ABSTRACT

Force Touch Gesture Based Interaction for Virtual Keyboards

Kuanysh Zhunussov, M.S.

Supervisor: G. Michael Poor, Ph.D.

With extreme popularity of touch screen mobile devices, the demand for effective one-handed text-entry on virtual keyboards is continually growing. To increase text-entry speed and decrease error rate, this thesis proposes force touch gesture based suggestion selection for virtual keyboards and analyzes the performance compared to standard keying and swyping keyboards. The prototype, called OctoType, was built for iOS smartphones to take advantage of the 3D-touch technology by implementing an interaction based on OctoPocus, a dynamic gesture guide. Two users studies were conducted to evaluate performance of OctoType. The results showed that OctoType outperforms standard keying keyboards by 13.1% in terms of text-entry speed. Moreover, OctoType registered force touch gestures with an accuracy of greater than 97%. Given these advantages of force touch gesture based interaction, this thesis also introduces the framework, called Gesturizer, that provides a conflict-free gesture interaction with arbitrary shapes to iOS applications running on devices with 3D-touch.

Force Touch Gesture Based Interaction for Virtual Keyboards

by

Kuanysh Zhunussov, BEngTech

A Thesis

Approved by the Department of Computer Science

Gregory D. Speegle, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

Approved by the Thesis Committee

G. Michael Poor, Ph.D., Chairperson

Greg Hamerly, Ph.D.

Dennis A. Johnston, Ph.D.

Accepted by the Graduate School
May 2018

J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would first like to express my appreciation to my thesis advisor, Dr. G. Michael Poor. His valuable discussions and suggestions steered me in the right the direction whenever he thought I needed it. I am very lucky to be pointed to the topic of gesture based interfaces that I really enjoyed working on.

I would also like to acknowledge thesis committee members, Dr. Greg Hamerly for his expert guidance in evaluation of gesture recognition algorithms and Dr. Dennis A. Johnston for his advises that helped me in analysis of the results. Also, I would like to thank Dr. Bill Booth for his help in recruiting of participants. Special thanks to Noble Applications team for provided resources during an internship and their guidance in acquiring necessary skills to build OctoType.

Finally, I must express my very profound gratitude to my parents and to my girlfriend for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

CHAPTER ONE

Introduction

With the expansion of computing technologies beyond the confines of the desktop, the demand for effective text input on handheld devices, such as the iPhone, the iPad, and Android devices, has been increasing over the last two decades. Among many interfaces studied by Human-Computer Interaction researchers to improve text entry performance (Leung and Aarabi 2014, Cheng, Liang, Wu, and Chen 2013), the standard keying keyboard remains the most commonly used, together with "swyping" keyboards that use gesture-based interaction. However, the performance of these virtual keyboards varies depending on the size and the orientation of a device (Nguyen and Bartha 2012).

Despite using the traditional QWERTY layout, hardware keyboards outperform virtual ones. Lack of strong tactile feedback in addition to the restricted size of mobile devices are key limitations of virtual keyboards (Henze, Rukzio, and Boll 2012). Moreover, smartphones are widely used in one-handed way (Hoober 2013), bringing additional challenge to researchers. To address these limitations, we further investigated existing gesture based user interfaces to improve efficiency of text-entry. We introduce OctoType, a virtual keyboard with force touch gesture interaction. OctoType was inspired from OctoPocus (Bau and Mackay 2008), a dynamic guide that combines on-screen feedforward and feedback to help users learn, execute, and remember gesture sets. Compared with other gesture guide mechanisms (Callahan, Hopkins, Weiser, and Shneiderman 1988, Kurtenbach 1993), OctoPocus does not take any extra screen space other than the gestures themselves. Since virtual keyboards in iOS and Android devices implement suggestion systems, OctoType uses suggestion selection as an example of the capabilities of force touch gesture interaction.

To address limitations of one-handed use of mobile devices, this thesis investigates the viability of one-handed force touch gesture interaction. In particular, we are proposing to apply the interaction for suggestion selection on virtual keyboards to improve one-handed text-entry speed. Due to high accuracy and low execution time of force touch gestures observed in our experimental results, we are also proposing to scale this kind of interaction for different applications by implementing a framework for conflict-free gesture interfaces.

Two experiments were conducted to estimate performance of force touch gesture interaction for virtual keyboards. In the first experiment, for group of people who type on standard keying keyboard faster than 22 words per minute (WPM), OctoType was more efficient than other keyboards as it yielded higher WPM and a lower error rate. Results and feedback collected from participants motivated us to conduct a second experiment with a modification to OctoType where force touch was used only for activation of gesture drawing mode, while gestures are drawn regardless of pressure level. As a result, the accuracy of force touch gesture drawing reached nearly to 99.9% for two out of three gestures. Moreover, execution time of single gesture execution was less than one second for all types of gestures used in the prototype.

In the next chapter we will review existing studies on gesture based user interfaces and virtual keyboards. In Chapter Three we will discuss experiments that we conducted to evaluate several gesture recognition algorithms and compare their performance. In Chapter Four we demonstrate the design of our prototype. Chapter Five discusses methods and procedures of our first experiment as well as its results. In Chapter Six we will show modifications made to OctoType and how results were improved. In Chapter Seven we introduce the Gesturizer, a framework that provides a conflict-free gesture interaction to iOS applications running on devices with 3D-touch. Then we briefly discuss its implementation and capabilities. Finally, in Chapter Eight there will be a discussion and conclusion of the thesis.

CHAPTER TWO

Related Works

2.1 Gesture Based User Interfaces

Gesture based user interfaces provide efficient forms of interaction with objects of interest on the screen. Depending on tasks and context, gesture based user interfaces are more efficient and convenient to use than standard buttons and pull-down menus. One of the main advantages of gesture based interfaces is that a user is not forced to execute a gesture on a particular static area of the screen. In fact, gestures can be executed anywhere on the display regardless a cursor, finger or pen position.

Gestures can be arbitrary with different directions and shapes which helps a user to memorize and execute gestures. Accordiing to Poppinga et al., gestures can be grouped into 5 shapes (Poppinga, Sahami Shirazi, Henze, Heuten, and Boll 2014). For instance, gestures with *letter* shapes correspond to a meaning of a command such as "M" for mail, and a *geometric* shape "circle" represents opening "camera" app. Other gesture shapes (see Figure 2.1) include *word* gestures which are sequence of letters or *icon* gestures representing an object's figure such as "+" sign for opening calculator app.

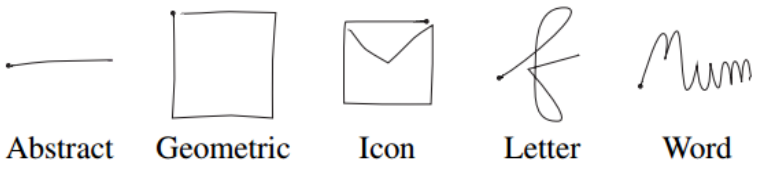

Figure 2.1. Examples of gestures for five shapes each gesture can be assigned to. Poppinga et al. empirically identified these shapes from different user-invented gestures. The beginning of gestures is represented with thick point, e.g. "line" gesture is started from left.

Depending on number of strokes, there are two types of gestures on touch screen devices: unistroke and multi-stroke. Unlike multi-stroke gestures which contain more than one stroke, single-stroke gestures can be executed only in two different ways, since there are only two possible starting points. On the other hand, execution of multi-stroke gestures can be extremely challenging from the perspective of recognition algorithms. In addition, multi-stroke gestures with more complex geometrical shape decreases consistency of gesture execution between-users (Anthony, Vatavu, and Wobbrock 2013). Therefore, unistroke gestures are commonly used in touchscreen devices.

There are many applications for gesture based interfaces, including web browsers (Moyle and Cockburn 2003), drawing applications (Igarashi, Kawachiya, Tanaka, and Matsuoka 1998) and text-document editing (Rodriguez, Sánchez, and Lladós 2007). However, most graphical user interfaces consist of standard buttons and pull-down menus, likely because gesture based interfaces require users to learn and execute gestures correctly. If users are to take advantage of gesture based interfaces in specific and/or common applications, guide systems for users must be provided.

2.1.1 Guides for Learning and Execution of Gestures

Existing systems designed to improve the usability and learnability of gestural interfaces provide two basic mechanisms: *feedforward* and *feedback* (Bau and Mackay 2008). Gesture guides with feedforward mechanisms visually provide information about a gesture's shape and a command that the gesture is associated with. For instance, help cards or pop-up "cheat sheets" illustrate the gestures and associated commands on the screen (Kurtenbach, Moran, and Buxton 1994). There are two main drawbacks of such systems. First, users must alternate attention repeatedly between current gesture state to the cheat sheet. Second, the number of gestures that can be used within an application is restricted, since displaying a full list of gestures and associated commands uses a wide amount of space on the screen.

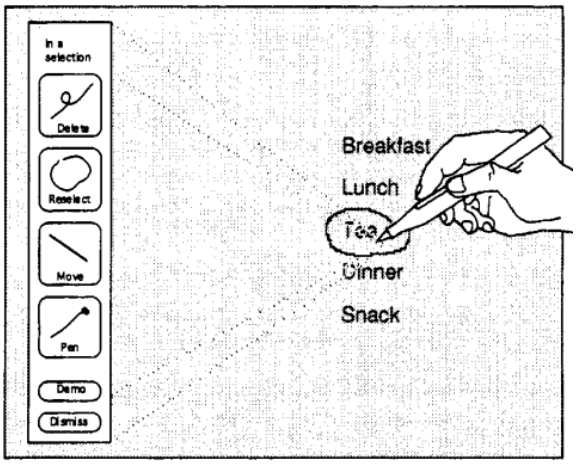

Figure 2.2. Kurtenbach's crib-sheet. When a user selects the word "tea", the crib-sheet pops up with set of possible gestures and associated commands.

