Project Number. GFP1403

Leap Motion Presenter

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

_____

_____

James Anouna

Johnny Hernandez

March 25, 2015

Approved:

_____

Professor Gary F. Pollice, Major Advisor

Table of Contents

Table of Figures

Table of Tables

Abstract

The Leap Motion controller is a device that reads the precise position of a person's hands in the space above it, and relays that data to a program running on the person's computer. The purpose of this project is to create a prototype of an application that uses the Leap Motion controller to create and run a presentation. The hand and gesture recognition of the Leap Motion device facilitates an intuitive application that enables presenters to interact less with their computer and more with their audience. The application is written in Java 8, and is heavily based on the JavaFX graphics library. While there remain imperfections in the application, the application is a working prototype that demonstrates the strengths of gesture-based presentation software and demonstrates how an existing software task can be enhanced with new technology.

Chapter 1: Introduction

There are many tools that can assist a person in giving a presentation. One of the most well-known and widely-used of these tools is Microsoft PowerPoint®. PowerPoint allows a presenter to put images and text on slides in a readable fashion. Advanced users can embed video files in the slides and set up animations. With careful planning, timers can also be set to perform animations and advance slides for you. The Leap Motion Presenter aims to modernize the presentation experience with gesture-based object manipulation intuitive to end users.

One of the downsides of PowerPoint is it is difficult to use some of the more complex features. As more features were added, it became more and more difficult for new users to find certain functions of the program. All of the main functions are visible, but there are some things PowerPoint can do that few people know about. Setting up effective animations can be challenging and time-consuming for a novice user. Timing animations and slide changes are conceptually simple, but require a lot of trial runs of the presentation. In addition, there is functionality for a presenter to draw on slides during the presentation. Few people know this functionality exists, let alone how to use it. Michael Buckwald, President and CEO of Leap Motion, said: "It's that the way users interact with [devices and programs] is very simple. And that, unfortunately, leads to things like drop-down menus and keyboard shortcuts… elements that require people to learn and train rather than just do and create" (Foster 2013).

The goal of this project was to create a presentation application that utilizes the Leap Motion as a means of delivering a presentation similar to Microsoft PowerPoint slide-based presentations. We made use of the Leap Motion control scheme to allow users to utilize gestures to manipulate objects during presentation for a more dynamic presentation. The ability for users to move objects, resize objects, and draw on the slides as needed throughout the presentation

were core features we thought were possible with the Leap Motion device and API. We believed this would give the user more control on the content or information presented to the audience. Our original goals did not include the use of the Leap Motion device when creating the presentation itself, but early on we had decided on using the Leap Motion as the primary input device for most of the application.

The Leap Motion Presenter app is designed to be intuitive to users, both in the setting up and delivery of presentations. Object animations in the presentation are performed in real-time by the presenter, so no tedious or time-consuming setups are required. The presenter grabs and moves objects, resize objects, draw on the screen, and plays and pauses videos with simple gestures using the Presenter app. Most of these tasks can only be done with preplanned animations in PowerPoint.

Another issue with PowerPoint is that it is sometimes difficult to effectively communicate the information you are presenting when it sits motionless on the slide or follows a predetermined animation. Animations are used to help demonstrate a concept the presenter imagines to be difficult to understand with static images alone. The presenter may not have an animation planned for every topic the audience misunderstands. One of the goals of the Leap Motion Presenter app is to incorporate gestures to allow the presenter to bring attention to certain text or images dynamically. These actions are not always very easy to perform with the mouse and keyboard. We have found from experience that certain functions of PowerPoint, such as drawing on presentation slides, are rarely used due to how unintuitive the mouse and keyboard work with them. Because these types of actions are difficult with the current technology, the addition of a new control scheme is required to make these features usable.

With the Leap Motion Presenter app, a presenter can shape a presentation to fit the

audience during the presentation with intuitive controls. This can make it much easier to emphasize or explain important or difficult to understand topics. The Leap Motion can also help presenters access functionality that is less known in PowerPoint, turning that functionality into common practice for every presentation. The Leap Motion device allows presenters to focus less on learning the software and more on presenting.

Chapter 2: Background

2.1 Leap Motion

The Leap Motion controller, also known as The Leap, was released in 2013 by Leap Motion, Inc. It was a highly-anticipated device that monitors the motion of a user's hands in an eight-cubic-foot space above the controller using three infrared cameras (Leap Motion). The device can track movements as small as 0.01 millimeters (Foster 2013). On its release, the device received mixed reviews from various technological and mainstream journals due to the lack of well-written apps in its store, called Airspace (Pogue 2013). While there is still hope for a potential explosion in popularity, the device, and the company, have yet to receive the public support originally anticipated (Hutchinson 2013).

David Holz began development on the Leap Motion controller in 2008, and spent nearly five years developing the software and cameras. After those five years, a prototype was demonstrated for Bill Warner who then invested $25,000 to further develop the technology. The project then gained support from several other individuals and venture capital firms. This support helped the company grow to over 80 employees and become popular among a large group of future users. The following was "bigger than the iPhone and the Facebook platform when they launched" (Richardson 2013).

The device was inspired by Holz experiences with 3D modeling software, noticing that what can be sculpted out of clay in five minutes takes hours using top of the line software. He knew that technology had been helping him do things he normally could not so he realized that the complexities of the common computer controls such as mice and keyboards complicate the process of creation when it should be as easy as using your hands. This desire to simplify the interface drove the development of what is now known as the Leap Motion, and one of the many

reasons for creating this application.

The Leap Motion Store contains roughly 200 applications, more than half of those being games. In this way the Leap Motion is seen more as a different type of input for games, rather than for fully-fledged applications. Only a small fraction of these applications are categorized as productivity or computer control applications that attempt to replace the mouse. The applications in the store demonstrate the Leap's ability to perform as an entertainment device, but rarely explore the usefulness of a control alternative for existing applications. Applications such as Sculptor, a 3D sculpting tool using the Leap Motion device, demonstrate the ability to manipulate 3D objects in 3D air space as if one were molding clay. An application like Touchless, which enables users to access normal system functions with the Leap Motion on Mac and Windows, demonstrates the ability to control common computer actions such as pointing, clicking, and scrolling using physical hand movements over objects on screen. The ability of the Leap Motion device to act as a new form of control for existing applications is vastly underrepresented with the existing application base, and the different controls could create new ways of performing existing tasks or new tasks altogether.

2.2 JavaFX

JavaFX is the framework used to write apps for the Leap Motion in Java. JavaFX was originally released in 2008 for creating rich Internet applications. A rich internet application is an application designed to run on the web, but with functionality normally associated with desktop applications (Rouse 2007). Other languages that are commonly used to create rich internet applications are JavaScript, Adobe Flash, Microsoft Silverlight, and more recently Python and HTML5. JavaFX is included in the standard Java Runtime Environment as of Java 7, meaning it does not need to be included separately in projects. The application programming interface (API)

is based heavily on Java's Swing libraries. Swing is Java's main graphical user interface framework, managing windows and containers with buttons, text boxes, labels, etc. In addition, the theme of JavaFX applications is handled with Cascading Style Sheets (CSS), which handles the look and format of the application, keeping style separated from actual coding (CSS Tutorial). There is also an extensible markup language (XML) called FXML which can be used to construct graphical user interfaces in JavaFX (Pawlan 2013). XML is used to define rules for web documents or applications in a form that is readable by both human programmers and computers. JavaFX is a content-rich and well-documented framework that will be well suited for developing a presenter app for the Leap Motion Controller.

2.3 Alternative Input Devices

The Leap Motion is not the only computer input device that is not a keyboard or mouse. The most common alternative input device is the touch screen. Older touch screens used pressure sensors on the screen to simulate "clicks" with the mouse pointer. In modern systems, capacitive touch is used instead. When a person touches a capacitive touch screen, electrical fields behind the screen are distorted, and the screen monitors these fields to determine where the touch occurred. Touch screens allow for some intuitive gestures, such as swiping to scroll and pinching to zoom. However, the lack of tactile feedback from on-screen keyboards has kept traditional keyboards popular.

