KTH Computer Science
and Communication

Designing a Better Touch Keyboard

JOEL BESADA (JBESADA@KTH.SE)

DD143X Degree Project in Computer Science, First Level
KTH Royal Institute of Technology, School of Computer Science and Communication
-
Supervisor: Anders Askenfelt
Examiner: Mårten Björkman

Abstract

The growing market of smartphone and tablet devices has made touch screens a big part of how people interact with technology today. The standard virtual keyboard used on these devices is based on the computer keyboard – a design that may not be very well suited for touch screens.

This thesis seeks to answer whether there are alternative ways to design virtual touch keyboards. An alternative design may require training to use, but the user should be able to use it viably after a reasonable amount of practice. Optimally, the user should also have the potential to reach a higher level of efficiency than what is possible on the standard touch keyboard design.

Analysis is done by researching current and previous attempts at alternative keyboard designs, and by implementing an own prototype based on observations made on the researched designs. The thesis concludes that there is an interest in solving issues encountered in this area, and that there is good potential in building viable virtual keyboards using alternative design paradigms.

Referat

Utformning av ett bättre pektangentbord

Den växande marknaden för smarttelefoner och surfplattor har gjort pekskärmen till en stor del av hur folket interagerar med dagens teknologi. Det virtuella standardtangentbordet som används på dessa enheter är baserad på datortangentbordet – en design som möjligen inte är särskilt bra anpassad för pekskärmar.

Denna uppsats söker att besvara om det finns alternativa sätt att designa ett virtuellt pektangentbort. En alternativ design kan kräva träning för att kunna användas, men användaren ska gångbart kunna använda den efter en rimlig övningsmängd. Optimalt sett ska användaren ha potentialen att uppnå högre nivåer av effektivitet än vad som är möjligt på pektangentbordets standarddesign.

Analysen görs genom undersökning av nuvarande och föregående försök till alternativa tangentbordsdesigner. Dessutom implementeras en egen prototyp baserad på observationer på de granskade designerna. Uppsatsen drar slutsatsen att det finns ett intresse för att lösa de problem som hör till detta område och att det finns bra potential till att bygga tillämpbara virtuella tangentbord med alternativa designparadigm.

Contents

Chapter 1

Introduction

Since the popularization of smartphone and tablet devices, touch screens have become a big part of how we interact with technology today. While the touch screen in many cases allows for a simpler and more intuitive interaction with applications, the process of typing text has not evolved much from its physical origins — the computer keyboard.

The smartphone and tablet market is constantly expanding, embracing new audiences of users with limited computer experience. Because of the expected familiarity with computer keyboards that current text input implementations depend on today, it is of interest to research alternative and better suited solutions for text input on touch devices.

1.1 Problem Statement

In the adaption process of the physical computer keyboard to touch devices, certain compromises have had to been made. Touch devices of today are in general much smaller than the size of a regular computer keyboard, which forces the keys to be shrunk down to fit on the screen. This leads to a need of very precise finger positioning when typing, often leading to many mistakes.

Another drawback with the virtual computer keyboard is the loss of tactility when put on a touch screen. Most experienced computer users rely on being able to feel where their fingers are positioned, allowing them to quickly move their fingers to the chosen keys. Without being able to feel where the fingers are in relation to other keys, the ability to type quickly and without looking at the keys is severely degraded.

With these inherent drawbacks of the standard touch input systems of today, the major advantage is the fact that it is of a familiar format to users with previous computer experience. However, with the increased number of inexperienced computer users that are starting to use touch devices, this advantage is becoming more and more irrelevant.

The familiar format certainly makes it easy for experienced computer users to start typing on touch devices without having much or any training. However, a user might be able to type more efficiently if efforts were put into learning to use an alternative text input system. This alternative system would optimally be designed specifically for touch screens, embracing their constraints instead of trying to fight against them.

This brings us to the primary questions that this thesis seeks to answer: Is it possible to design a text input alternative on touch devices that

- is viable,
- has the potential of being more efficient,
- and is not based on the ill-suited design of the computer keyboard?

1.2 Purpose

As previously described, there are a number of critical flaws in the design of current standard touch keyboards. This thesis explores alternative text input methods on touch devices with the aim of providing inspiration and ideas in disrupting the current popular, but arguably flawed, design paradigm. The ultimate goal of research and exploration within this area is to gain important insight into how a better text input system on touch devices can be designed for both experienced and inexperienced computer keyboard users.

The aspired practical results of exploration within this area of interaction design is not to allow us to build a system that instantly improves our ability to type on current touch keyboards. Instead, the ultimate goal is to be able to design input

methods that truly utilize the strengths of touch screens, and once learned, have the potential of providing a more efficient and pleasant typing experience than current systems.

1.3 Approach

The undertaken approach to answer the questions presented in the problem statement of this thesis is divided into multiple steps. As a starting point, background research of current and previous attempts at designing a keyboard alternative is conducted. This is presented to the reader in a summarized form in the *Background* chapter.

With the gathered information as inspiration, a prototype is built where the strengths and drawbacks of the researched systems are taken into consideration. This process and the results of it is detailed out in the *Method* chapter. The prototype is then user tested and measured, with results and analysis in the *Analysis* chapter.

Finally, based on the gained insights from the previous steps, conclusions are drawn and presented in the *Conclusions* chapter.

Chapter 2

Background

A number of attempts at creating alternative text input systems have been made since the introduction of the QWERTY keyboard, many of which had the goal of designing a system to be used with a single hand. The concepts explored within these systems have later found their way into implementations of alternative touch keyboards. This chapter is dedicated to give the reader an overview of significant entries in this area, both physical and virtual.

2.1 Chord Keyboards

A chord keyboard is an alternative to regular keyboards that builds around the concept of pressing multiple keys at once to produce different outputs. Much like how a chord is played on a piano, the user simultaneously presses down a number of keys to build a chord, producing a certain output.

The first chord keyboard is thought to have been created by Benjamin Livermore in 1857 with the invention of Livermore's Permutation Typograph[1]. This device featured six keys and a roll of paper inside on which the entered characters were printed.

The chorded keyboard concept is not unseen on regular keyboards, the SHIFT, CTRL and ALT keys on a PC are specifically meant to be used in chords. However, a chord keyboard uses chords at a larger extent, allowing a greater set of characters to be produced by a limited number of keys. Without any mechanic other than registering keys being pressed down at the same time, a set of n keys can produce a number of characters given by the formula:

$$2^n - 1 \tag{2.1}$$

This formula is devised from the simple realization that we can think of each key as a bit in a binary number of length n, with the digit 1 representing pressed down keys and the digit 0 indicating unpressed keys. This binary number can represent 2^n different values, but we need to subtract the state where none of the keys are pressed down (all bits are 0), leaving us with the formula above. As an example, a

5-key chord keyboard can produce 31 different characters, which is enough for every letter in the English alphabet.

