**Final Year Project**

**Gesture Interpreter**

Andrew Ladlow

33421889

Contents

1. [Introduction 3](#_Toc442749170)

[1.1 Motivation 3](#_Toc442749171)

[1.2 Aims 4](#_Toc442749172)

[1.3 Report Overview 4](#_Toc442749173)

1. [Background 5](#_Toc442749174)

[2.1 Leap Motion 5](#_Toc442749175)

[2.2 BSL in Education 6](#_Toc442749176)

[2.3 Gesture Recognition 6](#_Toc442749177)

[2.3.1 Dynamic Time Warping 6](#_Toc442749178)

[2.3.2 K-Nearest Neighbour & Support Vector Machines 7](#_Toc442749179)

[2.3.3 Hidden Markov Models 7](#_Toc442749180)

[2.3.4 Artificial Neural Networks 7](#_Toc442749181)

[2.3.5 $P Point-Cloud Recognizer 8](#_Toc442749182)

[2.4 Similar Applications 8](#_Toc442749183)

[2.4.1 LeapTrainer 8](#_Toc442749184)

[2.4.2 UNI 8](#_Toc442749185)

1. [Design 9](#_Toc442749186)

[3.1 User Requirements 9](#_Toc442749187)

[3.2 System Architecture 13](#_Toc442749188)

[3.3 User Interface Design 15](#_Toc442749189)

1. [Implementation 16](#_Toc442749190)

[4.1 Leap Motion Integration 16](#_Toc442749191)

[4.2 Graphical User Interface 16](#_Toc442749192)

[4.2 Hand Visualization 17](#_Toc442749193)

[4.4 Gesture Recognition 18](#_Toc442749194)

1. [The System in Operation 21](#_Toc442749195)

[5.1 Initial Application State 21](#_Toc442749196)

[5.2 Hand Enters Field Of View 22](#_Toc442749197)

[5.3 Recognition 22](#_Toc442749198)

[5.4 Final Score 23](#_Toc442749199)

[5.5 Calibration 23](#_Toc442749200)

1. [Testing and Evaluation 24](#_Toc442749201)

[6.1 A 24](#_Toc442749202)

1. [Conclusions 25](#_Toc442749203)

[7.1 Aims Analysis 25](#_Toc442749204)

[7.2 Limitations 26](#_Toc442749205)

[7.3 Future Work 26](#_Toc442749206)

[7.4 Summary 26](#_Toc442749207)

[References 27](#_Toc442749208)

**Chapter 1**

# Introduction

## 1.1 Motivation

Human-computer interaction has, for many years, been limited to mouse pointing and typing based devices. With the more recent development of smartphones we have seen numerous breakthroughs with touch based devices. These methods are all inherently unnatural when compared with real world interaction, however. In all of these cases users must interact through some kind of interface as opposed to naturally communicating via hand signals or body language.

Currently, there are several released devices which aid in monitoring, recording and displaying these more natural forms of communication. Devices such as the ‘Microsoft Kinect’, ‘ASUS Xtion Pro’ and ‘Apple Primesense Carmine’ all feature some form of motion capture – the issue lies in their level of accuracy, especially with more precise or intricate gestures – e.g. movement of fingers as opposed to movement of arms. The gesture interpreter aims to utilise one of these devices, the Leap Motion, in order to facilitate natural gesture communication – particularly gestures found in the British Sign Language (BSL).

According to Weichert, et al [13] the Leap Motion is capable of recognising movement with an overall average accuracy of 0.7mm, a degree that devices such as the Kinect were unable to reach in comparison. In addition, a recent V2 firmware update released for the Leap Motion improved tracking performance and enhanced the features of its API.

The proposed application will aid users who wish to learn or reinforce their current knowledge of the BSL. The low cost of the Leap Motion device means it could be purchased by users themselves or incorporated into current sign language courses in order to help streamline the overall learning process.

## 1.2 Aims

The application should fulfil the following main objectives in order to achieve the goal described above:

* Record gestures performed by a user
* Recognise the gestures representing BSL alphabet characters and distinguish between them
* Given a gesture input, display a matching gesture with a similarity score

## 1.3 Report Overview

The structure of the report is as follows:

Chapter 1 discusses the problem and the proposed solution, including its aims.

Chapter 2 analyses background research and related work as well as any similar applications.

Chapter 3 identifies the user requirements for the application, and subsequently details appropriate architectural designs and initial designs for the user interface.

Chapter 4 covers the implementation of the application’s features, providing an in-depth overview of the program code.

Chapter 5 describes the operation of the application with the aid of screenshots.

Chapter 6 covers the applications’ test procedure and test results, as well as an evaluation of the system.

Finally, chapter 7 concludes the report, analysing how well the application met its original aims, any identified limitations and scope for future work. The report ends with an overall conclusion summarising the project.

**Chapter 2**

# Background

## 2.1 Leap Motion

The Leap Motion is a palm sized USB device which tracks hands and fingers using optical sensors and infrared lights. The device was first released by Leap Motion, Inc. in July 2013[3]. A major software update, firmware version 2, was subsequently released for the device in May 2014[4]. The update claimed improved tracking performance and an enhanced API feature set. An image of the device is shown in figure 2.1.



**Figure 2.1:** Leap Motion device

The use of the Leap Motion controller in order to detect sign language gestures was previously investigated by Araullo, et al [8]. The authors noted that although the controller showed potential, further development of the API was required. This was mainly due to inaccurate hand detection in certain scenarios e.g. pinching fingers together, interlocking hands or holding one hand above the other – all of which created anomalous data due to the obscuration of finger positions. Despite this, recognition improvement is still certainly possible; these tests were carried out prior to the release of the aforementioned V2 firmware update for the Leap Motion in 2014 which led to improved tracking performance. As such it’s likely that the current accuracy of the device has also since improved.