In contrast, feedback mechanisms depict information after a user starts drawing a gesture and/or at the end of execution. This mechanism requires a gesture recognition algorithm. Feedback may be given by displaying the recognized gesture (see Figure 2.3) or providing incremental information as to the current state of the recognition algorithm (Mankoff, Hudson, and Abowd 2000). Another feature of feedback mechanisms is input correction or beautification. Guide systems such as Fluid

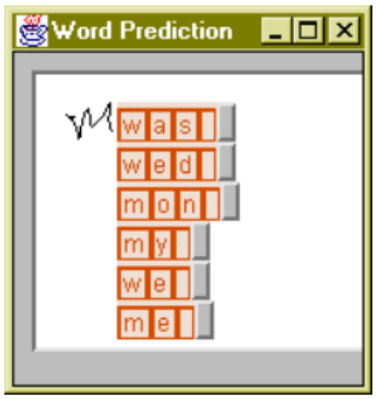

Figure 2.3. Mankoff's ambiguity resolution system shows how it interpreted an input and gives a list of possible words. In this example, the ambiguous shape was recognized as 'm' and 'w'.

Sketches (Arvo and Novins 2000) and Incremental Intention Extraction (Li, Zhang, Ao, and Dai 2005) correct user input using the perfect template of a gesture (see Figure 2.4).

Figure 2.4. Li et al's Incremental Intention Extraction *beautifies* user input to the original template of a gesture.

Considering the pros and cons of the two mechanisms, later studies showed best practices of dynamic gesture guides combining feedforward and feedback. Kurtenbach extended Pie menus (Callahan, Hopkins, Weiser, and Shneiderman 1988), which have better time efficiency compared to linear menus (Kurtenbach 1993). Compared to Pie menus, Kurtenbach's Marking menus (Figure 2.5) have learnability of gestures for novice users and faster speed of drawing for experts.

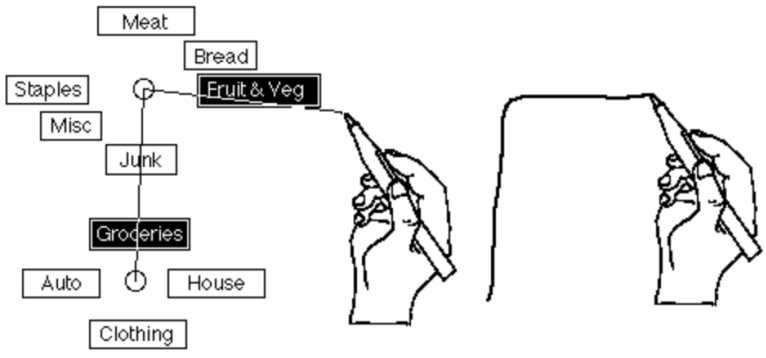

Figure 2.5. Once the user pressed the screen and waited (marked as circles), a set of possible selections appears. Selected items become highlighted, allowing the user to perform the next iteration in a sequence of actions.

Compared to other novel gesture guides developed during the last decade, OctoPocus significantly improved drawing speed and learning rate of gestures (Bau and Mackay 2008). Unlike Help menus, OctoPocus does not take any extra space other than displaying gesture options. Also, both feedforward and feedback are continuously updated as the gesture progresses (see Figure 2.6). After the user selects and begins to make a gesture, less likely gesture guide paths become thinner and disappear. OctoPocus therefore depicts ideal future path of each gesture as well as the path that is drawn by the user, decreasing error rate and increasing speed of gesture execution.

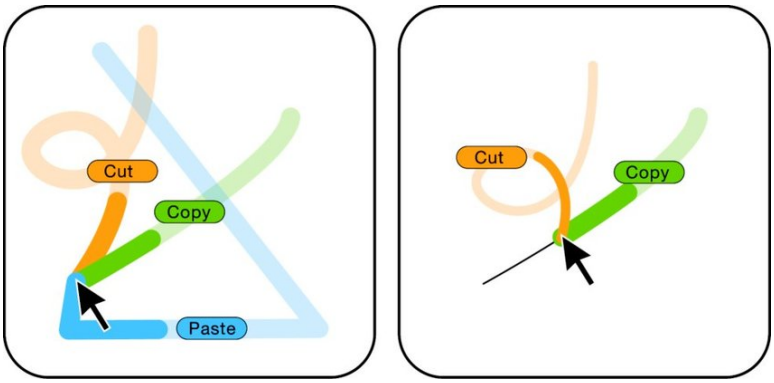

Figure 2.6. Example of OctoPocus with guide paths for three different gestures.

2.1.2 Gesture Recognition Algorithms

Several effective and efficient algorithms have been developed for unistroke gesture recognition using various approaches, including Hidden Markov Models (Sezgin and Davis 2005, Anderson, Bailey, and Skubic 2004), neural networks (Pittman 1991) and ad-hoc heuristic recognizers (Wobbrock, Wilson, and Li 2007). However, drawbacks of these algorithms include dependence on large training dataset as well as difficulty of programming and debugging. Therefore, these classifiers are not convenient for prorotyping user interfaces.

One of the simplest and accurate algorithms that was designed for prototyping gesture based user interfaces is the $1-recognizer. Basically, the $1-recognizer is an instance-based nearest-neighbor classifier with a Euclidean scoring function. In contrast to Rubine's classifier (Rubine 1991) that extracts geometrical features, the $1-recognizer uses every point of a gesture as a feature. The algorithm obtains 97% accuracy with 1 training template per gesture and 99% with 3+ templates per gesture (Wobbrock, Wilson, and Li 2007).

To evaluate speed of execution of gestures and confusion error between pairs of different gesture types, Vatavu et al. studied the performance of different gesture types using the $1-recognizer (Vatavu, Anthony, and Wobbrock 2014). The results of this study are extremely useful for constructing gesture sets when designing a prototype with a gesture based interface. For instance, a simple 'circle' gesture that looks like an English letter 'O', has high drawing speed over the whole gesture path, while the more complex 'car' gesture has the lowest drawing speed among many other gesture types. According to *confusion matrix* of gesture types, confusion error between gesture 'V' and 'caret' gesture (looks like inverted 'V') is low, i.e. it is safe to include these gesture types when designing a prototype with gesture based interface. In contrast, there is a high confusion between 'circle' and 'rectangle' gestures. Thus, it is important to choose distinct set of gestures to reduce error rate for recognition algorithms.

2.2 One-Handed Text-Entry on Touch Screen

Smartphones are used by billions of people around the world and text remains indispensable channel of communication. This is why academic researchers in HCI and commercial developers have been studying and inventing variety of text-entry methods for mobile devices. Although there are many novel solutions for text-entry without using keyboards, such as speech recognition, handwriting, and sign language

recognition (Kölsch and Turk 2002), virtual keyboards with standard QWERTY-layout are common in commercial mobile systems such as Apple's iOS and Google's Android.

One-handed interaction with touch screen mobile devices is dominantly used in different activities such as walking and standing (Karlson, Bederson, and Contreras-Vidal 2006), which brings additional challenge in implementing an efficient user interface for text-entry and other tasks. There are numerous research papers exploring alternative ways of controlling UI elements with one hand including but not limited to, using physical buttons on the sides (Wilson, Brewster, and Halvey 2013), force touch interaction (Heo and Lee 2011), bend interaction with deformable phones (Girouard, Lo, Riyadh, Daliri, Eady, and Pasquero 2015).

2.2.1 Keyboard Layouts

Standard QWERTY layout was designed for typewriters to prevent mechanical jamming by putting most commonly occurring consecutive letter pairs on different sides of the layout (Cooper 2012). This layout also was constructed to facilitate both right and left hands. To increase text-entry performance on touchscreen devices, researchers explored optimized keyboard layouts for one-handed input including OPTI (MacKenzie and Zhang 1999), Quasi-QWERTY (Bi, Smith, and Zhai 2010), ATOMIK (Zhai, Hunter, and Smith 2000) and others. However, QWERTY layout remains standard and familiar to users. Current commercial systems provides two types of touch screen keyboards using standard QWERTY layout: *keying* and *swyping*. While Apple provides standard keying keyboard by default, Google promotes swyping keyboard on the Android operating systems (see Figure 2.7). Essentially, swyping is gesture based interaction, where every word maps to an unique gesture. Compared to keying keyboards, gestures in swyping keyboards possibly involves muscle memory more efficiently.

Figure 2.7. Standard keying keyboards from Apple on the left and shape-writing keyboard from Google on the right. The gesture in the example on the right, starting from letter 's', corresponds to the word "swipe".

2.2.2 Suggestion Systems

To improve text-entry speed and decrease error rate, prediction systems, also called suggestion systems, are presented as an improvement to virtual keyboards. Augmentative and Alternative Communication research has investigated prediction systems to improve input speed for users with physical impairments, but has found that the benefits of suggestions are not always clear (Quinn and Zhai 2016). Current

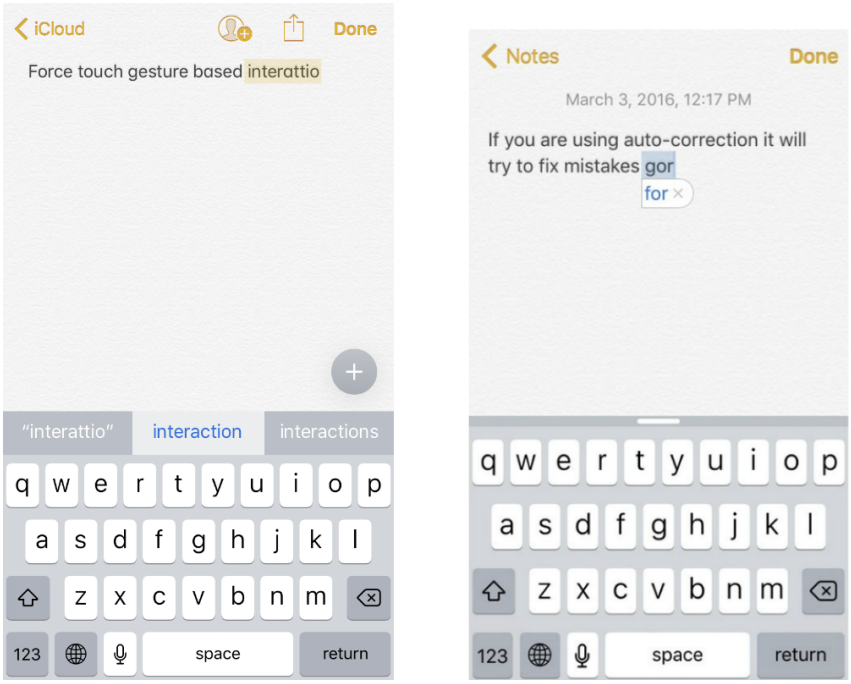

Figure 2.8. Two examples of auto-correction and suggestions implemented in iOS.

commercial practice uses suggestion interfaces by default. Google's Android platform (versions 5 'Lollipop' and higher) presents up to three suggestions on top of the keyboard. Conversely, Apple's iOS platform (versions 9 and higher) can display one suggestion below the text insertion cursor, or optionally, three in a bar above the keyboard (see Figure 2.8). The suggestion below the cursor is transient, and is accepted by tapping on the spacebar (tapping on the suggestion will reject it).

