University of Glasgow | School of Computing Science

Text Prediction Visualizer for Android

Daniel Budeanu

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 27th, 2015

Abstract

With a total of over 2 billions estimated smartphone users for 2015 [8] it can be stated with enough certainty that these small mobile devices have become crucial to the way in which the modern world communicates. One crucial component that aids us in our everyday discussions is represented by the correction system. But even with the help of the various advancements that have been made in the field, there are still situations in which these helping mechanisms fall short of what the user expects. The aim of the project is to alienate this particular issue and provide a way through which a person can influence the decisions taken by the correction system and even overwrite them if needed.

Acknowledgements

I would like to thank Dr. Simon Rogers for supervising throughout the entire development process of the project as well as providing feedback on the drafts of this report. I would also like to thank Daryl Weir for offering guidance in the incipient phases of implementation.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name:		Signature:	

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Background 2

1.2.1 Types of touchscreens 2

1.2.2 Auto correction systems 3

1.3 Aim 4

1.4 Outline 4

2 Requirements **5**

2.1 Functional Requirements 5

2.2 Nonfunctional Requirements 6

2.3 Existing products and studies 6

2.3.1 PressureText: Pressure Input for Mobile Phone Text Entry [21] 6

2.3.2 Exploring Continuous Pressure Input for Mobile Phones [7] 8

2.3.3 TypeRight: a Keyboard with Tactile Error Prevention [19] 9

2.3.4 Uncertain Text Entry on Mobile Devices [27] 10

2.3.5 SwiftKey: The keyboard that learns from you [3] 10

2.3.6 Swype: Type fast, Swype faster [4] 11

2.3.7 Flesky Keyboard: The fastest, most customizable keyboard [1] 12

2.3.8 Minuum Keyboard: Type anywhere [2] 12

3 Design **14**

3.1 System Architecture 14

3.1.1 Main Activity 15

3.1.2 Input Method Service 15

3.2 Use Case 16

3.3 User Interface/General Appearance 17

3.4 User Feedback 19

4 Implementation **20**

4.1 Familiarisation with Android 20

4.2 Touch Coordinates Manipulation Experiment 21

4.3 Researching Soft Keyboards 22

4.3.1 CustomIME 22

4.3.2 TPVKeyboardView 24

4.3.3 TPVKeyboard 24

4.4 Label Alteration 24

4.5 Highlight Creation 25

4.5.1 Fat-finger Approach 25

4.6 Creating the Language Model 26

4.7 Creating the Touch Model 27

4.8 Connecting the Components 29

4.9 Highlighting based on Probabilistic Models 30

4.10 Using Touch Duration as an Offset Factor 31

5 Evaluation **34**

5.1 Usability/Feasibility Test 34

5.2 Survey Findings 34

6 Future Improvements **36**

6.1 Appearance 36

6.2 Performance 36

6.3 Usability 37

7 Conclusion **38**

7.1 Summary 38

7.2 Lessons learned 38

A Full size diagrams **42**

B Usability Test Feedback Survey **47**

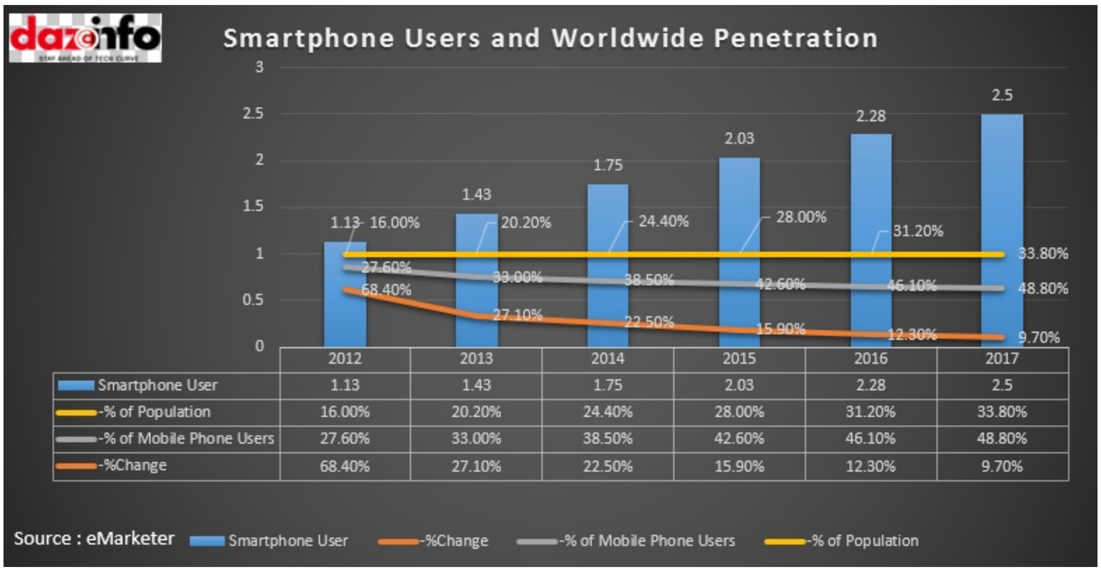
Chapter 1

Introduction

1.1 Motivation

Nowadays it seems to be the consensus that the majority of people are using a mobile device, and while there are still quite a few normal devices around, most of them are in fact smartphones. According to a study provided by the online website DazeInfo [8], in 2014, over 60% of the global population (4.55 billion) was using a mobile phone. Out of those mentioned previously, over 38% were actually smartphone users (Figure **??**).

When it comes to the market dedicated to these types of useful gadgets there are multiple manufacturers that come to mind, but as far as the underlying operating system that runs on them is concerned, the number of choices slims down considerably to the three main competitors: Android[28] (created by Google), iOS[32] (created by Apple) and Windows Phone[37] (created by Microsoft).



	2012	2013	2014	2015	2016	2017
Smartphone User	1.13	1.43	1.75	2.03	2.28	2.5
-% of Population	16.00%	20.20%	24.40%	28.00%	31.20%	33.80%
-% of Mobile Phone Users	27.60%	33.00%	38.50%	42.60%	46.10%	48.80%
-%Change	68.40%	27.10%	22.50%	15.90%	12.30%	9.70%

Figure 1.1: Figure which illustrates a graph showing the estimated increase of the smartphone user population over the span of 5 years

Every single one of those operating systems provides a way to input text through the means of having a virtual keyboard (software keyboard) or even a normal hardware keyboard (in the case of tablets). Since there is not that much feedback when it comes to typing on a glass like surface other than a small click like motion in the case of resistive touchscreens it is often hard to receive proper feedback in terms of where we are actually touching the

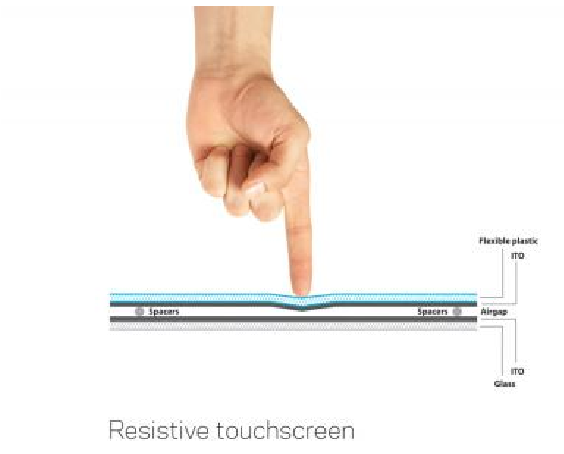


Figure 1.2: Structure of a resistive touch panel

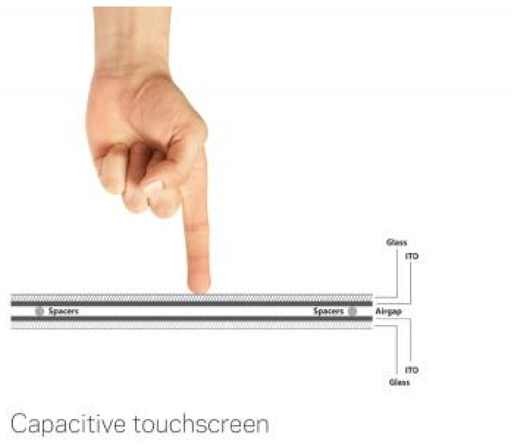


Figure 1.3: Structure of a capacitive touch panel

screen [6][17][22]. This is the main reason why auto correction background services have been implemented for these types of keyboards.

This brings us to the main reason behind the idea for this project. Auto correction systems are useful because they can be trained in order for the predictions to become more accurate for a specific user [3]. However, despite them improving over time there will still be moments when the recommended word/letter is not what the user actually wants to type (eg. alternating between two or more languages when typing, using regional vocabulary, using slang words etc.). This could in theory be overcome by implementing a way through which the input from the user would overwrite the prediction from the auto correction system.

1.2 Background

1.2.1 Types of touchscreens

There two main technologies that are still currently being used in producing the screens for smartphones: resistive[35] and capacitive[29] touchscreens (Figure 1.2 and Figure 1.3 respectively).

Even though the project itself does not focus on the importance of haptical feedback on mobile devices, it is important to notice how in the past, using a resistive touch panel meant that the user would receive some justification of the fact that a touch occurred due to its structure that would allow it to modify its shape given the flexible nature of the outer material.

However the drawback to the resistive screens was that, due to their nature, multi-finger gestures would not be possible to occur. As a natural result, necessity breeds innovation, hence the capacitive screens were invented in order to provide this type of functionality. Ever since, the majority of the manufacturers have decided to go for this option in order to provide useful functions to their users that the resistive screens being produced at the time could not due to them not being able to distinguish whether one finger or multiple were touching the surface at the same time (ie. pinch-to-zoom). Consequently this lead to an increase in importance for the correction systems since the user would not receive any haptic feedback from the device.

1.2.2 Auto correction systems

Having considered the way technology evolved over time, the user would now get to do more things with less effort on their smartphones but the physical feedback that the screens once provided was lost. This meant that auto correction became even more important and the need for it to be reliable was what drove innovation in this particular domain.

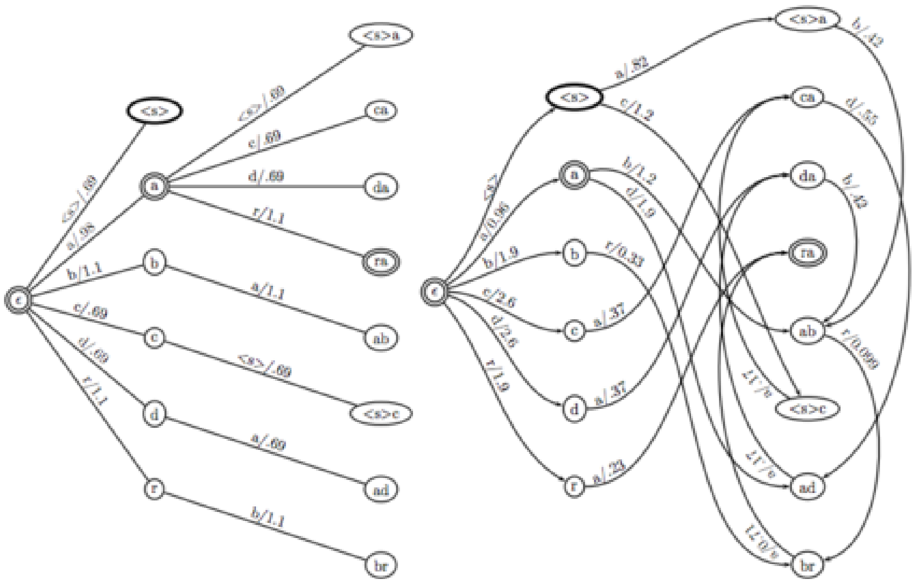


Figure 1.4: Figure which illustrates the tree structure of a generic language model

An auto correction system is usually composed out of two different components: a language model[33] and a touch model. Each of them provides a series of probabilities in regards to what the next input could be depending on: the touch position (for the latter) and the words or characters that were previously typed in (for the first case).

Language models (Figure 1.4) have experienced quite dramatic changes over the time but the same can be said about the existing touch models. If at the beginning they used to be based on predetermined dictionaries of words, with time more features began to be included such as user trained language models and even adaptive language models which can predict words from a number of different languages at the moment of typing (such as SwiftKey[3]).

However, correction errors will occur either due to user mistakes or simply because no matter how advanced the systems might be there will still be cases where the statistical solution is not necessarily the desired one. In order to correct this, the user should be given a way to see what the language model intends to predict at the time of pressing the key and have a mechanism available that would enable him to overwrite the choice determined by the system with the key that the user is pressing (assuming that the user is indeed pressing the correct keys).

When it comes to the above mentioned hypothesis, there are two solutions that come to mind in terms of the mechanism that could exist on the soft keyboards. One would involve detecting pressure in order to influence whether the user or the correction system should determine the next input, and the other involves touch duration. Since resistive touchscreens are no longer being used, the pressure option is not looking feasible, hence the only plausible solution being the touch duration.

1.3 Aim

The aim of this project are to develop an application that would illustrate whether or not it would be feasible to have such a mechanism in place that would show the user what the correction system has determined to be the most appropriate predicted result based on the probabilities supplied by the language model and touch model respectively. This should be done in a graphical manner and it will also enable the user to alter the result by using touch duration as the determinant factor.

In order to achieve this the following applications and services need to be implemented:

- a simple application needs to be provided in order to test the implemented solution
- a custom keyboard needs to be created that would enable us to modify its appearance based on the typing that is being performed on it
- a highlighting system that is adaptable to the duration of the touch on a given key

1.4 Outline

The remainder of this document will be divided into the following chapters:

Chapter 2 - **Requirements**

This chapter will discuss the requirements for the project as well as taking into account existing work that is closely related to the subject

Chapter 3 - **Design**

This chapter will discuss the decisions that went into creating the final product and the reasoning behind the appearance (UI) and functionality of it

Chapter 4 - **Implementation**

This chapter will elaborate the actual implementation of the project using the Android platform

Chapter 5 - **Testing**

This chapter will discuss the evaluation done on the project and the improvements made as a result of it

Chapter 6 - **Conclusions**

This chapter will summarise the report as well as describe ideas for future developments

Chapter 2

Requirements

The main focus for this chapter will be to document which were the crucial functional and nonfunctional requirements that needed to be met for the project and more importantly, to discuss how this project relates to other existing products. The last subject discussed within this chapter will be the issue of feasibility and why some of the approaches mentioned in the related systems section were not plausible implementations for Android (in particular, this meant using pressure as the determinant for the result of the keyboard input.

2.1 Functional Requirements

After performing a series of requirements gathering activities which involved the main stakeholders for the project, as well as having explored a series of research papers that were related to the subject in question (particularly the study published by Weir et al. [27]) and investigating the solutions that were implemented as a result, the following lists of requirements were produced in order to monitor the projects overall progress and to reflect the clients needs. The classification for the resulting requirements has been performed using the MoSCoW scheme.

FR1	Must Have	Allow highlighting of keys	The system should enable the user to see what both the language model and the touch model intend to predict as the next letter candidate.
FR2	Must Have	Has to be universally usable	Given the fact that the keyboard in Android is a service, this should work with any application that contains a text box which allows for text editing.
FR3	Must Have	Allow the user to alter letter that is predicted by the correction system	The system should enable the user to overwrite the letter provided by the correction system by pressing the key for a longer duration.
FR4	Must Have	Use bright red highlighting for highest probability	The key with the highest associated probability should be highlighted in red for it to be easily distinguishable.
FR6	Could Have	Allow different languages to be predicted	The application should allow the user to receive letter predictions for different languages provided that the appropriate LM / dictionary of words is provided.