A drawback with the extensive use of key combinations is the difficulty of settings labels to which letters a key produces. In a fully utilized n-key setup, a single key is used in producing

$$2^n - 2^{n-1} \tag{2.2}$$

different outputs (the difference in the number of outputs of $n-1$ and n keys). In our 5-key example, one single key would be used for typing 16 different characters.

This leads to the absolute necessity of memorizing the key outputs and learning to type without looking at the keys, a practice that is usually called *touch typing*. This contrasts to how a user can look down on a regular keyboard to find the correct key – *hunt and peck typing*.

2.2 NLS

The NLS, On-Line System, was a multi-user system demonstrated[1] by Douglas Engelbart in 1968. The system included a first of its kind computer mouse and a five key chorded keyset, which were to be used simultaneously. The goal of the system was not to make computing more accessible to the general public, but to enable experienced computer users to improve their productivity by learning a more optimized system.

In addition to the mouse and chorded keyset, the system also had a regular QWERTY keyboard. The QWERTY keyboard was to be used when typing longer sequences of text, while the mouse and chorded keyset combination would be used to more efficiently edit text. By pointing the mouse at different parts of the text and entering commands on the keyset, characters could be inserted and removed from the text in a more efficient manner. The mouse had three keys, two of which could also be used when building chords with the keyset to extend the range of available characters and commands.

Engelbart used a relatively simple mapping scheme between chords and the 26 English alphabetic letters. The letters A to Z were numbered according to their alphabetic order, and each chord would represent a 5-bit binary number. Pressing the rightmost key represented the binary number 00001, the decimal number 1, corresponding to the first letter of the alphabet, A. Pressing the first, second and fifth key forming 11001 (decimally 25), produced Y, the 25th letter in the alphabet.

Tests done in early 1960 concluded that the system could be mastered in less than two hours, and while regular keyboard are faster in normal typing, editing and text manoeuvring was much more efficient with Engelbart's mouse and keyset system.[2]

[1] This demonstration later became to be called *The Mother of All Demos*, since it introduced experimental versions of many common technologies of today, such as: the computer mouse, video and teleconferencing, hypertext and hypermedia, object addressing and dynamic file linking, word processing and a collaborative real-time editor.

2.3 Microwriter

The Microwriter was a portable word processing device sold in the early 1980s by Microwriter Ltd. The device featured a 6-key chorded keyboard and a small LCD display for a single line of text. The thumb had two keys, one for regular letter chords and the other for switching input modes. The rest of the four fingers of the hand had each a dedicated button for chords.

A notable feature of the Microwriter was its approach to mapping chords to letters. Instead of a simple method based on alphabetic order such as in Engelbart's keyset, most of the chords represented a shape that should be easy to remember for the user. The device came with a booklet of mnemonics that would aid in learning and remembering the letters. When ordering the main five keys to form a pattern resembling an upside down V, different letter shapes could be imagined to be overlaid on top of the corresponding keys. (See figure 2.1)

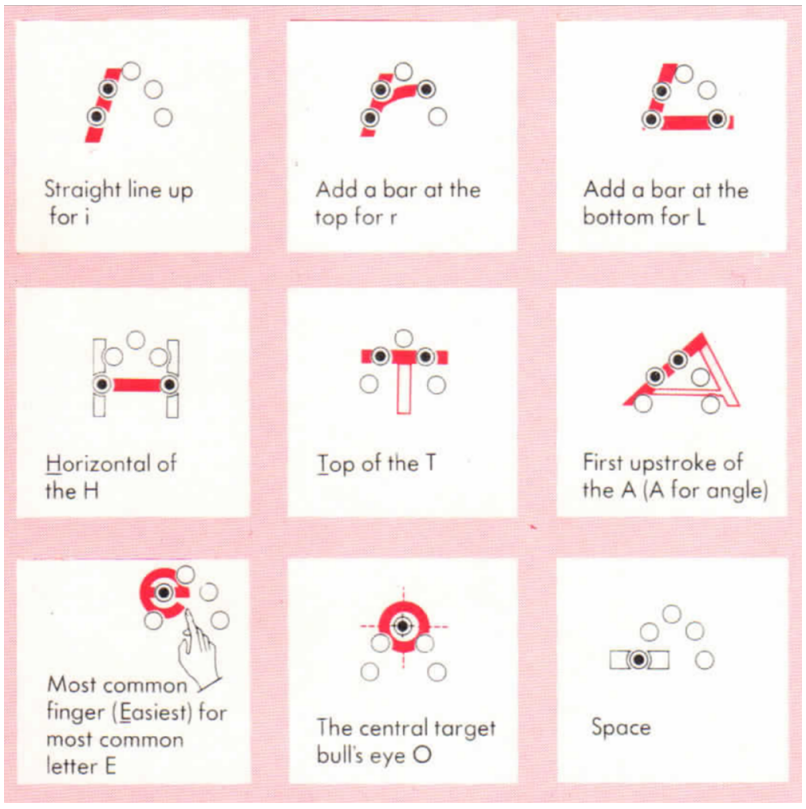

Figure 2.1. A sample of the letter mnemonics in the booklet *Microwriting in Practice*[3]

2.4 ASETNIOP

ASETNIOP is a chorded virtual keyboard built by Pointesa LLC. It has ten keys at fixed positions, meant to be used with all ten fingers simultaneously. The thumbs are used for space bar and shift while the rest are used to build chords. There are a total of 28 chords, with 18 for letters and 10 for commonly used punctuation marks. There is also an alternative layout that can be switched to on the fly to gain access to numbers and other symbols.

The name, ASETNIOP, represents the eight letters that can be typed without chords, assigned in order to the fingers from left to right. The placement corresponds to the fingers that would be used when normally typing on a regular QWERTY keyboard. These letters have been chosen to include the most frequently used characters in the English language. The developers claim that 65% of all keystrokes are accounted for without having to use any chords at all.

For the remainder of the 18 letters, two fingers chords are used with regards to letter frequency, fingers that would normally be used for that letter on a QWERTY keyboard, and chord difficulty. The purpose of designing ASETNIOP around the layout of the QWERTY keyboard is not only to make learning easier, but also to not affect the ability to type on a regular keyboard after learning to use ASETNIOP.[4]

On top of the aim of efficient letter placement, a set of other features are used to simplify the learning process, increasing the typing speed, and to reduce typing errors:

Disambiguation
A disambiguation scheme is used with aims to allow users to ignore the chording process when first getting started. As an example, if the user wanted to type the word *this* – requiring the letter H which is not included in the eight non-chorded letters – the word *tnis* could instead be typed and the system automatically replaces it with the correct word.

