.5 displays an example of the JavaFX scene graph.

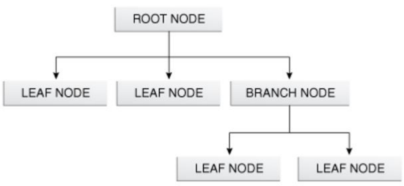

FIGURE 1.5: The Scene Graph architectural model for UI components in JavaFX applications.

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 1.6 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 1.7

```
<?xml version="1.0" encoding="UTF-8"?>
<?language JavaScript?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>

<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">
  <children>
    <Button fx:id="okBtn" alignment="CENTER_RIGHT" contentDisplay="CENTER"
        mnemonicParsing="false" onAction="#printHelloWorld" text="Say Hello
        World" textAlignment="CENTER" />
  </children>
</VBox>
```

FIGURE 1.6: A FXML file defining a simple layout and referencing a controller java class.

```
public class MyController
{
   @FXML
   private void initialize()
   {
   // this method runs first after all the UI components have been loaded and
       bound.
   }

   @FXML
   private void printHelloWorld()
   {
      System.out.println("hello world!");
   }
}
```

FIGURE 1.7: 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 1.8. 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

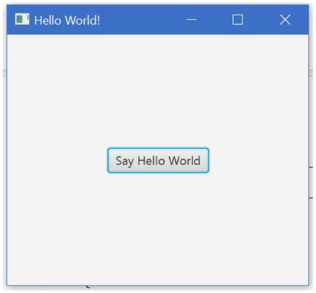

FIGURE 1.8: The output of the simple HelloWorld application.

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 1.9 this key difference between the two methods.

```
// The approach uses the static "load" method. Not recommended
Parent root = FXMLLoader.load(getClass().getResource("fxml_example.fxml"));

//This approach allows one to access the controller.
//create instance of FXMLLoader
FXMLLoader loader = new FXMLLoader(getClass().getResource("fxml_example.fxml"));
//get the controller
MyController controller = loader.<MyController>getController();
//initialize controller with custom parameters
controller.initData(data);
//object graph root handle
Parent root = (Parent) loader.load();
```

FIGURE 1.9: Retrieving a reference to the controller object for a loaded FXML file.

1.2.2 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 1.10 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.

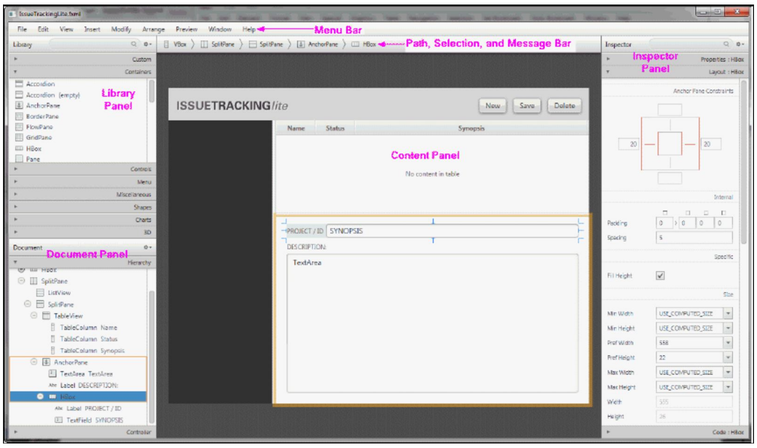

FIGURE 1.10: This shows the different areas of interest in the Scene Builder software.