2.2 Nonfunctional Requirements

This type of requirements are meant to describe how well the entire system performs its activities so it is mostly concerned with how well it performs and how maintainable and/or extensible it is. The priority classification scheme for the set of requirements has been MoSCoW[34] in this case as well.

NFR1	Must Have	Run on Android	The system need to be able to run on the Android operating system
NFR2	Must Have	It has to have a small footprint	The service itself should be used with any other application and not tied to a particular one
NFR3	Should Have	Allow for future extensions	The system itself needs to be extensible so that new functionality can be added without too much hassle.
NFR4	Should Have	It shouldnt obstruct typing	It has to be decently responsive so as to not make typing cumbersome for the user
NFR5	Should Have	It should support multiple screen sizes	The application should behave the same regardless of the orientation in which the device is being used or the size of the screen.
NFR6	Could Have	Highlighting should be consistent	The colour scheme used for the prediction provided by the system based on the LM and GP should be consistent with the one provided by the GP only when the user decides to overwrite the decision.
NFR7	Could Have	Display additional information	The system could display to the user, by the means of a text view, various information regarding the possible letter candidates and the touch location, in order to supplement the visual feedback provided by the keyboard.
NFR8	Would Like to Have	Provide different colour schemes	The user could possibly choose from different options of colour schemes in order to properly perceive the visual feedback in the case the where colour blindness is an issue.

2.3 Existing products and studies

There have been many attempts at trying to improve text input on mobile devices by making use of various factors (one of which was applying variable pressure on touch) in order to achieve different types of functionalities. In the following sections we are going to briefly discuss a series of studies and present some of their achievements.

2.3.1 PressureText: Pressure Input for Mobile Phone Text Entry [21]

This study, which was published by David C. McCallum in 2009 investigates how using a series of different pressure levels on pressure sensitive buttons from a mobile phone keyboard can possibly improve the rate at which an user can input text.

Given the fact that the normal mobile phone keyboard has the letters distributed in groups of three (excepting two digits, 7 and 9, which both contained four letters: the groups PQRS and WXYZ) the decision was taken to use three pressure levels in order to cycle through the letters that were afferent to each of the buttons. In order for the two exceptions to be satisfied, a certain workaround needed to be found since allowing for four levels of pressure for each of the other buttons as well would have been rather error prone.

In order to present the solution that was found in order to manage the two keys that presented groups of four letter we will have to first discuss the general approach for the other remaining buttons. The way this was implemented was by making usage of three pressure levels. In order for the user to pass from one level to another, there was a certain pressure threshold that needed to be overpassed before the system would record the fact that the next letter needs to be cycled through. So not only did pressure need to exceed the boundaries between any two pressure levels, but also it need to go over a certain threshold, which has been established through experimental means to be 10% of the previous pressure level.

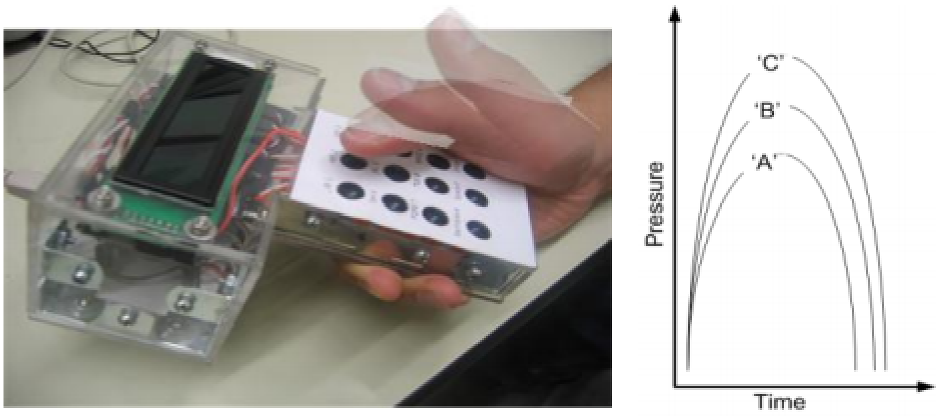


Figure 2.1: Figure showing the prototype used for the study as well as a graph which illustrates the different pressure levels [21]

Figure 2.2 shows a representation of how the pressure levels and their corresponding thresholds were distributed. The dark blue line represents the threshold limit, the light blue line represents the boundary between the different levels and the triangle from the side represents the current pressure value. In the case presented in the left side the system will record the fact that the first letter needs to be cycled through but not the second due to the fact that the current pressure value only exceeds the boundary between the levels but not the threshold established for the first level. On the other hand, in the case presented in the right side, the system will indeed proceed to cycle through the next letter found in the current group since the pressure value overpasses the threshold corresponding to the first level.

Now that the principles behind this study have been explained, we can further on elaborate on the solution to the four letter group exception case. Given the fact that adding another pressure level was not feasible, the workaround that was implemented was a 750 ms delay after successfully cycling through the third letter in the group. This way the user would perceive no noticeable difference in typing speed seeing as the value chosen was somewhat proportional to the amount of time it would take to pass from one pressure level to the other.

As a result of this study, it has been concluded that users who have been using the PressureText system for an extended amount of time were up to 33.6% faster than users who have just started using it. It has also been shown that expert users were 5% faster on average than using the already established multi-tap system of selecting letters on mobile phone keyboards.

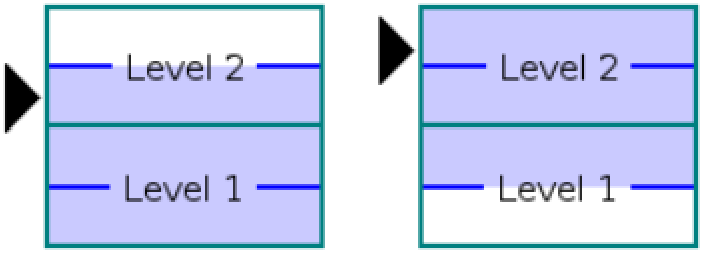


Figure 2.2: Figure illustrating pressure level boundaries and thresholds

2.3.2 Exploring Continuous Pressure Input for Mobile Phones [7]

For this particular study there have been many approaches to seeing how feasible it would be to introduce pressure input for mobile phone by inserting a series of capacitive sensors under each of the buttons of a standard keyboard for such devices.

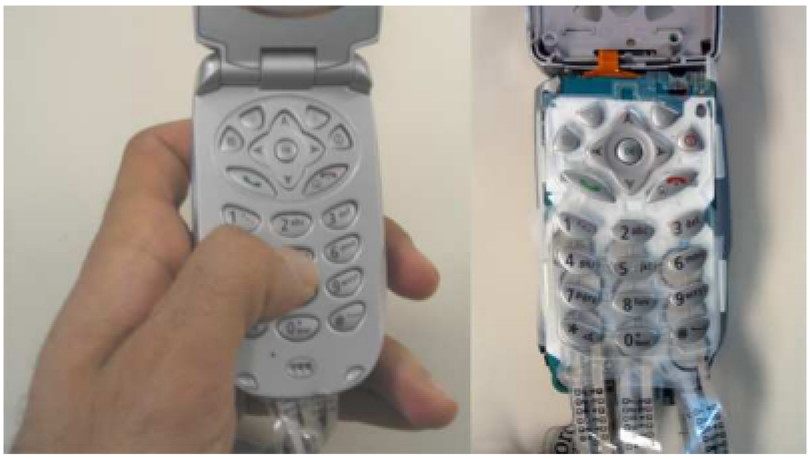


Figure 2.3: Figure illustrating how capacitive sensors were inserted under the buttons [7]

Most of the functionalities that were tested as a result of this study were based on using pressure levels as well (Figure 2.4), similar to the study previously mentioned by McCallum et al[21]. but to a lesser extent since the study took place before the first one and the pressure equipment used was not as precise and not as flexible in terms of calibration. To go into further detail, this meant that pressure was used in order to only distinguish between normal and hard presses. It has to be mentioned that there has been an attempt at cycling through the letters for a given button but it has been concluded that multi-tap was a faster process on the hardware that was created for the purpose of this research.

The first type of functionality that was tested and implemented using this approach was toggling between inserting the number associated with the button or the letter that the user was currently at in the multi-tap process associated with typing on a mobile phone keyboard. The second one was rather similar except the fact that the toggling was done between letter cases instead. Last but not least, and possibly the most interesting feature was the idea of marking the messages that were typed using hard presses as urgent. This could be applied in series of different circumstances. For instance, if we were to consider a hierarchy of message priorities in a system where only messages of a specific type would get displayed in the console, an administrator could easily communicate with other people connected to the system by hard-typing the messages while reverting to normal typing when it comes to resuming administrative tasks.

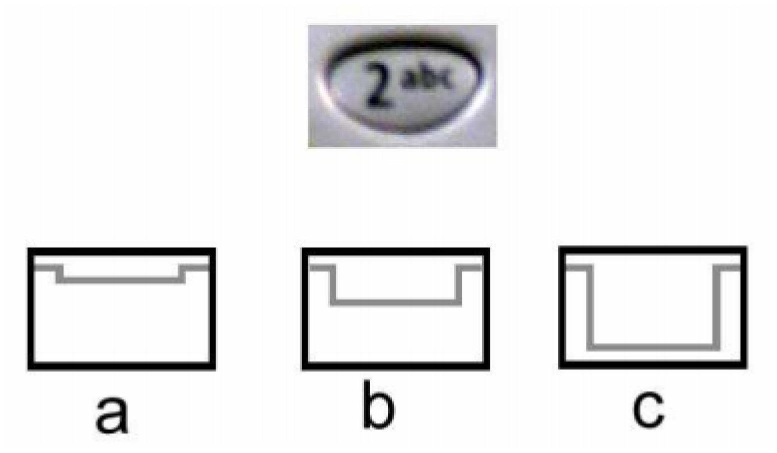


Figure 2.4: Figure displaying the different existing pressure levels

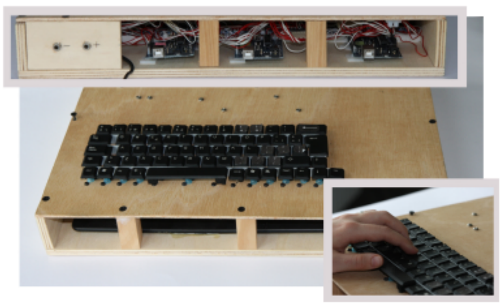


Figure 2.5: Prototype keyboard build for the purpose of the study [19]

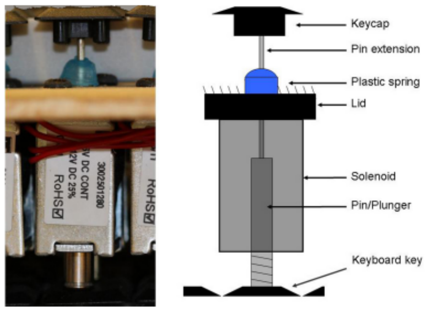


Figure 2.6: Cross-section through one of the keys on the TypeRight keyboard [19]

2.3.3 TypeRight: a Keyboard with Tactile Error Prevention [19]

This study adopts a rather original approach to using pressure as a determining factor in text input and error correction. At the basis of the study stands the concepts of a haptic keyboard. What the authors meant by this is the fact that a normal computer keyboard was modified so that it had resistive keys. What this would further enable users to do is to receive haptic feedback based on their typing or in simpler terms, every time a person would try to press a new key, the system would take advantage of the error correction software and assign a higher resistance to touch for the keys that would produce a typing mistake.

The study itself was rather successful. The main, long term, experiment that was performed involved having one member of the team test the system on two given pieces of text one which provided graphical feedback with the haptic system turned off and one which provided haptic feedback without and graphical indicators of the errors made in typing. After this first stage, the user then continued to use the system for everyday activities over the period of three months. Then after this training stage, the team member was asked to retake the initial test.

The results were rather conclusive, as for the first fragment of text which included graphical feedback, the completion time was 10% faster than first attempt. When it comes to the second fragment of text where haptic feedback was included, the difference was quite impressive since the user hasnt had any words that were not part of the correction softwares dictionary.

2.3.4 Uncertain Text Entry on Mobile Devices [27]

One of the most recent studies that is related to this subject in particular was the one published by Weir et al. In this study there are discussed two main approaches in terms of designing a touch model. The first model is based on the usage of a gaussian progress regression[31] (GPType) in order to improve typing accuracy on Android, and the second one is based on using pressure in order to explicitly control the uncertainty of the text that is currently being inserted (ForceType). The latter was implemented using a pressure plate as a means of determining the applied pressure. This solution was chosen purely because of the fact that there is no way to accurately determine pressure on modern day smartphones (since the touch panels that are used are capacitive).

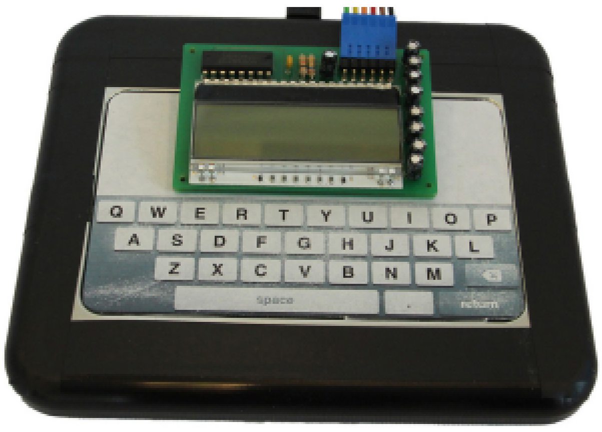


Figure 2.7: Figure showing the prototype built for the ForceType implementation [27]

The GPType approach takes advantage of the Gaussian Process regression for a given point in order to calculate a series of probabilities for all the keys based on the touch. This is then used in correlation with the probabilities provided by a language model in order to produce a suitable word or letter candidate for the text that is currently being composed. The system also makes good use of training data which was gathered by having users type a series of sentences using the custom keyboard that was created for the purpose of this study. This is then used to enhance the data provided by the GP and the LM and help to further calibrate it as a result.

The second approach (Figure 2.7), based on pressure sensitive input, takes the study one step further, incorporating the functionality present in the case of GPType and adding pressure detection as a factor that would help the user offset the bias that was currently leaning towards the results of the GPType system. What this actually means is that the user would get to influence whether the system decides the next letter or if this would be discarded and the key that was pressed would become the letter being added to the composed text.

2.3.5 SwiftKey: The keyboard that learns from you [3]

Given the fact that there are many custom types of keyboards available on the market, each an every one of them has to provide the potential users something that sets them apart from the other competitors in this rapidly increasing area.