## 2.2 BSL in Education

BSL courses such as those offered by the nationally accredited Signature [10] award a number of qualifications ranging from ‘Level 1’ to ‘Level 6’, with each level representing an incrementing complexity of known vocabulary required in order to qualify. Each level’s accompanying qualification is composed of a number of modules, each of which focus on a particular subtopic e.g. ‘BSL conversational skills’ and ‘Understand varied British Sign Language in a range of work and social situations’. All of these modules include guided contact time as well as additional work intended for private study. There is also an accredited online learning resource available for an additional cost.

The problem with this current system is that new users are incredibly reliant on their tutor for guidance – online resources help but only to a certain degree, users often need reassurance that they are performing gestures correctly and the instant feedback provided by the proposed software would do just that. In turn this would benefit both the students taking these courses and the schools offering them. The use of the low-cost Leap Motion could hence be used as an additional resource to augment the user’s understanding of the BSL in both tutor guided learning hours and private study.

## 2.3 Gesture Recognition

Probably the most important aspect of the system is its gesture recognition – as we’re dealing with data in real time, a suitable algorithm should calculate match results on the fly without causing delay or otherwise affecting the user when the application is running. Most approaches apply some form of machine learning on the Leap Motion frame data, or a specific subset of it. Generally it’s difficult to say with certainty that one algorithm is better than any other other due to the variance of testing conditions, input data etc. Some feasible examples are shown below.

#### 2.3.1 Dynamic Time Warping

The use of dynamic time warping (DTW) was explored by Vikram, et al [12]. The appeal of DTW is that it isn’t reliant on the time taken or speed of each input in order to accurately compare them – this is particularly useful with gestures which are often performed at varying speeds. The authors demonstrate how DTW could be applied to 2D handwriting gestures and suggested that it could be extended for use with 3D data, i.e. gestures. They conclude that the DTW approach used is suitable for real-time comparison but it remains to be seen whether that is still the case when dealing with the increased complexity of gestures compared with just handwriting.

#### 2.3.2 K-Nearest Neighbour & Support Vector Machines

K-Nearest Neighbour (KNN) and Support Vector Machines (SVM) were proposed by Chuan, et al [2] as recognition algorithms for the American Sign Language using the Leap Motion. Tests were carried out using the 26 letters of the alphabet - results showed a recognition accuracy rate of 72.78% and 79.83% for the two methods, respectively. The authors mention some possible reasons for the low accuracy with both algorithms; compared with the BSL alphabet, the ASL is signed using only one hand – as a result some letter representations are very close to one another which led to misclassifications of the Leap Motion data. The use of BSL with these methods could show improved results as all of its gestures require two hands to perform and are therefore more varied.

#### 2.3.3 Hidden Markov Models

Markov models, in particular Hidden Markov Models (HMM), are generally known for their use in pattern matching algorithms for speech recognition or typing prediction. A Markov model is a network of states with each state being connected to another with a specific weight or probability. In a Markov model based system the future state of the system is only dependant on its current state and the probability of the states it’s linked to. A HMM differs in that its state is partially obscured. An example of this could be found in a speech recognition system where we are able to observe a waveform of speech but the actual spoken words is hidden. This can be compared to a gesture recognition system where we are given the movement data of a gesture but the actual intended gesture is hidden.

The use of a HMM was proposed by Chen [1] to support 2D and 3D motion recognition, achieving recognition rates of 91.9% in user-dependant testing and 96.9% in user- independent testing.

#### 2.3.4 Artificial Neural Networks

The use of artificial neural networks (ANN) was previously proposed by Mohandes, et al [5], in particular a Multilayer Perceptron neural network (MLP) for use with the Arabic Sign Language (ArSL). The proposed system resulted in a classification accuracy of over 99%. An artificial neural network is a type of machine learning algorithm which bears similarity to the human brain in that it is composed of a series of simple processing units, neurons. These neurons are interconnected and each of these connections have a determined weight. The network learns from experience when provided with test data, calculating specific outputs for given inputs.

It’s noted that the testing produced some erroneous results, similar to those found in the KNN/SVM with ASL testing described earlier. In this case this was due to fingers being occluded by the palm of the hand or by other fingers during recognition, as opposed to gestures just being too similar to one another to discern between. The authors suggest the use of a second Leap Motion device positioned to the side of the user. Combined with the Leap Motion in front of the user this should theoretically resolve the observed issues, though further work has yet to be carried out.

#### 2.3.5 $P Point-Cloud Recognizer

The $P recognizer ($P), designed by Vattavu, et al [11], is another example of a gesture recognition algorithm. As described in the paper, the $P is “a 2-D gesture recognizer designed for rapid prototyping of gesture-based user interfaces”. The $P aims to overcome the complex task of matching user gestures by instead treating them as groups or “clouds” of points and evaluating each one in turn. Even the simplest of gestures could be created in many different ways depending on the properties of its strokes e.g. start and end points, order or time, and direction. The use of a point cloud helps remove any ambiguity from the gesture which simplifies comparison and recognition. According to the paper, the algorithm requires only 70 lines of code to function and delivered over 99% accuracy in user-dependant testing.

## 2.4 Similar Applications

#### 2.4.1 LeapTrainer

‘LeapTrainer.js’, created by O’Leary [7], is a browser based gesture and pose learning and recognition framework for the Leap Motion. Developed in JavaScript, LeapTrainer allows users to create and store gestures then replay them at will. The software also allows gesture data to be exported for use in other applications. LeapTrainer recognises a gesture using a choice of geometric template matching and artificial neural networks.

From initial testing the trainer seems to recognize gestures accurately. Unfortunately the software struggles to discern between more intricate gestures. This is likely due to a low level of accuracy in the comparison algorithm. This could be improved by comparing gestures more thoroughly, though it is unknown how severe an effect this would have on the application’s performance.