Other alternative computer inputs are voice recognition and game controllers. Voice recognition has been used in the Windows operating system since Windows 7 was released in 2009 (History 2015). It has limited functionality with the ability to open certain programs and dictate text. Voice commands have also become popular with modern smart phones. Game controllers can be mapped to controls on a computer, but often require complex emulators to

accomplish anything specific. Additionally, special adapters are required to connect many game

controllers to a computer, usually using a USB port.

Chapter 3: Methodology

The Leap Motion Presenter utilizes Java as its core programming language, with only a small use of markup languages, such as CSS, for interface purposes. The Leap Motion API supports several different programming languages. These languages are JavaScript, C#, Java, C++, Python, and Objective C. We decided not to use Objective C because of its proprietary nature and that neither of us are familiar with the Apple development environment. We did not believe that Python had the functionality or speed to make our application run smoothly, nor is it suited for a larger scale application. C++ is a very large language, and the Leap Motion API was originally designed in C++, but there are more modern alternatives that we believed would be better suited for our application. While we recognize that C++ was one of the more valid candidates for this application, we decided against using this language due to time constraints and its difficult learning curve when compared to other languages.

We felt similarly about JavaScript as we did about Python, that it was not a robust enough language for our application, and that we would end up writing a lot of functionality that is built into some of the other available languages. This left us with Java and C#, both of which are modern, object oriented languages with enough functionality for our application which makes building large applications more easily. In the end, we decided against the use of C# due to its restriction of only being used through the Unity game engine, which was designed more for gaming than the slower, more productive nature of our application.

Java is well-suited for our application for a few reasons. Java runs on most modern operating systems. The only differences when building on different operating systems is the inclusion of system-specific libraries and minor differences in parameters among the Leap API. Java is a very widely-used language which includes a large community for getting help or

finding third-party libraries that assist with the development of our application. It is also important for many other applications, so most end users have the necessary tools needed to run our application without installing other software.

3.1 Leap API & JavaFX

The Leap Motion API follows a simple architecture design. It is based on a listener design pattern where methods are called when certain actions are performed on the Leap Motion device, similar to mouse listeners. The most important Leap Motion listener method in our application is the *onFrame()* method, though there are several other methods provided by the Leap *Listener* class. The *onFrame()* method has access to the current frame application data such as hand positions, finger positions, and gestures being performed. Leap Motion API has built in functionality for certain gestures, returning a list of any recognized gestures on a given *onFrame()*. The commonly used gestures that Leap supports are swipes, taps, and circle gestures. This data is then handed off to the JavaFX threads for the interface to "react" to.
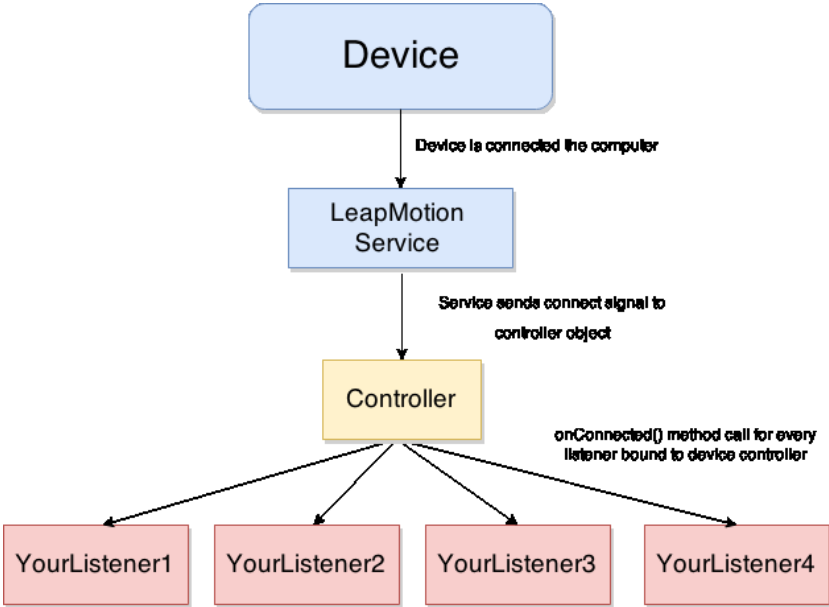

Figure 1: Example of listener pattern with device connection.

The Leap Motion listeners in our application call JavaFX *runlater()* methods. These

create new GUI threads that our application is built upon. This is a relatively simple way of

passing important data from the Leap Motion *Listener* threads to the JavaFX threads. Once the

data is passed from a Leap *Listener* to a JavaFX thread, that data can be used to change the

interface of the program. The JavaFX threads never access the raw Leap Motion data and the

Leap Motion threads do not alter the JavaFX application state. The threads are independent of

each other and are interchangeable.
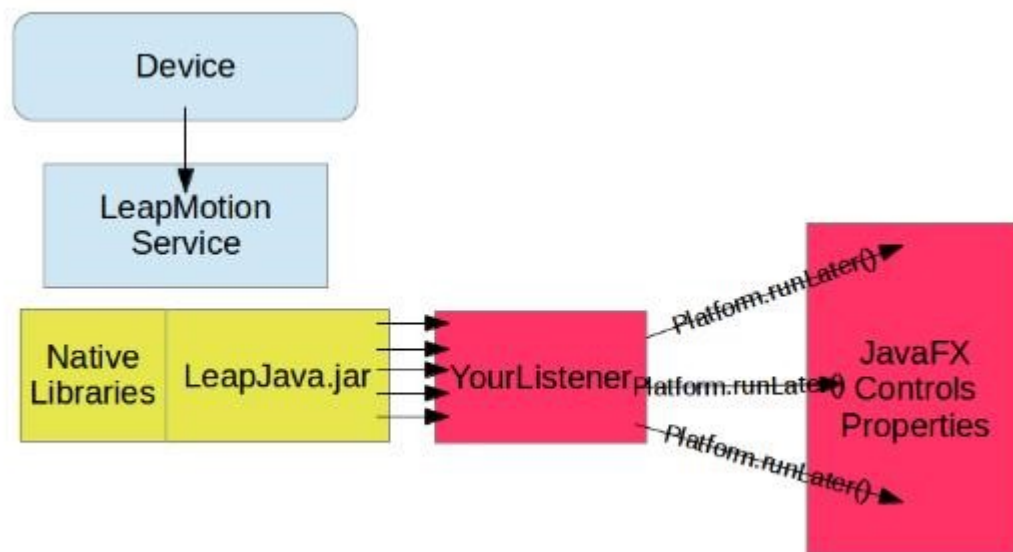
Figure 2: The Leap Motion API architecture (Vos 2014)

3.1.1 Coordinate Mapping

Mapping the coordinates of a user's hands to the screen was one of the first interactions

with the Leap Motion controller that we had attempted as it is a fundamental feature in Leap

Motion applications. There are two methods of mapping the user's x and y-coordinates relative

to the screen. One is found in example code on the Internet and we highly discourage its use

(Samarth 2015). The other is more supported by the Leap Motion API, and is detailed later in

this section.

The method commonly found online involves the use of the *locatedScreens()* method

found on the controller's frame within an *onFrame()* call. This method returns a list of *Screens* based on the number of monitors the user has and most examples will simply use the first *Screen* found on the list. The *locatedScreens()* method should not be used for mapping coordinates on screen as it makes assumptions about the physical space between the device and the monitor and assumes a standard size monitor. In certain environments we found that this method would return an empty list, rendering that frame data useless as it has nothing to map to.