CHAPTER THREE

Gesture Recognition Algorithms

This chapter evaluates different approaches to gestures recognition to pick the right algorithm for implementation of the keyboard prototype. Criterion for choosing algorithms included accuracy and speed of recognition as well ease of programming and debugging. Thus, the goals of this chapter are to:

(1) evaluate different gesture classification approaches to achieve sufficient results

(2) adapt gesture classifiers in order to be able to recognize noisy gestures

Since the $1-recognizer is able to classify with 99% accuracy using 3 and more templates per gesture (Wobbrock, Wilson, and Li 2007), the target accuracy of gesture recognition for our keyboard prototype was extremely high. The second goal is important in terms of accessibility of gestures. In particular, accurate classification of noisy gestures potentially makes gesture based interfaces accessible for users with motor disabilities. However, the high priority was achievement of the first goal, because the prototype of our keyboard was not specifically targeting users with physical impairment.

In the rest of this chapter, we will discuss classifiers in the context of gesture recognition. In methods section, there will be discussions about design of experiments for evaluation of gesture recognition algorithms followed by analysis of results. We conclude with a discussion and future enhancements.

3.1 Methods

3.1.1 Data and Feature Extraction

In this study we used a dataset with 5,040 gesture templates (14 participants $\times$ 18 gestures $\times$ 20 executions) collected using a Wacom DTU-710 Interactive Display

(Vatavu 2011). It is worth mentioning that in both our experiments we used only 14 gesture types. Since there was significant increase in the error rate classification of other gestures. For example, all three algorithms misclassified 'a' and 'g' gestures, increasing error rate by 5-10%. Illustrations of gesture types from the dataset are shown in Figure 3.1.

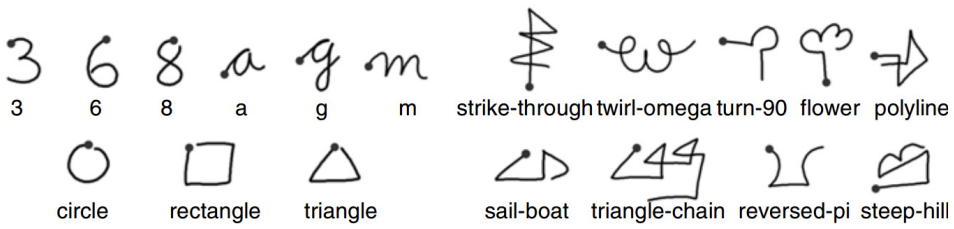

Figure 3.1. Different gesture types from Vatavu dataset.

Pen stroke gestures are represented as a sequence of coordinates and time of every point of a gesture in (x, y, t) form. The number of points per gesture template ranges 120-150. As part of normalization of data, we re-sampled them to gestures with 32 points so that distances between points were equally distributed. We also tried resampling to 16 and 64 points, but 32 was optimal in terms of time efficiency and accuracy of classification. From every resampled gesture template extracted 8 geometrical features were also used in Rubine's classifier:

(1) Cosine of initial angle with respect to the X axis:

$f_1 = cos\alpha = (x_2 - x_0)/d$, where $d = \sqrt{(x_2 - x_0)^2 + (y_2 - y_0)^2}$

(2) Length of the bounding box diagonal:

$f_2 = \sqrt{(x_{max} - x_{min})^2 + (y_{max} - y_{min})^2}$

(3) Angle between diagonal and x-axis:

$f_3 = arctan\frac{y_{max} - y_{min}}{x_{max} - x_{min}}$

(4) Distance between first and last point:

$f_4 = \sqrt{(x_{p-1} - x_0)^2 + (y_{p-1} - y_0)^2}$,

where p is the number of points in a template

(5) Cosine of angle between first and last point:

$f_5 = cos\beta = (x_{p-1} - x_0)/f_4$

(6) Total gesture length:

$f_6 = \sum_{i=0}^{p-2} \sqrt{\Delta x_i^2 + \Delta y_i^2}$, where $\Delta x_i = x_{i+1} - x_i$ and $\Delta y_i = y_{i+1} - y_i$

(7) Maximum speed (squared):

$f_7 = \max \frac{\Delta x_p^2 + \Delta y_p^2}{\Delta t_p^2}$, where $\Delta t_p = t_{p+1} - t_p$

(8) Path duration:

$f_8 = t_{p-1} - t_0$

We also calculated *intensity features*, which are intensities in four equally separated areas in the bounding box of a gesture, where center point O calculated as mean of maximum and minimum of x and y values of the gesture. For example, point intensity features for the gesture in Figure 3.2 are $f_9 = 0.20$, $f_{10} = 0.22$, $f_{11} = 0.31$, and $f_{12} = 0.27$. Note that the sum of four intensity features is always equals to one.

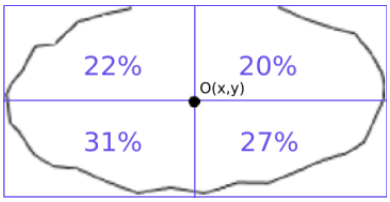

Figure 3.2. Points intensity features calculated from four equally separated rectangles.

As it was mentioned in the introduction, one of the goals is to classify noisy gestures. However, we didn't find any data set that was collected from participants with a physical impairment. Which is why we applied some simple calculations on the original gesture templates to generate noisy gestures. Firstly, we added random noises, in a range from -10 to 10, to x and y values of every point in a gesture.

Then gestures were squeezed twice, either by x-axis or y-axis (axis was also picked randomly), by halving values of x or y of a gesture template. Finally, we removed the first N and last M points from a gesture, where N and M are random integers from 0 to 10 inclusively. Visualization of such transformations can be seen in Figure 3.3.

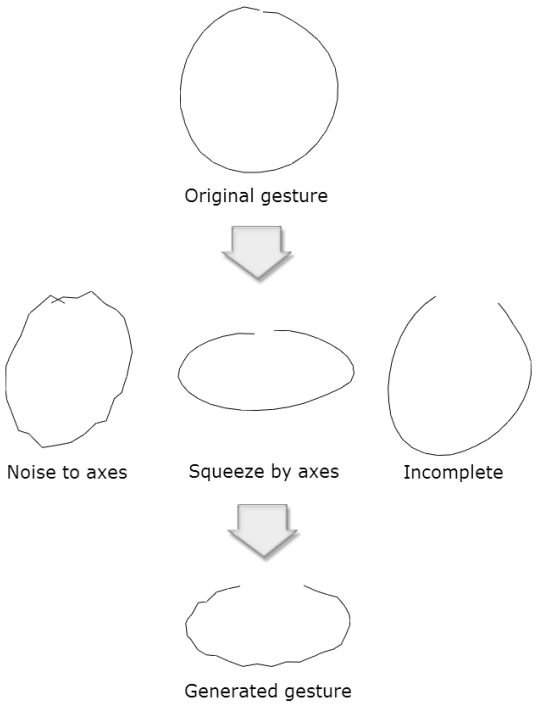

Figure 3.3. Generation of noisy gestures.

3.1.2 SVM and K-NN

The $1-recognizer was compared to other classification algorithms: Support Vector Machines (SVM) and K-Nearest Neighbor (K-NN) algorithm. SVM is used due to good generalization and stability for noisy data after adapting regularization parameters. On the other hand, K-NN is very sensitive for noisy data (Ougiaroglou and Evangelidis 2015), providing us opportunity to observe dependency of classification accuracy on noise of gesture templates.

Implementation of the $1-recognizer written in Java[1] was used in our experiments, while for SVM and K-NN we used *Weka* Java-library (Eibe Frank and Witten 2016), collection of machine learning algorithms for data mining tasks developed at University of Waikato, New Zealand. Using the Weka library, configuration of different classifiers, including SVM and K-NN, can be easily managed.

Sequential minimal optimization (SMO) was chosen to solve the quadratic programming problem that arises during the training of SVM (Zeng, Yu, Xu, Xie, and Gao 2008). The pairwise coupling method was used to multiclass classification (Hastie and Tibshirani 1998). This method creates $\frac{n(n-1)}{2}$ classifiers, where n is a number of gesture types, and chooses the most probable class from all pairwise results. In fact, for number of gesture types used in our study, pairwise coupling method performed in ~ 5.5 nanoseconds for classification of a single gesture template.

Both K-Nearest Neighbor and SVM were trained with features described in previous section. However, we did not study any feature selection algorithms nor correlation of them. Initially, we tested K-NN using our dataset to pick a value of k with a low error rate. The result showed that 5-NN is optimal for our dataset using the 12 features (see Figure 3.4). In the following sections, all results regarding K-NN are achieved by 5-NN.

3.1.3 Procedure

We conducted 2 experiments to evaluate our goals stated early in this chapter. In the first experiment we ran $1, 5-NN and SVM on both original and noisy data. To measure how accuracy of gesture classification depended on number of training templates per gesture, we ran each algorithm 2240 times with increasing size of training data from 1 to 224 per gesture type, i.e. 10 runs for every training template size.