#### 2.4.2 UNI

A commercial application, UNI, is currently in development and is scheduled to be released in summer 2016 by MotionSavvy [6]. UNI bares similarities to the proposed application in that it utilises the Leap Motion in order to translate gestures into spoken text. An included ‘crowd sign’ library would allow users to add and share gestures with other users via a cloud based dictionary system.

The software is closed source and based on a subscription model of $20 / month with an initial up front cost. This severely limits further adaptation or extension by like-minded developers, instead users are solely reliant on MotionSavvy to support the application in the future. Unfortunately, with no demo or other proof of concept available, it’s unknown how well the UNI will achieve the goals described on the website.

**Chapter 3**

# Design

## 3.1 User Requirements

When considering the overall design of an application there are several sections which must be considered. The first of which is the clear definition of user requirements. In software development, a requirement is a “property that a system must contain or exhibit in order for it to satisfy a user”. Before any implementation occurs it’s crucial to ensure these requirements are clarified.

Requirements are grouped into two categories; functional and non-functional. The former describes the features of a system (what it does) whereas the latter describes how the system behaves (performance, reliability etc.). These requirements can be more easily identified via the creation of use case diagrams (shown in figure 3.1) and use case tables (figure 3.2). Diagrams aim to visualize the relationships between users of a system and possible use cases, as well as between use cases themselves. Use case tables specify the function of each use case, but don’t consider their implementation.



**Figure 3.1:** Use case diagram







**Figure 3.2:** Use case scenarios

From the use case analysis above the following functional and non-functional user requirements were established:

Functional

* R1.1: The system shall display a real time interpretation of the user’s hands during operation
* R1.2: A user shall be able to record and store their own data for a given gesture
* R1.3: The system shall recognize a gesture provided by a user
* R1.4: The system shall present user feedback, a normalized score, based on the similarity between a given gesture and stored gestures

Non-Functional

* R1.5: The system’s real time display of a user’s hands shall be updated with a latency of 5ms or less
* R1.6: The recognition of gestures shall take no longer than 200ms to complete
* R1.7: The system shall recognize gestures with an accuracy of at least 80%
* R1.8: The system shall implement a simplistic interface which is easy to use without relying on mouse and keyboard input
* R1.9 The system shall operate robustly (should not crash or otherwise close without user’s request)

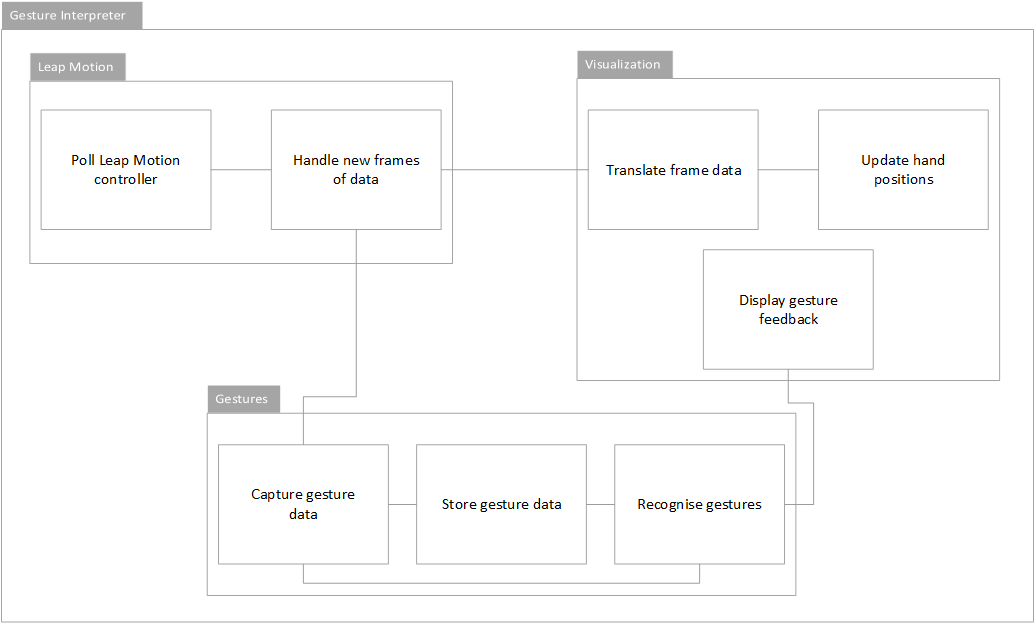
Regarding R1.6, the time value was chosen based on a statistical analysis of the average human reaction time from the human benchmark website [17]. The site currently holds records for 26,404,960 reaction time tests, with an average reaction time of 271ms. If we take into consideration the additional overhead of other factors e.g. Java VM, screen refresh rate then the time taken should be lower to compensate. Therefore it’s reasonable to assume that users won’t notice a delay in the recognition of gestures, provided that it takes less than 200ms to complete.

## 3.2 System Architecture

Considering the user requirements specified in the previous section, the application will consist of three main subsystems:

* **Leap Motion subsystem**: Updates application with new data as the physical Leap Motion device generates it
* **Gesture subsystem**: Handles storage and recognition of performed gestures
* **Visualization subsystem**: Displays a visual interpretation of the current Leap Motion data, providing the user with feedback of their actions

A high level overview of each of these subsystems is shown in figure 3.3. The diagram shows only which subsystems should be present within the application and how they should be linked together, rather than how they should be implemented. This diagram doesn’t show the actual classes but rather what each subsystem should accomplish as a whole.