What SwiftKey bring through the table is a very established design that makes it easy to type on it due to its familiarity to the normal Android user, but what makes it actually special is it's ability to provide increasingly better accuracy on its predictions thanks to its adaptive language model that will adopt new words as the user

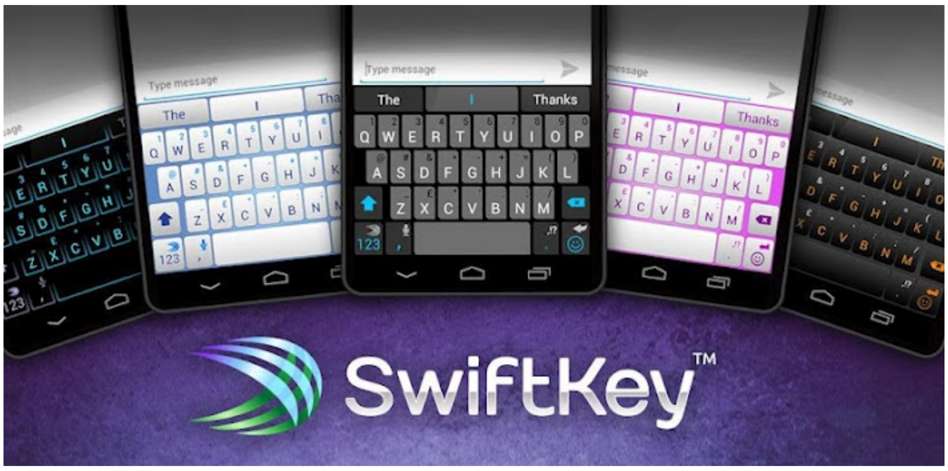


Figure 2.8: Figure showing the UI of the SwiftKey keyboard, offered in various different themes [3]

continues to communicate using the product. On top of that it provides unrivalled prediction accuracy when it comes to typing in alternative languages.

There are of course many other features provided by this particular product but they resemble features that are present in other products as well, hence not adding to the overall originality of the concept.

2.3.6 Swype: Type fast, Swype faster [4]

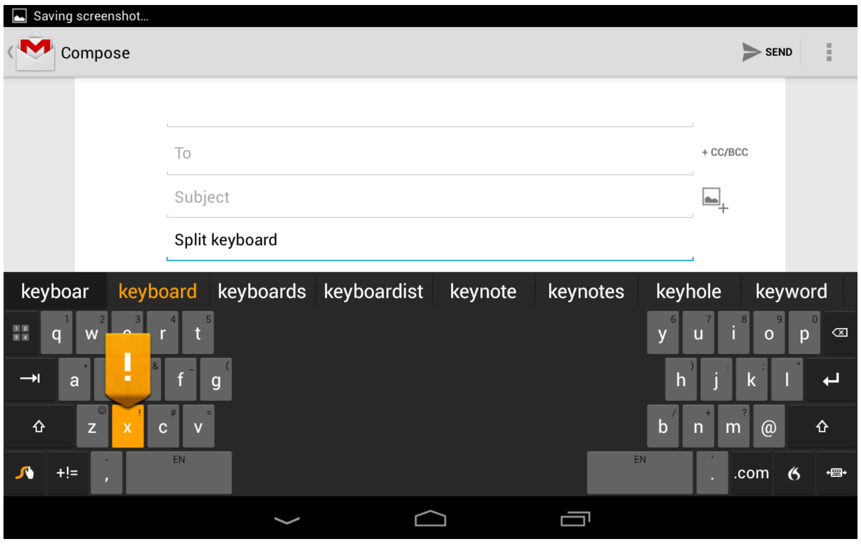


Figure 2.9: Figure showing the UI of the Swype keyboard, as well as its ability to split the layout for more comfortable typing [4]

As the name suggests, Swype Keyboard's most important asset is its ability to allow user input by using the swype gesture. In order to enter a word, the finger should just glide across the surface of the keyboard. The novelty of this concept stands in the algorithm used by the developers in order to determine what word corresponds to a given path drawn across the screen by an user.

Another very interesting feature is the ability to not only customise the way in which the colour scheme and style of the keyboard looks but also the way in which the layout of it is displayed. Users can choose to have the keyboard separated in the middle in order to have the keys closer to the thumbs for a more comfortable experience (Figure 2.9).

The same as with the previously mentioned case, there are as well other features offered by the product but they are common occurrences in other applications available on the market (typing using voice recognition, normal typing as well as the bilingual typing offered by SwiftKey. [3])

2.3.7 Flesky Keyboard: The fastest, most customizable keyboard [1]

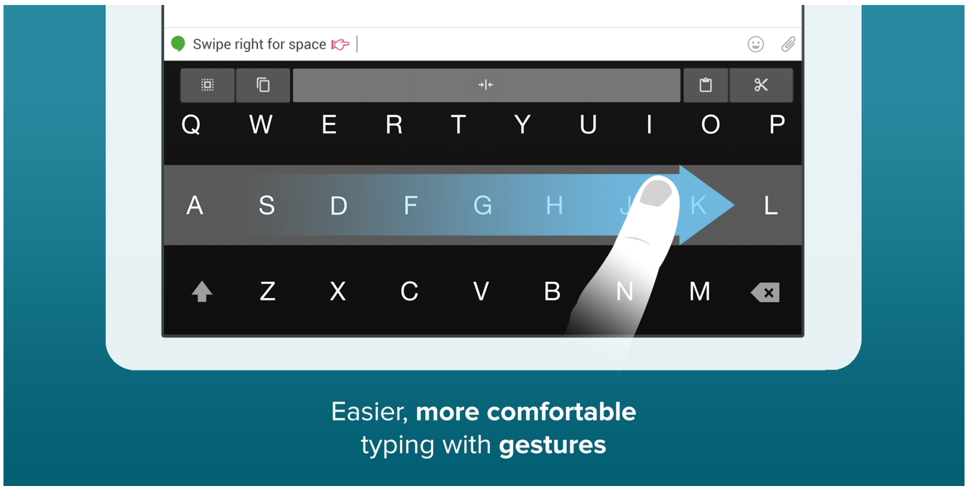


Figure 2.10: Figure showing the UI of the Flesky keyboard, as well as illustrating one of the many included gestures for an easier and faster typing experience [1]

Classified by many as the best custom keyboard available on the market before the apparition of SwiftKey, Flesky is another very good example of a product that brings novelty to an already familiar experience.

What makes this product very special is its improved experience by the use of gestures. While Swype uses a single type of motion exploited to the absolute maximum in order to create an unique product based around it, Flesky goes for the diversity and adds multiple gestures (eg. swype right for space, drag down for cycling through the auto-correct suggestions etc.) that would allow an user to increase their typing speed after growing accustomed with the system itself (Figure 2.10).

Similar to the way in which Swype allows its users to modify the layout of the keyboard, Flesky provides the same functionality but only to a certain extent. The layout cannot be separated but only resized in order to allow for an extra row at the top which is highly customizable (it can host a numerical row, links towards popular applications, search bars etc.).

2.3.8 Minuum Keyboard: Type anywhere [2]

Possibly one of the oddest yet efficient products that are currently available on the market, Minuum keyboard treasures screen space more than anything else, with the core concept being that the keyboard should not intrude with our ability to see the rest of the content on the screen. Based on that principle, the developers have created an incredibly compact layout which only consists of a single row of continuous characters through which the user can cycle through in order to reach to the desired letter (Figure 2.11).

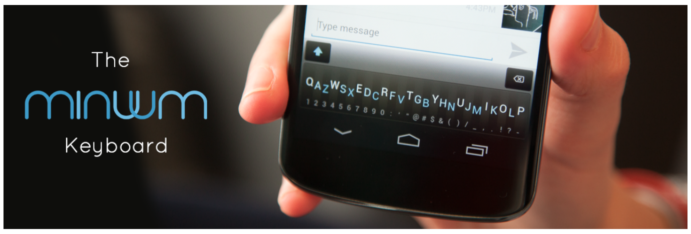


Figure 2.11: Figure showing the UI of the Minuum keyboard [2]

What sets this product apart from the competitors however, is its ability to implement the typing mechanism on virtually any device using their own dedicated SDK. There are examples of it being used on smartwatches, Google Glass, Nintendo Wii consoles and many other unexpected places, living true to the their defining motto: ”Type anywhere”.

Chapter 3

Design

Since the implementation for the project would differ depending on the platform that was chosen, Android was selected due to its variety of customisation features and the fact that developing applications on it is a more easier process to get into as opposed to iOS and Microsoft Phone.

The following chapter will describe the design of the application as well as the decisions which were taken in terms of appearance and layout. However, before we proceed further with the details of the system, it is necessary to present the overall expected functionality. What the project was supposed to achieve is to combine the idea of using a gaussian process (GP) regression for the touch model with the information provided by the language model (LM). The user would then be able to alter this using the length of the touch on a particular key as the deciding factor for choosing between the result of the press and the result of the two models (LM and GP).

3.1 System Architecture

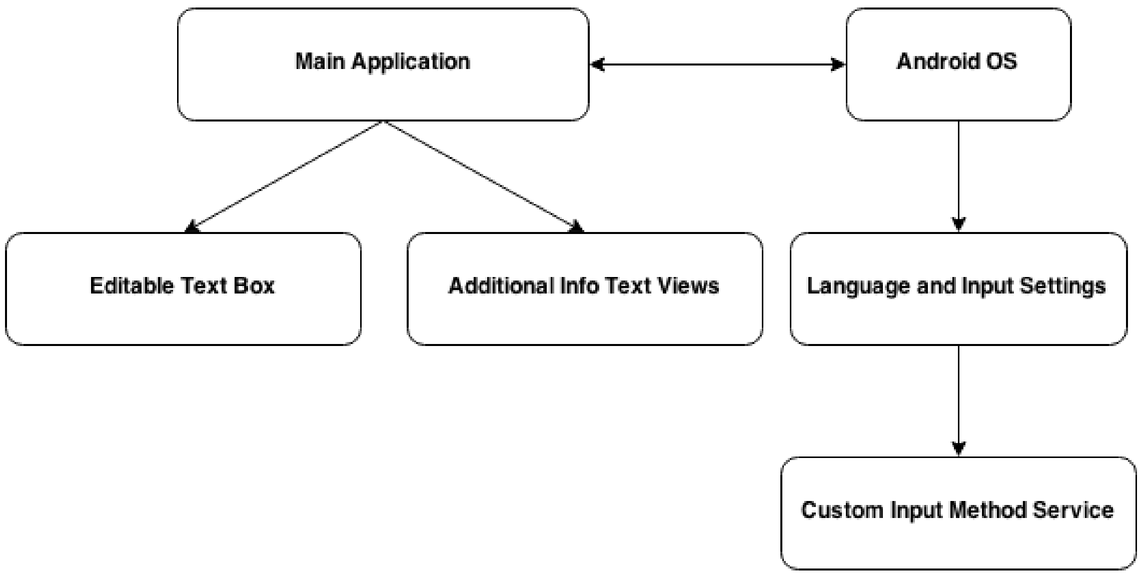

Main Application
Android OS
Editable Text Box
Additional Info Text Views
Language and Input Settings
Custom Input Method Service

Figure 3.1: Figure which displays the outline of the system architecture

During the requirements gathering process, two possible alternatives have been discussed in terms of how the system itself should be structured in order to achieve the desired functionality.

The first option that was proposed was to have the two main components (an activity containing the text resulted from the keyboard input and a custom keyboard service) contained within the same application. The reasoning behind this was that the service itself would be tightly coupled to the application, hence it would be easier to access and interact with at the application level but it would render it to be restricted in its usage with any other application.

Conversely, the second option was to separate the two in order to enable the custom keyboard to be used with any generic application as long as it provided an editable text box. After implementing the first option it has been discovered that the new input method service was registered in the devices settings, hence allowing it to be used with all of the other existing applications, however with certain limitations due to different requirements and behaviours for the keys that the key.

3.1.1 Main Activity

Figure 3.2: Figure showing some different types of activities existent in Android applications (from left to right: settings activity, fullscreen activity, login activity)

The way an activity would be defined, according to the documentation presented on the developer Android website, is that it represents an application component that allows its users to do various things on the screen. If we were to make a parallel to the structure of Java Swing [36], the most accurate correspondent would be the main container in the application, often being the main JFrame which holds all the other declared components within.

The purpose of the activity in our application is to host the editable text box which we will use to trigger the custom keyboard service and also display the result of its input. Optionally it could also include a series of text views meant to show the users meaningful information regarding the computations being performed in the background by the keyboard service.

3.1.2 Input Method Service

The input method service is as the name suggests, an Android service which is in charge of manipulating key events, interpreting them and ultimately sending them through to any applications that may request them by the means of the common channel of communication existing between any application and the keyboard service, called InputConnection.

The main focus of the project is directed at this component in particular because most of the functionality is going to be implemented within the classes that are part of it. The highlighting will be based on the background of the keys that together form the layout for our custom keyboard. The computations for the gaussian probability distribution will be performed using the coordinates provided by the keys within the context of their container. Last but not least the initial input (coming from the combined results of the language model and the touch model) or the modified one (obtained mainly from the touch model) at any given key press will be defined by the duration of pressing that specific letter.

3.2 Use Case

In this section we will go through a usage scenario for the system and show how the application should behave in a normal situation when an user tries to input text using the custom created keyboard. We will separate each event in its own case and present the sequence following the natural order of occurrence.

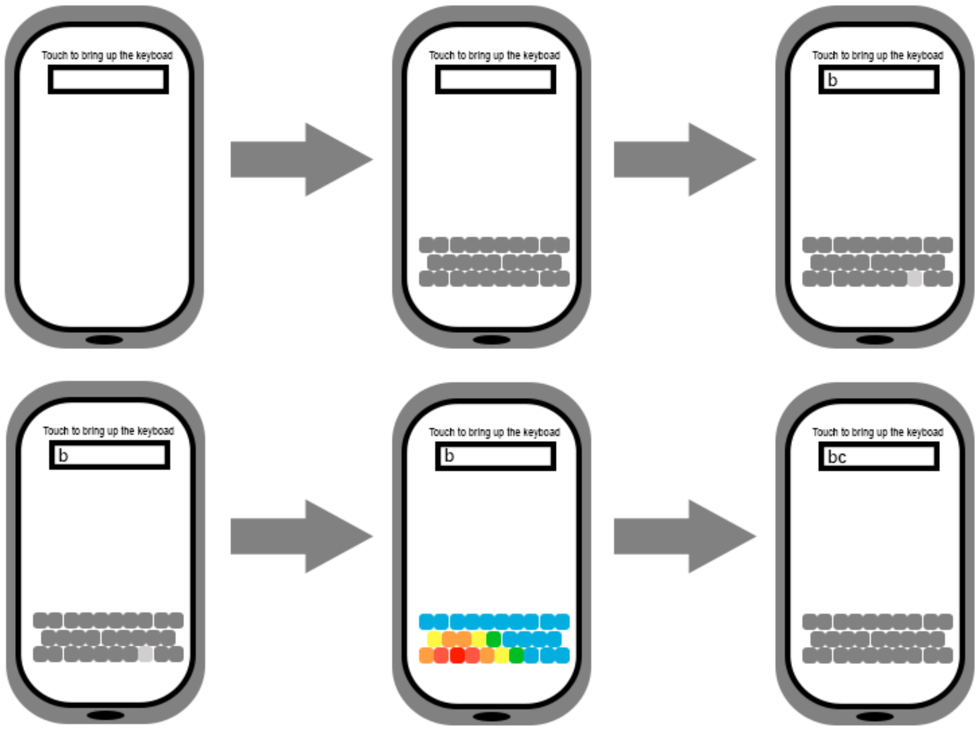

Touch to bring up the keyboad
Touch to bring up the keyboad
Touch to bring up the keyboad
b
Touch to bring up the keyboad
b
Touch to bring up the keyboad
b
Touch to bring up the keyboad
bc

Figure 3.3: Figure which shows a general use case for the system

Figure 3.2 shows how the system will behave in a regular typing situation. The first image presents us with the initial screen that the application will have without any interaction from the user. As soon as the user touches the text box the keyboard service will be initiated and the layout for it will popup into view. The third image presents the case in which the first letter is touched, and as a result of that the pressed letter will be highlighted and the value of it (associated code which corresponds to the key in question) will be sent to the text box which will modify its contents to display the changes. The next image is repeated for the purposes of keeping track of the sequence, being followed by the case in which the user touches the next letter on keyboard. At this time the highlighting process will begin, first showing the results of the language model and the GP regression combined, shortly followed by a change in the layout of the highlights if the user hold the key pressed long enough for the bias to shifted completely towards the GP regression only. Lastly, the remaining case presents the fact that the letter tapped by the user has been sent towards the text box and that it has been updated accordingly to reflect this.