Given results from the first experiment, for time efficiency and accuracy of algorithms we used optimal number of gesture templates for training, then validated

[1] http://depts.washington.edu/madlab/proj/dollar/

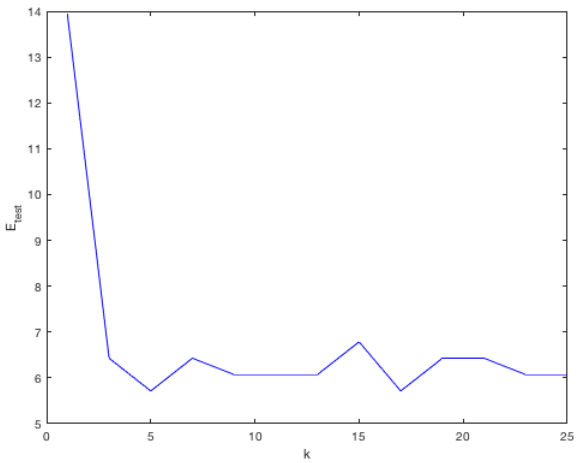

Figure 3.4. Impact of k in K-NN on error rate.

all three classifiers with increasing number of gesture types (on both original and noisy gestures).

3.2 Results

Using 1 to 25 training data per gesture type, $1 algorithm yielded a significantly lower error rate than SVM and K-NN (see Figure 3.5). Although 5-NN reached 8.9% error rate, with SVM we also achieved high accuracy, i.e. $\sim$ 3% error rate or $\sim$ 97% accuracy. For noisy data, SVM reached lower error rate than the $1-recognizer with 10.5% and 11.5% respectively.

In the second experiment, the $1-recognizer always performed with error rate less than 1% on original data (see Figure 3.6). SVM also showed sufficient results steadily increasing over number of gesture types from 0.7% to 1.95% error rate. SVM clearly outperforms other classifiers when running validation on noisy data.

3.3 Conclusion and Discussion

In this chapter, we evaluated 5-NN and SVM with features used in Rubine's classifier as well as features describing points intensity. Compared to the $1-recognizer, 5-NN and SVM showed higher error rates for original data. We achieved

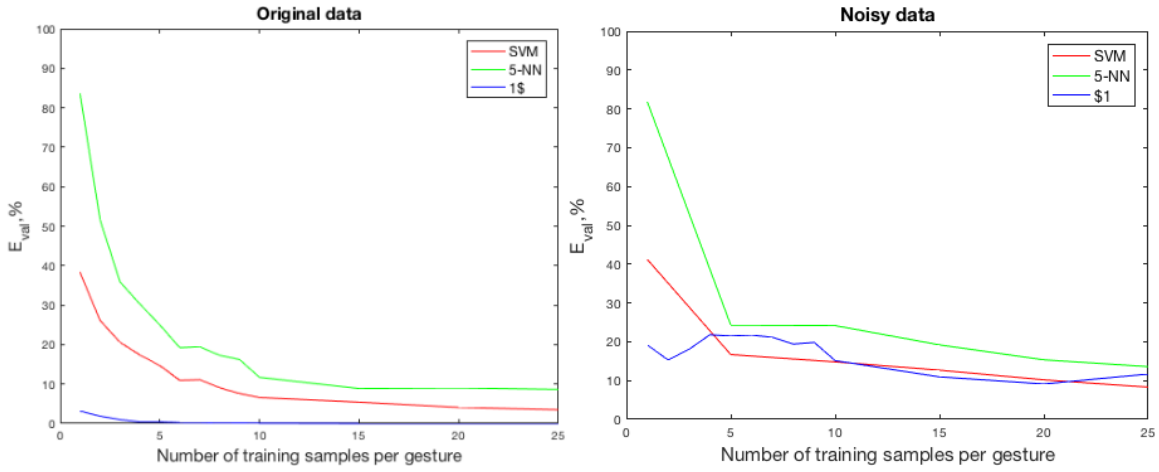

Figure 3.5. Error rate dependency on number of training samples.

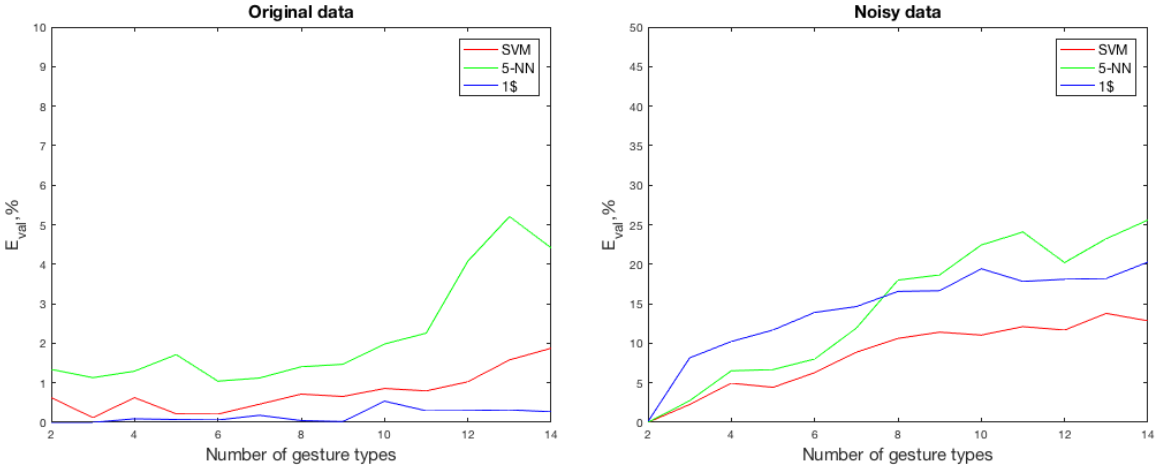

Figure 3.6. Error rate dependency on number of gesture types.

relatively accurate classification of noisy gestures using SVM, while the $1-recognizer had higher error rate than SVM as well as 5-NN. In fact, even 5-NN had lower error rate than the $1-recognizer when validating on noisy data for up to 7 gesture types. The main difference in settings of these three classifiers is that SVM and 5-NN used geometrical features, while the $1-recognizer uses every point of template as features. Therefore, using extracted geometrical features shows better generalization than using raw points of a gesture template as features.

Given high accuracy and robustness of classification as well as ease of implementation of the $1-recognizer, in designing of our keyboard we proceeded with this algorithm. Although the keyboard use fewer gesture types than the number of gestures used in this chapter, the results of accuracy for $1-recognizer shows that the number of gestures can be extended in the prototype.

On the other hand, if target audience for our keyboard would be changed in future for people with physical impairment, the use of another gesture recognition algorithm must be considered, possibly using geometrical features. In this case, we should think about smarter way of generation noisy gestures and/or collecting real data involving people with a physical impairment to validate different algorithms in a real-world context.

CHAPTER FOUR

Design of Keyboard Prototype

4.1 Prototype

The prototype of OctoType is supported on iPhones with 3D-touch[1] technology (e.g. iPhone 6s, 7, 7+, and 8) and has two modes of operation: Default Mode and Gesture Mode. In Default Mode, which is activated initially, OctoType performs as standard keying keyboards on iOS and Android devices with suggestions directly above the keyboard (see Figure 4.1). In the example below, three suggestions appear on the suggestion bar located on top of the keyboard according to a dictionary constructed from the phrases set specifically designed for evaluation of text-entry methods (MacKenzie and Soukoreff 2003). The details of the dictionary are discussed later in this section.

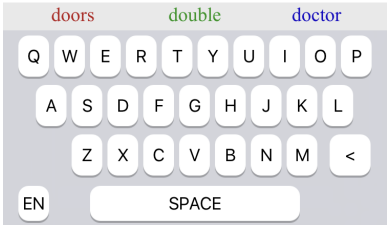

Figure 4.1. OctoType in Default Mode after typing 'do' prefix.

In contrast, Gesture Mode is a state of the keyboard when different gestures are displayed on the screen and can be executed to select a suggestion (see Figure 4.2). Gesture Mode can be activated by applying additional pressure to any area on the screen within the keyboard's scope. Apple's iOS development kit gives the ability to read screen pressure level from 0 to 1. Force touch threshold for OctoType is

[1]https://developer.apple.com/ios/3d-touch/

0.5. Thus, when a user touches the screen with pressure level ≥ 0.5, Gesture Mode becomes active. Once gestures become visible, the user can execute their prefered gesture with pressure level ≥ 0.5. If the pressure level goes lower than 0.5 while drawing a gesture, OctoType stops Gesture Mode and returns back to Default Mode.

There are three different static gestures with red, green, and blue colors that matches with first, second, and third word in the suggestion bar respectively. Different variations of *curve* and *angle* gestures were picked due to their simplicity and ease of drawing (Vatavu, Anthony, and Wobbrock 2014). These three gestures are highly distinguishable by the $1-recognizer, reducing possibility of errors. Also, speed of drawing for this gesture is high as well as their learnability. Once a user finishes to draw one of these gestures, OctoType associates a word matching to the gesture executed (see Figure 4.2, Figure 4.3 and Figure 4.4).

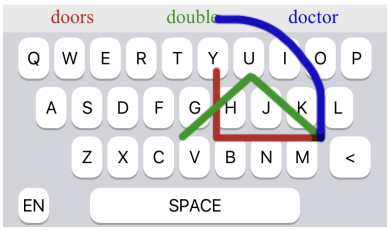

Figure 4.2. OctoType in Gesture Mode after typing 'do' prefix at the beginning.

The gesture interaction of the keyboard in Gesture Mode is based on OctoPocus. Once Gesture Mode is active and a user starts to execute a gesture, less likely gesture guide paths become thinner and disappear. Likeliness of a gesture based on a path that a user already drawn is calculated similarly to the algorithm of OctoPocus. Assuming T is the template or 'perfect' gesture for class C, the algorithm proceeds as follows for each template:

(1) Subtract the prefix of the length of user's input from the full template T, resulting in a sub-template $subT$.

(2) Concatenate the current user's input with sub-template $subT$. The resulting shape $perfT$ is the user's completed input together with a perfect drawing for a given class.

(3) Use the $1-recognizer to compute the distance between the resulting shape $perfT$ and the gesture class C associated with the template.

(4) Compute the difference between the computed current value and a given threshold. This gives user's room for error before reaching the distance threshold i.e. before the input no longer resembles an element of class C, from perspective of the recognizer.

In the examples given below, a user starts to draw the *blue* gesture, i.e. select the word 'doctor'. While user starts drawing the blue gesture, the probability and thickness of red and green gestures decrease. As a result, the red and green gestures disappear as well as the first two suggestions.