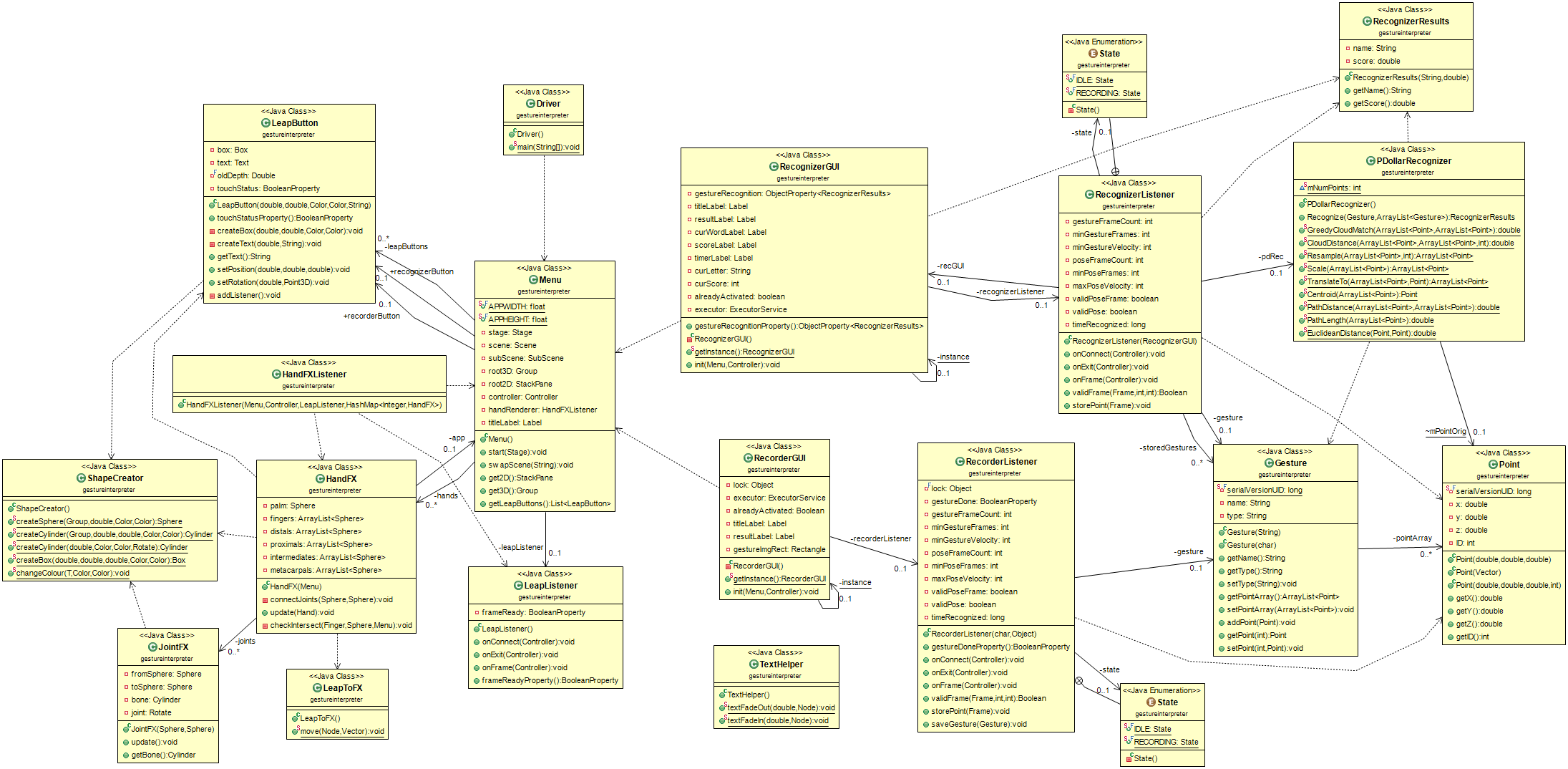
**Figure 3.3:** Overview of subsystems within the application

Figure 3.4 shows a more technical overview of the complete application. This helps to clarify the relationships between classes, as well as the data and features each class should contain. At this point only skeleton classes are used but it’s still beneficial to establish the layout of the program before any full code is written.

## 3.3 User Interface Design

As the application will contain a user interface, it’s beneficial to draft initial design plans to get a general idea of what the application’s appearance will be and how a user is able to interact with it. Looking back at the user requirements, R1.8 states that the system should be easy to use without relying on a keyboard and mouse – in other words, the user should be able to use the application through only the Leap Motion.

[USER INTERFACE DESIGN HERE]



**Figure 3.4:** Application UML class diagram

**Chapter 4**

# Implementation

## 4.1 Leap Motion Integration

The Leap Motion controller records tracking data in a series of frames. Each frame contains the positions of any detected hands or other pointable objects. Frames can be acquired by simply polling the device or via a call back method from an event listener which is assigned to the device. In the latter case, the Leap Motion will create a new thread for each new frame, and will pause execution until the current thread’s call back method (‘onFrame()’) has returned. This prevents an occurrence of thread-flooding, where threads are created at a faster rate than the device is able to process them.

Comparing the two integration choices, I found the second to be the most intuitive. The use of the event listener allowed me to manipulate or store the current frame’s data without having to consider the poll rate, which could in some cases be causing frames to be skipped, or duplicate frames to be requested, depending on the rate of frame polling compared with the rate of frame returns by the Leap Motion controller.

## 4.2 Graphical User Interface

The application’s GUI is handled by JavaFX, a library of graphics and media packages which is written as a Java API, meaning it can be referenced from any standard Java based program. The properties of JavaFX allow external devices such as the Leap Motion to be easily incorporated within an application. Vos [14] described this combination in a schema shown in figure 4.1.



**Figure 4.1:** Interaction between Leap Motion and JavaFX

As described in Oracle’s documentation of JavaFX [15], a system using JavaFX will run two or more of the following threads at any given time:

* **JavaFX application thread**: Primary thread used by JavaFX. Essentially any content which can be seen by the user must be managed in this thread.
* **Prism render thread**: Allows the application to perform concurrent processing. While one frame is being rendered, the next can be pre-processed to help off-load the work required.
* **Media thread**: Runs in the background and synchronizes the latest frames using the application thread.