The recommended alternative and the method we use to map coordinates on screen is to use the *InteractionBox* of the device, also found within the controller's frame. The *InteractionBox* is the bounded recognizable area of the Leap Motion directly above the device. Using the positions within the *InteractionBox*, one could normalize the coordinates of the hand within the bounding box and then map that to the application's window size.
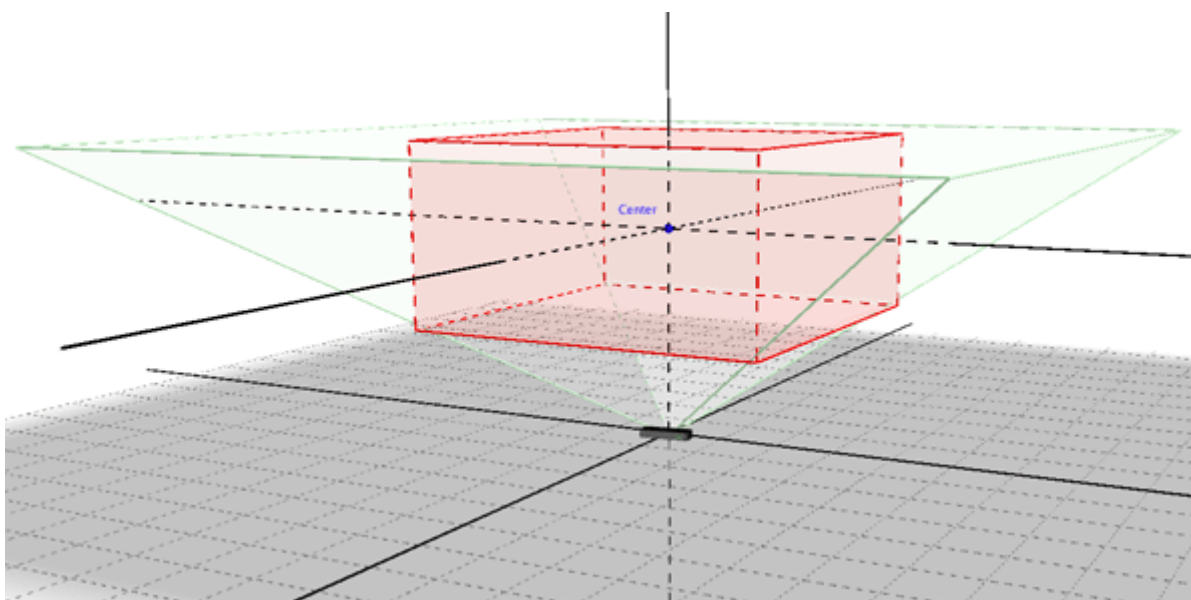

Figure 3: Leap InteractionBox (Coordinate Systems)

The *normalizePoint()* method in *InteractionBox* is used to convert the xyz coordinates of the desired object, whether it be the palm of the hand, point of a finger, or a Leap recognizable tool, into floating point numbers between 0 and 1. The code snippet below is a simple example of how mapping can be achieved using the Interaction Box.

```
Frame frame = controller.frame();
InteractionBox screen = frame.interactionBox();

// Validity checking and looping through HandsList should be done before using the data

final Hand thisHand = hands.get(0);

final Vector intersect = screen.normalizePoint(thisHand.stabilizedPalmPosition());

final double x = intersect.getX() * SCREEN_WIDTH;

final double y = (1 - intersect.getY()) * SCREEN_HEIGHT;
```

A hand/finger/tool object has many options when retrieving its position. The two most important methods in the above case are *palmPosition()* or *stabalizedPalmPosition()*. We found that x and y-coordinates for our project work best when using the stabilized method, and any uses of the z-axis worked better using the raw position.

Every frame provides new data of where the hand is relative to the screen of the user. All that is left to do is, within the *onFrame()* call in the controller, use these coordinates to map an object on screen to represent the hand. To simplify the code needed to use the Leap Motion to control the application while also retaining support for the mouse we decided to use the AWT *Robot* class in order to move the mouse cursor in response to the movement of the hand during the *onFrame()* method call. There are certain benefits and drawbacks to using this approach.

The AWT *Robot* allows for a more generalized code base. Code that works for the Leap Motion device should also be working for the mouse if the user decides to switch devices. Using the *Robot* class also enabled us to use built-in mouse listeners. That saved us a lot of time that would have been spent writing listeners specific to the Leap Motion device. It also lets us take advantage of the API for the cursor, such as switching icons based on context (an open or closed hand) with relatively little code.

The drawbacks of using the AWT *Robot* class mainly involve performance issues and slightly limited functionality. Translating positions and certain hand states into the mouse have a significant impact on performance. This also limits certain actions via the Leap Motion controller to some gestures that can be translated to mouse actions, such as clicking. There is currently a workaround in place, mostly seen in object interaction, which allows us to capture gestures performed on an object as long as the cursor is above the given object. Any and all recognized gestures will fire the method on that object allowing for a multitude of gestures for that object. With this modification, an object can be manipulated through several gestures. However, if the user decides to use the mouse, the manipulations possible with the mouse are limited as there is no method of translating those Leap gestures easily to the mouse.

3.1.2 Gestures

Once the coordinates of the hands are mapped on screen, code is needed to react to certain gestures the user performs on top of a scene object or a menu item. One of the first and most important gestures we implemented in the beginning of development is a grab gesture. The act of making a fist changes a JavaFX property variable that reflects the hand state (open or closed). Using this property, objects are able to respond to the changed property variable via an Observer pattern and perform a certain method.

Table 1: A list of gestures in our application.

Gesture	Source	Description
Circle	Built-In	Make a quick vertical circle with one or more fingers while hand is pointed towards the screen. Clockwise and counterclockwise are different gestures.
Grab	Custom	Make a fist with the cursor over an object to select, and open the fist to release.
Key Tap	Built-In	"Tap" finger down vertically towards the keyboard. Not currently used by the Presenter application.
Resize	Custom	With each pointer finger extended, move hands closer to shrink and object, and farther apart to make the object larger.
Rotate	Custom	With the first two fingers of one hand extended, rotate hand to desired angle.
Screen Tap	Built-In	With one finger extended, "poke" towards the screen.
Swipe	Built-In	Quickly move open hand across screen. Right-to-left and left-to-right are different gestures.

