Chapter 1

Application User Interface

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

1.1 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 1.1 shows the layout of the home screen.

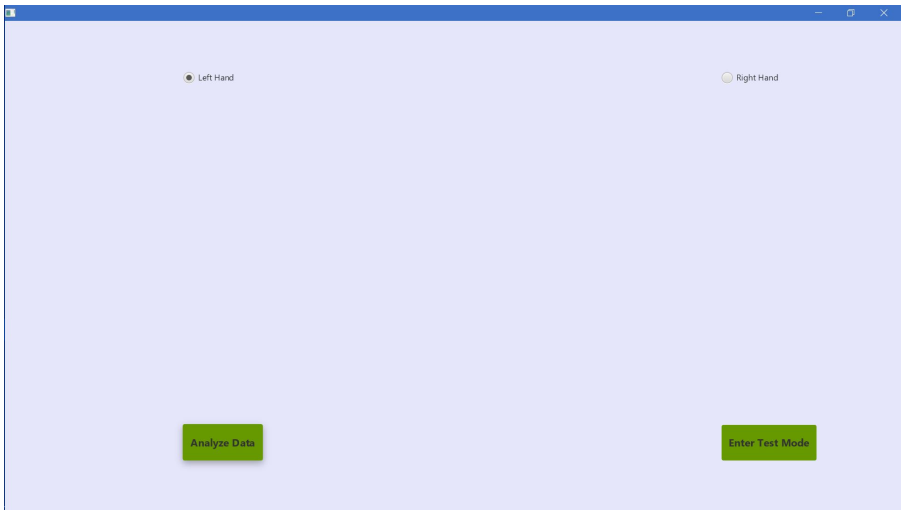


FIGURE 1.1: The scene that the user sees initially when they load the application.

1.1.1 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 1.2 and Figure 1.3.

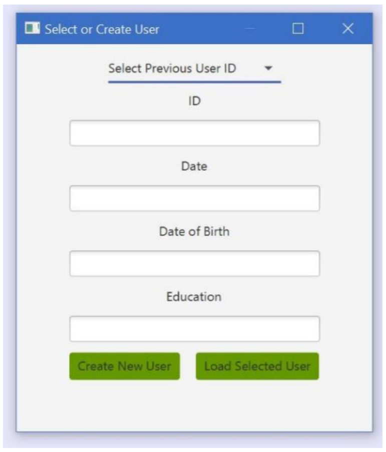


FIGURE 1.2: Pop-up window showing options to create or select user.

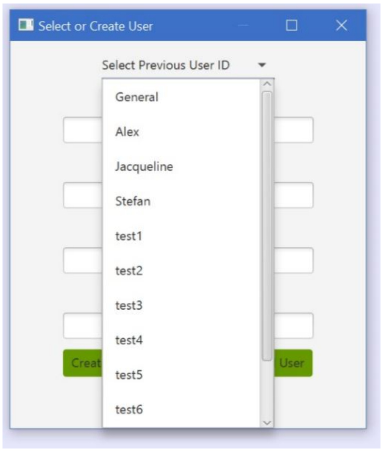


FIGURE 1.3: Pop-up window showing the previously created users.

The users shows 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.

1.1.2 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 1.4. 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.