With JavaFX, any property that modifies a window’s live content can only be changed through a ‘Platform.runLater()’ call – this ensures that these modifications only occur on the application thread. The combination of this system and the Leap Motion listener lead to an approach where each call of ‘onFrame()’ on a listener includes one or more Platform.runLater() calls in order to modify the content shown in the JavaFX window, either directly or indirectly. This ensures the application runs safely in regards to multithreading so as not to attempt to modify values in incorrect threads.

## 4.2 Hand Visualization

The inclusion of a visualization system to represent a user’s hands when they’re interacting with the application was a crucial initial design consideration. The user should be able to see their actions in real time, without having to swap between the Leap and the keyboard/mouse to e.g. navigate through menus – or having no visual feedback from the Leap entirely.

The visualization is setup through the use of a dedicated listener class (as described in section 4.1). When each new frame is received by the Leap controller, this listener’s onFrame() method checks the content of the frame to find the number of hands (if any) within it. If there is at least one hand visible in the frame, the listener indirectly fires an event through the ‘HandFXListener’ class that one or more hands should be updated in the JavaFX window.

The HandFXListener acts as an interface between the Leap Motion and the displayed content in the application window. The class contains a listener linked to an ObservableValue contained in the Leap Motion listener class. An ObservableValue is an object unique to JavaFX. Its function is to wrap a given primitive value and then allow this value to be observed for changes. This ability is particularly useful in this case, where an ObservableValue’s wrapped value is modified within the Leap Motion listener class if there are hands present within the frame. In turn, this calls updates on ‘HandFX’ objects within the HandFXListener class.

A HandFX is simply a representation of a user’s hand, generated by taking the raw data from the Leap Motion and converting it into co-ordinates. These co-ordinates are subsequently used to update collections of sphere and cylinder shapes, representing a hand’s joints and bones respectively. This solution provides a 1 to 1 mapping of the Leap Motion data to the application, meaning the user can be certain that what they see on the screen is a true representation of what the Leap Motion sees at any given time.

## 4.4 Gesture Recognition

The application builds on the work of Vattavu, et al. [11], using an adapted version of the $P recognizer (discussed briefly in section 2.3.5). The recognizer was originally written in Javascript and C# for use with handwriting recognition. The code used has been converted to the Java syntax and modified to support points in three dimensions in order to function correctly with the Leap Motion.

At its highest level, the $P is an “instance-based nearest neighbour classifier with a Euclidean scoring function”. Breaking this down, the $P selects the most appropriate category for an object, given a selection of objects, by calculating the Euclidean difference in positions between all available objects in order to find the closest and hence the most likely match. The $P is instance-based, meaning it compares a given unknown object against a set of known objects which are stored in memory.

In machine learning, objects are grouped into specific ‘categories’ based on a number of their ‘features’. Features refer to the properties of the object which make it unique, compared with other objects. In the context of the application, the features shown in table 4.2 are used to distinguish gestures from one another.

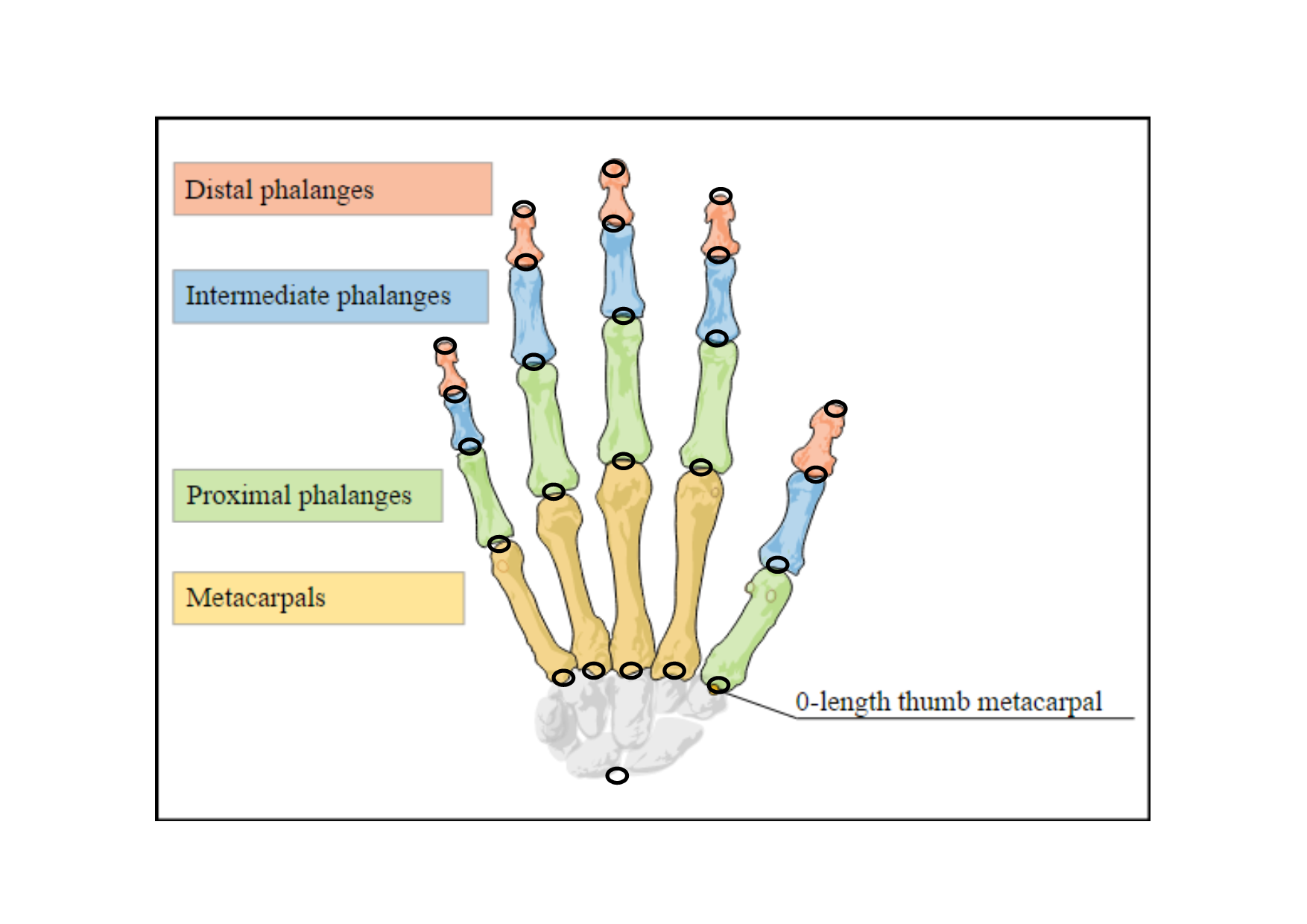
**Table 4.2:** Hand features used for gesture recognition