3.3 User Interface/General Appearance

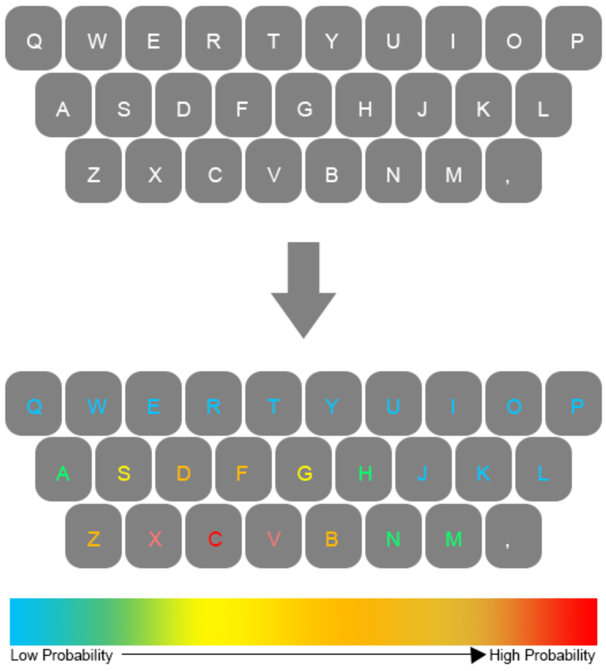


Figure 3.4: Figure showing the label highlighting approach

The user interface for the application itself is rather simplistic and was not modified from the default look that Android will provide, given the fact that the main focus of the project is on the keyboard service rather than the application which is displaying the input coming from it. However, design decisions needed to be made in terms of how the user will be notified of what is happening in the background when a tap on a certain letter occurs.

Regarding this subject, there were two different approaches to be taken. The first one involved modifying the labels of the keys in order to reflect which letter was determined to be the most suitable candidate based on the results provided by the two probability models (LM and GP regression). When it comes to what the changed label will reflect there were two distinct choices as well: will the keys have modified colours of the label based on a scale of probabilities or will only the letter with the highest probability have its label changed to emphasize this.

The figure 3.3 illustrates a mockup of how the first approach would have appeared to the user if it were to be chosen for implementation. However, since the visibility is rather reduced due to the size of the label itself, a second path was chosen, in which the background of the key is being changed to reflect the probability value instead. From the point of view of feasibility the first situation offered a much easier solution in terms of implementation but given the size of the average smartphone screen, the highlighting needed to be far more obvious given also the short duration of it (the amount of milliseconds that passed between two consecutive presses).

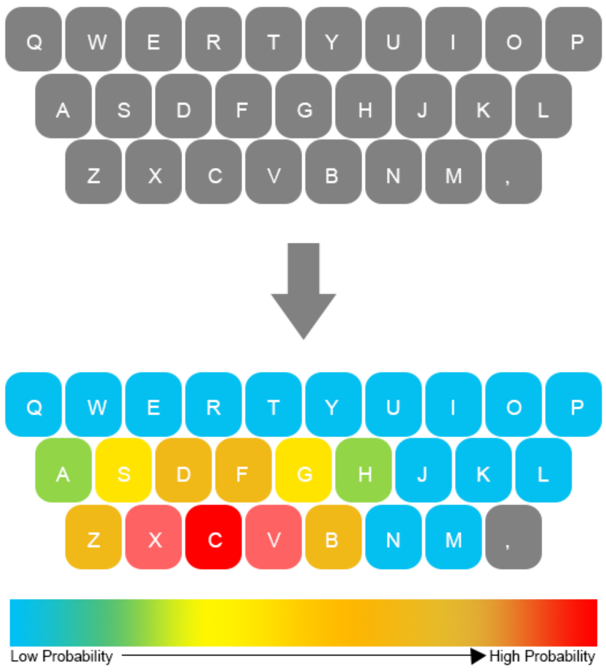


Figure 3.5: Figure showing the key background highlighting approach

Figure 3.4 presents us with an illustration of the second approach and it is immediately obvious that the highlighting is much emphasized from the previous version, hence it will make it much easier for the user to understand what the predictions will be just by taking a glance at the rest of the keyboard while tapping on a letter.

Even though this seemed to be apparent from the mockup designs, a series of users have been consulted for feedback and the result was the majority preferring the key background highlighting approach over the initial one.

Another very important aspect when it comes to the design part of the project was ensuring that the application would be able to be ran on various screen sizes and orientations without sacrificing the way in which the keyboard performs or it looks (pixelization). In order to manage this there was a need for research to be done, hoping to uncover the most efficient way of creating the graphical assets for the keys without introducing too much redundancy in the process, as this could potentially lead to making the typing seem sluggish due to a decrease in performance and timing of the highlighting.

Android provides a series of graphical entities through the means of the Drawable class. What this class represents is an abstraction of objects that can be drawn on the screen, meaning that every single shape can be traced back to being an instance of Drawable. After having tested a series of different objects, a compromise had to be made between two choices whose implementations were rather divergent from one another.

The first choice was using the shape drawable[15] which is also in turn an abstractization for a series of different known polygons and ellipses. This enabled the usage of a variety of different attributes and structures that would allow creating strokes, drop shadows, gradients and many other useful characteristics.

On the other hand, the second choice was represented by the NinePatchDrawable[14] image type. What made this type of drawable special is the fact that a border could be declared (offset from the edge towards the inner parts of the image) and this would allow it to be resized in order to fit the desired bounds specified programmatically.

What the compromise consisted of was weighing the advantages of the raster type (NPD) versus the vector type

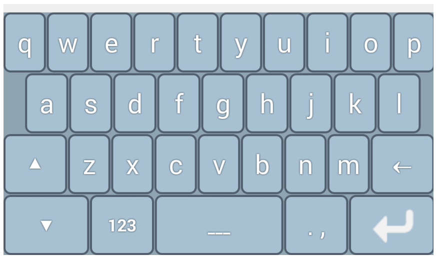


Figure 3.6: Figure showing usage of shape drawable and its styling capabilities

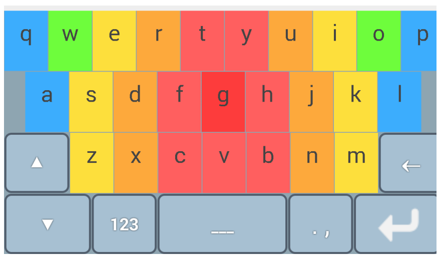


Figure 3.7: Figure showing usage of NPD and its simplicity

(shape drawable) and choose where each of the two would be best suited. The scalability of raster based image and the lower impact on performance make it desirable when it comes to dynamic drawings such as displaying the highlights in a timely manner. For this reason, the choice was made to use NPD for displaying the highlights as they have to be done dynamically and would have a lesser impact on performance given the fact that users normally type quite rapidly and this would mean that a large number of redrawing processes would need to occur in a short amount of time.

As far as the shape drawable type is concerned, the best usage for it was for displaying the normal and pressed state of the keys on the TPV keyboard, seeing as these would be displaying without the user interfering with it, hence they did not need to be updated dynamically nor as frequent.

In terms of the footprint variation for the application that these specific image types will cause, it can be argued that the best balance was chosen between performance, storage and quality. Even though the shape drawables are computationally more intensive than the NPD's, they have a much smaller footprint considering the case that all the image needs to store is a predefined path and the algorithm used to indefinitely scale it. Conversely, the NPD offers much better performance result at the expense of quality and most importantly storage needed. Since the type itself is based on the PNG format, depending on how the image was compressed at the time of creation, the size of the file can be quite large in some situations.

3.4 User Feedback

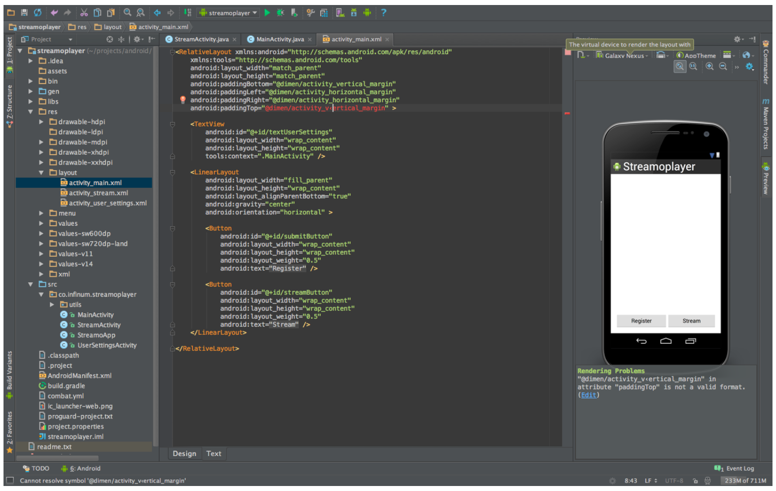
Following a short user evaluation session, in which the participants were asked to provide some feedback regarding the visual aspect of the keyboard service, it has been concluded that the overall design decisions which were taken were rather useful as far as emphasizing the background result of the computations performed by the prediction system but there were a few issues addressed with the timing of the highlights. Subsequently the matter of colour blindness has also been addressed by a few of the participants.

Chapter 4

Implementation

During the next sections of this chapter, we are going to discuss into more detail how the development for the project evolved over time, debating the choices that were made as well as explaining some of the intricacies that were encountered during this process.

4.1 Familiarisation with Android



```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_v‹ertical_margin" >

    <TextView
        android:id="@+id/textUserSettings"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        tools:context=".MainActivity" />

    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:gravity="center"
        android:orientation="horizontal" >

        <Button
            android:id="@+id/submitButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="0.5"
            android:text="Register" />

        <Button
            android:id="@+id/streamButton"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="0.5"
            android:text="Stream" />
    </LinearLayout>

</RelativeLayout>
```

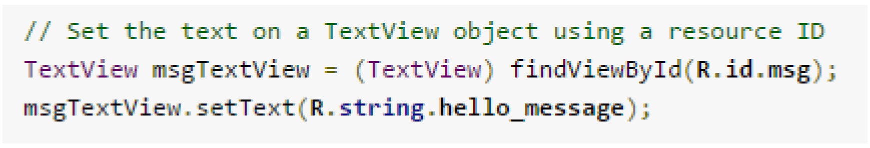
```
Rendering Problems
"@dimen/activity_v‹ertical_margin" in
attribute "paddingTop" is not a valid format.
(Edit)
```

```
Cannot resolve symbol '@dimen/activity_v‹ertical_margin'
```

Figure 4.1: Figure showing the interface of Android Studio

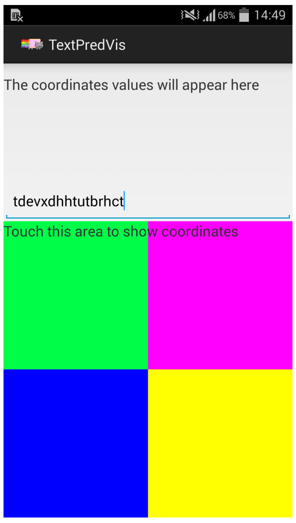
Due to the lack of experience as far as Android development was concerned, the first action needed to be taken before doing any of the implementation was to start getting accustomed with the SDK for this particular platform. Ensuring that we had a good understanding of the various concepts presented in Android was crucial before starting to develop the different components which resembled the final system.

Even if most of the programming is done in Java, the interactions with resource files provided in xml format which all come with Android specific attributes is a concept that is novel to Android. Each of the resources that we create will have associated with them an unique id which is kept in an auto-generated class symbolically named "R" [10]. This class will then be referenced statically in order to access the components from the Java side (Figure 4.2).



```
// Set the text on a TextView object using a resource ID
TextView msgTextView = (TextView) findViewById(R.id.msg);
msgTextView.setText(R.string.hello_message);
```

Figure 4.2: Figure showing an example of the R class usage [10]

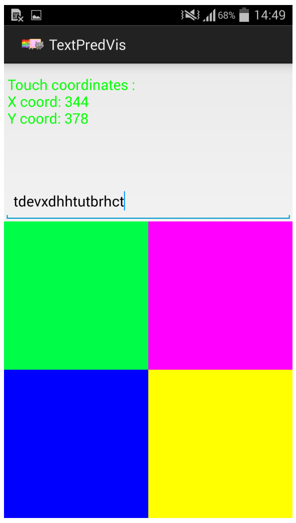


The coordinates values will appear here

tdevxdhhtutbrhct

Touch this area to show coordinates

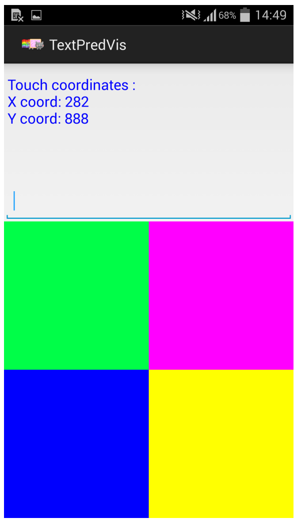
Figure 4.3: Touch manipulation app - default screen



Touch coordinates :
X coord: 344
Y coord: 378

tdevxdhhtutbrhct

Figure 4.4: Touch manipulation app - touch taking place in green area



Touch coordinates :
X coord: 282
Y coord: 888

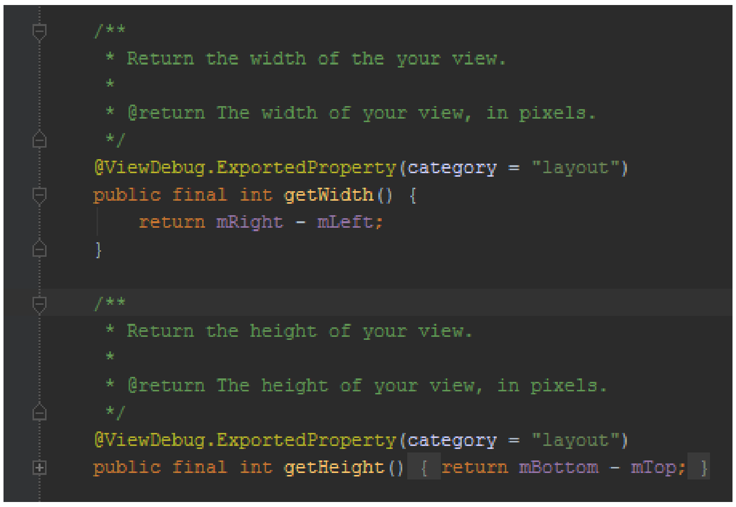
Figure 4.5: Touch manipulation app - touch taking place in blue area

The first step in the process was to go through the documentation and the tutorials that are being hosted on the developer Android website in order to produce basic applications and get a feel for how applications will behave. After having done that, the next logical step was to start developing a basic app but in order for this to happen, another learning process needed to take place. This was represented by getting accustomed with using Android Studio, the main IDE for this platform (Figure 4.1).