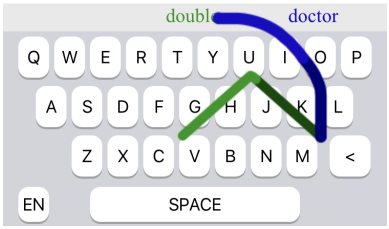

Figure 4.3. OctoType in Gesture Mode after typing 'do' prefix in the middle of drawing the blue gesture.

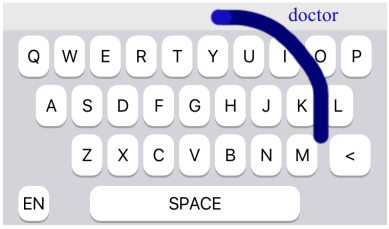

Figure 4.4. OctoType in Gesture Mode after typing 'do' prefix at the end of drawing the blue gesture.

4.2 Dictionary

Suggestions are shown according to a dictionary that was built from MacKenzie and Soukoreff phrases set (MacKenzie and Soukoreff 2003). Since the suggestion mechanism itself is not the focus of the study, the dictionary is constructed by simply sorting words in the set by frequency. However, words that are not long enough were not included in the dictionary. In particular, if the sum of the prefix length of a word and T was less than the length of the whole word, then it was added to the dictionary (where T is average gesture drawing time). The value of T was approximately picked to be equal to the typing time of 3 letters.

In total, there are 409 prefixes (see Table 4.1) that were generated from the MacKenzie and Soukoreff set that contains 255 phrases with a mean value of 5.3 words per phrase and $\sigma = 1.1$ words. The theoretical improvement of entry speed using this dictionary was equal to 9.4%, which was estimated by the total time needed to finish the whole phrases set.

Table 4.1. First 10 prefixes in the dictionary.

Prefix	Suggestions
a	about, always, agree
ab	abandoned, aboard
ac	accident, according, accept
acc	accompanied, acceptance, account
acu	acutely
ad	adult, addition, advance
af	afraid, after
ag	again, agreement
ah	ahead
ai	airport

CHAPTER FIVE

Experiment I

The main objective of this experiment was to examine the text entry speed and error rate of OctoType compared to other types of virtual keyboards. Our hypothesis was that OctoType outperform standard keying and swyping keyboards in terms of text-entry speed and accuracy.

5.1 Participants

A user study was conducted with 20 participants (two were left-handed and 18 were right-handed) between ages 18 to 20. They were recruited from an introductory computer science course taught at the Department of Computer Science at Baylor University. All participants, including 3 female and 17 male, use a smartphone (Android or iOS) on daily basis.

5.2 Materials

To evaluate virtual keyboards, experiments were conducted using the iPhone 7 (4.7-inch diagonal screen) with the iOS 11.1 operating system. Together with OctoType, we used GBoard [1] (swyping keyboard developed by Google). Also, OctoType was used in three keying modes: without suggestions, suggestion selection by tapping, and suggestion selection by gesture drawing. Thus, the study was conducted using 4 types of virtual keyboards. In addition, suggestion selection types included normal and training modes. Since the experiment evaluates the performance of virtual keyboards using only one dominant hand, the iPhone 7 was chosen due to smaller screen size than other versions of iPhone with 3D-touch technology. We used short phrases from MacKenzie and Soukoreff (MacKenzie and Soukoreff 2003) set that were

[1] https://itunes.apple.com/us/app/gboard/id1091700242

specifically designed to evaluate text entry techniques. Five randomly chosen phrases were assigned to each type of keyboard.

5.3 Measurements

To evaluate keyboard performance, we used Soukoreff and MacKenzie (Soukoreff and MacKenzie 2003, Soukoreff and MacKenzie 2004) metrics:

$$WPM = \frac{T}{S} \times 60 \times \frac{1}{5} \tag{5.1}$$

where T is the length of transcribed string and S is the total amount of time (in seconds).

$$Error = \frac{IF}{C + IF} \times 100 \tag{5.2}$$

where C is correct keystrokes and IF is incorrect but fixed keystrokes.

5.4 Procedure

The study is within-subject where each participant individually performed a single session lasting approximately 30 minutes. At the beginning of a session, a participant was asked to self-report his/her experience with different types of virtual keyboard on Likert scale. After the introduction of the experiment, a tutorial about OctoType and its usage was provided. Each participant was given 5 minutes of training time to practice with OctoType in training mode (see Figure 5.1). In training mode, the testing application highlights a prefix that needs to be typed and a color of a gesture that needs to be drawn. In contrast, the testing application in normal mode highlights only the current word. Additionally, in both modes the testing application highlights letters in the text field with blue or red colors, indicating that a word is typed correctly or not.

Once the training is over, participants typed randomly assigned phrases using 4 different types of keyboards. Keyboards with suggestion selection by tapping and

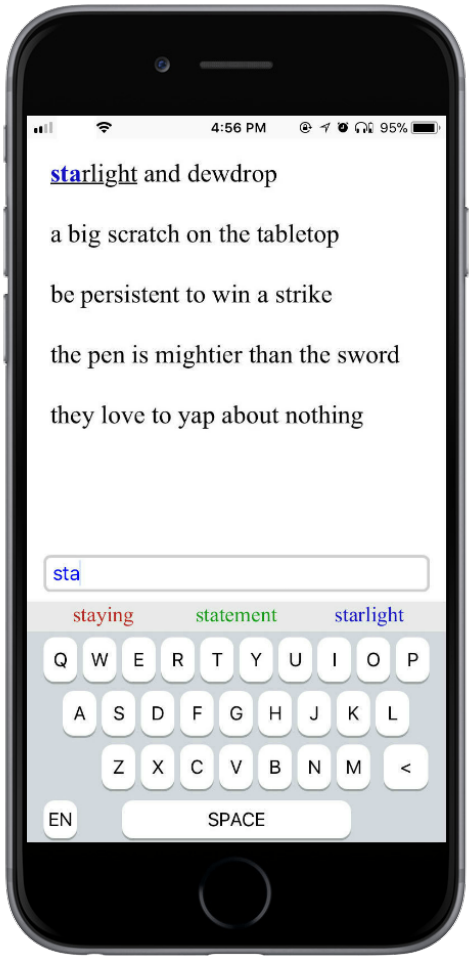

Figure 5.1. Testing application in training mode.

gestures were used in two different modes: training and default, i.e. 6 runs per participant. The order of keyboards to be tested was randomized in each session. Since participants typed using only one hand, they were allowed to take a break in case of fatigue. At the end of a session, each participant filled out a survey about his/her experience with OctoType.

5.5 Results

The means and standard deviations of WPM for different keyboards are shown on Figure 5.3 (for abbreviations of different keyboards see Table 5.1). The results of paired t-test for each pair are shown on Table 5.2.

Table 5.1. Abbreviations for different keyboards.

Abbreviation	Description
GM-T	Gesture based suggestion selection in training mode
GM	Gesture based suggestion selection in default mode
Tap-T	Suggestion selection by tapping in training mode
Tap	Suggestion selection by tapping in default mode
DM	No suggestion in default mode
GB	GBoard in default mode

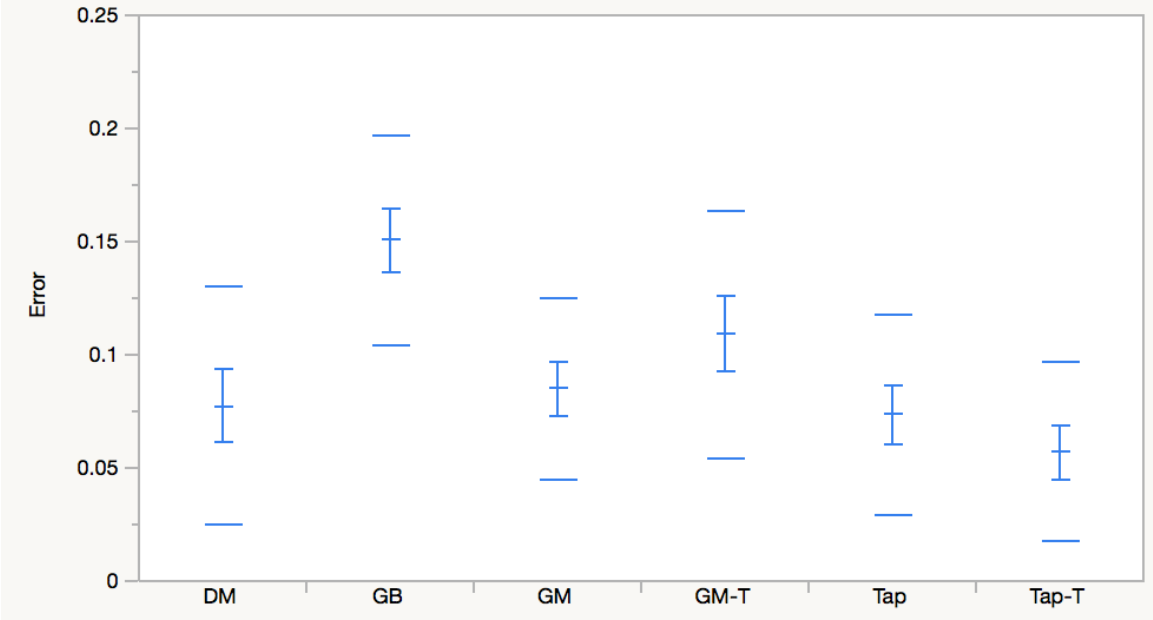

Figure 5.2. Error rate for all keyboards.

Taking into consideration statistically significant outcomes of paired t-test, we see that both Tap-T and DM have a better performance than GB and GM. Also, it appears that DM performs better than GM and GB.

However, according to differences in entry speed between DM and GM-T, there were two groups of typers:

- Group-1: entry speed in DM was at least 22 WPM and faster than in GM-T
- Group-2: entry speed in DM was at most 22 WPM and slower than in GM-T

For the Group-2 data, the results appear to be different (see Figure 5.4). Tap-T still shows the best performance, while GM-T outperforms DM and GB.