|  |  |  |
| --- | --- | --- |
|  | **Palm** | **Fingers** |
| **Features used** | Stabilized palm position | Finger direction |
| Palm normal | Metacarpal start position |
| Palm direction | Metacarpal end position |
|  | Proximal end position |
|  | Intermediate end position |
|  | Distal end position |

The meaning of each of these features is as follows:

* Stabilized palm position: The distance between the centre point of a hand’s palm and the Leap Motion controller origin, in millimetres. Smoothing and stabilization is applied to this value.
* Palm normal: A vector pointing in the same direction as the palm’s normal (orthogonal to the palm). For example, if the hand is held flat the normal would point downwards.
* Palm direction: A vector pointing from the palm position towards the fingers.
* Finger direction: The direction in which the finger is pointing.
* Finger start position: The base of the bone, closest to the wrist.
* Finger end position: The end of the bone, closest to the fingertip.

Each position feature is highlighted in figure 4.3, a hand bone structure image from the LeapMotion API page [16].



**Figure 4.3:** Bone model with highlighted position features

Gestures are recognized by the application implicitly based on a certain criteria. I had considered to fire recognition events explicitly via input from the user (e.g. pressing the spacebar to signal the start and stop points of a gesture) but felt that this would become cumbersome to operate, more so in cases where the keyboard might be difficult to access or not available entirely.

This criteria is the velocity of each hand within the frame. When each new frame of image data is received by the Leap Motion listener, the velocity of all detected hands is checked – this is found by checking the velocity of the palm of the hand, as well as the maximum velocity of each of the fingers on the hand. If this velocity is above a threshold (300 millimetres per second was chosen as the default value) then the listener will recognize that a gesture is being performed. If this threshold is not met, the listener will instead recognize that a ‘pose’ is being performed.

Although the BSL is usually referred to as a set of hand gestures, this isn’t necessarily the case in practice. Figure 4.4 shows an example of the alphabet interpreted in BSL where this can be seen more clearly. From the 26 letters of the English alphabet, only two (‘H’ and ‘J’) involve any hand motion. The majority of letters involve the hands remaining stationary in a specific pose, hence the requirement of pose and gesture categories. If a recognition includes any degree of hand motion then it’s a waste of time to compare against any stored recognitions which don’t, and vice versa.



**Figure 4.4:** BSL interpretation of English alphabet

If a pose is detected, the listener will wait for 50 frames. If the hand’s velocity remains below the motion velocity threshold for the entire 50 frames, a single frame will be captured to represent this pose. This frame is stored as a Gesture object in-memory, containing each of the features described in Table 4.2.

The process is essentially identical when considering a gesture which contains motion, such as for the ‘H’ or ‘J’ letters – for each frame that the hand’s velocity is above the threshold, a frame is recorded. If at least 10 frames are recorded in succession the gesture is considered to be valid. In the case of motion gestures, recognition will trigger once the hand’s velocity eventually falls below the threshold. In theory this allows for gestures which continue for an indefinite length of time, though in the case of the BSL they typically last a second or two at most.

**Chapter 5**

# The System in Operation

## 5.1 Initial Application State

Upon starting the application, the window displays this content. This screen acts as a main menu of the application, allowing access to the recognition and calibration screens. The user is able to access either of these screens by making a tap motion with their hand in the Leap Motion controller’s field of view, as if they were actually pressing the button. As discussed in chapter 3, I had aimed to design the user interface to be over-simplistic in order to promote ease of use with the Leap Motion.



## 5.2 Hand Enters Field Of View

When a hand enters the Leap Motion’s field of view, a skeletal model is shown in the application window. This model mimics the motion of the user’s hand in a 1 to 1 fashion. A side effect of this is that any blemishes in the Leap Motion’s accuracy are also represented in the application. For example, if a hand is unable to be correctly identified by the Leap Motion it will appear distorted in the application. This is beneficial as it highlights cases where the Leap Motion is at fault, rather than the user.



## 5.3 Recognition

When the user touches the left-most ‘recognition’ button, the following screen will be displayed. The user is asked to perform a given gesture from the set of alphabet letters. When a gesture is performed, the closest match and associated accuracy score is displayed to the user.

For each successful gesture match, the score value shown in the top-right is increased by 10. The goal is to correctly match the most amount of gestures within the time limit provided. As shown in the screenshot below, the user was asked to perform a ‘C’ gesture and upon recognition the application displays a similarity rating of 66%, by calculating the distance to all of the stored gestures in memory.

The objective of this screen is to provide an example as to how gesture recognition could be integrated into an application. There are of course a variety of uses which could be expanded upon but this provides a useful starting point and proof of concept.



## 5.4 Final Score

When the timer expires, the user is shown their final score. This screen remains for 5 seconds before returning the user to the main menu screen.



## 5.5 Calibration

When the user touches the right-most ‘calibration’ button, the following screen is displayed. Calibration is a means of improving the application’s recognition by a set of gestures which are categorized from A through Z. For each alphabet letter, an image is displayed in the centre-top of the screen, displaying its recognized BSL equivalent. Users are able to perform their own gestures for each letter if desired, the image only serves as a suggestion.