Stenographic Combinations
While all the letters of the alphabet are available with a maximum of two fingers presses together, three finger chords or more allow digraphs, trigraphs and sometimes entire words to be entered instantly. For example, pressing down the fingers for typing T, H and E at the same time, regardless of order, produces the word *the*.

Autocorrection
The system tries to autocorrect words that may have been mistyped as a consequence of mistakenly using a chord instead of two subsequent letters, or the other way around. Whenever the user inputs a word that cannot be found in a standard English dictionary, the system tries to replace pairs of single letters into the corresponding chord or chord letters into their component parts to see if a word is produced.

2.5 UpSense

UpSense by Inpris Ltd is another recently introduced touch keyboard replacement to be used on smartphones and tablets. In addition to using chords to form letters, swipes with one or more fingers in four different directions are also used.

The keyboard can be configured to be used with four, five or ten fingers. Every letter can be customized to the user's own preferred chord or gesture, but the defaults have the aim of being as intuitive as possible, with gestures that follow the shape of the letters.

As an example, the letters N and M are formed by swiping two and three fingers downward, respectively, while swiping up with two and three fingers forms the letter V and W. The swipes represent the downward tips in the letters N and M, and the upward tips of V and W. Many of the other letters follow this similar approach.

If the user ever forgets how to type a particular letter, the bottom of the screen has a row of all letters lined up. Pressing any of these reveal the gesture to form the letter, but does not actually type the letter out.

To distinguish between the different fingers of the user's hand, the input needs to first be calibrated by placing down all fingers on the screen. This places out rings on the screen that mark the touch area for each finger. When performing swipe gestures, the swipe needs to start inside the marked rings and moved outside before lifting the fingers.[5]

Chapter 3

Method

To explore the capabilities of alternative virtual keyboards and the possible difficulties in designing one, a relatively simple prototype was implemented for the project. The implementation, hereby also referred to as *TactionType*, runs in a web browser and uses chords on five touch keys to produce the letters of the English alphabet.

3.1 Scope

Because of the expected scope of this project, the prototype has certain limitations:

- The prototype runs in a limited environment, with the aim of demonstrating the design rather than running as a deeply integrated process on the system. This means that the prototype cannot be used everywhere on the operating system as a replacement to the regular touch keyboard.
- Only English letters are available, and a very limited set of non-letter characters are assigned. When designing for efficiency, only the character frequency of the English language is considered.
- The prototype is specifically designed towards touch screens with enough room to comfortably contain all five fingers of one hand, which in general means tablet devices.

3.2 Technologies

The prototype is built using web technologies – HTML, CSS and JavaScript[1] – and therefore runs inside the device's web browser. Developing the prototype in this environment was a natural choice for the author due to his previous experience and

[1] In reality, the implementation uses LESS and CoffeeScript instead of CSS and JavaScript. Both of these provide syntactic sugar and mechanisms for a more pleasurable working environment, and ultimately compile down to their plain counterparts.

level of comfort in working with web technologies, but it also provided set of other benefits:

- The amount in friction of setting up and letting other users test the prototype is greatly reduced. With the server running, all the user needs to do is to connect to the server IP in the device's web browser, which can be done on any supported device without any installations or pre-configurations.
- The web browser provides a set of APIs for easily handling touch input on the device, reducing the amount of work in building surrounding systems.
- The use of *WebSockets* allows the input from the touch device to be synced to other connected devices, setting up a great environment for testing and demonstration purposes.

The back-end is powered by *Node.js*. As most of the application logic lies in the front-end, its main tasks is only to serve files and direct the WebSocket communication between clients.

The prototype was mainly tested on a third generation iPad, but it should be compatible on any tablet device running a web browser implementing the required set of HTML5 features.

3.3 The Application

Here follows a run through of the different aspects and features of TactionType. Figure 3.1 should also give a good overview of how the application looks in its running state.

3.3.1 Touch Keys

The interaction starts with the user placing their five fingers on the touch screen, which places out the five touch keys for future use. Should the user at any later point feel discontent with the position of the keys, holding down all five fingers again for one second allows the placement to be re-calibrated.

The design decision of letting the user place out the keys themselves borrows from that of UpSense. This allows for a more comfortable hand placement once typing, instead of being restricted to a set position as is the case with ASETNIOP.

The touch key areas are each marked with a uniquely colored circle. However, the keys are not absolute hit points and only serve as guidance, as any touch made outside of these is automatically assigned to the nearest unpressed key. As an example, this allows the user to press down with all five fingers anywhere on the screen, even outside any of the keys, and the system still registers that as all five keys being pressed down simultaneously.

To group touches together as chords, a small grace period is needed between the time a touch is registered and a letter is typed. This grace period is reset between

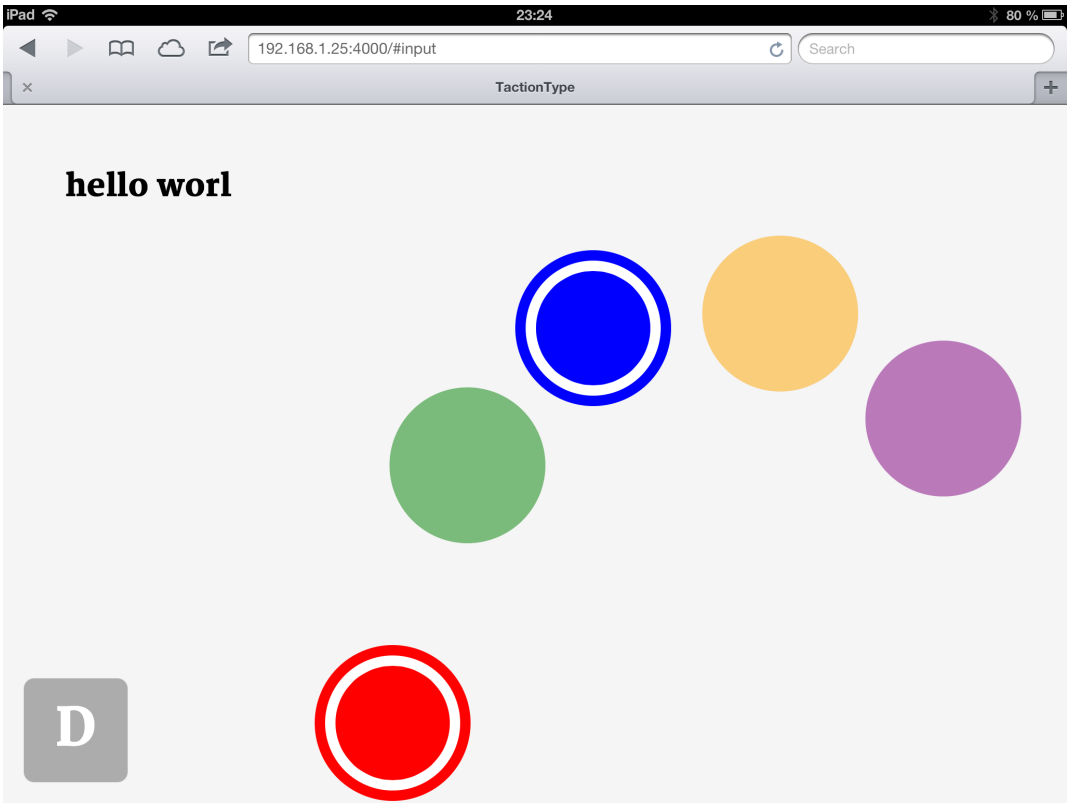

Figure 3.1. A screenshot of TactionType, with the first and third button highlighted – the chord for producing the letter D.