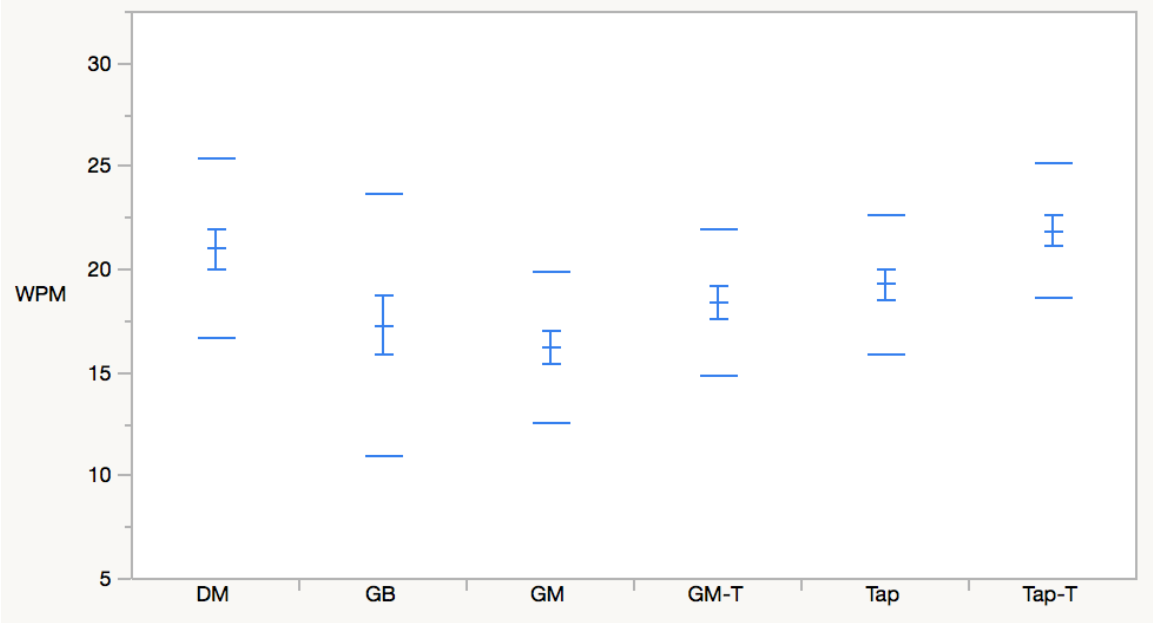

Figure 5.3. A comparison of overall WPM.

The error rate for all type of keyboards are shown in Figure 5.2. Tap, Tap-T and DM have the lowest error rate, while GB has the highest. Also, there is a significant decrease in GM's error rate after training.

Table 5.2. Paired t-test for WPM.

	Diff	**p-value**
Tap-T – GM	5.63	0.0001
DM – GM	4.76	0.0005
Tap-T – GB	4.58	0.0008
DM – GB	3.71	0.0064
Tap-T – GM-T	3.49	0.0104
Tap – GM	3.04	0.0248

5.6 Conclusion and Discussion

Gesture selection had statistically significant lower performance than selection by tapping, because single touch takes less time than executing a gesture. However,

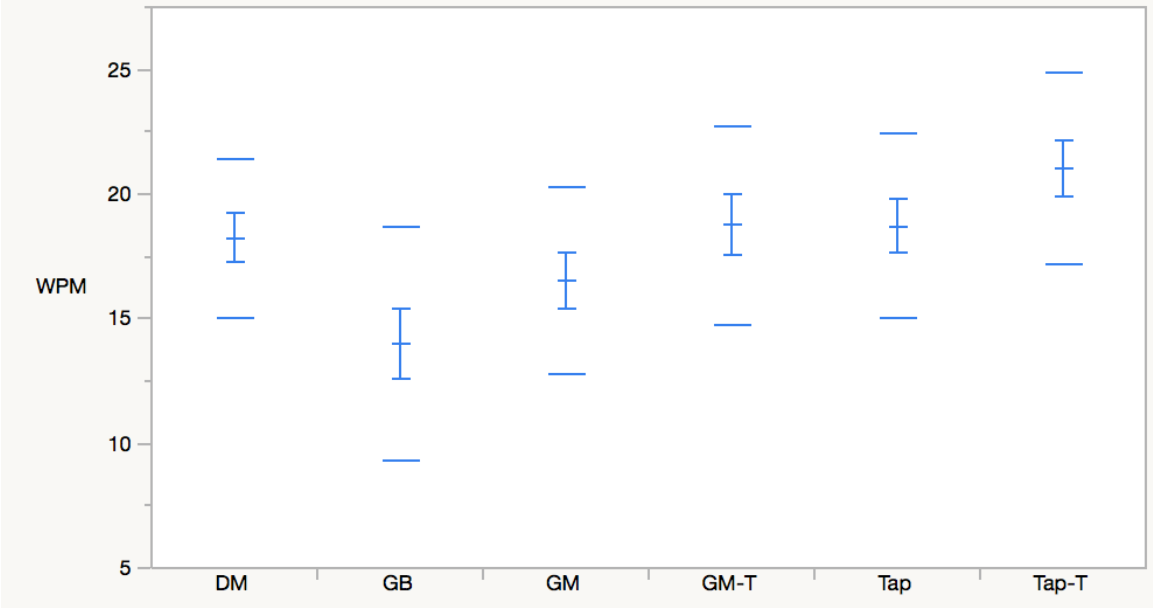

Figure 5.4. A comparison of WPM for Group-2.

for the group of participants who typed slower than 22 WPM in standard keying keyboard (Group-2), OctoType had higher WPM than default keying without suggestion selection. In fact, suggestion selection has benefits for Group-2, because it decreases the number of letters required to type a word which allows the user to complete the word with one final motion. Unfortunately, for the group of participants who typed faster than 22 WPM in standard keying keyboard (Group-1), users were able to complete the word faster using their normal input mechanism. The additional movement that OctoType would provide caused the users to hesitate enough that understanding the action slowed down their overall WPM.

From the feedback collected from participants, they suggested that drawing a gesture with force touch becomes complicated at the end of drawing a gesture. This suggestion aligns with the differences seen between groups 1 and 2. Simplifying the gesture might improve the times of both groups. Additionally, it was noted that force touch should be used only for activation of Gesture Mode, while gestures drawn

regardless of pressure level. This was motivation for the second experiment with minor modifications to OctoType.

CHAPTER SIX

Experiment II

According to the participant feedback from Experiment I, executing a gesture while pressing the display harder causes fatigue. Therefore, this property of OctoType was changed so that force touch is required only for activation of Gesture Mode, rather than throughout the entire gesture, and a gesture can be executed regardless of pressure level. Our hypothesis was that the modification made to OctoType increases text-entry speed and decreases error rate. Moreover, we expected that accuracy of drawing gestures increases significantly, while their execution time decreases.

6.1 Participants, Materials and Procedure

A user study was conducted with 10 right-handed participants between ages 22 to 30. Each uses a smartphone (Android or iOS) on a daily basis. The participants of this experiment were not participants in Experiment I. All materials and procedures were the same as in Experiment I including the iPhone 7 smartphone, MacKenzie and Soukoreff phrases set (MacKenzie and Soukoreff 2003), and 4 types of keyboard.

6.2 Measurements

In addition to WPM and $Error$, we measured accuracy and execution time of gestures. For a given gesture (*red*, *green*, or *blue*), accuracy is calculated as follows:

$$Accuracy = \frac{S}{U + S} \tag{6.1}$$

where S and U are the numbers of successful and unsuccessful executions of the gesture.

The execution time of a gesture was calculated as a time difference between activation of Gesture Mode and a finger release. Additionally, we calculated *Gesture*

Mode activation time which is equal to time spent from the first touch until gestures become visible. The results of accuracy, Gesture Mode activation time, and gesture execution time from Experiment I were obtained from log data and video records of the iPhone 7 screen.

6.3 Results

Gesture based suggestion selection in training mode showed 18.27 WPM, while for default mode with no suggestions it was 16.17 WPM. Thus, there is a 2.1 WPM difference, which is $\sim 13.1\%$ improvement in text entry speed using OctoType. With t-Ratio equal to 3.41 and significance level $p = 0.0056$ for the paired t-test, the improvement is considered to be statistically significant. Similarly to results of Experiment I, TAP-T still shows the highest WPM among all keyboards (see Figure 6.1). Surprisingly, TAP-T and GM-T had similar WPM, since paired t-test failed to reject the difference between them. The full list of differences with statistically significant results are shown on Table 6.1.

Error rates for DM, GM, GM-T, TAP, TAP-T are similar with mean values 4.1%, 6.2%, 5.4%, 3.9%, 3.6% respectively (see Figure 6.2). In contrast, GB showed the highest error rate with 8.3% mean value and 4.5% standard error mean. Also, paired t-test gave no statistical difference between error rates for any pair.

Table 6.1. Paired t-test for WPM.

	Difference	t-Ratio	p-value
GM-T – GM	3.52	5.28	0.0006
GM-T – DM	2.1	3.42	0.0056
TAP-T – GM	4.92	4.02	0.0025
TAP-T – TAP	2.71	2.04	0.0401
TAP-T – GB	4.91	2.08	0.0377
TAP-T – DM	3.49	4.28	0.0018

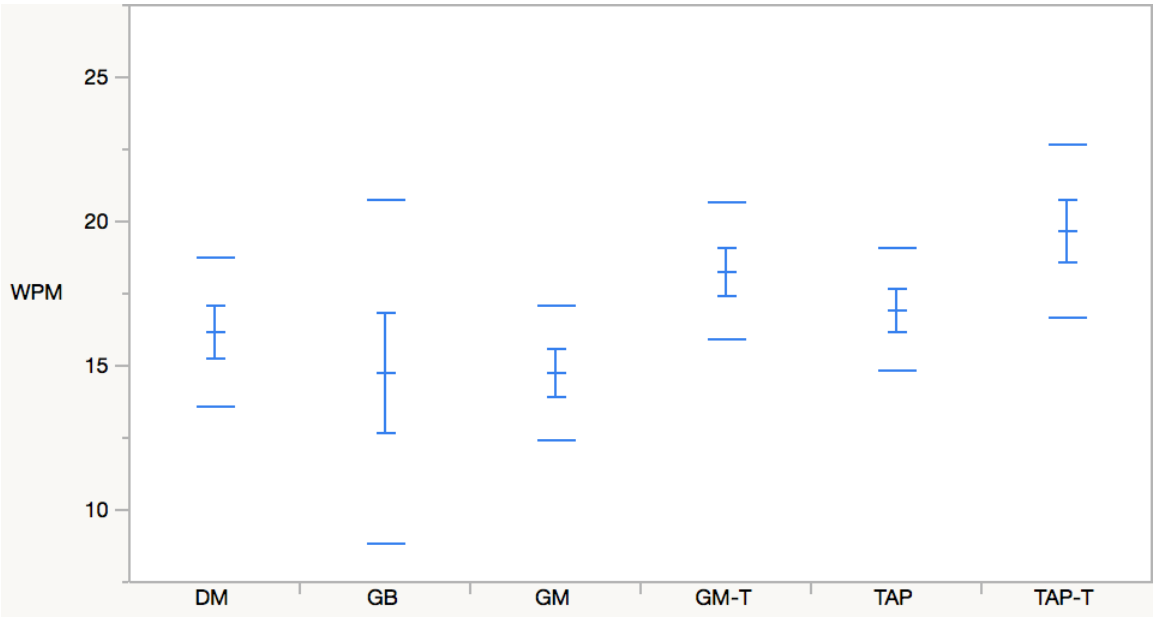

Figure 6.1. A comparison of WPM for all keyboards.