Initially, we believed that grabbing would be a simple concept to implement by counting the number of extended fingers. The Leap API does provide an *onFrame()* call that retrieves a list of extended fingers or an empty list if none are extended. This implementation quickly ran into problems as the thumb and index finger would frequently count as extended even though they were not, counting a thumb not tucked into the fist as extended. We then attempted a different approach using the *sphereRadius()* method, which calculates the size of a sphere based on the curvature of a person's hand. Demo representations of the sphere are shown below.
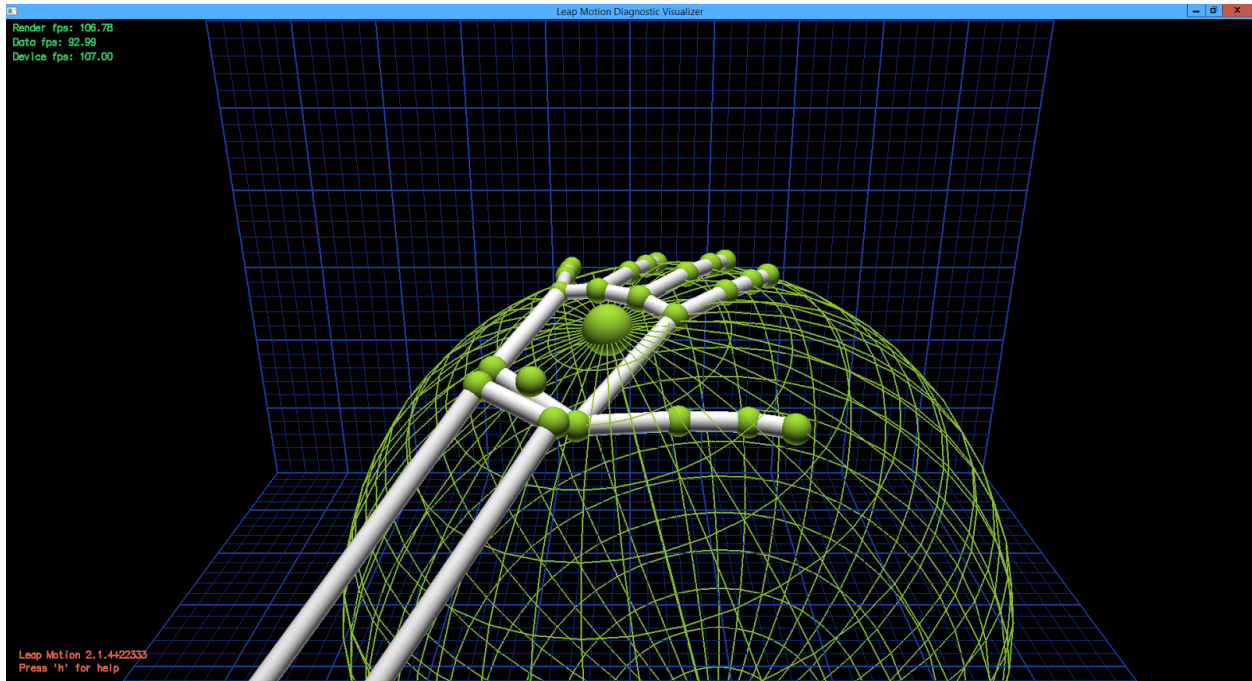
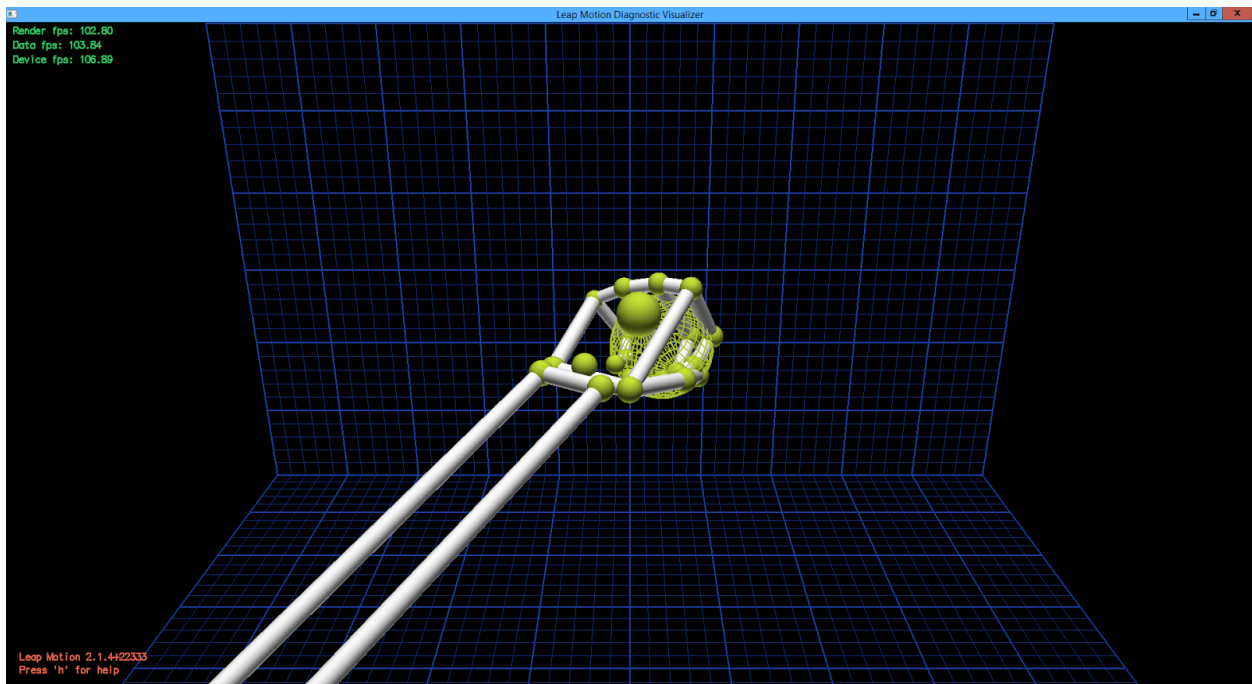Figure 4: Open Hand Sphere

Figure 5: Closed Hand Sphere

Testing with a few volunteer users, we found a sphere radius threshold that appeared to satisfy as many user hands as possible. After a few more user tests however, we found that one user with exceptionally large hands could not grab an object without modification to the sphere

radius threshold. Changing this threshold for one user affected several other different hand sizes.

By this time the Leap API had a few updates, one which noted an improved *grabStrength()* method. The method returns a float between 0 and 1 that defines how strong of a fist the Leap detects. After some testing with this new value, we found a threshold that allowed grabbing without completely being a fist, for users who may have difficulty making a fist for extended periods of time, and that also satisfied all of the users that tested the software.

The Leap update also boasted a *pinchStrength()* method that returned a float between 0 and 1 that defined how strong of a pinch the Leap detected between two fingers. Originally this was the intended gesture for moving objects, but we settled for the grabbing gesture as it was easier to perform for users during an extended period of time. A separate application, made from code ripped from the Leap Motion Presenter, used Blizzard's Hearthstone trading card game to test the pinching gesture to drag cards across the screen and place them on the board. Certain users grew tired of performing the pinch gesture after playing a few games, so we decided to keep the current grabbing implementation.

Other built-in gestures within the Leap API are used to increase the functionality of our application. The current system involves using the cursor to hover over the desired object to manipulate. This object is then passed into a Leap controller that listens for certain gestures that can be performed on an object. When they are performed, the corresponding method for that gesture is run for that object. The actions performed can vary between objects as the entity passed to the control is an *IObject*, the interface for our scene objects. These methods are implemented in the interface for our scene objects.

The Leap API supports *Swipe, ScreenTap, KeyTap,* and *Circle* gestures detected by the controller. These gestures can have methods in the interface that are called whenever the gesture

is detected. For example, the specific object manipulations currently in place are screen taps on video objects to play/pause and counter clockwise circles to rewind to the beginning. A screen tap on a text object will bring up a window to edit the content of the text; during presentation mode the screen tap will highlight the text to draw attention to it. We also implemented swipe gestures to be performed on the presentation screen itself to move forward or back a scene in the presentation. This brought up an issue of needing a specific swipe to move forward or backward based on direction and strength. Swipe direction was simple enough to calculate using the line *swipe.direction().getX() > 0*. The swipe strength was slightly problematic as it seemed to be based on the operating system used to run the software. The Leap API states that the minimum length of a swipe needed to recognize it as a valid gesture could be set via the *onConnect()* method call using the line below. This line could also be used to change any other native gesture configuration options by changing the *set()* methods parameters.

controller.config.set("Gesture.Swipe.MinLength", 200.0);

controller.config().save();

The problem with switching the value of the minimum swipe length was that the initial value tested in a Linux environment was not sensitive enough in the Windows environment. The units of the minimum swipe length are measured in millimeters, therefore we are not sure why this should be an issue. We found that the value for a full screen swipe in the Linux environment was ideally 400mm while the ideal value for the Windows operating system was 200mm.

3.1.3 Custom Gestures

The Leap API only supports swipe, key and screen tap, circle, pinch, and grabbing gestures through its *onFrame()* calls. One of the most basic actions required by our application, resizing, needed a gesture that was more flexible and intuitive for the user to adequately size

17

objects precisely. When deciding a good gesture to use for resizing, we looked at new interface design ideas that have been adopted by a large consumer base. Mobile smartphone and tablet interfaces use a pinch and stretch style combination of gestures to zoom in and out of certain objects on screen. To accommodate a larger screen size for a desktop application, we took the idea of pinch and stretch to apply to a two-handed gesture that has similar motions to the single-handed pinch and stretch.
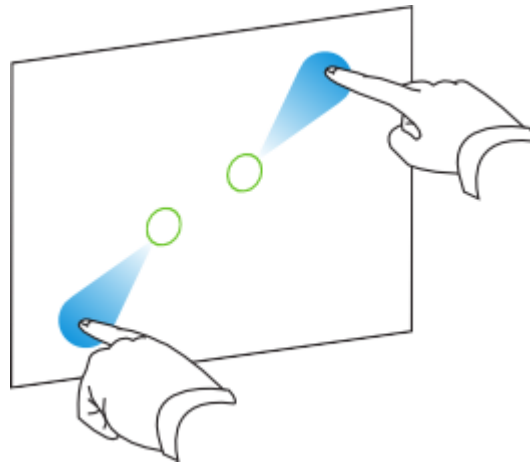

Figure 6: Two Handed Stretch Gesture (Using Gestures)

The Leap API does not currently have any gesture similar to this, nor does it provide two-handed gestures. For this gesture, as well as any other type of custom gesture, it is necessary to create a subclass of the *Listener* object supplied by the Leap API and use the available data on the *onFrame()* calls sent by the Leap controller. In the case of the resize listener, every *onFrame()* will begin by retrieving a list of *Hand* objects detected by the Leap controller. If two are found each with two extended fingers, then the resize gesture will calculate the distance between the two hands as a start point. While the two hands are in view of the device, the space between the hands will change based on whether the user would like to increase or decrease the size of an object. If the user wishes to decrease the size of the object, the user will bring their hands closer together, performing the pinch motion. The listener will then calculate the

percentage change of distance and pass that value to the appropriate JavaFX thread to manipulate the object being resized. As this gesture was custom-made for this application, several user tests were performed to ensure an appropriate level of control over this gesture can be attained.