each new registered touch. In essence, this means that a set of keys being pressed in a quick succession are registered as a single chord, rather than separate key presses. A value between 50 and 100 milliseconds for the length of the grace period usually produces the best results.

3.3.2 Cross-client Communication

By utilizing WebSockets and a publish-subscribe messaging pattern, communication and syncing between different devices connected to the web page can be managed with a relatively small amount of code.

Once a touch device has registered as the input system, other connected clients, regardless of having a touch screen or not, can display all the touch actions and typed letters from the input device.

This setup provides a great environment for testing and demonstrating the virtual keyboard with other users. It also allows the application to be user tested remotely, without sacrificing the test supervisor's ability to see what the user is doing in real-time.

3.3.3 Chord Highlighting

To help the user with remembering the chord for each letter, a system of highlighting chords for corresponding letters is available. By connecting to the web page with a computer with a physical keyboard, the user or test supervisor can press any of the assigned letters on the computer keyboard to highlight the chord keys for that particular letter. (This is seen in action on figure 3.1)

This method of chord guidance is clearly contrived for this specific prototype environment, but a real-world version could be designed similarly to the strip of letters at the bottom of the screen given on the UpSense system. By providing a guidance system that does not type out any characters itself, the user is encouraged to eventually memorize all of the chords.

3.4 Character Mapping

A big challenge in designing a character mapping scheme is finding a good balance between memorability and efficiency. Let us imagine a system that reads input by letting the user write the letters in handwritten form on the screen. This would be very intuitive and memorable since the user presumably already knows how to write by hand, but the level of efficiency would be very low since large gestures are required for each letter input. The gesture complexity is also not in any way related to how often the letter is typed.

An efficient system would on the other hand map frequently used letters to simple commands that are quick to execute, while assigning the less used letters to more complicated inputs. The drawback of this system is that the letter gestures would be more difficult to remember, because they are not based on something that has already been taught to the user.

3.4.1 Observations

Before we get to the character mapping of TactionType, let us analyze the input systems from the *Background* section.

NLS

Douglas Engelbart's binary mapping scheme is not specifically designed with efficiency in mind. The mapping is easy to understand to anyone acquainted to the binary numeral system, which many of the people within the target group of NLS would likely have been.

However, understanding the mapping system does not necessarily make it easier for users to quickly remember the chord for a specific letter in the midst of typing. Apart from the first and possibly last couple of letters in the alphabet, quickly knowing the exact positional order of a letter in the middle of the alphabet is not something that most people are capable of without any training, and even less when the number needs to be in binary form.

Microwriter

The Microwriter featured a blend between memorable and efficient letter mapping. The letter E, which is the most commonly used letter in the English language, was mapped to a single press with the index finger, an efficient mapping.

The mnemonics served as an aid in learning the chords, even though many of them could often feel contrived or far-fetched. As an example, the mnemonic "most common finger for most common letter E" does not make much sense – are not all fingers equally common?

However, the point of mnemonics is to aid the brain in translating information into a more retainable form, regardless of its silliness. Even if the user were to take offense at the lacking sensibility of the mnemonic above, then it has already served its purpose in making that particular letter mapping memorable.

ASETNIOP

While ASETNIOP is mostly designed around typing efficiency, the mapping of the 8 single finger letters are based on a regular QWERTY keyboard, making it more intuitive and memorable for an experienced computer user. The name of the system itself also serves as a mnemonic for remembering the order of these 8 letters. When combined with the disambiguation system, the user is able to start typing a majority of words without needing to learn any chords.

UpSense

UpSense bases its character mapping on an intuitive and memorable system. Once the user learns that a downwards swipe with three fingers produces the letter M, it is easy to see that the swiping up with the same fingers produces W. Even though the use of swipes in addition to chords gives a greater freedom in assigning gestures to letters according to their shapes, the shape tracing approach does not work with all letters of the alphabet.

The creators of UpSense do not seem to consider typing efficiency in their default mapping. As an example, the most common letter E is mapped to a swipe to the right with three fingers, which is far from being as efficiently performed as pressing a single key.

3.4.2 The TactionType Mapping

When mapping out the characters to chords, the frequency of the letter in the English language was used as a big factor in determining the number of keys to be used in the chord. A chord with a lesser number of keys should map to a more frequently used letter. All of the chords and their corresponding letters can be seen in figure 3.2.

In addition to using the usage frequency of letters in the mapping, the letter shapes are also taken into consideration. Similar to the design of UpSense, mapping

Character	Chord
A	1
B	1-3-4
C	2-5
D	1-3
E	2
F	1-4-5
G	1-2-4
H	2-4
I	5
J	1-2-5
K	2-3-5
L	1-4
M	2-3-4
N	3-4

Character	Chord
O	4
P	2-4-5
Q	1-3-4-5
R	2-3
S	1-2
T	3
U	3-5
V	4-5
W	3-4-5
X	1-3-5
Y	1-5
Z	1-2-4-5
BACKSPACE	1-2-3
SPACE	1-2-3-4-5

Figure 3.2. A table of all the mapped characters. The chord lists the keys numbered by their position from left to right.

chords in relation to the shape of the letter helps in making the system more intuitive and easier to remember for the user.

Peter Norvig, Director of Research at Google Inc, used data based on the English books that have been scanned by Google to produce the following ordered sequence of letters based on their frequency: *ETAOIN SRHLDCU MFPGWYB VKXJQZ.*[6]

The five most frequent letters E, T, A, O and I are all mapped to single keys. The ten two-key chords are mostly mapped to the following ten most frequent letters, with some exceptions to favor memorability over typing efficiency.

The only letters mapped to four-key chords are Q and Z, the two most infrequently used letters in the English language.