The user is given a three second countdown timer before recording begins, at which point they are able to store a gesture. Upon successful storage of a gesture, the application will move onto the next letter and the suggestion image will update as necessary. When the last letter (Z) has been recorded, the application returns to the main menu.



**Chapter 6**

# Testing and Evaluation

## 6.1 Testing Procedure

Testing was conducted as a multi stage process. ..

## 6.2 Aims / Requirements Analysis

* Record gestures performed by a user
* Recognise the gestures representing BSL alphabet characters and distinguish between them
* Given a gesture input, display a matching gesture with a similarity score

## 6.3 Recognition Analysis

Achieving a high recognition accuracy is the primary and most important goal of the application. If the system is unable to distinguish between gestures then it’s not fit for purpose no matter how well it performs in other areas.

Testing was carried out with a data set of 10 of each of the 26 BSL alphabet letters, for a total of 260 samples. From this, 1 set of letters was used as the test data, with the remaining sets used as training data. The amount of training data used was varied from 1 set to 9 sets in order to check how the accuracy scales with the number of training sets used. The number of points used was 32 for every data set, unless otherwise stated. The time taken to generate each recognition was recorded by taking the time difference between calling the recognition method and receiving a result. This time value was calculated via Java’s ‘nanoTime()’ method, then converted to milliseconds and rounded to 2 decimal places. It can be assumed that this time value is therefore accurate to the nearest nanosecond.

Figure 6.1 shows the performance of the application with a varying number of training sets. Figure 6.2 shows the average time taken to complete all 26 recognitions of the test set, with a varying number of training sets. Figures 6.3 and 6.4 show the performance and time taken (respectively) with a varying number of points, with both using 9 training sets.

**Figure 6.1:** Recognition accuracy with a varying number of training sets.

**Figure 6.2:** Average time taken to recognise a full data set with a varying number of training sets.

**Figure 6.3:** Recognition performance with a varying number of points and 9 training data sets.

**Figure 6.4:** Average time taken to recognise a full data set with a varying number of points and 9 training data sets.

Firstly, looking at figure 6.1 – there’s a clear correlation between the system’s recognition accuracy and the number of training sets used. As the number of training sets used increases, the range of acceptable values for each feature of a gesture widens. This means a given gesture has more margin for error when comparing to stored gestures. In comparison, with a single training set there is only a single accepted value for each feature. As a result, the testing data set has to match the training data set much more closely in order to generate a match. This is particularly problematic when considering 3D gestures, where it’s unlikely that a given gesture will be performed in exactly the same manner every time. Section 6.4 highlights the concerns regarding the recognition of features themselves by the Leap Motion controller.

Figure 6.2 shows a linear increase in time taken (with the time taken with 3 sets being a slight outlier) as the number of training data sets increases. It’s important to note that even with 9 training sets used, the recognition of all BSL letters is still completed in only 14.57ms. One non-functional requirement of the application, R1.6, specified that recognition should take less than 200ms, in this regard the application is therefore successful.

I had chosen to use 32 points as a default value prior to testing but considered the effect of other point values. Figures 6.3 and 6.4 confirm that to be the best choice, however. There appears to be an exponential increase in time taken as the number of points used is increased. This is most evident with the difference between 16 points (2.9ms) and 32 points (14.59ms). If speed of recognition is more of a concern than absolute accuracy, the use of 16 points could be considered as it still resulted in a 96% accuracy rating, meaning it misrepresented only a single alphabet letter. The use of more than 32 points may only be relevant for data which contains more feature values. Splitting features into too many points can have a negative effect, as seen with a 92% recognition rate when using 64 points – not to mention that the time taken also greatly increases.

## 6.4 Leap Motion Performance

When actually using the application, the results obtained in figure 6.1 may seem deceptive. Unfortunately the recognition capabilities of the Leap Motion itself certainly leave a lot to be desired.

The Leap Motion is a brilliant device but its software still needs work. The recent v2 firmware upgrade helped alleviate some issues through the inclusion of unique bone tracking, but many more still remain. The Leap Motion recognizes hands based on a best guess system so can very easily misrecognize hand motions – or fail to recognize them entirely. This is evident in the case of this application due to the intricate poses required for some BSL letters.

Figure 6.5 shows three rows of gestures, with the gestures in each row bearing many similarities. In practice a new user is unlikely to generate correct matches with these gestures unless they make a deliberate attempt to exaggerate or modify the appearance of each. Looking at the first row, ‘I’ and ‘O’, the features available from the Leap Motion API make it impossible to reliably distinguish between them. If we consider the differences between them, ‘I’ requires the user to raise the middle finger on their left hand against the index finger on their right hand. ‘O’ is nearly the same pose, the only difference being the left hand’s ring finger is raised instead.

It’s a similar scenario when looking at the second row of figure 6.5. When we consider the features that are recorded to represent each gesture, the majority are the same for both gestures. Looking at ‘F’ and then ‘X’, the middle fingers on each hand are different but the palm and other finger positions are very similar (clasped in the palm). A test gesture representing ‘F’ or ‘X’ is therefore close in distance to both ‘F’ and ‘X’.

The third row is related more to the lack of inference capability in the Leap Motion. When one hand is above another, the Leap Motion is essentially blind to the motion of the upper hand due to occlusion by the back of the lower hand. This makes it difficult to reliably recognize ‘L’, ‘M’, ‘N’, and ‘V’ without generating false positive results. I would usually have to make these gestures several times and adjust finger positions slightly each time before the correct match is given.







**Figure 6.5:** Frequently mismatched gestures

A second problem with the Leap Motion relates to its misrepresentation of hand position and motion. Figure 6.6 shows to examples of letters which are difficult to recognize due to this. As mentioned earlier, the Leap Motion functions using a best guess system. In some cases this system will fail to accurately represent hand motions. For lack of a better term, this will cause hands to become glitched and distorted as the Leap Motion is unable to process them. This usually occurs when hands are pressed close together, fingers are interleaved, or hands are placed on top of one another.