The first iteration of this gesture immediately began calculating distances between the hands when the user introduced two hands on top of the Leap controller. Many users found this to cause drastic changes immediately to an object before they were prepared to make any changes. To facilitate this we changed the gesture to begin once the user has extended both pointer fingers as shown in *Figure 6: Two Handed Stretch Gesture*. While this helped, this was not intuitive for most users who generally bring both hands into view with their pointer fingers already extended. The current version of the resizing gesture now begins resizing when hands are brought together, within a threshold, on the z-axis and stops when the hands are no longer close to the same z value. This allows users the time to position their hands and allows them to manipulate objects when they are ready to perform resizing.

3.1.4 JavaFX Threads and runLater()

Tracking and gesture data have been observed from the Leap API controller and listener perspective. For changes to be seen on the interface portion of the application, this data must be supplied to a JavaFX thread to perform. No other thread may manipulate the values of the user interface, and this is one of the main reasons we chose JavaFX as our user interface library. By using JavaFX, we can separate the Leap threads, the UI threads, and JavaFX's *runLater()* method call. This allows us to easily hand off data from the Leap threads to the JavaFX threads.

3.2 User Interface

Our basic user interface is a simple white screen with nothing but an icon for the hand cursor. Off screen there are two menu bars. The one at the top of the screen is the main toolbar. It

has options to open, save, and exit the application, as well as add new scenes and objects, and start presentation mode. The toolbar at the bottom is the scene toolbar. The scene list with thumbnails of each scene is stored here.

3.2.1 Main Toolbar

The main toolbar contains all of the general actions for the presentation as a whole. It is broken up into a couple sections. The first section contains options to open and save presentations, and to close the application. The next section contains options to add scenes and other objects. There are also options to begin presentation mode, and to exit the menu.
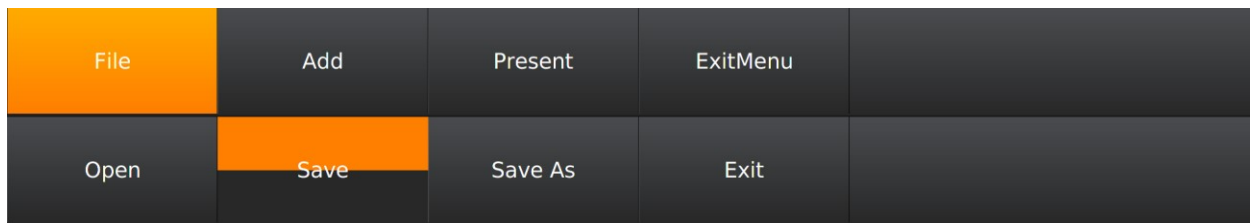

Figure 7: The main toolbar; save is halfway selected.

To access the main toolbar, the user needs to move the hand cursor up to the top of the screen. The first level of the menu will drop down. Options are selected from the menu by hovering over the option and dragging the hand cursor down again. For the first two options, this opens a submenu which functions the same way as the main toolbar.

We decided on the design of the main toolbar based on our own experiences and from individuals with some helpful input. The menus in the Presenter application need to follow the same intuitive principals that the rest of the application adheres to. A similar design was suggested in the Leap Motion developer blog (Plemmons 2013). Using the top of the screen as a way to access the menu is a very simple motion for users, but the threshold for opening the menu is small enough that it is not opened accidental by the user. Once the menu is open, the user can easily slide between menu options and down to sub-menus to get to the option they want. This design seemed to be easier for users than alternatives such as a menu opened by performing a

circle gesture.

3.2.2 Scene Toolbar

The Scene toolbar functions similarly to the main toolbar. Moving the hand cursor to the bottom of the screen triggers the toolbar to rise from the bottom. On the toolbar are thumbnails of each scene in order. There is also a counter of how many scenes there are, and the index of the current scene.
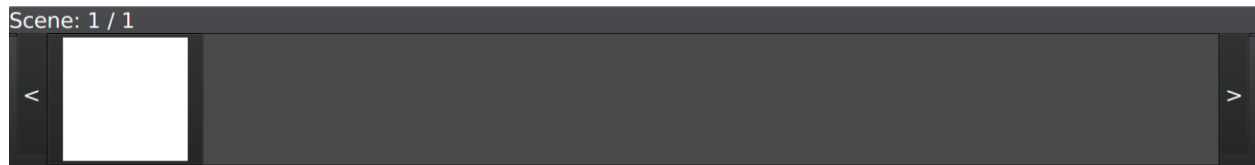
Figure 8: The Scene toolbar

3.2.3 Icons

There are a few custom icons for the hand cursor. The default cursor is an image of an open hand. When the user closes their hand to grab an object, the hand cursor changes to an image of a closed hand. There is also a cursor for pointing, which is used for resizing and when the Screen Tap action is performed.
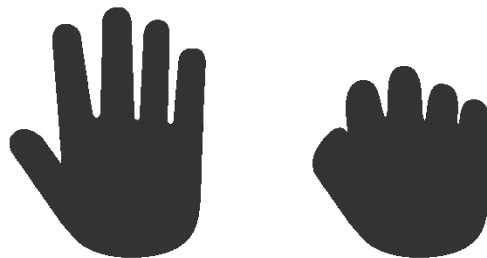
Figure 9: The open and closed hand cursor icons.

3.2.4 ControlsFX

Early in development we faced issues when displaying notifications or prompts to the user. Notifications are required for alerting the user to changing state of the application, such as beginning a presentation, or as a way of allowing the user to input text. Initially we tackled these problems by incorporating Swing popup elements, *JOptionsPane.showDialog()* being one of our

first solutions to text input through a pop up prompting input. As JavaFX is built on the Swing API, we believed that calls to the Swing API from a JavaFX thread would be handled well, but undefined behavior occurred across operating systems. In the Linux environment, the call was handled gracefully and input could be successfully recorded from the dialog, but in the Windows environment this caused the application to hang in most cases. JavaFX lacked the API required to create simple dialog windows for input, and this problem was then solved by incorporating the ControlsFX library into the project.

ControlsFX is a third party, open source library that allows for a larger selection of UI controls for Java 8 and later versions. Among these controls are Dialog windows, similar to the ones provided in the Swing API. These dialog windows are also customizable to create a wide variety of dialogs, which allowed us to create dialog windows that were consistent to the look and feel of the existing UI elements.
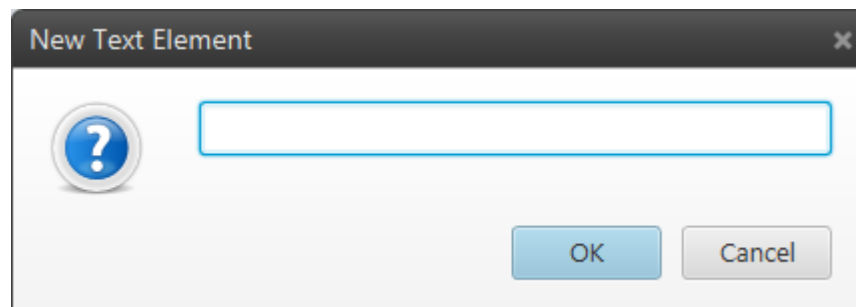
Figure 10: The text input dialog, created with ControlsFX

There were also other features within the library that allowed us to create seamless notifications for changes in application state. Due to the nature of the applications control scheme, both presenting and the creation of a presentation are full screen with hidden toolbars to allow the entirety of the screen to manipulate objects. ControlsFX allowed us a notification element to alert the user to a changing state when entering or exiting presentation mode. These are helpful in the cases where users may possibly enter present mode without intending to, as before no notification would be displayed and certain UI elements, such as toolbars, are disabled

in this mode. This UI element helped to solve the user confusion during these situations.