While not all chords have any greater significance behind their mapping, there are some notable ones:

- N and V map to two fingers, representing the two tips of the letters. Their three-tipped counterparts, M and W, are naturally mapped to three fingers by adding a finger to the left of the chord for N and V.

- The American Sign Language has signs for each letter in the alphabet. The letter I is signed by holding up the little finger, and Y is signed by holding up the thumb and little finger. This is reflected in their chord mapping. While the user cannot be assumed to know American Sign Language, informing the

user of the significance behind these letter mappings will serve as a mnemonic for remembering them.

- The shapes of the letters J and G both have a low center of mass, while P and F have their center of mass at the top. This is used in their chords, if the fingers are imagined to be positioned vertically, with the thumb at the bottom.

- The backspace key uses the three leftmost keys, to resemble going back to the left when deleting characters.

3.5 User Testing

The prototype was user tested three participants with varying levels of training with the system. While the prototype should be trained and tested with users over a long time period to truly measure its potential, the expected scope of this project has been a limiting factor in regards to the extent of the user testing. That said, the relatively limited user testing has still been a valuable resource for identifying drawbacks and strengths of the design.

To measure the user's typing efficiency on the system, the elapsed time of typing two separate sentences were used as benchmarks. The first of these sentences is the famous English pangram[2] "The quick brown fox jumps over the lazy dog". This sentence elegantly measures the user's ability and efficiency in finding the chords for all the letters of the alphabet. This sentence would also be used in training for memorizing all the chords.

The second of the benchmarking sentences is a nonsensical statement consisting only of letters producible by using chords with two keys at most – "The salty coastlines to the north are trendy". This sentence was used to more accurately portray the way a user would regularly type in a real-world scenario, because of the correlation between the number of keys in a chord and the letter's frequency of appearance in the English language. Based on the data gathered by Peter Norvig[6], the set of letters producible by single keys or two-key chords cover approximately 85% of the combined total of letters used in a body of regular English text, on average.

Users were allowed to use the chord guidance mechanisms during the tests, but only as a last resort if they were unable to remember the chord for the current letter. Both sentences were typed and timed a number of times, out of which the average speed and best attempts were recorded. The users were then asked to type the same sentences with the device's standard QWERTY keyboard, once again recording the average speed and the best entry.

When measuring the typing speed on the device QWERTY keyboard, auto-correction had been turned off. While this may be a feature the user has turned

[2] A pangram is a sentence using every letter of the alphabet at least once.

on when typing regularly on the device, it is not a feature that is specific to the design of that input system. An auto-correction mechanism could just as well be implemented with TactionType, without altering the design of the system itself.

Chapter 4

Analysis

This chapter is dedicated to analysis of the insights gained from building the prototype and from the user feedback. Both data and observations are discussed to build a basis from which conclusions can be drawn in the final chapter.

4.1 Usability

An aspect that is difficult to quantify, but nonetheless very important, is the general usability of the prototype. While the topics of memorability and efficiency are a part of application usability, these two will be covered in greater depth in later sections. What this section aims to focus at is more specifically the interaction process of pressing the keys of TactionType.

4.1.1 Key Calibration

As described in the previous chapter, the interaction starts with the user placing down all five fingers to set the positions of the keys. This allowed the users to easily find a comfortable hand position for using the keyboard.

This follows the design of the UpSense keyboard to a certain extent, however avoiding a core issue that the author found with it. The author would often miss one or more of the UpSense keys when typing, leading to a completely different character or no character at all being produced. By automatically assigning a touch to the nearest unpressed key on the TactionType keyboard, typing errors caused by missed keys were less likely to occur.

Of course, the system could still assign a touch to the unintended key. However, the size of the key markers encouraged the user to leave a good amount of space between each finger, reducing the risk of pressing the wrong keys.

With the five keys placed where the user's fingers would naturally touch when pressing down, the user is much more likely to focus their sight on the written text rather than the keys themselves. By constantly looking at the written text, typing errors are caught and fixed more quickly by the user. This is something

that an experienced computer user would beneficially do when typing on a physical keyboard, but often struggle with on regular touch keyboards. Auto-correction on touch keyboards often helps in fixing minor typing errors without requiring any action to be taken by the user, but the auto-correction system can sometimes produce unintended results that completely change the meaning of the written sentences.

4.1.2 Pressing Chords

Because of the improbability of the user being able to press down all keys in a chord at exactly the same time, an added grace period between touches and a chords being registered is used. The length of the grace period proved to be a difficult value to calibrate for every user – it needs to be short enough to distinguish between separate letters being typed in a quick succession, but not so short that typing chords with multiple keys becomes too difficult. While occurrence of the described issues certainly were minimized with proper grace period calibration, more work would be required to reduce these errors to a negligible level.

One alternative would be to register chords once the user lifts all fingers off the screen, but this also limits the ability to fluidly type chords in a quick succession, since all fingers need to be removed from the screen before starting on the next chord.

4.1.3 User Satisfaction

One aspect that is often discussed in regards to usability is the satisfaction that the user feels when using the application. Depending on the user's hand dexterity, certain chords could feel very cumbersome or awkward to type. Extended periods of use could also feel straining, more so than using the regular keyboard on the device. This is most likely the cause of the user not being used to performing these types of gestures in a repeated fashion, something that the user could adjust to over time.

Despite the described discomforts associated with using TactionType, users would commonly describe the virtual keyboard of being "fun" to use. A certain level of satisfaction and accomplishment could be felt once the user was beginning to learn the chords, something that can be compared to the feelings experienced when learning to play a new instrument. While the enjoyment gained in using a text input system might not be the first thing a user looks for, it is nonetheless an interesting aspect to take note of.

4.2 Chord Memorability

As discussed in the previous chapter, finding a good blend between efficient and memorable character mapping is a tough task. TactionType focuses primarily on mapping the letters to efficient chords rather than memorable ones. However, the few mnemonics that are used proved to be a good help for the user in remembering

those corresponding chords. The chords that do not carry any significance, such as the chord for the letter D (thumb and middle finger), would be the ones that users struggled the most with.

When the user would forget a specific chord, the chord highlighting system quickly helped in refreshing the user's memory. However, the user would often within a short duration forget the chord again, having to look it up once more. It is probable that a more extensive use of mnemonics could help alleviate this issue. These mnemonics could then be displayed together with the highlighted keys when the user forgets a chord, to more efficiently educate the user.

It would take the average user around one to two hours to learn and memorize all the chords, a time period very similar to the reported learning time of the NLS keyset. This might not seem to be a very long time to learn a completely new system, but it does cause a friction for new users that should preferably be reduced to its minimum. However, such friction reductions should not be at the cost of efficiency or other usability aspects.