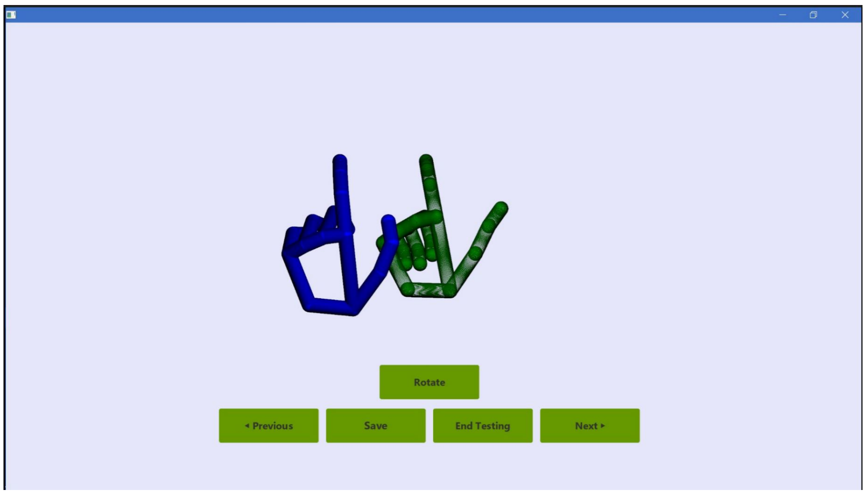


FIGURE 1.4: This scene is where the user can complete the shown gestures on the screen and the clinicians can collect the user's gesture data.

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 1.5 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.

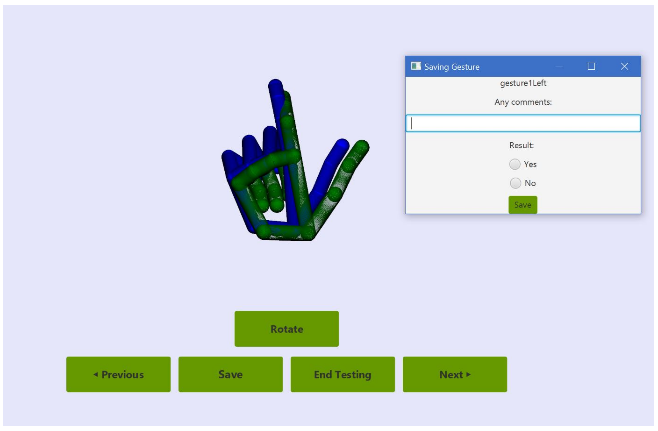


FIGURE 1.5: This dialog lets the clinicians grade and comment on the user's gesture before saving it.

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.

1.1.3 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 1.6.

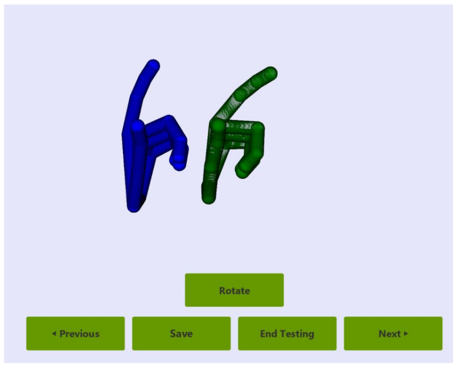


FIGURE 1.6: This figure shows instance of the rotation in action. The hands are shown at 90 degree angle because of the camera rotating around them.

The way this is achieved via code is shown in Figure 1.7. 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 1.8.

```java
//set up rotation timeline
Timeline timeline = new Timeline(
   new KeyFrame(Duration.seconds(0), new KeyValue(rotateAroundY.angleProperty(),
       0)),
   new KeyFrame(Duration.seconds(7), new KeyValue(rotateAroundY.angleProperty(),
       -360)));
timeline.setCycleCount(1);
...
//rotate button plays animation
rotateButton = new Button("Rotate") {
   @Override
   public void fire() {
   timeline.play();}
};
```

FIGURE 1.7: 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.

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

FIGURE 1.8: 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 to 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.

1.1.4 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 1.9.