4.2 Touch Coordinates Manipulation Experiment

After having gained some experience with using Android Studio it was time to put together a simple application that would obtain touch coordinates from the touch position. Based on the location of the touch, a text view would display them in different colours according to a mapping which was displayed within the application (Figure 4.3, Figure 4.4, Figure 4.5). This was mainly a proof of concept in order to see how to obtain and manipulate touch information (in particular the coordinates) for various purposes.

When the screen is touched in Android, the component that we interact with is not the main activity as we might initially think but it is often times a container called `View` [18]. This was the case in our situation as well, as we had an empty `View` whose background was set to be the png image containing the colour mapping for the areas. The logic behind deciding within the surface of which colour we were currently touching was rather simple, we would use the `getHeight()` and `getWidth()` methods (Figure 4.6) in order to determine the dimensions and successfully split the empty `View` into four quadrants. After this was in place, at the moment of touch we would call the method `onTouchEvent(MotionEvent e)` in order to get access to the touch coordinates within the `View` itself and then simply assess whether we were situated into one of the previously defined quadrants and update the color of the `TextView` [16] which was in charge with displaying the information related to this.



```
/**
 * Return the width of the your view.
 *
 * @return The width of your view, in pixels.
 */
@ViewDebug.ExportedProperty(category = "layout")
public final int getWidth() {
    return mRight - mLeft;
}

/**
 * Return the height of your view.
 *
 * @return The height of your view, in pixels.
 */
@ViewDebug.ExportedProperty(category = "layout")
public final int getHeight() { return mBottom - mTop; }
```

Figure 4.6: Figure showing the implementation for the `getWidth()` and `getHeight()` methods [18]

4.3 Researching Soft Keyboards

By starting with the default sample provided by Android for the skeleton implementation of a `SoftKeyboard`, the next step in the implementation process was to get touch coordinates from within the area of the keyboard. In order to do so, a proper understanding of how to implement a custom `InputMethod` [12] was needed so that a decision could be taken in terms of which component would offer us the correct information.

Even though the documentation website for Android offers a decent series of guidelines regarding how to go about developing a new `InputMethod`, this is only suitable in order to achieve the basic functionality that we would expect from such a service. A much more comprehensive guide was found online on the website dedicate to Android Development of Maarten Pennings [25]. It described in detail how to produce various alterations to a `Soft Keyoard` for it to suit the different contexts in which it was going to be used. From the many things that were taken into account the most relevant for the purpose of this project was explaining the structure of such an implementation and describing what can be achieved with each of containing elements.

From a structural point of view, an implementation of a custom keyboard contains the following JAVA components: an extension of the base implementation for an `InputMethodService` which in this case was called `CustomIME`, a view for the layout of the new keyboard (`TPVKeyboardView`), the layout itself (`TPV-Keyboard`), and a helper class that will allow for generating co-occurrence matrices from the dictionary of words for the language model named `BigramCoocMatrix` (Figure 4.7).

4.3.1 CustomIME

Within this class, an extension from the offered base implementation for the InputMethodService was created. At this level we can control various different types of behaviours, from which we will mention the following:

- communication with text fields (through the means of the `getCurrentInputConnection()` method (Figure 4.8);

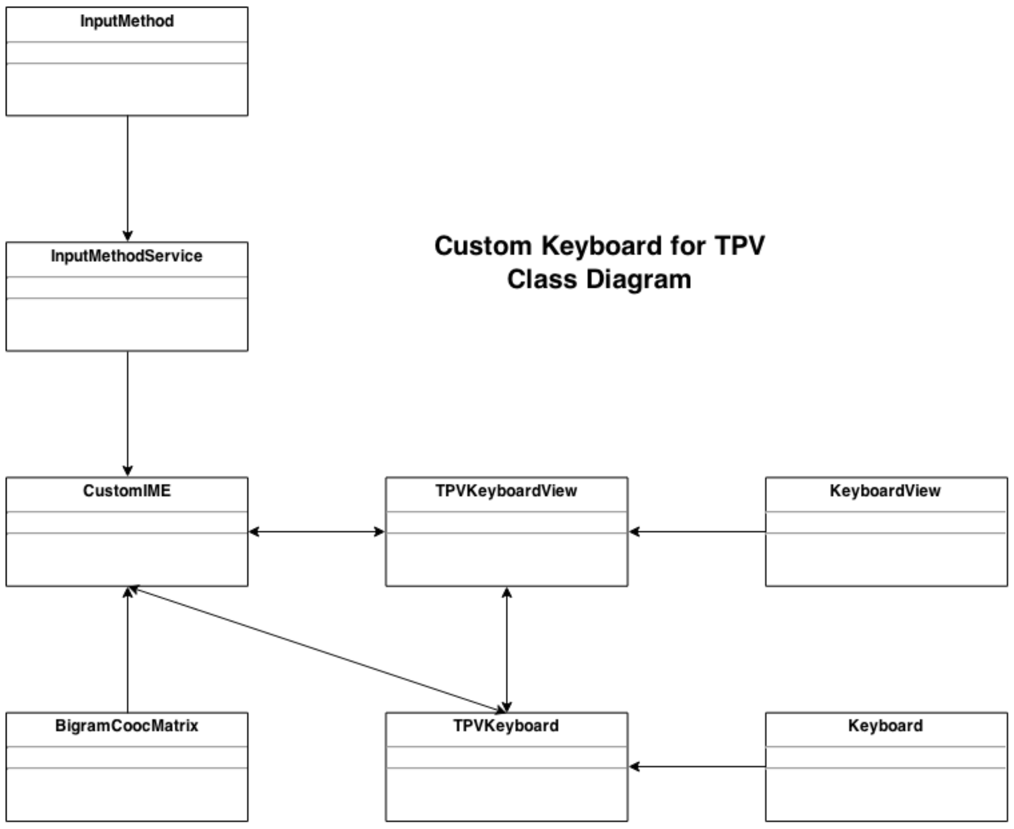
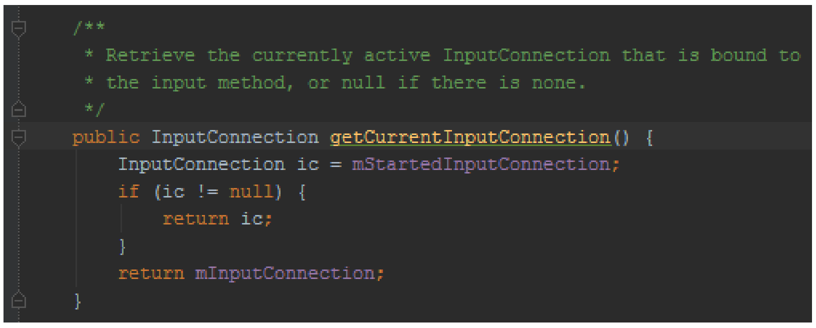


Figure 4.7: Figure showing the class diagram for the custom keyboard

- manipulate key events;
- generate the keyboard layout at the start of the service (implemented within the `onCreateInputView()` method);
- modify the layout according to events which include modifier keys such as `Shift` or `Caps`

Basically everything that is tied into actual functionality and interaction with the system and the other applications will be presented in here while the appearance of the layout and it's properties will be managed in the other two classes previously mentioned (`TPVKeyboardView` and `TPVKeyboard`).



```
/**
 * Retrieve the currently active InputConnection that is bound to
 * the input method, or null if there is none.
 */
public InputConnection getCurrentInputConnection() {
    InputConnection ic = mStartedInputConnection;
    if (ic != null) {
        return ic;
    }
    return mInputConnection;
}
```

Figure 4.8: Implementation of getCurrentInputConnection()

4.3.2 TPVKeyboardView

This class is in charge of generating and hosting the keyboard layout which is defined in the three resource xml files associated with the alphabetic, numerical and symbol layouts (`qwerty.xml`, `symbols.xml`, `symbols_shift.xml`). The draw method that controls how the keyboard will be generated on the `Canvas` [11] is also implemented here, hence all the dynamic highlighting will be generated using it.

4.3.3 TPVKeyboard

As the last major component of a custom keyboard service for Android, this class has the sole purpose of generating an instance of a `Keyboard` [13] object (or `TPVKeyboard` in our case) from using the information and specified attributes in the xml files associated with the keyboard itself (`input.xml`) and the three layout defining xml files mentioned previously).

4.4 Label Alteration

The first stage in trying to alter the behaviour of the standard keyboard service was to implement the functionality that would enable a key to change label after it was pressed in order to signify the user that a change occurred. Since this meant that the way in which the keyboard behaved was targeted, the implementation was done in the class designated with controlling this (`CustomIME`).

Initially, this was implemented within the `onKey()` method and as a result the label would change only after the user had released the key and while this was suitable for showing which key were touched, it was not necessarily viable for the approach needed in the project. The intended behaviour was to have the label modify its value during the press and revert to its initial state after release. In order for this to be done programmatically the entire process had to be split into smaller atomic actions as it would be the case with a normal mouse or physical key press.

As a result of that approach, the implementation code had to be moved from the `onKey()` method to the corresponding methods associated with the behaviour of the press and release of a given key (`onKeyDown()` and `onKeyUp()`). This achieved the desired outcome (Figure 4.9) but it was ultimately removed due to the feedback provided by the users and the client in terms of how much better it would be to have the key backgrounds change when highlighting changes instead of using the key labels (see discussion about UI and appearance in Chapter 3 Subsection 3.3)

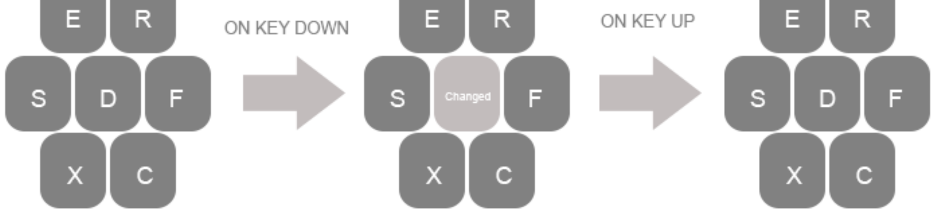
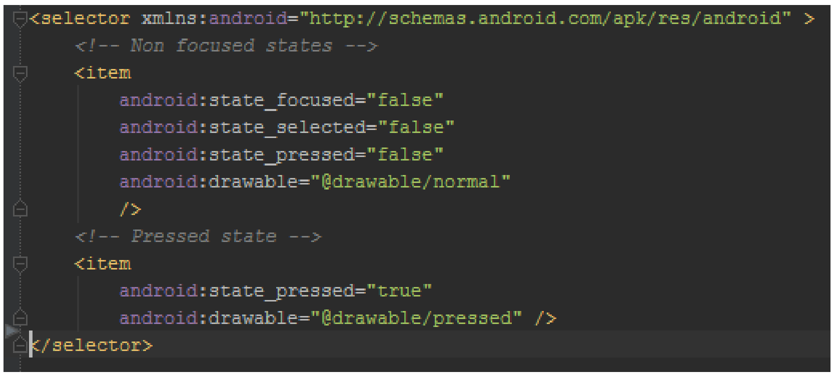


Figure 4.9: Label alteration process

4.5 Highlight Creation

Due to the design decisions that were taken as a result of the discussions with both the client and some prospective users, the next step in the development process was to find a feasible way to implement key background highlighting.

The first attempt at providing this type of functionality was by trying to alter the `key_state_selector.xml` file in order permit another state that was dedicated to creating the highlight (Figure 4.10). However, this proved not be feasible as the attributes that needed to be set in order for the highlight to occur would overlap with the characteristics of the pressed state, hence making the distinction problematic and random. The second approach was to try and imitate the implementation of the label alteration but this only worked up to a certain extent since there was no way to access the key background from within the `CustomIME` class and as a result this reasoning had to be dropped as well in hopes of a better alternative.



```
<selector xmlns:android="http://schemas.android.com/apk/res/android" >
    <!-- Non focused states -->
    <item
        android:state_focused="false"
        android:state_selected="false"
        android:state_pressed="false"
        android:drawable="@drawable/normal"
        />
    <!-- Pressed state -->
    <item
        android:state_pressed="true"
        android:drawable="@drawable/pressed" />
</selector>
```

Figure 4.10: Label alteration process

Ultimately, a working solution has been found and this consisted of sending repaint requests for the `Canvas` object destined to hold the keyboard layout and instead of actually modifying the key background itself, and overlay would be created at the exact position of the key. This behaviour would look much like the pop-up that would appear initially in the case of a long press in order to allow cycling through letters with different accents attached to them or other similar actions. The most suitable place to host this part of the implementation was the `onDraw()` method from within the `TPVKeyboardView` class.

Now that the mechanism for creating the highlights dynamically was in place, some structure that would allow us to control whether highlighting should be enabled or not need to be created. The best solution was to create a `boolean` flag named `highlightOn` which would always have its initial state be `false` and would be turned on only when one of the alphabetic keys on the keyboard was in a pressed state.

4.5.1 Fat-finger Approach

A very suitable test for the dynamic highlighting was represented by the fat-finger approach. The assumption that stands at the base of this is the idea that more often than not people will touch a key that's situated in the immediate proximity of the actual key that they wanted to tap on and by using this, we tried to highlight all the keys adjacent to the pressed key (Figure 4.11). This was a very good experiment in terms of showing how feasible dynamic highlighting would be with this approach and in order to allow for possible optimisation issues to surface as well.