1.2.3 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:

```
<dependency>
    <groupId>com.jfoenix</groupId>
    <artifactId>jfoenix</artifactId>
    <version>1.4.0</version>
</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 1.11. Selecting this option will allow for the Jfoenix JAR file to be specified and the custom components to be loaded into the Scene Builder.

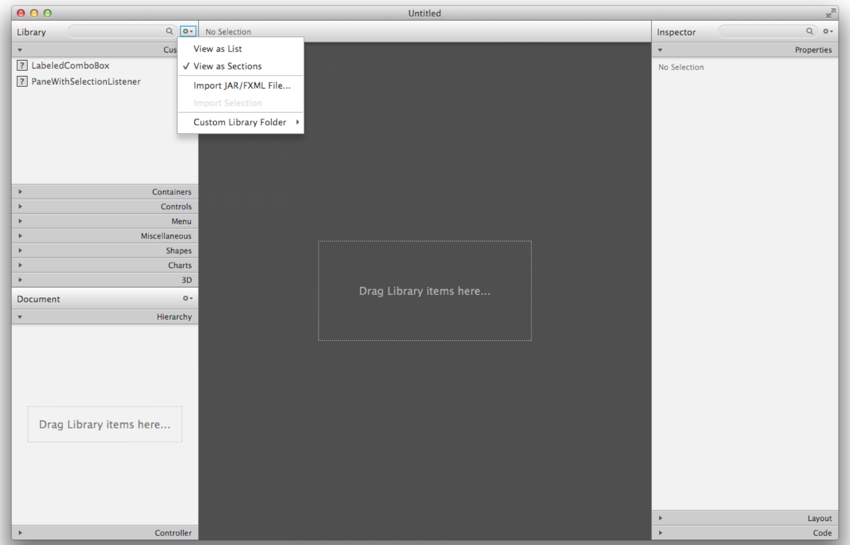

FIGURE 1.11: This is how to add a external library of custom components to the Scene Builder for UI prototyping.

Chapter 2

Constructing Hand UI Model

2.1 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 heirarchical 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 neigboring 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 object endpoint closer to the tip of the finger. The figure below shows the bones and joints of the hand for which the Leap Motion sensor records data.

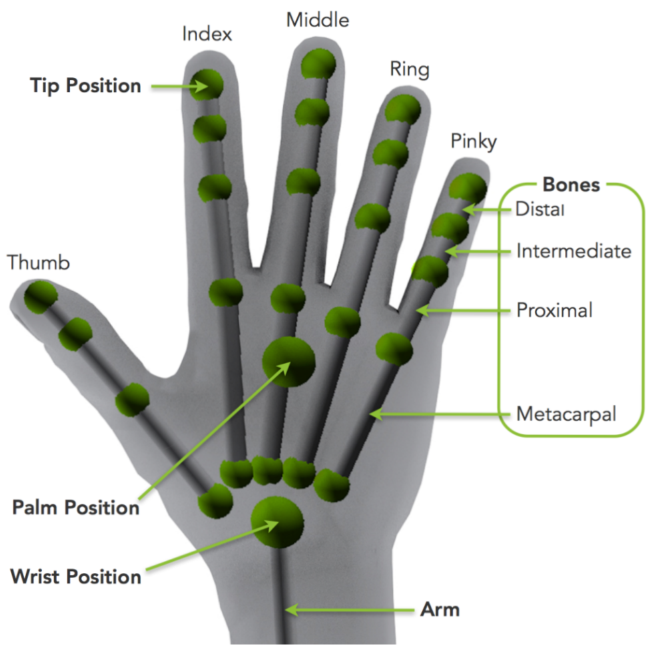

FIGURE 2.1: The Bones of the hand that Leap Motion device records data on.

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_Simple`Java 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 below shows an example of how a Group Node can can contain multiple children nodes.

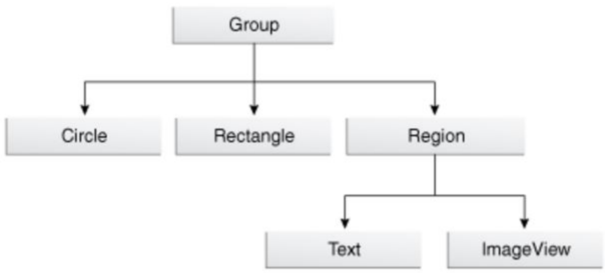

FIGURE 2.2: The Group Node structure in the JavaFx framework.

The `UIHand_Simple`class 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 Phong-Material object is set as the hand's material property. Below is a figure showing an excerpt from code which shows the initialization of the `UIHand_Simple`and part of its constructor.

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.

2.2 Set Location Method

The `UIHand_Simple`class also contains a method that allows for the graphical hand to be positioned according to the exact postions 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

```java
public class UIHand_Simple extends UIHand {

    private static float fingerRadius = 14f;

    private Cylinder[][] fingerBones;
    private Sphere[][] fingerJoints;
    private Cylinder[] knuckleSpans;
    private Cylinder palmWrist;
    private Cylinder palmPinky;
    private Sphere pinkyJoint;

    public UIHand_Simple(Color color, boolean wireframe) {
        super();

        PhongMaterial dark = new PhongMaterial(color);
        PhongMaterial light = new PhongMaterial(color.brighter());

        fingerBones = new Cylinder[5][];
        for (int i = 0; i < 5; ++i) {
            fingerBones[i] = new Cylinder[3];
            for (int j = 0; j < 3; ++j) {
                fingerBones[i][j] = new Cylinder();
                fingerBones[i][j].setMaterial(dark);
                fingerBones[i][j].setRadius(fingerRadius / ViewMath.radiusScaleFactor);
                if (wireframe) fingerBones[i][j].setDrawMode(DrawMode.LINE);
            }
```

FIGURE 2.3: 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. Below is a snippet of code that shows the setRotationByVector method.

```java
//This method rotates a given JavaFx node to point in the direction passed in
public static void setRotationByVector(Node node, Vector direction) {
    //Correct the direction to correspond to JavaFx Coordinate system
    Vector correctedDirection = new Vector(direction.getX(), direction.getY(), -direction.getZ());

    //Find the angle of the direction to the y-axis; in degrees
    double angle = correctedDirection.angleTo(Vector.yAxis()) * 180 / Math.PI;

    //Find the axis of rotation by taking the cross product of the corrected direction with the y-axix
    Point3D axis = vectorToPoint(correctedDirection.cross(Vector.yAxis()));

    //Set the axis and angle of rotation on the Node object
    node.setRotate(angle);
    node.setRotationAxis(axis);
}
```

FIGURE 2.4: A snippet of code showing how the rotation is set for an arbitrary Node object of the JavaFX Hand Model.

2.3 Concurrency

-testing refresh –read some code about javafx application threading stuff. research. write it in a paragraph and explain what it means.

–look at the code that does task setting. read it a little try to understand. make some notes about what it means and how its wired. write about it.

– also write about the frame controller methods offered by leap ui. and talk about the controller2. read code, understand, make notes. and then write about it.

Appendix A

Frequently Asked Questions

A.1 How do I change the colors of links?

The color of links can be changed to your liking using:
`\hypersetup{urlcolor=red}`, or
`\hypersetup{citecolor=green}`, or
`\hypersetup{allcolor=blue}`.
If you want to completely hide the links, you can use:
`\hypersetup{allcolors=.}`, or even better:
`\hypersetup{hidelinks}`.
If you want to have obvious links in the PDF but not the printed text, use:
`\hypersetup{colorlinks=false}`.