Figure 11: Notification presented to the user when entering a different state

3.3 Scenes & Objects

A scene is a group of objects in the Leap Motion Presenter. It can be compared to a slide in Microsoft PowerPoint, but it has more functionality in the Leap Motion Presenter. The objects supported by the Leap Motion Presenter are images, videos, and text. Text is editable, videos can play and pause, and all objects can be moved and resized with hand gestures. Additionally, a gesture can be used to change scenes. All control of the presentation is controlled with gestures through the Leap Motion controller.

Scenes are built from the JavaFX *Parent* class. This is a JavaFX *Node* subclass that is specifically designed to contain other *Nodes*. *Parent* is an abstract class that has methods for storing and managing children nodes, and we have added to these by being able to differentiate between regular nodes and our presentation objects. Additionally, being a subclass of *Node*, it has access to all of *Node*'s protected methods, which can be used to define the size and shape of the scene, as well as the ability to set Properties to monitor instance and local variables in the scene. The *Scene* class also has the ability to take an image as the background of the scene, though this has not yet been implemented on the Leap side of the application.

We used the Factory Pattern to handle object creation within scenes. The class *ObjectFactory* is a singleton that returns an object of type *IObject*, the interface that all scene object classes implement. This interface contains methods that facilitate the moving and resizing of scene objects, gesture specific methods that are called when the cursor is on top of the object

and the specified gesture is performed, and methods that help in saving and opening presentation files. All scene objects are in a scene, and they are referenced by accessing the current scene. This is done through the *LeapSceneManager*, another singleton that keeps a list of scenes, facilitates adding and removing scenes, and changes and stores the current scene viewed by the user.

3.3.1 Objects

There are currently three types of scene objects supported by the Leap Motion Presenter. They are images, video, and text. All of these scene objects support movement by the grab gesture, and resizing with our custom resize gesture. Additionally, videos can be played and paused with a tap gesture. The tap gesture also can be used to edit text in presentation creation mode, or used to highlight a piece of text in presentation mode. Any object can perform a specified action when a gesture is performed on it by editing the corresponding method within that object. For now, *onScreenTap()* and *onCounterCircle()* are implemented within all scene objects via the interface and are called when these gestures are performed on them.

We started creating scene objects by creating image objects the user could instantiate. The image objects in the Leap Motion Presenter are inherited from the JavaFX *ImageView*. *ImageView* inherits from *Node*, and has all of the position, size, and Property methods from *Node*. Being a subclass of *Node* also makes it very easy to add *Image* objects to the parent *Scene* object.

The initial position of the image is set by the X and Y instance variables from *Node*, but any movement after the initial positioning is handled by the *TranslateXProperty*() and *TranslateYProperty*() methods from *Node*. Using these property methods allows us to bind the position of the hand or mouse to this object when a fist is detected to the position of the hand so

that it moves as the hand or mouse moves. Resizing is handled by the width and height variables from *Node* for both initial size and future size changes. Resizing is determined by a percent change sent from the Leap Motion Controller by a custom made resizing gesture. The resize gesture uses both hands and waits for the user to align these hands before any resizing is performed. Once the hands are aligned, the percentage distance change between the hands are used to determine the percentage height and width increase of the desired object. Once the hands are not aligned, the resizing gesture is stopped.

The next scene object we implemented was videos. The *Video* class extends the JavaFX *MediaView* class, which is another *Node* subclass. To play a video, the *MediaView* needs media to play, generally in the form of an .MP4 or .FLV file. This media is then played using a *MediaPlayer*. The *MediaPlayer* is then displayed on the screen by *MediaView*. Being a subclass of *Node*, *MediaView* was easy to set up with moving and resizing, which was implemented in the same fashion as *Image*. Additionally, the built in Leap gesture Screen Tap was implemented to play and pause videos. All of the *MediaPlayer, MediaView,* and *Media* classes needed to form a viewable video object on the screen are encapsulated by our scene *Video* object.

Our *Video* object at this point possibly supports playing .MP4 or .FLV files found on the web by supply a direct URL to the video, but it is not implemented as a possible option to the user. There is also the possibility of supporting YouTube© playback by supplying the embed link of the video and the use of JavaFX's *WebView* object to load the video. This was ultimately scrapped as the *WebView* would not behave like the other objects with the Leap Motion device and a significant amount of work would need to be put in to make it work well within the presentation environment. The possibility of adding the *WebView* as an online media player within presentations is also another possible future addition.

The last scene object we implemented was text. *Text* is the most complex of the scene objects supported by the Leap Motion Presenter. When a new *TextBox* object is created, an input dialog is presented for the user to input the text to be shown in the *TextBox* object. Moving text is set up the same way as moving images and videos. However, while *Text* is an indirect subclass of *Node*, it is also a subclass of *Shape*. *Shape* changes the way its objects are sized, so the height and width instance variables from *Node* do not necessarily do what they should. Instead, when the *resize()* method is called, we change the font size of the text. We originally thought that approach would be visibly slow and not work well with our application, but after some testing we decided it was the functionality we wanted with no visible latency or performance decrease. In addition to the move and resize gestures, the Tap gesture from Leap opens a new window to edit the text box when not in presentation mode. In presentation mode, tapping the text box highlights the text, and tapping again will put the text back to its original color. As of writing, there is no way to change color or font of the text.

3.4 File System

An important function in a presentation application is the ability to save and open presentations. Java does not present an easy method for saving and opening files, and the intricacies of JavaFX positioning makes the task even more difficult. After a bit of research, we determined that an XML-based save file would be best suited for our application.

When the user clicks the save button, a few different things happen. Each scene object type has its own save an open functions that either give or receive a snippet of XML. This miniature XML document stores information about the object including position, size, and the file location of the object if it is a video or image. That data is then saved in a master XML file, with all of the data from all of the other scene objects and all of the scenes. That XML file is

saved in a folder in a location of the user's choice. Copies are made of all of the video and image files (in the case of a Save As…) and those are also put into the folder. The information about file location stored in the XML file points to the files in the save location, not the original file locations.

The opening process is very similar, but happens in reverse. Each scene from the XML file is created, along with all of the objects in each scene. The objects are created based on the information stored in the XML file, including file location, position, and size. After the opening process is complete, the presentation is the same as it was when it was last saved.

Chapter 4: Results & Analysis

4.1 What did we accomplish?

In designing and implementing the Leap Motion Presenter, we identified several important strategies for designing an application that uses a completely new control scheme. The most crucial area of focus when designing this application was the user experience and interactions between the user and the application. With a new medium of interaction, new user expectations for interaction arise. A good example is using the grab gesture to initiate movement of objects, as the relationship between grabbing an object in the real world is mapped to grabbing an object in the presentation. Most of the other gestures, on the other hand, do not map to an action performed in daily life, but still successfully map to expectations of users through other established applications.

Many desktop applications follow common design elements, such as clickable buttons that appear pressed when clicked to signal a button press to the user or toolbars at the top of the application for the bulk of menu items. These applications follow these design patterns as users have been accustomed to using applications that make use of these interface elements, and the same can be done with Leap gestures. The resize gesture was inspired by mobile applications that support zooming in and out through pinching and squeezing. By slightly modifying it to work with two hands, but still retaining the idea of pinching and squeezing, we created a way of resizing objects that the user is familiar with. Being able to tie in gesture actions with other relatable actions, whether it be mimicking real life actions or existing implementations, translates to actions that users will be able to perform and remember with ideally as little frustration as possible.