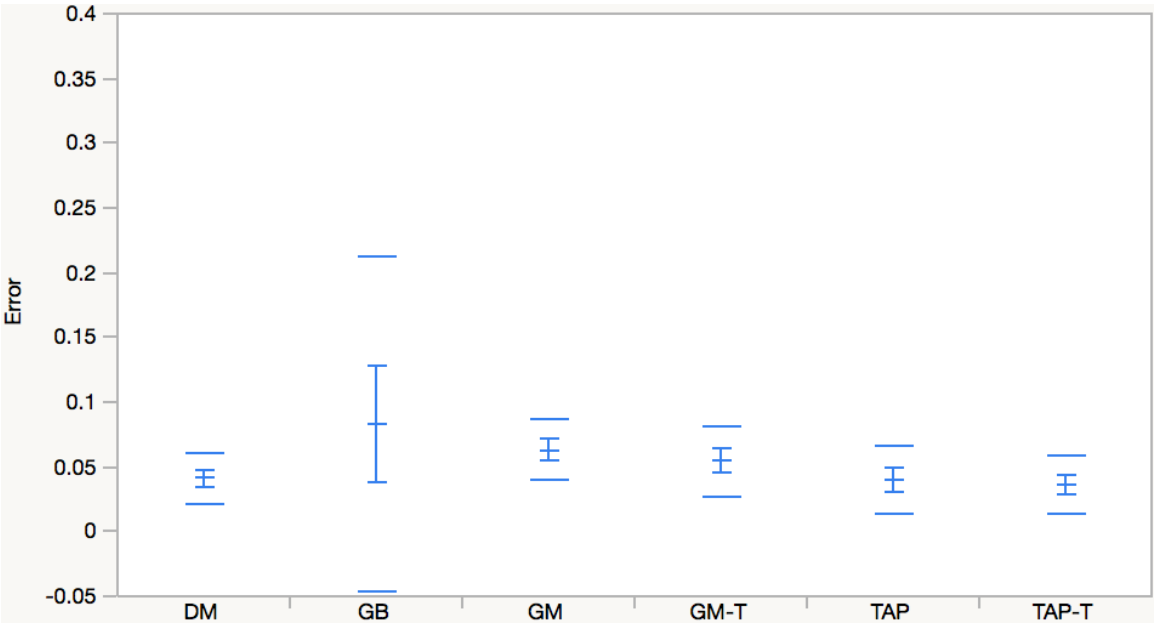

Figure 6.2. Error rates for all keyboards.

Overall, gesture drawing had high accuracy and low execution time. The average Gesture Mode activation time was equal to 0.37 seconds and average time per character entry was 0.6 seconds. The accuracy of gestures for all three gestures were drastically high compared to Experiment I (see Table 6.2). Which implies that force touch should be used only for activation of Gesture Mode.

Table 6.2. Gesture Drawing Accuracy.

	Red	Green	Blue
Experiment I	93%	85.2%	89.3%
Experiment II	97.5%	99.9%	99.9%

The execution time decreased in Experiment II. In particular, execution time of *blue* gesture (0.35 seconds) were even lower than per character entry time. Thus, execution time for all three gestures do not differ a lot from entry speed of a single character with a keyboard in normal keying mode (see Table 6.3).

Table 6.3. Gesture Execution Time.

	Red	Green	Blue
Experiment I	0.78	0.8	0.6
Experiment II	0.5	0.6	0.35

6.4 Conclusion and Discussion

In this experiment, we evaluated OctoType with modifications to the keyboard. As a result, our proposed force touch gesture based suggestion selection outperformed standard keying and swyping methods. Moreover, in training mode there were no significant differences between OctoType and suggestion selection by tapping in terms of WPM and error rate.

Measurements of accuracy and execution time of gesture drawing showed promising results. When a user applies additional pressure to the screen only for

activation of Gesture Mode, there was nearly no error for gesture drawing. Additionally, the speed of gesture drawing was almost as fast as single tap movement. In particular, the *curve* gesture showed the very small execution time compared to other *angle* gestures. Thus, the combination of force touch for gesture mode activation and drawing a gesture regardless of pressure level potentially has high usability.

CHAPTER SEVEN

Force Touch Gesture Interaction Framework

This chapter describes the framework for iOS applications that we developed based on the results of experiment II. The combination of using force touch for gesture mode activation and then drawing a gesture regardless of pressure level may have high usability in touch screen mobile devices. This combination can potentially be used not only at keyboard level, but at application and/or an operating system's UI level. In fact, this kind of gesture interaction does not conflict with already existing UI elements in apps or in mobile OS, since force touch is required for gesture mode activation. The framework provides to researchers and developers a tool for fast prototyping and designing conflict-free gesture based interfaces on pressure-sensitive touch screen devices.

7.1 Framework Features

The framework, named *Gesturizer*, enables unistroke gesture based interaction in any iOS apps running on 3D-touch devices without interfering with existing UI elements. It is written in Swift[1] programming language and can be imported to an existing iOS app in relatively few lines of code.

Once the framework is imported to an application, there are two modes of operation for gesture interaction: *default* and *training*. In default mode, there is no visual representation of gestures. Instead, when a user applies additional pressure to the screen, a light vibration feedback is given indicating that the user can start executing a gesture.

On the other hand, training mode is activated after a user hesitates to draw a gesture. The *hesitation* in this case is defined as 1 second delay after force touch.

[1] https://developer.apple.com/swift/

In training mode illustration of gestures on the screen is similar to OctoType (see Figure 7.1), intending to provide dynamic guide for learning and execution of different gestures.

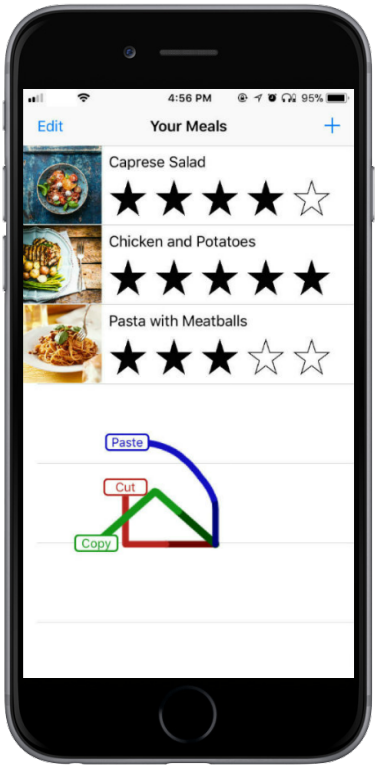

Figure 7.1. Example of using Gesturizer framework on sample app with scrollable and clickable UI elements.

By default, the framework provides three different gestures that we used in OctoType. The number of gestures and their templates can be changed by providing a list of (x, y) coordinate points for a gesture. Additionally, parameters of the framework such as pressure level for gesture mode activation and hesitation time for training mode activation can be customized as well.

7.2 Architecture and Design

Architecture of the Gesturizer framework contains two main components (Figure 7.2):

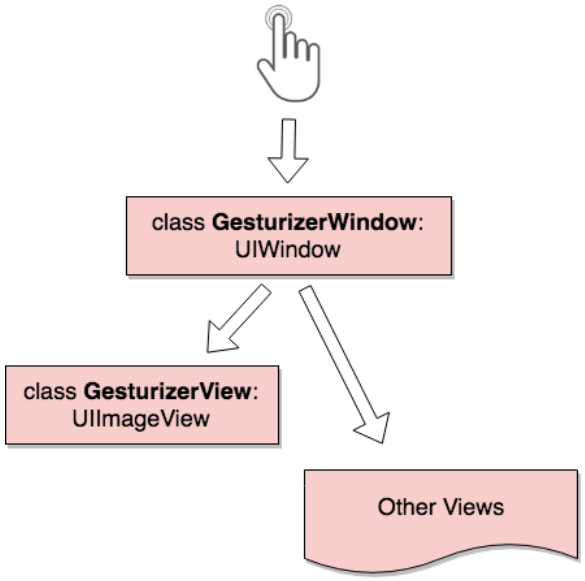

Figure 7.2. Architecture of Gesturizer Framework.

- *GesturizerWindow* handles all touches and decides which UI element will receive a touch event.
- *GesturizerView* is used for illustrating colorful gestures in training mode.

When a user touches the screen, all touch events firstly go to GesturizerWindow. Depending on the pressure level, GesturizerWindow decides whether a touch is associated with gesture interaction or common touch event that must be handled with UI elements of an app. GesturizerView is only needed for displaying gestures in training mode, while the processing of the force of pressing and the recognition of gestures occurs in the GesturizerWindow. GesturizerWindow does not create delays in the application due to robustness of the $1-recognizer.

Samples of code to import GesturizerWindow and GesturizerView are shown on Figure A.1 and Figure A.2 in Appendix A.