Once the user has memorized all the chords, there is still a lot of room to improve the user's efficiency with the system. These improvements can only be achieved through extensive use, by allowing the chords to be put into muscle memory and improving hand dexterity to be able to more elegantly form the different gestures. Because of the limited project scope, the long term aspects of TactionType could unfortunately not be analyzed.

4.3 Typing Speeds

Approximate Training Time

User A	1 hour
User B	1-2 hours
User C	2-3 hours
Author	4-5 hours

Figure 4.1. The approximate time each user had spent using TactionType before the typing speed tests were performed.

When measuring users' typing speeds on TactionType compared to the device's standard QWERTY keyboard, both an average and best entry were recorded. The best entries serve as frame of reference for the potential attainable typing speeds. The results from the first sentence, seen in figure 4.2, show large speed differences between different users on TactionType, with more consistent results on the QWERTY touch keyboard. While the amount of training with TactionType varied

"The quick brown fox jumps over the lazy dog"

TactionType

User	Average	Best
User A	95s	80s
User B	90s	76s
User C	40s	33s
Author	35s	29s

Device QWERTY Keyboard

User	Average	Best
User A	16s	12s
User B	15s	11s
User C	10s	8s
Author	12s	10s

Figure 4.2. The average and best typing speeds for the sentence "The quick brown fox..." on TactionType and the device's standard QWERTY keyboard.

greatly between users (see figure 4.1), everyone were already experienced with typing on QWERTY keyboards on touch devices.

The second sentence, with results displayed in figure 4.3, is a more accurate representation of the letters that a user would more commonly type when writing regular English text. The results show clear speed improvements for most users compared to results in figure 4.2, something that is expected based on the simpler chords that this sentence consists of. It is apparent that the typing speeds of the two sentences on the device's QWERTY keyboard do not vary much at all, the set of used letters does not change the attainable typing speeds to any considerable extent.

"The salty coastlines to the north are trendy"

TactionType

User	Average	Best
User A	100s	85s
User B	55s	40s
User C	28s	23s
Author	25s	20s

Device QWERTY Keyboard

User	Average	Best
User A	15s	11s
User B	14s	11s
User C	10s	8s
Author	12s	9s

Figure 4.3. The average and best typing speeds for the sentence "The salty coastlines..." on TactionType and the device's standard QWERTY keyboard.

From these results, a potential TactionType typing speed ranging between 30-50% of speeds on the QWERTY touch keyboard can be observed. This converts to a range of around 20-25 WPM[1] on TactionType, and 35-50 WPM on the standard

[1]Words per minute, a typing measurement where every five characters are counted as one word.

device keyboard.

This does show a great potential, considering the relatively small amount of training with TactionType compared to the users' many years of experience with QWERTY keyboards. An important aspect to remember is the also fact that TactionType only uses one hand for text input, while the device's QWERTY keyboard is built for use with two hands.

Chapter 5

Conclusions

Going back to the introduction, the question that this thesis seeks to answer is whether a viable touch keyboard alternative can be designed, one that is not based on the computer keyboard and has the potential of being more efficient. It is apparent that UpSense, ASETNIOP and TactionType all build on design paradigms that are far separated from the design of the physical keyboard. The three alternatives are designed to not require precise finger positioning to hit the correct keys on the keyboard. This makes it easier for the user to focus their sight on the written text rather than their fingers, which is crucial in being able to quickly adjust potential typing errors.

It is difficult to objectively determine the viability of a text input system. The viability of the keyboard design is very dependent on the dedication of the user, but this is also true for inexperienced users learning to use a QWERTY keyboard. Having all the chords memorized, a feat usually achieved within two hours of use, allows the user to successfully type on TactionType at a steady speed. Although the user's typing speed at that point might be much slower than the speed on the device's QWERTY keyboard, it should be considered a viable keyboard alternative.

From the results seen from measuring the crude TactionType prototype, potential typing speeds reaching up to a half of the speed on the device's QWERTY keyboard can be observed. These speeds were achieved relatively quickly, and even greater speeds should be attainable with further training. This shows good potential in allowing users to become efficient with the system, especially considering that the system only utilizes one hand.

The design of TactionType may not be the optimal one within the area of alternative keyboard designs, but it displays what can be achieved with a relatively limited amount of work and research. The UpSense and ASETNIOP keyboards show that there is an interest in exploring this area, and the author is hopeful of seeing more virtual keyboards with alternative designs entering the market in the future.

Bibliography

[1] Bill Buxton, Feb 2012, *Haptic Input: Chord Keyboards*
http://www.billbuxton.com/input06.ChordKeyboards.pdf
Downloaded 2013-02-10

[2] Doug Engelbart Institute, 2008, *"Father of the Keyset"*
http://www.dougengelbart.org/firsts/keyset.html
Downloaded 2013-02-10

[3] Microwriter Ltd, 1980, *Microwriting in Practice*
http://research.microsoft.com/en-us/um/people/bibuxton/buxtoncollection/a/pdf/Microwriting%20in%20Practice.pdf
Downloaded 2013-02-10

[4] Pointesa LLC, 2012, *About ASETNIOP*
http://www.asetniop.com/about.html
Downloaded 2013-02-10

[5] Inpris Ltd, July 2012, *UpSense User Guide*
http://www.inprisltd.com/media/UpSenseGuide-English.pdf
Downloaded 2013-02-10

[6] Peter Norvig, Jan 2013, *English Letter Frequency Counts: Mayzner Revisited or ETAOIN SRHLDCU*
http://norvig.com/mayzner.html
Downloaded 2013-02-15

Appendix A

Source Code

Here follows the relevant parts of the TactionType source code. The source code in its entirety is also available at https://github.com/JoelBesada/taction-type.

A.1 Server-side Code

A.1.1 app.coffee

```
###
# Express Bootstrap
###
express = require "express"
async = require "async"

app = express()

require("./config/environments").init app, express

webServer = require("./webserver").init app
webSocketServer = require("./websocketserver").init webServer
```

A.1. The server bootstrap code.

A.1.2 environments.coffee

```
###
# Environment Configuration
###
exports.init = (app, express) ->

  # General
  app.configure ->
    app.use express.logger "dev"
    app.use express.cookieParser()
    app.use express.methodOverride()
    app.use app.router
    app.use require("connect-assets")()
    app.use express.static("#{__dirname}/../public")
```

```
    app.set "port", process.env.PORT or 3000
    app.set "views", "#{__dirname}/../views"
    app.set "view engine", "ejs"

  # Development
  app.configure "development", ->
    app.use express.errorHandler()
    app.set "port", process.env.PORT or 4000
```

A.2. The server environment settings.