Along with gestures, menu design needs to ensure the actions that can be performed on

the device are utilized to successfully navigate through user options, rather than attempting to mimic current desktop menu trends that have been designed with the mouse and keyboard in mind. One of the earliest examples of this are the specialized toolbars, as selections are not made by having the user push downward on the desired menu item with their hand. This type of design attempts to utilize the device's strengths in hand tracking as well as use a gesture that falls into the expectations of the user to a certain degree.

4.1.1 Performance Metrics

In our application, we have 92 java classes. These are spread across 15 packages, and have 304 methods in all. There is an average of 3 ⅓ methods per type with a minimum of zero and a maximum of 43, and a total of 225 fields with an average of 2.18 per type. In the methods, there are an average of about 10 lines of code per method, with a minimum of 1 and a maximum of 82. All told, there are 3676 lines of code, 390 comments, and 2092 semicolons. The comments-to-code ratio is 10.6%.

Additionally, our application had no unit tests. While we could have written unit tests for many of the functions in the application, the application as a whole is a graphical interface. We decided that testing the application would be easier to accomplish from the user interface than by running hundreds of unit tests. While we understand that unit tests may have helped us identify issues in some of the functions, we implemented enough real-time debugging code that we felt the unit tests to be less necessary.

4.2 How do accomplishments compare to our original plans?

When we started this project, we intended it to be a user application on the Leap Motion app store. Many of our original design decisions were based on how we envisioned the program to work in the app store. For example, we started out by making the top-level main function as

modular as possible, so that when the time came to release the application, it would be easy to integrate. However, about halfway through the project, we realized that making an application that works without bugs and that users would really want to use instead of an existing application would take a lot more time than we had available to us. We do believe that what we did create can be built upon to one day become the user application we envisioned it to be.

One of the features we had planned for our application was the ability to draw on the presentation area, either while creating a presentation or during the presentation itself. While planning the application, we intended to make use of the Leap's ability to recognize tools to allow the user to draw anywhere on the screen. We believed that this would allow both a way of triggering drawing as well as giving the user the experience of drawing with an object that mimics the behavior of drawing with a pen or pencil. We quickly recognized that the Leap is unable to recognize common tools, such as pens, pencils, or markers, with the precision we required.

With the lack of tools, we lost a simple way of differentiating between the user drawing on the screen and performing other gestures, and at that stage of development we had begun to run out of possible gestures to initiate certain actions. Without the use of a tool, several gestures that would control the drawing portions of the application, such as selecting a color or a brush type, would need to be done with bare hand gestures that are already in use by other actions. Hopefully in the future, the Leap Motion API will support more robust tool recognition to reliably perform gestures we have already implemented, but with the addition of the tool to tailor these gestures for drawing. Some examples of these gestures are tapping with the tool to bring up a color wheel or a screen wide swipe with the tool to clear the screen of all drawings.

Another design alteration we had to make in the middle of the project was to abandon the

idea that the Leap Motion would be the only input device. When we were planning out the saving and opening of presentations, we realized that there was no great way to use the Leap Motion as a file chooser. Having to use the operating system's file system with Leap controls was unacceptable, and creating an entire new file system viewer specifically tailored for use with the Leap Motion was out of the scope of this project. We decided that while the Leap Motion is a good device for our application, but for actions like choosing files and typing in text were tasks best left to the keyboard and mouse.

Chapter 5: Future Work

While the base of the application is well-developed, there are still several features that we believe would make the application more complete. Some of the features described may be vital to the user when using the application, while others may be completely optional. Several of these features may also have progress towards them, but were left unfinished as we either could not finish them on time or believed that the current progress was enough to demonstrate the application's capabilities. We believe that these features are all possible to implement and would help the application reach a more complete state.

A core feature that is currently still in progress is the movement between currently selected scenes and changing the order of scenes. Currently, moving between scenes is done either by swiping left or right, or quickly grabbing and releasing a scene icon in the scene bar at the bottom of the application. We are currently unsure of whether the grabbing of a scene is desirable to the user and it is possible that this gesture for selecting scenes would need to be changed to a more comfortable gesture, such as a screen tap. Additionally, the reordering of scenes has not been implemented. One possible way of reordering scenes would be to make use of the grab gesture to pick up scenes in the scene bar and move them to the desired position.

Customizable fonts and colors for text elements also remained unimplemented. Currently, the controls for text are prompted through a radial menu, spawned by performing a counterclockwise circle on a piece of text. That allows the user to customize the text, such as allowing bold or italics text. The menu items for fonts and colors are there, but do not perform any actions. We left these features out as the other menu items demonstrate the capabilities of the radial menu as a method of allowing the user to manipulate properties of an object. Fonts and colors are also much trickier when using radial menus, as there are several system fonts that may

be available to the user and colors would need their own element like a color wheel. Possible future work would be to implement a method for the user to customize the fonts and colors of text. A quick and simple way to allow for both would be to spawn new radial menus specific to the desired action, such as a radial menu filled with a limited selection of fonts and a radial menu with color options similar to a color wheel. While spawning a new color wheel for changing colors is possible in the current application, creating a menu to select the various amounts of fonts on different platforms would be difficult to implement just by using Leap. It is possible that this is another action that is only suited to the mouse and keyboard, or one might select a number of the most commonly used fonts and instantiate a radial menu with only those fonts as options.

Another possible modification to existing objects would be the addition of embed links for videos found online, such as through the YouTube or DailyMotion© services. While creating the video object, we believed that the current implementation of *Media* and *MediaView* in JavaFX would support embeddable video. While this was not possible through the *MediaView*, it is possible to implement through the JavaFX *WebView* object. An example of embedding a YouTube video with its embed link in a JavaFX program would be:

// Embed link from YouTube (Could be from any service that provides one)

final String content_Url = "<iframe width=\"560\" height=\"315\"

src=\"http://www.youtube.com/embed/C0DPdy98e4c\" frameborder=\"0\"></iframe>";

// Initiate new WebView loading the embed from above

final WebView webView = new WebView();

final WebEngine webEngine = webView.getEngine();

webEngine.loadContent(content_Url);

// Add this web object to the root which is the root container in our application

root.getChildren().add(webView);

The code above will place a web view in the application with the desired video object. Difficulties using this as a solution are the controls using the Leap Motion device, as the *WebView* has its own browser-like controls which override the application controls. Another problem with this is the controls vary with the players provided from the many different video hosting services. The web view must also be properly sized to display only the contents of the video and the player controls.

Drawing is a feature that unfortunately was high on the priority list, but has several problems that we are currently unsure on how to solve. Initially we believed that the use of tools would help make this feature possible, but tools are currently too unstable to be considered. Therefore, to implement drawing we would need a gesture or menu item to allow the user to begin a drawing state. While in this drawing state, gestures would be needed to allow the user to draw, erase, customize brushes or brush size, and select colors. This would be especially difficult in the present state, as all toolbars are disabled during this state. Unfortunately, without the use of tools, this feature would need several new gestures that are not available natively and would require a significant amount of time to implement. These new gestures would also need significant user testing to ensure they function properly with user actions as well as being comfortable for the user to perform and remember. Perhaps it would be best to wait for the Leap Motion API to improve tool detection before attempting to implement this feature.

Another feature that we did not get to implementing was scene backgrounds. There is a constructor in *LeapScene* that takes an image file that can be used as a background, but currently nothing calls it. There are a couple design decisions for scene backgrounds, such as what kind of images should be allow, if there should be a default set of backgrounds, and if different scenes

can have different backgrounds. That being said, the actual implementation of scene backgrounds is pretty straightforward, and some of the code is already in place.

The last feature that we believe would make the application much more complete is changing the gesture for rotating. Currently, rotating is done by extending two fingers, and the object being selected is rotated to the angle of the user's hand. Unfortunately, there are times when the Leap Motion sees two fingers even if the user is making a fist or has their hand completely open. Additionally, it is extremely difficult to rotate an object back to its original orientation once it has been rotated. A better gesture for rotating would definitely make the process smoother. Changing the way rotating currently works may or may not be required, depending on the gesture chosen to replace it.