The ‘G’ gesture in particular is difficult for the Leap Motion to recognise due to the complete occlusion of the right hand when it is placed above the left hand. It usually requires several attempts to perform the gesture. Rotating the hands backwards / forwards aids in recognition by allowing more of the upper hand to be seen by the Leap Motion sensors - but then it is not the true representation according to the BSL.

The ‘W’ gesture sees the fingers on both hands being interleaved / locked together. This makes it difficult for the Leap Motion to distinguish which fingers belong to which hand and will frequently result in the hands becoming undetected until they are separated again.



**Figure 6.6:** Frequently misrepresented gestures

I feel the performance of the application in general could be problematic for some users. The majority of testing was carried out on a high-spec desktop system, with further testing on a weaker MacBook Air notebook.

The majority of testing was carried out on a

## 6.5 User Requirements Evaluation

## 6.6 User Study

Although the application has a deliberately simple user interface, there are other aspects of the system on which user feedback is helpful. In order to gather feedback on the application’s interface, and the application in general, the questionnaire shown in figure 6.7 was sent to 5 participants. Each participant was asked to try the calibration and recognition features of the application (with no guidance from myself). Subsequently they were asked to answer the questionnaire and prompted for any additional feedback if possible.

**Figure 6.7:** User feedback questionnaire

**Chapter 7**

# Conclusions

## 7.1 Aims Analysis

Accuracy?

Limited to gestures not requiring body or face?

Must be stationary to use - possible combination with oculus VR or other head mounted display?

## 7.2 Limitations

## 7.3 Future Work

There are a number of areas which could be explored further in order to enhance both the application in general and the accuracy of its recognition.

* **Data inference:** In its current state,
* **Alternate recognition algorithm:**
* **Multiple Leap Motion devices:**

## 7.4 Project Evaluation

### References

[1] Chen, M. (2013). Universal motion-based control and motion recognition. On line publication, Georgia Institute of Technology, <http://www.dtic.mil/dtic/tr/fulltext/u2/a344219.pdf>. Last accessed 1 November 2015.

[2] Chuan, C. H., Regina, E., & Guardino, C. (2014, December). American Sign Language Recognition Using Leap Motion Sensor. In Machine Learning and Applications (ICMLA), 2014 13th International Conference on (pp. 541-544). IEEE, 2014.

[3] Etherington, D. (2013, April). Leap Motion Controller Ship Date Delayed Until July 22, Due To A Need For A Larger, Longer Beta Test. Website, <http://techcrunch.com/2013/04/25/leap-motion-controller-ship-date-delayed-until-july-22-due-to-a-need-for-a-larger-longer-beta-test/>. Last accessed 4 December 2015.

[4] LeapMotion. (2014) Leap Motion V2 Tracking Now in Public Developer Beta. Website, <http://blog.leapmotion.com/leap-motion-v2-tracking-now-in-public-developer-beta/>. Last accessed 16 November 2015.

[5] Mohandes, M., Aliyu, S., & Deriche, M. (2014, June). Arabic sign language recognition using the leap motion controller. In Industrial Electronics (ISIE), 2014 IEEE 23rd International Symposium on (pp. 960-965). IEEE. Chicago, 2014.

[6] MotionSavvy. (2015) UNI. Website, <http://www.motionsavvy.com/>. Last accessed 9 October 2015.

[7] O’Leary R. (2013) LeapTrainer.js. GitHub repository, <https://github.com/roboleary/LeapTrainer.js>. Last accessed 9 October 2015.

[8] Potter, L. E., Araullo, J., & Carter, L. (2013) The leap motion controller: a view on sign language. In Proceedings of the 25th Australian Computer-Human Interaction Conference: Augmentation, Application, Innovation, Collaboration (pp. 175-178). ACM, 2013.

[9] Leap Motion. (2016) Platform Integrations & Libraries. Website, <https://developer.leapmotion.com/integrations>. Last accessed 8 February 2016.

[10] Signature. (2015) British Sign Language. Website, <http://www.signature.org.uk/british-sign-language>. Last accessed 1 November 2015.

[11] Vatavu R.D., Anthony L., & Wobbrock J. O. (2012) Gestures as Point Clouds: A $P Recognizer for User Interface Prototypes. On line publication, University of Washington, <http://faculty.washington.edu/wobbrock/pubs/icmi-12.pdf>. Last accessed 13 October 2015.

[12] Vikram, S., Li, L., & Russell, S. (2013) Writing and sketching in the air, recognizing and controlling on the fly. In CHI'13 Extended Abstracts on Human Factors in Computing Systems (pp. 1179-1184). ACM, 2013.

[13] Weichert, F., Bachmann, D., Rudak, B., & Fisseler, D. (2013) Analysis of the Accuracy and Robustness of the Leap Motion Controller. On line publication, PubMed Central, <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3690061/pdf/sensors-13-06380.pdf>. Last accessed 16 October 2015.

[14] Vos, J. (2014) Leap Motion and JavaFX. Website, <http://www.oracle.com/technetwork/articles/java/rich-client-leapmotion-2227139.html>.

Last accessed 2 February 2016.

[15] Oracle. (2014) JavaFX: Getting Started with JavaFX. Website, <https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-architecture.htm>.

Last accessed 2 February 2016.

[16] LeapMotion. (2015) Introducing The Skeletal Tracking Model. Website, <https://developer.leapmotion.com/documentation/java/devguide/Intro_Skeleton_API.html>.

Last accessed 5 February 2016.

[17] Human Benchmark. (2016) Reaction Time Statistics. Website, <http://www.humanbenchmark.com/tests/reactiontime/statistics>

Last accessed 9 February 2016.