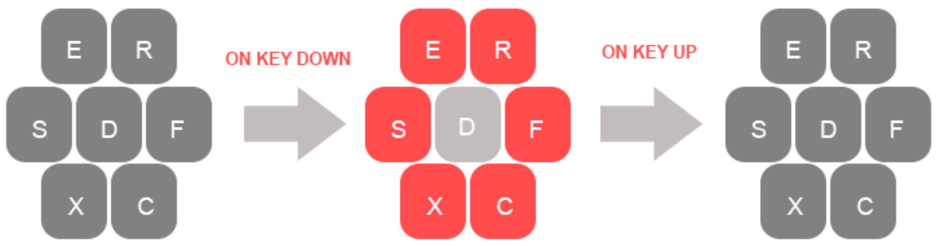


Figure 4.11: Illustration of fat-finger approach test

4.6 Creating the Language Model

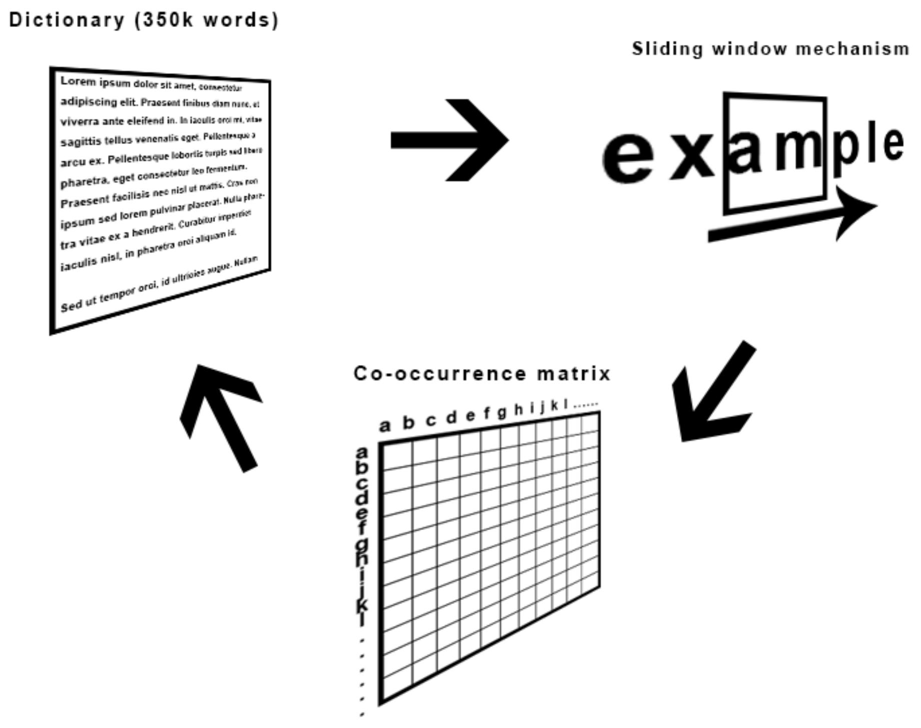


Figure 4.12: Illustration of language model approach

As soon as the highlighting was working as desired and the client was satisfied with its behaviour, a discussion took place in order to decide what the best approach was for designing the language model [33] in regards to the time left for development. It was decided that the language model should provide probabilities for the next letter by only looking at the previously entered one as the determining factor. At the same time, the system would have to be extensible enough in order to allow for a more complex language model to be put in place as a future improvement. Once this was established, some research was done for the purpose of discovering a good approach for developing this part of the project.

The final implementation was to determine probabilities based on having a file containing a dictionary of words as input data. Two dictionaries were used in order to improve to alternate between faster generation time and more accuracy for the probabilities depending on the context (the sizes were 58k [5] and 350k words [9]). The result would then be a co-occurrence matrix in which the rows would represent the first letter in a `bigram`, consequently having the columns represent the last letter (Figure 4.12).

A `bigram` is defined by a pair of tokens which in our case would be a pair of characters (which in the context of our implementation would have been the last letter in the word that was being composed and the letter that could be suggested as a result for the next possible candidate).

The values for the matrix itself would be generated using the following rational: a `Scanner` [24] would go through every word present in the dictionary and then extract them into an `ArrayList` [23] of `String`. Once this was performed we would iterate through the matrix to form each possible `bigram` and then we would cycle through the words present in the array using a sliding window of two characters. If the `bigram` was found into a word then its value from a map that contained all the possible pairs and their associated count would be incremented.

As far as the probabilities were concerned, the matrix should have not contained any zero values as this would nullify the results of the overall prediction process when the value supplied by the language model was combined with the one provided by the touch model (process which consisted in the multiplication of the two values). For this to be counteracted, the counts for the character pairs in the map destined to store this information started all from one. This would ensure that after computing the probability distribution, the final value would be very small but never zero.

This entire functionality was separated into the class `BigramCoocMatrix` in order not to perform these computations in one of the classes that is concerned with the rendering and behaviour of the keyboard. The data structure chosen for the matrix was a `HashMap<String,HashMap<String,Double>>` in which the inner map represents a row in the matrix with its associated values, while the outer map holds the projections from the last letter in the text to the map that holds the possible candidates and their probabilities. The matrix can be generated externally using a separate Eclipse project that will be delivered together with the main project and will produce a file that needs to then be stored inside the `assests` folder for the Android project. The other alternative is to create it at the time of launching the application which is one of the reasons for which the `BigramCoocMatrix` was included in the package for the TPV Keyboard, even though the second situation is undesirable. More importantly, another reason for which the class was included in the project was to enable the `CustomIME` to produce an instance of it within the `onCreateInputView()` method that would contain the generated matrix for it to be used in the computations to follow.

4.7 Creating the Touch Model

The touch model for the system was very much inspired from the GPType approach that was taken in the study conducted by Weir et al. [27]. What this means is that the probabilities would be generated using a Gaussian probability distribution based on the position of the touch.

The equation used for calculating the probability distribution for the touch model using a system with two coordinates x and y (Figure 4.13) is the following Gaussian function:

$$f(x,y) = Aexp(-(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2})) \tag{4.1}$$

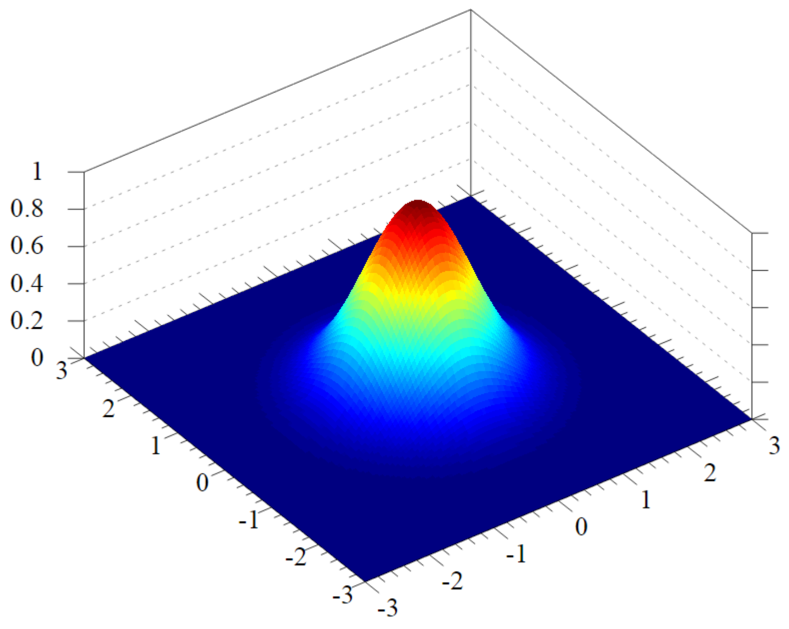
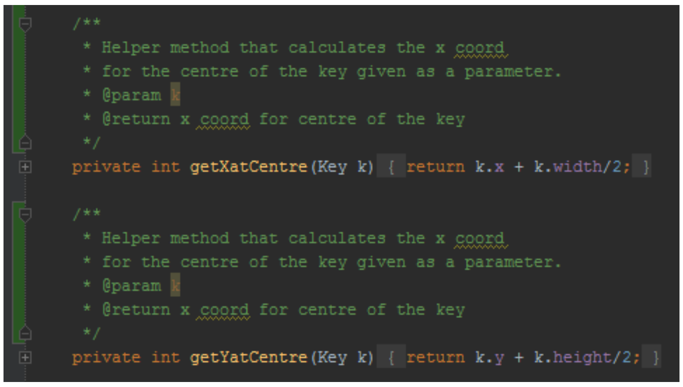


Figure 4.13: Plot of a gaussian funtion in a system with two axes of coordinates [30]

However, there were a series of adaptations made in order to simplify the process and decrease the amount of complexity and resulting computations that were necessary to be performed every single time a touch occurred. This was, of course, done due to exiting time constraints and in order to improve the performance of the system, considering the fact that this process would take place very frequently during the usage of the TPV Keyboard. The amplitude A was discarded with the consideration that it will considered constant (with a value of 1), hence the variation will not be taken into account. Another assumption that was made was the fact that the variance (σ^2) will have the same value on both axes, resulting into the distribution having a circular shape instead of resembling an ellipse which would be angled differently depending on the values for the standard deviation (σ) on both axes.

Since this had to be performed on each key press, the implementation for calculating the distributions was set to reside also inside the `TPVKeyboardView` class within the `calcFinalProb()` method. There were also a few helper methods created in order to provide some efficiency as well as reduce redundancy in terms of frequently occurring segments of code. The methods created were the following: `getXatCentre(Key k)`, `getYatCentre(Key k)` and `getVariance(Key currentKey)`. The first two methods (Figure 4.14) were created to avoid having to calculate the centre of a given `Key` repeatedly within the computations performed by the touch model, while the last method was initially created with the purpose of calculating the variance value given the reference to the key that was currently being pressed. This was discarded when deciding to have the same variance for both axes, leading to the value being considered a constant. The logic for it was kept however, for it to be made use of during possible further improvements or extensions made to the project.

Given the fact that σ^2 was now considered a constant, the challenge was to find a suitable value for it so that the results of the Gaussian probability distribution were proportional with the ones provided by the language model. Taking into consideration the values of the squared distances between the touch position and each of the keys as well as the values for the probabilities provided by the LM, the best suiting number for the variance was found to be 10000.



```java
/**
 * Helper method that calculates the x coord
 * for the centre of the key given as a parameter.
 * @param k
 * @return x coord for centre of the key
 */
private int getXatCentre(Key k) { return k.x + k.width/2; }

/**
 * Helper method that calculates the x coord
 * for the centre of the key given as a parameter.
 * @param k
 * @return x coord for centre of the key
 */
private int getYatCentre(Key k) { return k.y + k.height/2; }
```

Figure 4.14: Implementation of getXatCentre() and getYatCentre()

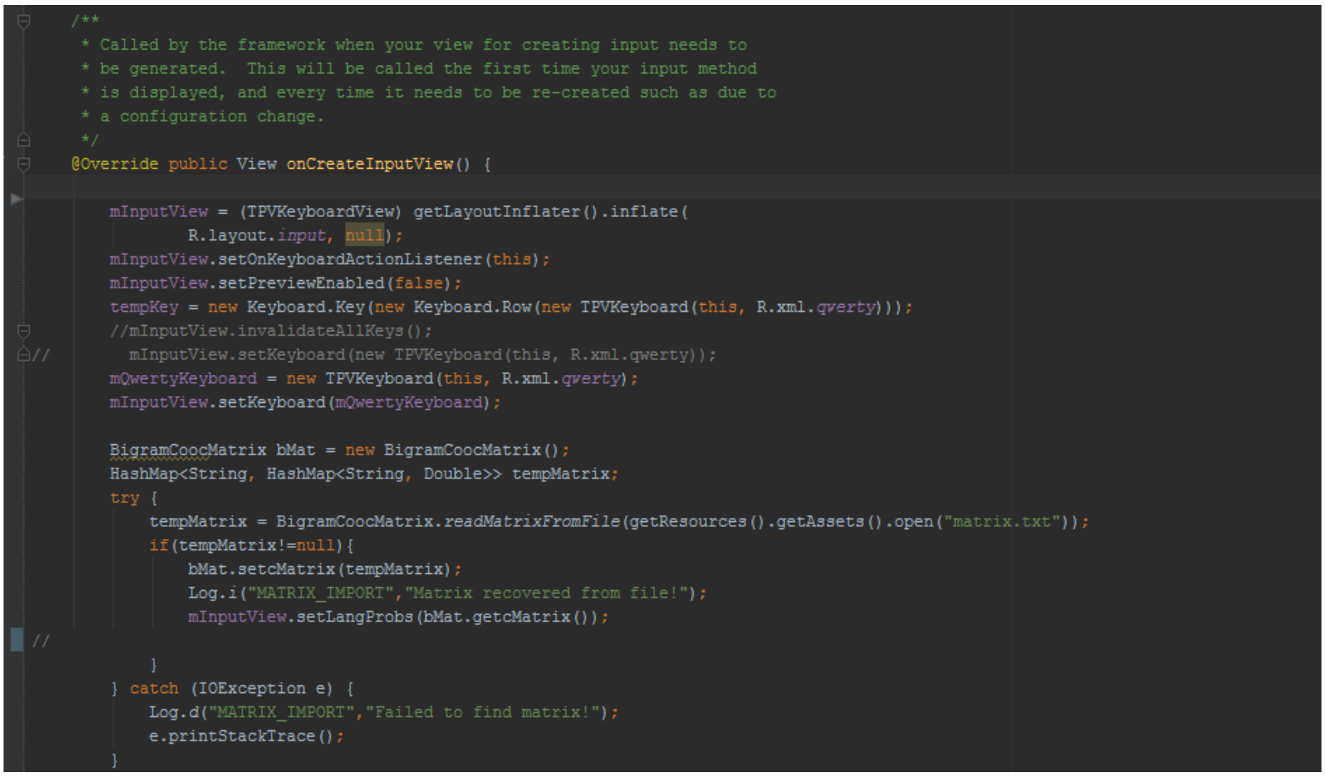
4.8 Connecting the Components

Having all the main parts of the system implemented (language model, touch model and highlight mechanism) it was time to connect them together in order provide the expected functionality. Before all of this could be put in practice however, the data obtained from the two models needed to be normalized before being combined for the final results.

The sequence to be performed was rather similar for both the language model and the touch model. In the case of the latter, after having calculated the preliminary values of the exponent for the Gaussian function, the maximum result needed to be extracted from that series and then subsequently subtracted from the results for every key before actually performing the exponentiation. Then a sum of the resulting values would be calculated, followed by obtaining the probabilities as a result of dividing each previously generated value to the determined sum. The thought process was almost the same for the normalisation of the data provided by the language model. Here, we would calculate the sum of counts of occurrences for a given row, without first determining what the maximum value was, and then obtain the probabilities by dividing each individual count to the sum.

In the last part of the process for determining the final probabilities, the two sets of data need to be combined in order to provide the highlighting system with the information that it needs for displaying each key. This sequence of actions starts in the `CustomIME` class where, at the time when the start of the input is detected, an instance of `BigramCoocMatrix` is created and within that object, the data from the file previously generated in Eclipse will be loaded. Immediately after, the `langProbs` attribute from the `TPVKeyboardView` is initialized with the instance of the co-occurrence matrix extracted from the object that resides locally in the `onCreateInputView()` method from `CustomIME` (Figure 4.15).