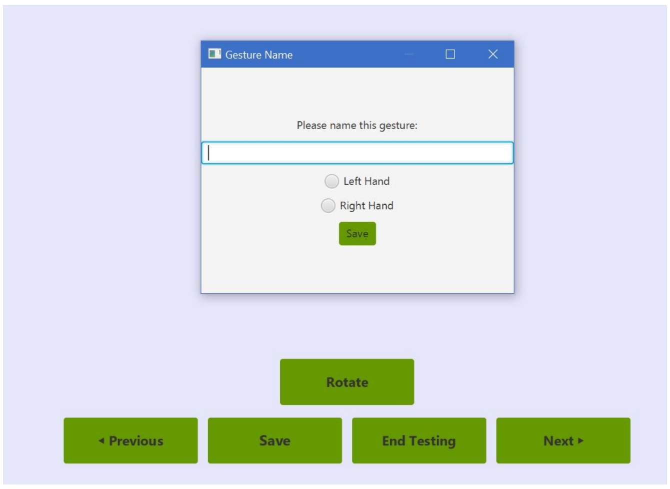
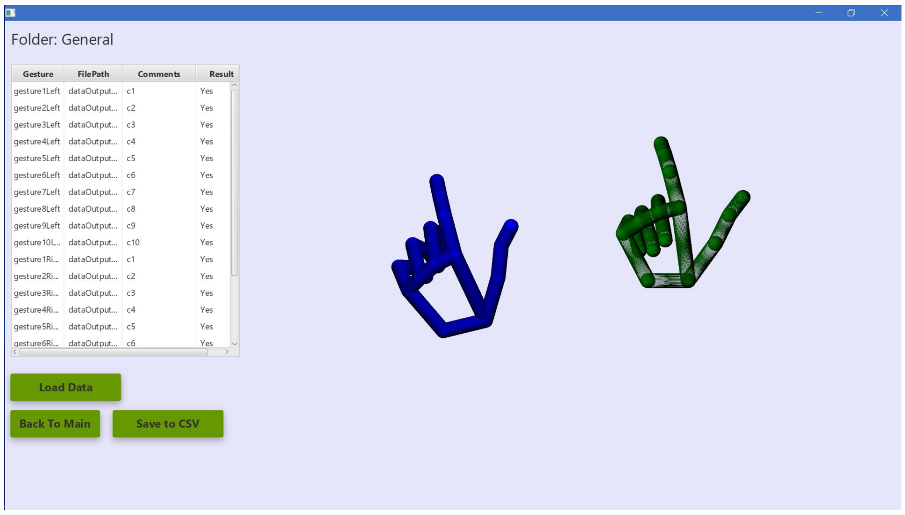


FIGURE 1.9: This dialog window will save the current user hand as a new gesture, in the list of target gestures.

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

1.2 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 1.10. 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.



Folder: General

Gesture	FilePath	Comments	Result
gesture1Left	dataOutput...	c1	Yes
gesture2Left	dataOutput...	c2	Yes
gesture3Left	dataOutput...	c3	Yes
gesture4Left	dataOutput...	c4	Yes
gesture5Left	dataOutput...	c5	Yes
gesture6Left	dataOutput...	c6	Yes
gesture7Left	dataOutput...	c7	Yes
gesture8Left	dataOutput...	c8	Yes
gesture9Left	dataOutput...	c9	Yes
gesture10L...	dataOutput...	c10	Yes
gesture1Ri...	dataOutput...	c1	Yes
gesture2Ri...	dataOutput...	c2	Yes
gesture3Ri...	dataOutput...	c3	Yes
gesture4Ri...	dataOutput...	c4	Yes
gesture5Ri...	dataOutput...	c5	Yes
gesture6Ri...	dataOutput...	c6	Yes

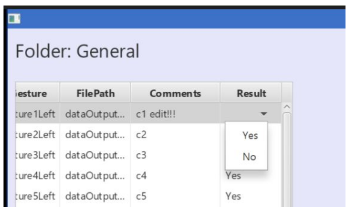
FIGURE 1.10: The scene which allows user to view and edit some of the meta-data on the hand objects saved for the currently loaded user.