A.1.3 webserver.coffee

```
###
# Web Server Setup
###
http = require "http"

exports.init = (app) ->
  http.createServer(app).listen app.get("port"), ->
    console.log "Express server listening on port #{app.get "port"}"

  app.get "/", (req, res) ->
    res.render "index", port: req.app.get "port"
```

A.3. The web server setup.

A.1.4 websocketserver.coffee

```
###
# WebSocket Server Setup
# WebSockets are used within the application to communicate with
# other connected devices. Only one input device can be active
# at a time, all other devices act as "audience" devices.
#
# Communication between devices is handled by sending a JSON
# object with an event and data field. These events will then
# be triggered on all receiving applications, like any other
# JavaScript event.
###
_ = require "underscore"
WebSocketServer = require("websocket").server
clients = {}
inputDeviceID = null

exports.init = (webServer) ->
  webSocketServer = new WebSocketServer
    httpServer: webServer
    autoAcceptConnections: false

  webSocketServer.on "request", (req) ->
      setupConnection req.accept(null, req.origin)

setupConnection = (connection) ->
  # Give a unique ID to every connected device
  ID = _.uniqueId()
```

```
  clients[ID] =
    connection: connection

  connection.on "message", (message) ->
    # Allow only one connected device be registered as the input
    if message.utf8Data is "INPUT_DEVICE" and inputDeviceID is null
      console.log "New input device set: #{ID}"
      inputDeviceID = ID

      # Trigger a a newinput event on all other connections
      for clientID, client of clients
        continue if clientID is ID
        client.connection.sendUTF JSON.stringify(event: "newinput")

      return

    # Forward all messages to other clients
    for clientID, client of clients
      client.connection.sendUTF(message.utf8Data) unless clientID is ID

  connection.on "close", (reasonCode, description) ->
    delete clients[ID]
    # Allow a new input device to connect
    inputDeviceID = null if ID is inputDeviceID
```

A.4. The WebSocket server setup.

A.2 Client-side Code

A.2.1 main.coffee

```
#= require_tree lib
#= require_tree src

# Extend the global TactionType object with a couple of methods
_.defaults TactionType,
  # jQueryfied version of TactionType, used to listen to and trigger events
  $: $ TactionType

  # Is this an input device?
  inputDevice: "ontouchstart" of window and window.location.hash is "#input"

  # Trigger an event across all connected devices through the WebSocket.
  # This also triggers the event locally.
  triggerSynced: (event, data) ->
    @connection.send JSON.stringify
      event: event
      data: data

    TactionType.$.trigger event, data

$ ->
  TactionType.connection =
    new WebSocket("ws://#{window.location.hostname}:#{TactionType.SOCKET_PORT}")

  # Trigger events for all incoming messages on the WebSocket connection
  TactionType.connection.addEventListener "message", (e) ->
    message = JSON.parse e.data
    TactionType.$.trigger(message.event, message.data)
```

```
  # The application is ready
  TactionType.$.trigger "ready"
```

A.5. The client bootstrap code.

A.2.2 app.coffee

```
$ ->
  # Refresh when input devices refreshes
  TactionType.$.on "newinput", -> window.location.reload()

  # Send keyboard key press events to all devices
  $(document).on "keypress", (e) ->
    TactionType.triggerSynced "charpress", keyCode: e.which
```

A.6. The application (non-input device) code.

A.2.3 input.coffee

```
# Listen for touch events and send them out through the socket
setupTouchListeners = ->
  # Register with the server as an input device
  TactionType.connection.send "INPUT_DEVICE"

  $(document)
    .on("touchstart touchmove touchend touchcancel touchleave", (e) ->
      e.preventDefault()
      e.stopPropagation()
    )
    .on("touchstart", (e) ->
      trigger "touchstart",
        touches: formatTouches e.originalEvent.changedTouches
    )
    .on("touchmove", (e) ->
      # Throttle the events for a consistent update rate
      triggerThrottled "touchmove",
        touches: formatTouches(e.originalEvent.touches, true)
    )
    .on("touchend touchcancel touchleave", (e) ->
      trigger "touchend",
        touches: formatTouches e.originalEvent.changedTouches
    )

# Local shorthand for triggering a synced event
trigger = -> TactionType.triggerSynced.apply(TactionType, arguments)

# A throttled version of trigger
triggerThrottled = _.throttle(trigger, 10)

# Pick out the info we are interested in from the list of touches
formatTouches = (touches, move) ->
  list = []
  pressedNow = {}
  for touch in touches
    formatted =
      id: touch.identifier
```

```
      x: touch.pageX / document.width
      y: touch.pageY / document.height

    unless move
      key = determineKey touch, pressedNow
      pressedNow[key] = true
      formatted["key"] = key

    list.push formatted
  list

# Return the closest key for the given touch
determineKey = (touch, pressedNow) ->
  return null unless TactionType.TouchKey.calibrated
  key = $(touch.target).data("id")
  return key if key and not (TactionType.TouchKey.isPressed key or pressedNow key)
  x = touch.pageX
  y = touch.pageY

  availableKeys = _.filter TactionType.TouchKey.unpressedTouchKeys(), (touchKey) ->
    not pressedNow[touchKey.key]

  # Not actual distances, but good enough for finding the closest key
  distances = _.map availableKeys, (touchKey) ->
    key: touchKey.key
    distance: Math.abs(touchKey.x * document.width - touch.pageX) +
              Math.abs(touchKey.y * document.height - touch.pageY)

  _.min(distances, (item) -> item.distance)?.key

$ ->
  # This code should only run on the input device
  return unless TactionType.inputDevice

  TactionType.$.on "ready", ->
    $("body").addClass "input"
    TactionType.connection.onopen = setupTouchListeners
```

A.7. The input device code.

A.2.4 touchpoint.coffee

```
# Displays a single touch point on the screen
class TactionType.TouchPoint
  constructor: (@id, @x, @y) ->
    @$el = $("<div>").addClass("touch-point")
    $("body").append @$el
    @$el.addClass "show"
    @move x, y

  move: (x, y) ->
    @x = x
    @y = y
    @$el.css
      "webkit-transform": "translateX(#{x * document.width}px)
                           translateY(#{y * document.height}px)
                           translateZ(0)"

  remove: ->
    @$el.removeClass "show"
```

```
    setTimeout( =>
      @$el.remove()
    , 250)

  @touchPoints = {}
  @touchCount = -> _.keys(@touchPoints).length

  @init: =>
    # Touch points are only rendered on "audience" devices, i.e.
    # not the touch device itself
    return if TactionType.inputDevice

    TactionType.$
      .on("touchstart", (e, data) =>
        for touch in data.touches
          @touchPoints[touch.id] = new @(touch.id, touch.x, touch.y)
      )
      .on("touchend", (e, data) =>
        for touch in data.touches
          @touchPoints[touch.id]?.remove()
          delete @touchPoints[touch.id]
      )
      .on("touchmove", (e, data) =>
        for touch in data.touches
          @touchPoints[touch.id]?.move touch.x, touch.y
      )

$ ->
  TactionType.$.on "ready", TactionType.TouchPoint.init
```

A.8. The code to render touches on connected devices.