While conducting informal user testing, we noticed that certain settings that work for some users did not work as well for others. As it is nearly impossible to develop gestures that work exactly the same for all users, a settings menu that allows the user to set custom sensitivities or values for certain gestures would be a great addition in the future. Grabbing and swiping gestures have threshold values that dictate how closed the hand must be before it detects that the user wants to grab or how quick a swipe is performed before the program recognizes the user wishes to swipe. These values may change from user to user and in some cases from one machine to another, and a settings menu that allows a user to either select values or to automatically calibrate the application for the user is highly desirable.

Chapter 6: Conclusion

The Leap Motion Presenter is a working prototype of a presentation application using the Leap Motion device. The application takes the more intuitive gestures the Leap Motion offers and channels them into a more creative and reactive presentation than is currently offered by other software. The ability of a presenter to move and resize objects on the fly with precision allows a greater connection with the audience than would be afforded with pre-determined animations. Additionally, the flexibility of the Leap Motion device allows the presenter to take a step back from the podium, and focus more on their audience, and less on the keyboard and mouse.

The combination of JavaFX and Leap Motion listener threads is the driving force of this application in its current state, and an understanding of their relationship is needed to develop the application further. Once the listeners required to capture certain information from the device are created, all the listener must do is send the data to a JavaFX application thread. Once the JavaFX application thread has the data, the application can then change the UI state to reflect the captured device input. In this way, a JavaFX application can be made independent of the method of input, whether it be mouse, touchscreen, or a Leap Motion device.

Another key concept in the Leap Motion Presenter are JavaFX Nodes. Nodes are the visual portions of the JavaFX application, both UI controls and presentation objects. They are also the containers for UI elements and it is important to understand Nodes and the relationship between Nodes and container Nodes to create an effective user interface when developing on a new input device.

While this application is far from complete, the current prototype can easily be built upon. With some additional features and some tweaking of the current gestures, we believe that

this could be a functional user application, suitable for use in real-world presentation scenarios. While we have not created an application that solely uses the Leap Motion device, certain actions are not easily performed with the Leap, such as text input. However, we have created an application that uses the Leap to the best of its ability, and other actions that currently use the mouse as an input device, such as file choosing, can be expanded upon to use the Leap Motion device. The application is far from perfect, but it demonstrates that certain tasks are better completed with the Leap Motion device than with the traditional keyboard and mouse. We believe that with more time and effort, the Leap Motion can become a well-known addition to the common user experience.

Glossary

Airspace: The store for downloading or buying applications designed for the Leap Motion, or having Leap Motion support. These applications may be standalone programs or a web application.

API: Application Programmable Interface, a set of standards followed when writing an application.

AWT Robot: This Java class simulates button presses and mouse movements systematically. Using this class, controlling mouse movements through the Leap is made fairly simple.

Binding: Binding links the values of JavaFX Properties to other instance or local variables. If a variable bound to a JavaFX Property changes, the value of the Property and all other variables bound to it is changed to the new value. The same happens when the Property value is changed directly.

Circle: A Leap Motion gesture performed by making a circle with one or more fingers. The circle can be clockwise or counterclockwise.

ControlsFX: Custom JavaFX library designed to improve the quality of the UI controls.

CSS: Cascading Style Sheet, used for describing the formatting and look of a document written in a markup language such as HTML or XML.

File Chooser: A dialog box provided by the operating system for choosing a location to open or save files.

FXML: JavaFX's XML implementation.

Grab: A custom gesture created for the Leap Motion Presenter. It is performed by closing your fist to activate, and opening your fist to deactivate.

GUI: Graphical User Interface, the view of an application the user normally sees.

***InteractionBox*:** The area in space above the Leap Motion that the device can see. Information about hand positioning and other data can be accessed through the interaction box.

***IObject*:** A Java interface that all presentation objects (images, video, text boxes, and shapes) implement. Includes methods for moving, resizing, rotating, saving, and opening.

JavaFX: A set of graphical interface packages to develop user interfaces for Java applications. Uses FXML, a type of XML based language, to design the interface while keeping functionality in code. CSS may also be used to style elements in the FXML document.

JavaFX Properties: Monitored instance variables that automatically update linked variables,

and change the values of the related object in the user interface. See binding.

Java Swing: Java's main graphical user interface library.

Key Tap: A Leap Motion gesture performed by "tapping" your finger down toward the Leap Motion device.

Leap Motion: The company that worked on and released the current version of the Leap Motion device. Started by David Holz, the creator of the Leap prototype, and Michael Buckwald.

The Leap: The controller released by Leap Motion. This device connects to a computer and reads hand motions above it. The device uses three infrared cameras to track movements as small as 0.01 millimeters.

Listener **Class:** A Leap Motion class containing the functionality to monitor for Leap Motion gestures.

*locatedScreens()***:** Gets a list of all screens currently accessible by the program.

*Node***:** The main JavaFX object base class. All presentation objects, including scene, inherit from the *Node* class.

*onFrame()***:** Called on every frame the Leap Motion device obtains data. Data about hand positions and other information at that instant is available in the *onFrame()* method.

*runLater()***:** Creates a new JavaFX GUI thread. Code in the *runLater()* method is executed separately from the rest of the program code. It can be assumed that code in the method is run later.

*Scene***:** A class that contains a group of presentation objects and a background image. Data on the location and size of each presentation object is stored in the Scene.

Screen Tap: A Leap Motion gesture performed by "poking" in the direction of your computer screen.

Swipe: A Leap Motion gesture performed by moving your hand horizontally or vertically above the Leap Motion, parallel to the screen. The swipe can be performed from right-to-left, left-to-right, or up and down.

XML: Hybrid human and computer readable markup language for formatting documents.

References

Coordinate Systems. n.d. Web. 2015.

    <https://developer.leapmotion.com/documentation/csharp/devguide/Leap_Coordinate_M
apping.html>.

CSS Tutorial. n.d. Web. 2014. <http://www.w3schools.com/css/default.asp>.

Foster, Tom. *Will These Guys Kill The Computer Interface As We Know It?* 22 July 2013. Web.

    2014. <http://www.popsci.com/technology/article/2013-07/will-these-guys-kill-
computer-interface-we-know-it>.

History of Microsoft Commitment to Accessibility . 2015. Web. 24 March 2015.

    <http://www.microsoft.com/enable/microsoft/history.aspx>.

Hutchinson, Lee. *Hands-on with the Leap Motion Controller: Cool, but frustrating as hell*. 27

    July 2013. Web. 2014. <http://arstechnica.com/gadgets/2013/07/hands-on-with-the-leap-
motion-controller-cool-but-frustrating-as-hell/>.

Leap Motion. n.d. Web. 2014. <https://www.leapmotion.com/>.

Pawlan, Monica. *What Is JavaFX?* April 2013. Web. 2014.

    <http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm>.

Plemmons, Daniel. *Rethinking Menu Design in the Natural Interface Wild West*. 16 December

    2013. Web. 2014. <http://blog.leapmotion.com/rethinking-menu-design-in-the-natural-
interface-wild-west/>.

Pogue, David. *Leap Motion Controller, Great Hardware in Search of Great Software*. 24 July

    2013. Web. 2014. <http://www.nytimes.com/2013/07/25/technology/personaltech/no-
keyboard-and-now-no-touch-screen-either.html?pagewanted=2&_r=1>.

Richardson, Nicole Marie. *One Giant Leap for Mankind*. 28 May 2013. Web. 2014.

<http://www.inc.com/30under30/nicole-marie-richardson/leap-motion-david-holz-michael-buckwald-2013.html>.

Rouse, Margaret. *Rich Internet Application (RIA)*. September 2007. 2014.
<http://searchsoa.techtarget.com/definition/Rich-Internet-Application-RIA>.

Samarth. *Not able to perform a tap or any gesture on a jbutton!* 2 December 2013. Forum. 2015.
<https://community.leapmotion.com/t/not-able-to-perform-a-tap-or-any-gesture-on-a-jbutton/348>.

Using Gestures. 25 April 2011. Web. 2015.
<http://onlinehelp.smarttech.com/english/mac/help/notebook/10_0_0/UsingGestures.htm>.

Vos, Johan. *Leap Motion and JavaFX*. May 2014. Web. 2015.
<http://www.oracle.com/technetwork/articles/java/rich-client-leapmotion-2227139.html>.