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 1.11. 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.

```java
//bind references with fxml annotations
@FXML
private TreeTableView<HandInfo2> treeTableView;
@FXML
private TreeTableColumn<HandInfo2, String> col3;
...
//set up results column; using lambda function
col3.setCellValueFactory((TreeTableColumn.CellDataFeatures<HandInfo2, String>
    param) -> param.getValue().getValue().result2Property());
//set up results column to be editable
ObservableList<String> list = FXCollections.observableArrayList();
list.add("Yes");
list.add("No");
col3.setCellFactory(ChoiceBoxTreeTableCell.forTreeTableColumn(list));
col3.setOnEditCommit(new EventHandler<TreeTableColumn.CellEditEvent<HandInfo2,
    String>>() {
   @Override
   public void handle(TreeTableColumn.CellEditEvent<HandInfo2, String> event) {
      TreeItem<HandInfo2> h =
          treeTableView.getTreeItem(event.getTreeTablePosition().getRow());
      h.getValue().setResult(event.getNewValue());
   }
});
//make table editable
treeTableView.setEditable(true);
```

FIGURE 1.11: 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. It's 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.



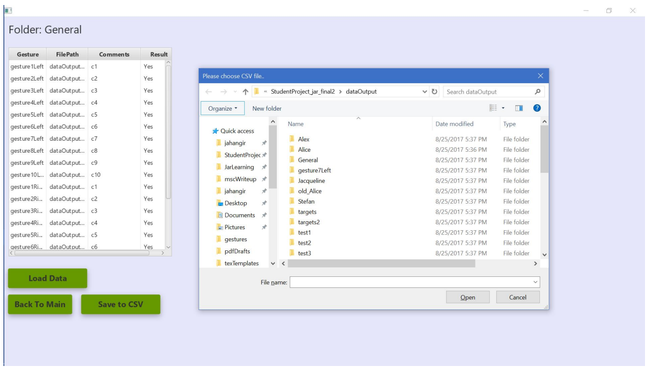
Folder: General

esture	FilePath	Comments	Result
ure1Left	dataOutput...	c1 edit!!!	
ure2Left	dataOutput...	c2	Yes
ure3Left	dataOutput...	c3	No
ure4Left	dataOutput...	c4	Yes
ure5Left	dataOutput...	c5	Yes

FIGURE 1.12: This demonstrates the editable feature of the table.

1.3 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.



Folder: General

Gesture	FilePath	Comments	Result
gesture1Left	dataOutput...	c1	Yes
gesture2Left	dataOutput...	c2	Yes
gesture3Left	dataOutput...	c3	Yes
gesture4Left	dataOutput...	c4	Yes
gesture5Left	dataOutput...	c5	Yes
gesture6Left	dataOutput...	c6	Yes
gesture7Left	dataOutput...	c7	Yes
gesture8Left	dataOutput...	c8	Yes
gesture9Left	dataOutput...	c9	Yes
gesture10L...	dataOutput...	c10	Yes
gesture1Ri...	dataOutput...	c1	Yes
gesture2Ri...	dataOutput...	c2	Yes
gesture3Ri...	dataOutput...	c3	Yes
gesture4Ri...	dataOutput...	c4	Yes
gesture5Ri...	dataOutput...	c5	Yes
gesture6Ri...	dataOutput...	c6	Yes

Please choose CSV file..

« StudentProject_jar_final2 › dataOutput

Name	Date modified	Type
Alex	8/25/2017 5:37 PM	File folder
Alice	8/25/2017 5:36 PM	File folder
General	8/25/2017 5:37 PM	File folder
gesture7Left	8/25/2017 5:37 PM	File folder
Jacqueline	8/25/2017 5:37 PM	File folder
old_Alice	8/25/2017 5:37 PM	File folder
Stefan	8/25/2017 5:37 PM	File folder
targets	8/25/2017 5:37 PM	File folder
targets2	8/25/2017 5:37 PM	File folder
test1	8/25/2017 5:37 PM	File folder
test2	8/25/2017 5:37 PM	File folder
test3	8/25/2017 5:37 PM	File folder