A.2.5 touchkey.coffee

```
# The touchkeys visible on the input device
class TactionType.TouchKey
  # The interval time in ms for grouping touches together as chords
  KEY_GROUPING_INTERVAL = 120

  # The time in ms that the user should hold down all five
  # fingers on the screen before keys are recalibrated
  CALIBRATION_TRIGGER_TIME = 1000

  $charBox = null
  calibrationTimeout = null
  previousTouchKeys = null

  constructor: (@id, @x, @y) ->
    @$el = $("<div>").addClass("touch-key")
    $("body").append @$el
    @$el.addClass "show"
    @move x, y

  move: (x, y) ->
    @x = x
    @y = y
    @$el.css
      "webkit-transform": "translateX(#{x * document.width}px)
                           translateY(#{y * document.height}px)
                           translateZ(0)"
```

```
  remove: ->
    setTimeout @$el.remove, 500

  @calibrated: false
  @calibrating: false
  @touches: {}
  @touchKeys: {}
  @isPressed: (key) -> _presses[key]

  # Get all keys that are not currently pressed
  @unpressedTouchKeys: -> _.filter @touchKeys, (touchKey) ->
    not _presses[touchKey.key]

  @init: =>
    $charBox = $ ".char-box"
    TactionType.$
      .on("touchstart", (e, data) =>
        pressKeys data.touches
        @touches[touch.id] = touch for touch in data.touches

        if _.keys(@touches).length is 5
          if @calibrated or @calibrating
            calibrationTimeout = setTimeout startCalibration, CALIBRATION_TRIGGER_TIME
          else
            startCalibration() if _.keys(@touches).length is 5
      )
      .on("touchend", (e, data) =>
        for touch in data.touches
          key = @touches[touch.id].key
          $key(key).removeClass("pressed") if key
          delete @touches[touch.id]

        clearTimeout calibrationTimeout
        endCalibration() if @calibrating
      )
      .on("touchmove", (e, data) =>
        return unless @calibrating
        @touchKeys[touch.id]?.move touch.x, touch.y for touch in data.touches
      )
      .on("charpress", showCharacter)
      .on("keypressed", hideCharacter)

  # Calibrate/place out the touch keys
  startCalibration = =>
    # Remove the unintented space that was added on re-calibration
    TactionType.KeyHandler.backspace()

    @calibrating = true
    previousTouchKeys = @touchKeys
    touchKey.$el.hide() for id, touchKey of previousTouchKeys

    @touchKeys = {}
    for id, touch of @touches
      @touchKeys[id] = new @(id, touch.x, touch.y)

  # Lay down the keys and give them identifiers based on their positions
  # from left to right
  endCalibration = =>
    @calibrating = false
    @calibrated = true
    touchKey.remove() for id, touchKey of previousTouchKeys
```

```
    touchKeyList = _.sortBy((touch for id, touch of @touchKeys), "x")
    touchKey.$el.attr("data-id", touchKey.key = i + 1) for touchKey, i in touchKeyList

  _presses = {}

  # Keypresses within small intervals of each other
  # are grouped together as one chord
  _pressKeys = _.debounce( =>
    keys = _.keys(_presses).sort()
    _presses = {}
    $key(key).addClass "pressed" for key in keys
    _.defer =>
      $(".pressed").each (index, element) =>
        unless _.findWhere(@touches, key: $(element).data("id"))
          $(element).removeClass("pressed")

    if TactionType.inputDevice
      TactionType.triggerSynced "keypressed", keys: keys

  , KEY_GROUPING_INTERVAL)

  pressKeys = (touches) ->
    for id, touch of touches
      _presses[touch.key] = true if touch.key
    _pressKeys()

  # Retrieve the jQuery element for a given key
  $key = (key) -> $(".touch-key[data-id=\"#{key}\"]")

  # Show the guide highlight for a key pressed on a remote keyboard
  showCharacter = (e, data) ->
    char = String.fromCharCode(data.keyCode).toUpperCase()
    keys = TactionType.KeyDefinitions.lookup[char]
    return unless keys
    hideCharacter()

    for key in keys.split("-")
      $key(key).addClass "highlight"

    $charBox.text(char).addClass("show")

  # Hide the character highlight
  hideCharacter = ->
    $(".touch-key.highlight").removeClass("highlight")
    $charBox.removeClass("show")

$ ->
  TactionType.$.on "ready", TactionType.TouchKey.init
```

A.9. Code for handling the touch keys on the input device.

A.2.6 keyhandler.coffee

```
# Handle and render the characters of pressed keys
class TactionType.KeyHandler
  $textArea = null

  # Interpret and render key presses as characters
  handleKeyPress = (e, data) =>
    char = TactionType.KeyDefinitions.presses[data.keys.join("-")]
```

```
    return unless char
    if char is "BACKSPACE"
     @backspace()
    else
      $textArea.append(char) if char

  # Remove the last letter
  @backspace = ->
   $textArea.text $textArea.text().substring(0, $textArea.text().length - 1)

  @init = ->
    TactionType.$.on "keypressed", handleKeyPress
    $textArea = $ ".text-area"

$ -> TactionType.$.on "ready", TactionType.KeyHandler.init
```

A.10. Code for rendering pressed characters.

A.2.7 keydefinitions.coffee

```
# Definitions of all keys.
# Fingers:
#   1: Thumb
#   2: Index finger
#   3: Middle finger
#   4: Ring finger
#   5: Little finger

class TactionType.KeyDefinitions
  @presses:
    "1": "A"
    "2": "E"
    "3": "T"
    "4": "O"
    "5": "I"

    "1-2": "S"
    "2-3": "R"
    "3-4": "N"
    "4-5": "V"
    "1-3": "D"
    "2-4": "H"
    "3-5": "U"
    "1-4": "L"
    "2-5": "C"
    "1-5": "Y"

    "1-2-3": "BACKSPACE"
    "2-3-4": "M"
    "3-4-5": "W"
    "1-2-4": "G"
    "2-3-5": "K"
    "1-3-4": "B"
    "2-4-5": "P"
    "1-2-5": "J"
    "1-4-5": "F"
    "1-3-5": "X"

    # "1-2-3-4":
    # "2-3-4-5":
```

```
    "1-3-4-5": "Q"
    "1-2-4-5": "Z"
    # "1-2-3-5":

    "1-2-3-4-5": " "

  @lookup: _.invert @presses
```

A.11. The chord to character mapping.