CHAPTER EIGHT

Conclusion, Discussion and Future Works

In this work we evaluated use of force touch gesture interaction as an input mechanism during one-handed virtual keyboard input. More specifically, force touch gesture interaction was added to allow suggestion selection. Unfortunately, gesture selection had lower performance than selection by tapping. However, OctoType had better performance than default keying without suggestion selection in terms of WPM. Because OctoType decreases number of letters required to type, it allows the user to complete the word with one final motion. In contrast, for group of people who typed faster that 22 words per minute on standard keying keyboard, users could complete the word faster using their normal input mechanism. The additional movement that OctoType required caused the users to hesitate enough that unfamiliarity with the action reduced their WPM. Possibly text-entry speed using OctoType increases in long-term use, since gestures become more intuitive to be executed.

Modifications made to OctoType showed that force touch gesture interaction is accurate and fast, achieving more than 97% accuracy and execution time less than 1 second. These promising results were a motivation to build the Gesturizer framework that enables conflict-free gesture based user interfaces to iOS applications running on devices with 3D-touch technology. The framework is designed in a way that researchers and developers can use it for fast prototyping of force touch gesture interfaces. In fact, this is one of the biggest results achieved during the research. The interaction implemented in the framework has high potential in mobile user interfaces. Therefore, further study should be conducted to evaluate usability of force touch gesture based interaction in different applications on mobile devices.

In regards to future implementations of force touch gesture interaction, there are numerous things that would require improvement. Further study is needed to investigate the level of pressure to activate Gesture Mode. Also, the dictionary of suggestions need to be improved by using existing approaches that is common for these kinds of purposes. Lastly, since the mechanisms used in OctoType can be expanded to use a higher number of gestures and suggestions, functionality of OctoType can be improved by increasing number of gestures.

APPENDICES

APPENDIX A

Framework

```
var window: UIWindow? = GesturizerWindow()
```

Figure A.1. AppDelegate.swift

```
override func viewDidAppear(_ animated: Bool) {
    super.viewDidAppear(animated)
    let window = UIApplication.shared.keyWindow as! GesturizerWindow
    let view = GesturizerView()
    view.gestureHandler = {index in
      // implement a command related to a gesture
      // with given index in the gesture list
    }
    window.setGestureView(view: view)
}
```

Figure A.2. ViewController.swift

APPENDIX B

Vita

Name:	Kuanysh Zhunussov
Date of Birth:	1994
Place of Birth:	Pavlodar, Kazakhstan
Email:	kuanysh.zhunussov@gmail.com
M.S. (seeking):	August 2016 - May 2018 (Expected)
	Computer Science
	Department of Computer Science
	Baylor University, Waco, TX, USA
BEngTech:	August 2012 - June 2016
	Computer Systems and Software
	Faculty of Information Technologies
	Kazakh-British Technical University, Almaty, Kazakhstan

BIBLIOGRAPHY

Anderson, D., C. Bailey, and M. Skubic (2004). Hidden markov model symbol recognition for sketch-based interfaces. In *AAAI fall symposium*, pp. 15–21.

Anthony, L., R.-D. Vatavu, and J. O. Wobbrock (2013). Understanding the consistency of users' pen and finger stroke gesture articulation. In *Proceedings of Graphics Interface 2013*, pp. 87–94. Canadian Information Processing Society.

Arvo, J. and K. Novins (2000). Fluid sketches: continuous recognition and morphing of simple hand-drawn shapes. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pp. 73–80. ACM.

Bau, O. and W. E. Mackay (2008). Octopocus: A dynamic guide for learning gesture-based command sets. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, UIST '08, New York, NY, USA, pp. 37–46. ACM.

Bi, X., B. A. Smith, and S. Zhai (2010). Quasi-qwerty soft keyboard optimization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 283–286. ACM.

Callahan, J., D. Hopkins, M. Weiser, and B. Shneiderman (1988). An empirical comparison of pie vs. linear menus. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pp. 95–100. ACM.

Cheng, L.-P., H.-S. Liang, C.-Y. Wu, and M. Y. Chen (2013). igrasp: grasp-based adaptive keyboard for mobile devices. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pp. 3037–3046. ACM.

Cooper, W. E. (2012). *Cognitive aspects of skilled typewriting*. Springer Science & Business Media.

Eibe Frank, M. A. H. and I. H. Witten (2016). The weka workbench. online appendix for "data mining: Practical machine learning tools and techniques". *Morgan Kaufmann, Fourth Edition*.

Girouard, A., J. Lo, M. Riyadh, F. Daliri, A. K. Eady, and J. Pasquero (2015). One-handed bend interactions with deformable smartphones. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*, pp. 1509–1518. ACM.

Hastie, T. and R. Tibshirani (1998). Classification by pairwise coupling. In *Advances in neural information processing systems*, pp. 507–513.

Henze, N., E. Rukzio, and S. Boll (2012). Observational and experimental investigation of typing behaviour using virtual keyboards for mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, New York, NY, USA, pp. 2659–2668. ACM.

Heo, S. and G. Lee (2011). Force gestures: augmented touch screen gestures using normal and tangential force. In *CHI'11 Extended Abstracts on Human Factors in Computing Systems*, pp. 1909–1914. ACM.

Hoober, S. (2013). How do users really hold mobile devices. *Uxmatters (http://www.uxmatter.com). Published: Feburary 18*.

Igarashi, T., S. Kawachiya, H. Tanaka, and S. Matsuoka (1998). Pegasus: a drawing system for rapid geometric design. In *CHI 98 conference summary on Human factors in computing systems*, pp. 24–25. ACM.

Karlson, A. K., B. B. Bederson, and J. L. Contreras-Vidal (2006). Studies in one-handed mobile design: Habit, desire and agility. In *Proc. 4th ERCIM Workshop User Interfaces All (UI4ALL)*, pp. 1–10. Citeseer.

Kölsch, M. and M. Turk (2002). Keyboards without keyboards: A survey of virtual keyboards. In *Workshop on Sensing and Input for Media-centric Systems, Santa Barbara, CA*.

Kurtenbach, G. (1993). *The Design and Evaluation of Marking Menus*. Ph. D. thesis, Dept. of Computer Science, University of Toronto.

Kurtenbach, G., T. P. Moran, and W. Buxton (1994). Contextual animation of gestural commands. In *Computer Graphics Forum*, Volume 13, pp. 305–314. Wiley Online Library.

Leung, L. and P. Aarabi (2014). Mobile circular keyboards. In *Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on*, pp. 1–4. IEEE.

Li, J., X. Zhang, X. Ao, and G. Dai (2005). Sketch recognition with continuous feedback based on incremental intention extraction. In *IUI*.

MacKenzie, I. S. and R. W. Soukoreff (2003). Phrase sets for evaluating text entry techniques. In *CHI '03 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '03, New York, NY, USA, pp. 754–755. ACM.

MacKenzie, I. S. and S. X. Zhang (1999). The design and evaluation of a high-performance soft keyboard. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pp. 25–31. ACM.

Mankoff, J., S. E. Hudson, and G. D. Abowd (2000). Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, UIST '00, New York, NY, USA, pp. 11–20. ACM.

Moyle, M. and A. Cockburn (2003). The design and evaluation of a flick gesture for'back'and'forward'in web browsers. In *Proceedings of the Fourth Australasian user interface conference on User interfaces 2003-Volume 18*, pp. 39–46. Australian Computer Society, Inc.

Nguyen, H. and M. C. Bartha (2012). Shape writing on tablets: Better performance or better experience? *Proceedings of the Human Factors and Ergonomics Society Annual Meeting 56*(1), 1591–1593.

Ougiaroglou, S. and G. Evangelidis (2015). Dealing with noisy data in the context of k-nn classification. In *Proceedings of the 7th Balkan Conference on Informatics Conference*, pp. 28. ACM.

Pittman, J. A. (1991). Recognizing handwritten text. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 271–275. ACM.

Poppinga, B., A. Sahami Shirazi, N. Henze, W. Heuten, and S. Boll (2014). Understanding shortcut gestures on mobile touch devices. In *Proceedings of the 16th international conference on Human-computer interaction with mobile devices & services*, pp. 173–182. ACM.

Quinn, P. and S. Zhai (2016). A cost-benefit study of text entry suggestion interaction. In *Proceedings of the 2016 CHI conference on human factors in computing systems*, pp. 83–88. ACM.

Rodriguez, J., G. Sánchez, and J. Lladós (2007). A pen-based interface for real-time document edition. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, Volume 2, pp. 939–943. IEEE.

Rubine, D. (1991). *Specifying gestures by example*, Volume 25. ACM.

Sezgin, T. M. and R. Davis (2005). Hmm-based efficient sketch recognition. In *Proceedings of the 10th international conference on Intelligent user interfaces*, pp. 281–283. ACM.

Soukoreff, R. W. and I. S. MacKenzie (2003). Metrics for text entry research: An evaluation of msd and kspc, and a new unified error metric. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '03, New York, NY, USA, pp. 113–120. ACM.

Soukoreff, R. W. and I. S. MacKenzie (2004). Recent developments in text-entry error rate measurement. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '04, New York, NY, USA, pp. 1425–1428. ACM.

Vatavu, R.-D. (2011). Stroke gesture datasets. [Online; accessed 11-March-2018].

Vatavu, R.-D., L. Anthony, and J. O. Wobbrock (2014). Gesture heatmaps: Understanding gesture performance with colorful visualizations. In *Proceedings of the 16th International Conference on Multimodal Interaction*, pp. 172–179. ACM.

Wilson, G., S. Brewster, and M. Halvey (2013). Towards utilising one-handed multi-digit pressure input. In *CHI'13 Extended Abstracts on Human Factors in Computing Systems*, pp. 1317–1322. ACM.

Wobbrock, J. O., A. D. Wilson, and Y. Li (2007). Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, New York, NY, USA, pp. 159–168. ACM.

Zeng, Z.-Q., H.-B. Yu, H.-R. Xu, Y.-Q. Xie, and J. Gao (2008). Fast training support vector machines using parallel sequential minimal optimization. In *Intelligent System and Knowledge Engineering, 2008. ISKE 2008. 3rd International Conference on*, Volume 1, pp. 997–1001. IEEE.

Zhai, S., M. Hunter, and B. A. Smith (2000). The metropolis keyboard-an exploration of quantitative techniques for virtual keyboard design. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pp. 119–128. ACM.