Finally, the last part involves multiplying the two corresponding sets of probabilities, and this is performed within the `calcFinalProbs()` method. Depending on the key that was passed on as a parameter, the relevant row will be extracted from the `langProbs` matrix (which, considering the fact that the data structure used is `HashMap<String,HashMap<String,Double>>`, will be the value of the key in the first map that resembles the label of the keyboard key given as a parameter to the method.). Afterwards, the values for the touch model will be computed using the algorithm described in the Section 4.7. Last but not least, we will iterate through the obtained map for the touch model and populate the map `finalProbs` with the results of the multiplication between each value in the temporary map generated for the touch position and the corresponding probability from `langProbs`.



```java
/**
 * Called by the framework when your view for creating input needs to
 * be generated.  This will be called the first time your input method
 * is displayed, and every time it needs to be re-created such as due to
 * a configuration change.
 */
@Override public View onCreateInputView() {

    mInputView = (TPVKeyboardView) getLayoutInflater().inflate(
            R.layout.input, null);
    mInputView.setOnKeyboardActionListener(this);
    mInputView.setPreviewEnabled(false);
    tempKey = new Keyboard.Key(new Keyboard.Row(new TPVKeyboard(this, R.xml.qwerty)));
    //mInputView.invalidateAllKeys();
//      mInputView.setKeyboard(new TPVKeyboard(this, R.xml.qwerty));
    mQwertyKeyboard = new TPVKeyboard(this, R.xml.qwerty);
    mInputView.setKeyboard(mQwertyKeyboard);

    BigramCoocMatrix bMat = new BigramCoocMatrix();
    HashMap<String, HashMap<String, Double>> tempMatrix;
    try {
        tempMatrix = BigramCoocMatrix.readMatrixFromFile(getResources().getAssets().open("matrix.txt"));
        if(tempMatrix!=null){
            bMat.setcMatrix(tempMatrix);
            Log.i("MATRIX_IMPORT","Matrix recovered from file!");
            mInputView.setLangProbs(bMat.getcMatrix());
//
        }
    } catch (IOException e) {
        Log.d("MATRIX_IMPORT","Failed to find matrix!");
        e.printStackTrace();
    }
```

Figure 4.15: Implementation createInputView() showing the creation of the BigramCoocMatrix object within

4.9 Highlighting based on Probabilistic Models

Once the `finalProbs` map has been populated with values (provided the fact that the key that is being pressed is a letter), the next part that follows in terms of delivering the highlights to the user that would be currently tapping on a letter is to iterate through the keys and assign a different highlight colour depending on a series of threshold levels that have calculated based on a proportion of the highest probability (Figure 4.16).

Having considered the above mentioned, the highest probability will first be extracted by cycling through `final-Probs` and storing it into a local variable `max`. This won't happen however, unless the `highlightOn` flag is set to `true`. For this situation to occur the system will inspect if the keyboard key that is currently being pressed is a letter and if the text that is currently being composed meets a series of requirements. The two conditions that need to be respected are the following:

- the text must not be empty (this is rather obvious, since the prediction is based on the last letter in the text, the language model would not be able to return its results without having any letter to start the process with);
- the last character not be anything else other than a letter (again this is a requirement because of the same rational as in the previous condition).

There are six levels of colouring in the existing system and five thresholds based on the highest probability. The boundary between levels is determined exponentially by starting with the maximum value and then each threshold is calculated by multiplying the maximum amount with a negative power of 10 (obtained by subtracting 1 from the level number that we are trying to calculate the boundary for) (Figure 4.17). So, generally speaking, if we wanted to know the threshold for the any of the other six levels of highlighting we would have to calculate $max * 10^{1-level}$.

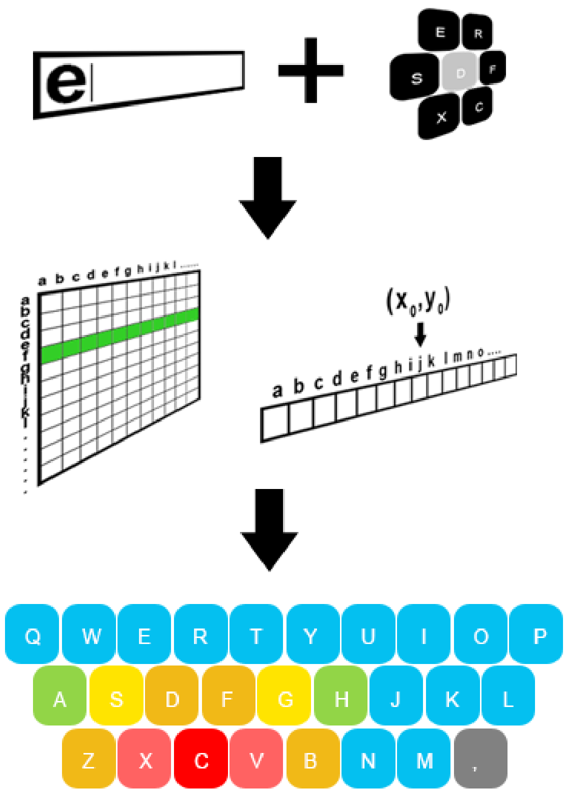


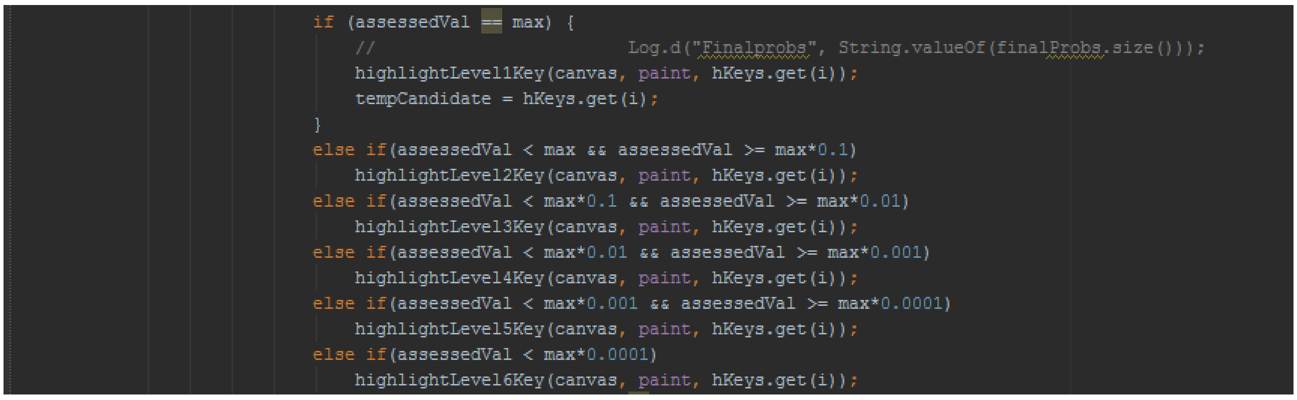
Figure 4.16: Diagram showing the steps of the highlight process for TPV

Apart from the implementation of the threshold differentiation and the ability to create the highlight levels proportional with the maximum registered value, a series of helper methods were created that would aid to create the correct highlight at the appropriate location of a given key. The methods are indexed based on the probability level that they are destined for (`createHighlightLevel2`, `createHighlightLevel3`, etc.) (Figure 4.18).

4.10 Using Touch Duration as an Offset Factor

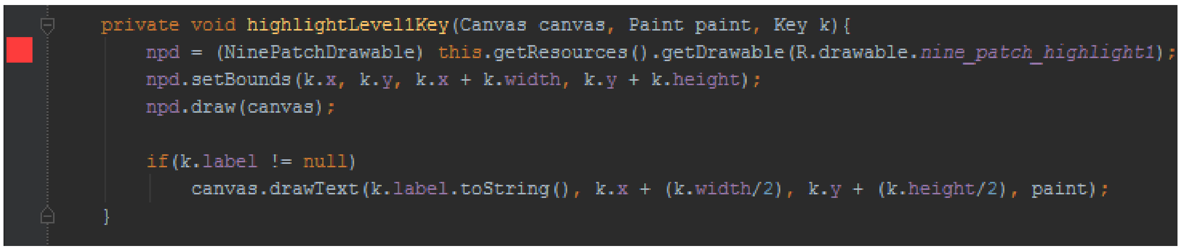
The second phase of highlighting feature for the TPV keyboard was supposed to be represented by its ability to allow the user to offset the bias in terms of the power of decision upon what the next letter being sent to the text box will be. The reasoning for this is that the products that are currently on the market value the input of the correction system slightly higher than the input of the user. Due to the numerous advancements that have been made in this particular area of interest, some would argue that it is perfectly reasonable for them to do so but the reality of the situation is that the exceptions that were presented in the motivation section of this paper (Chapter 1, Section 1.1) will keep occurring regardless of the product that we are using. Hence, some factor that would allow us to slowly alter the decision taken by the system (in order to reflect the key that we are pressing) could be nothing but beneficial.

The determining factor for our project was the duration of pressing a certain key. Conceptually speaking, the way in which the system would react to a longer touch would be to slowly alter the variance value for the touch model, which should in turn result into a smooth transition towards higher probabilities coming from the it, which would



```
if (assessedVal == max) {
    //                      Log.d("Finalprobs", String.valueOf(finalProbs.size()));
    highlightLevel1Key(canvas, paint, hKeys.get(i));
    tempCandidate = hKeys.get(i);
}
else if(assessedVal < max && assessedVal >= max*0.1)
    highlightLevel2Key(canvas, paint, hKeys.get(i));
else if(assessedVal < max*0.1 && assessedVal >= max*0.01)
    highlightLevel3Key(canvas, paint, hKeys.get(i));
else if(assessedVal < max*0.01 && assessedVal >= max*0.001)
    highlightLevel4Key(canvas, paint, hKeys.get(i));
else if(assessedVal < max*0.001 && assessedVal >= max*0.0001)
    highlightLevel5Key(canvas, paint, hKeys.get(i));
else if(assessedVal < max*0.0001)
    highlightLevel6Key(canvas, paint, hKeys.get(i));
```

Figure 4.17: Code showing the threshold levels determined based on the highest probability



```
private void highlightLevel1Key(Canvas canvas, Paint paint, Key k){
    npd = (NinePatchDrawable) this.getResources().getDrawable(R.drawable.nine_patch_highlight1);
    npd.setBounds(k.x, k.y, k.x + k.width, k.y + k.height);
    npd.draw(canvas);

    if(k.label != null)
        canvas.drawText(k.label.toString(), k.x + (k.width/2), k.y + (k.height/2), paint);
}
```

Figure 4.18: Code for helper highlighting method

then be reflected through a visible, gradual change in the way the highlighting system would produce the output.

However, when it comes to the actual implementation of this feature, a compromise had to be made because of the limitation imposed by the way in which Android registers presses. There is no way in which the platform can determine the length of the press. There are two states in which a press can find itself in at any given time: normal or long. Within the `TPVKeyboardView` we can assess whether a press is in one of the mentioned states by call the method `onLongPress(Key k)`. The direct consequence of this is the idea that we wouldn't be able to have a smooth evolution in terms of highlighting over time but rather a sharp change in the way it looks as soon as the press state would be modified.

The initial functionality for this method was to trigger the rendering of the popup with alternative outcomes for a key (such as different accented letters or quickly switching through a small collection of punctuation). In order to cause a change in the way in which the highlighting system determined its values, we created a `boolean` flag named `isLongPress` which would enable us to decrease the predetermined value for the variance. This should in theory lead to an increase of the values produced by the touch model, hence giving them a greater weight in determining the final probabilities after being combined with the values from the language model. If the variance decrease was significant enough this would lead to a desired outcome, in which the letter being sent to the recipient text box would be the one the user is actually pressing.

The initialisation value of `isLongPress` would always be `false` and it would only get set to `true` in the case where the system detected a call for the `onLongPress(Key k)` method. Its value would then be reset whenever a new letter was commited to the text box as a result of a call towards the `handleCharacter(int primaryCode, int[] keyCodes)` method from with the `onKey()` method belonging to the `CustomIME` class.

After having tested the implementation of the above rational it has been discovered that the behaviour would not be as consistent as it should have been. Even after performing another set of tests using smaller and smaller

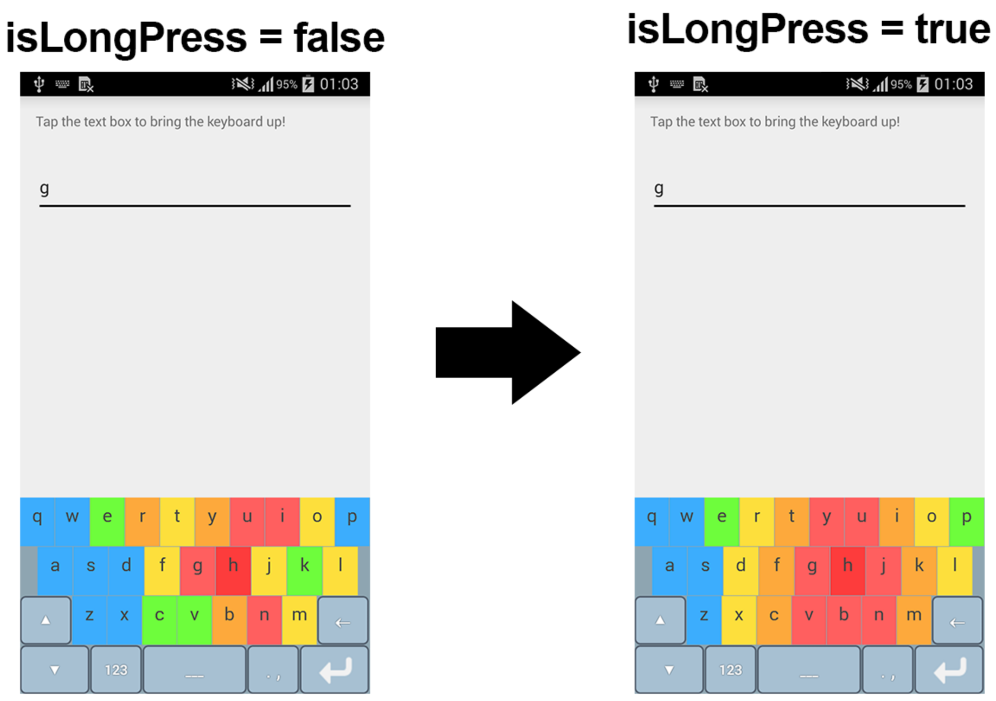


Figure 4.19: Illustration of highlight transition depending on length of press

values for the variance, there would still be cases in which after the long press event was detected the highlighting would not indicate the pressed letter as the most suitable candidate. The solution to this was to change focus from decreasing the variance value towards completely denying the language model to contribute to the final results in the particular case that a long press was detected. By doing so, the result of the long press would always be the key that the user is typing while keeping the initial predicted result for the case in which a short tap on a letter occurred (Figure 4.19).

Chapter 5

Evaluation

5.1 Usability/Feasibility Test

The process of evaluation consisted of a simple usability test, which was followed by a short survey in which some feedback on the experience was requested as well as possible suggestions for areas of improvement. There were 10 participants with ages varied between 18-25 years old. All of the participants were avid smartphone users and were experienced in typing on soft keyboards.

Initially, the participants were asked to input a few sentences without using the highlighting system, in order for them get accustomed with the layout and the look of the TPV Keyboard. The second part of the test presented them with the same task, with the difference being the fact that now they would be making use of the highlight mechanism. Even though, at the beginning most of them were typing rather slow, as they started entering more text, they started to improve their typing speed. This is a natural occurrence since the participants were faced with an unknown type of assistance when typing on a smartphone.

After the test had finished everyone was informed with regards to a future survey and asked about their availability to take part in it. The reasoning provided was that they would be asked to offer some feedback based on their experience with the system. As a result, all of the participants agreed to take the survey at a later date and the next section will discuss the findings that were accumulated after analysing the answers that were given by everyone.

5.2 Survey Findings

For the purpose of this paper we will go over each question that was addressed in the feedback survey (Appendix, Figure B.1) and present the conclusions and overall impressions that were extracted after reading through the supplied answers.

Question 1: Have you ever used other types of availble Soft Keyboards on the market (SwiftKey, Swype, Minuum, Flesky etc.)? If so, how did you find the process of getting acccustomed with typing on such a keyboard? If not, motivate your choice.