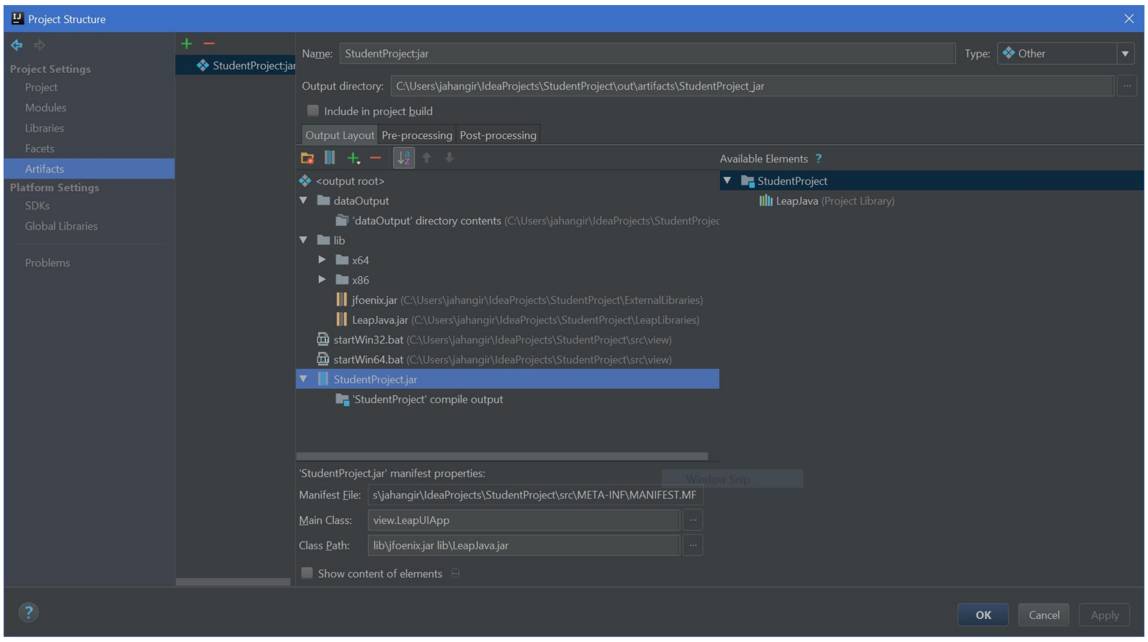
FIGURE 1.13: Load data from CSV file.

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.

1.4 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 1.14.



Name: StudentProject:jar
Type: Other
Output directory: C:\Users\jahangir\IdeaProjects\StudentProject\out\artifacts\StudentProject_jar
Include in project build

```
<output root>
dataOutput
    'dataOutput' directory contents (C:\Users\jahangir\IdeaProjects\StudentProjec
lib
    x64
    x86
    jfoenix.jar (C:\Users\jahangir\IdeaProjects\StudentProject\ExternalLibraries)
    LeapJava.jar (C:\Users\jahangir\IdeaProjects\StudentProject\LeapLibraries)
startWin32.bat (C:\Users\jahangir\IdeaProjects\StudentProject\src\view)
startWin64.bat (C:\Users\jahangir\IdeaProjects\StudentProject\src\view)
StudentProject.jar
    'StudentProject' compile output
```

'StudentProject.jar' manifest properties:
Manifest File: s\jahangir\IdeaProjects\StudentProject\src\META-INF\MANIFEST.MF
Main Class: view.LeapUIApp
Class Path: lib\jfoenix.jar lib\LeapJava.jar
Show content of elements

FIGURE 1.14: 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 1.15.

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

FIGURE 1.15: 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 1.16. 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.

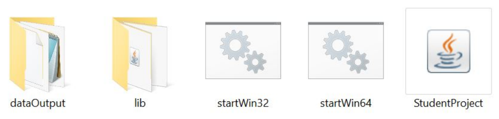


FIGURE 1.16: The older structure of the application containing all of its dependencies.

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.

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).