As far as this question is concerned, 8 out of the 10 participants have admitted to having used one of the mentioned products or other similar ones, with the remaining two preferring to use the default keyboard provided with the device (with the motivation being that nowadays, most of the smartphone manufacturers provide quite capable standard services that are equally as good in terms of normal typing and regular corrections). For the second part of the question, the participants who answered positively have had mixed experiences in terms of using the various tools that came along with the keyboards. While most appreciated the effort of trying to introduce novelty in the experience of typing , 2 participants considered the layout of Minuum in particular to be difficult to get accustomed with due to its very peculiar layout which took a lot of time to grow familiar and proficient with.

Question 2: As a follow up to the previous question, how did you find using the TPV Keyboard and how feasible do you think this approach is?

For this question all of the participants agreed that it is a good concept but that it needs some refinement in order to make it feasible for everyday typing. Given the fact that the layout is one that they were familiar with, they did not find it hard to perform basic text input but what took more to get accustomed was the timing of having to hold down on a key in order to enforce that to be the result.

Question 3: Lastly, if you were to offer some suggestions that could improve the look and feel of the TPV Keyboard, what would those be? If you could provide some reasoning with your answer as well, this would be greatly appreciated.

As expected this question provided the highest amount of insight into what could be improved and/or done differently in terms of this approach. Most of the suggestions will be detailed into the following chapter, dedicated to future improvements. However, here are some of the more interesting answers:

- reduce area that is being highlighted since colouring the entire keyboard can be distracting at time and/or redundant
- provide a potential words view, similar to the ones which most of the products nowadays will provide
- provide different ways of highlighting, and not necessarily colour themes but more in terms of dimming the letters that are less probable and increasing the brightness of the ones that are

Chapter 6

Future Improvements

6.1 Appearance

In terms of appearance the first improvement that could be considered is offering various colour palettes that the prospective users could choose from in order to enable persons that suffer from colour blindness to enjoy the same experience as everyone else (as suggested following the feedback received when investigating the design related aspects of the project in Chapter 3, Section 3.4).

Another improvement that could be implemented would be to provide the users with a choice of selecting whether they prefer to have the highlighting distributed over all the keys or if they would like the highlighting area to be more restricted (not necessarily referring to the area around the key that the highlighting would be performed on but rather limiting the number of levels that we would compute this for, such as only showing the most suitable candidate and possibly one more level).

6.2 Performance

The first, most obvious improvement that would greatly benefit the application would be to take advantage of the ever increasing power of mobile devices and separate the different types of operations (UI rendering, probability computation being the most important ones) and have them running on distinct threads in order to eliminate the slight latency between the moment in which the tap occurs and the instance when the highlights start to show.

The second adaptation that could be done as part of a future extension would be to introduce the concept of touch calibration, similar to the way in which Weir et al. [27] have implemented it. A very simple outline would consist of the users having to type a number of sentences in order to ensure that the entire surface of the keys is covered and that proper data is acquired in terms of touch probabilities. This would eliminate the need of constantly having to compute these values as we are typing and would probably lead to major improvements in terms of the speed at which the highlighting will be generated.

Lastly, another aspect that could be improved is the language model and the way in which we obtain the probabilities from it. The system could be modified in such a way that, given the appropriate adaptations, it could make use of one of the many available frameworks for handling language models that are available on the market (such as the SRI Language Modelling Toolkit [20], CMUSphinx [26]).

6.3 Usability

In order to improve the look and feel of the overall product, a suggested words view could be added on top of the keyboard, managing to supplement the information provided by the highlighting and at the same time, provide an alternative for the users who are having a hard time getting accustomed to the idea of seeing the results of the prediction presented in a graphical way rather than a textual manner.

Refering again to the study created by Weir et al. [27], one of the conclusions that were drawn during the testing of their GPType system, was that users are rather accurate at realising when the correction system will return a wrong result. Based on that observation, another extension that could be added to the TPV Keyboard would be to display what the language model will predict when nothing is being pressed. This would allow the users to have know in advance whether a tap or a long tap would be necessary without having to adapt during the press itself.

Chapter 7

Conclusion

7.1 Summary

The main goal of the project was to develop a custom Android keyboard service that would allow its users to visualize what the correction system has determined in terms of what the next prediction should be. The process as a whole included two experimental implementations (coordinate manipulation - Chapter 4, Section 4.2 and fat finger approach - Chapter 4, Section 4.5) which were rather useful in determining how some of the functionality should be created. The rest of the development period was rather challenging at times due to having to adapt to the way the system performed during the numerous testing sessions and find the most appropriate solutions to some of the behaviours that were observed (determining the best value for variance in - Chapter 4, Section 4.7 as well as having to give the touch model much more weight than initially anticipated for obtaining the right result - Chapter 4, Section 4.10).

Following the usability testing and after taking into account some of the feedback provided as a result of it, the author is rather confident that the final product has managed to implement most of the identified requirements in Chapter 2 of this paper. Even though some very intriguing features surfaced as possible improvements and were not able to be implemented during this development cycle, the product has great potential to be polished into a great service for Android users.

7.2 Lessons learned

Starting from the idea that the author's knowledge in terms of Android development at the beginning of the project was rather limited, the entire development process has been a great learning experience and it has revealed new interests in areas that are continually evolving (language modelling and Android development). The author is now confident that tackling future issues related to these subject will only make it so that his experience with the Android platform will increase gradually as he will be exposed to more opportunities.

Bibliography

[1] Flesky - The fastest, most customizable keyboard. http://fleksy.com/. [Online; accessed 24-March-2015].

[2] Minuum - Type anywhere. http://minuum.com/. [Online; accessed 24-March-2015].

[3] Swift Key - The keyboard that learns from you. http://swiftkey.com/en/. [Online; accessed 24-March-2015].

[4] Swype - Type fast, Swype faster. http://www.swype.com/. [Online; accessed 24-March-2015].

[5] Dictionary of words – Size: 58k words. http://www.mieliestronk.com/corncob_lowercase.txt, 2015. [Online; accessed 24-March-2015].

[6] Apple. Developer Apple – Human Interface Guidelines, 2015. [Online; accessed 24-March-2015].

[7] Edward C Clarkson, Shwetak N Patel, Jeffrey S Pierce, and Gregory D Abowd. Exploring continuous pressure input for mobile phones. *Proceedings of ACM UIST 2005*, 2006.

[8] DazeInfo. 2 billion smartphone users by 2015 : 83mobiles [study]. http://dazeinfo.com/2014/01/23/smartphone-users-growth-mobile-internet-2014-2017/, 2015. [Online; accessed 24-March-2015].

[9] Leslie Foster. Dictionary of words – Size: 350k words. http://www.math.sjsu.edu/ foster/dictionary.txt, 2015. [Online; accessed 24-March-2015].

[10] Google. Developer Android – Accessing Resources, 2015. [Online; accessed 21-March-2015].

[11] Google. Developer Android – Canvas. http://developer.android.com/reference/android/graphics/Canvas.htmll, 2015. [Online; accessed 24-March-2015].

[12] Google. Developer Android – Creating an InputMethod, 2015. [Online; accessed 24-March-2015].

[13] Google. Developer Android – Keyboard. http://developer.android.com/reference/android/inputmethodservice/Keyboard.html, 2015. [Online; accessed 24-March-2015].

[14] Google. Developer Android – NinePatchDrawable, 2015. [Online; accessed 21-March-2015].

[15] Google. Developer Android – ShapeDrawable, 2015. [Online; accessed 21-March-2015].

[16] Google. Developer Android – TextView. http://developer.android.com/reference/android/widget/TextView.html, 2015. [Online; accessed 24-March-2015].

[17] Google. Developer Android – Touch feedback. http://developer.android.com/design/style/touch-feedback.html, 2015. [Online; accessed 24-March-2015].

[18] Google. Developer Android – View. http://developer.android.com/reference/android/view/View.html, 2015. [Online; accessed 24-March-2015].

[19] Alexander Hoffmann, Daniel Spelmezan, and Jan Borchers. Typeright: A keyboard with tactile error prevention. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 2265–2268, New York, NY, USA, 2009. ACM.

[20] SRI International. SRILM — SRI Language Modelling Toolkit. http://www.speech.sri.com/projects/srilm/, 2015. [Online; accessed 25-March-2015].

[21] David C. McCallum, Edward Mak, Pourang Irani, and Sriram Subramanian. Pressuretext: Pressure input for mobile phone text entry. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '09, pages 4519–4524, New York, NY, USA, 2009. ACM.

[22] Microsoft. Windows – touch interactions for windows. https://msdn.microsoft.com/en-us/library/windows/apps/hh465415.aspx, 2015. [Online; accessed 24-March-2015].

[23] Oracle. Java – ArrayList. http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html, 2015. [Online; accessed 24-March-2015].

[24] Oracle. Java – Scanner. http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html, 2015. [Online; accessed 24-March-2015].

[25] Maarten Pennings. Android Development – Custom Keyboard. http://fampennings.nl/maarten/android/09keyboard/index.htm, 2015. [Online; accessed 24-March-2015].

[26] Carnegie Mellon University. CMU Sphinx — Building a Language Model. http://cmusphinx.sourceforge.net/wiki/tutoriallm, 2015. [Online; accessed 25-March-2015].

[27] Daryl Weir, Henning Pohl, Simon Rogers, Keith Vertanen, and Per Ola Kristensson. Uncertain text entry on mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2307–2316, New York, NY, USA, 2014. ACM.

[28] Wikipedia. Android (operating system) — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Android_(operating_system)&oldid=653292427, 2015. [Online; accessed 21-March-2015].

[29] Wikipedia. Capacitive sensing — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Capacitive_sensing&oldid=646818415, 2015. [Online; accessed 24-March-2015].

[30] Wikipedia. Gaussian function — wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Gaussian_function, 2015. [Online; accessed 24-March-2015].

[31] Wikipedia. Gaussian process — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Gaussian_process&oldid=651885788, 2015. [Online; accessed 22-March-2015].

[32] Wikipedia. Ios — wikipedia, the free encyclopedia, 2015. [Online; accessed 21-March-2015].

[33] Wikipedia. Language model — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Language_model&oldid=649108617, 2015. [Online; accessed 21-March-2015].

[34] Wikipedia. Moscow method — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=MoSCoW_method&oldid=647183815, 2015. [Online; accessed 22-March-2015].

[35] Wikipedia. Resistive touchscreen — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Resistive_touchscreen&oldid=643499178, 2015. [Online; accessed 21-March-2015].

[36] Wikipedia. Swing (java) — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Swing_(Java)&oldid=646477250, 2015. [Online; accessed 24-March-2015].

[37] Wikipedia. Windows phone — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Windows_Phone&oldid=651182438, 2015. [Online; accessed 21-March-2015].

Appendix A

Full size diagrams

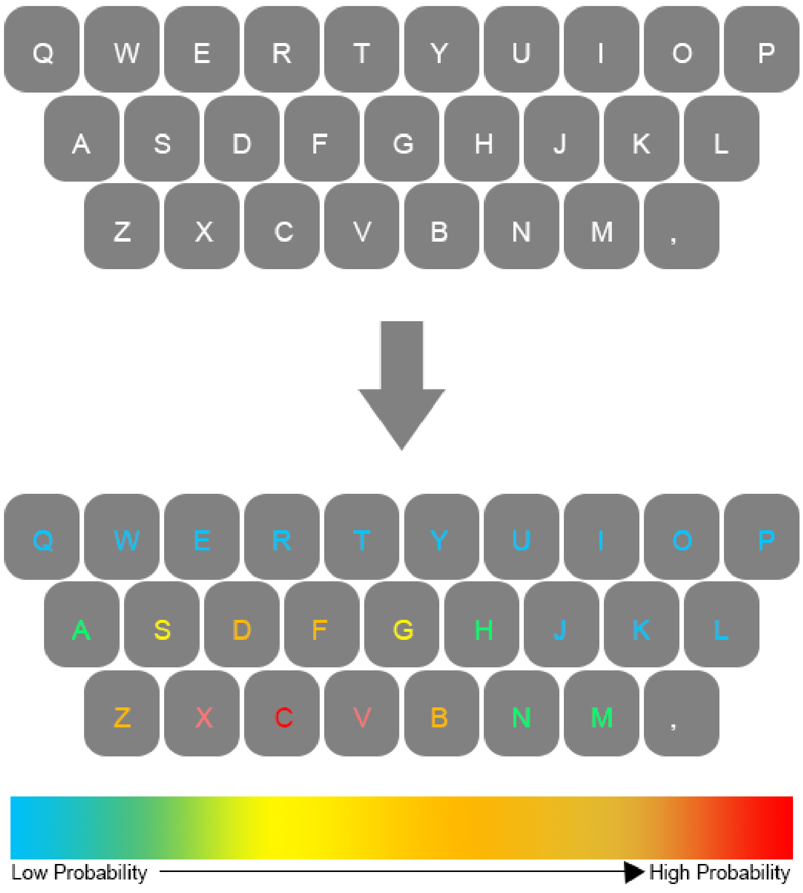


Figure A.1: Diagram showing highlighting using labels

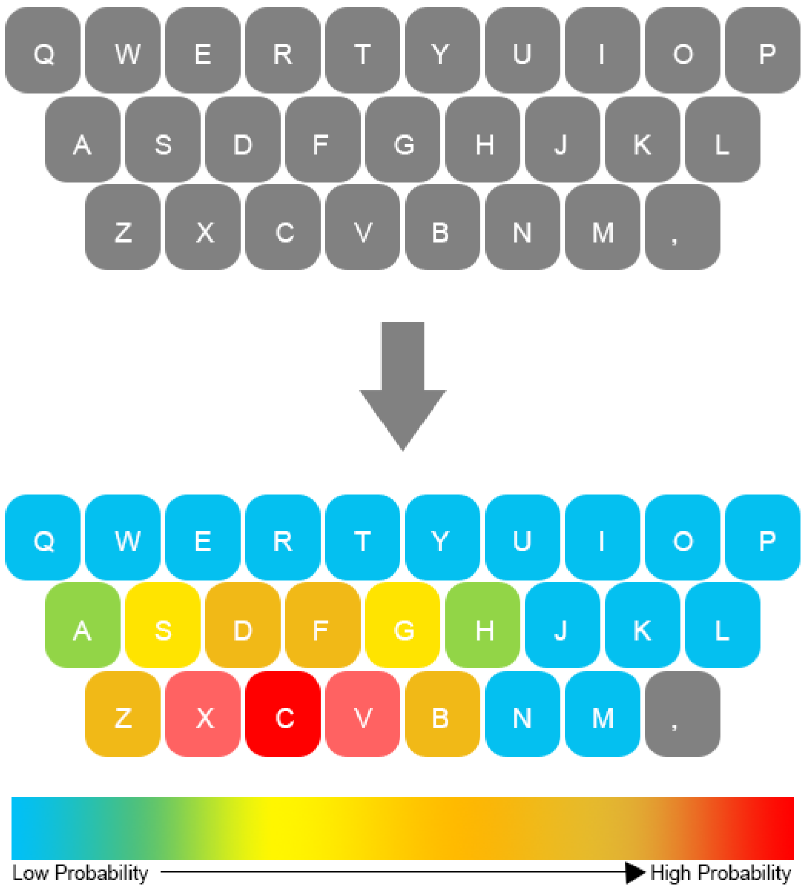


Figure A.2: Diagram showing highlighting using key background

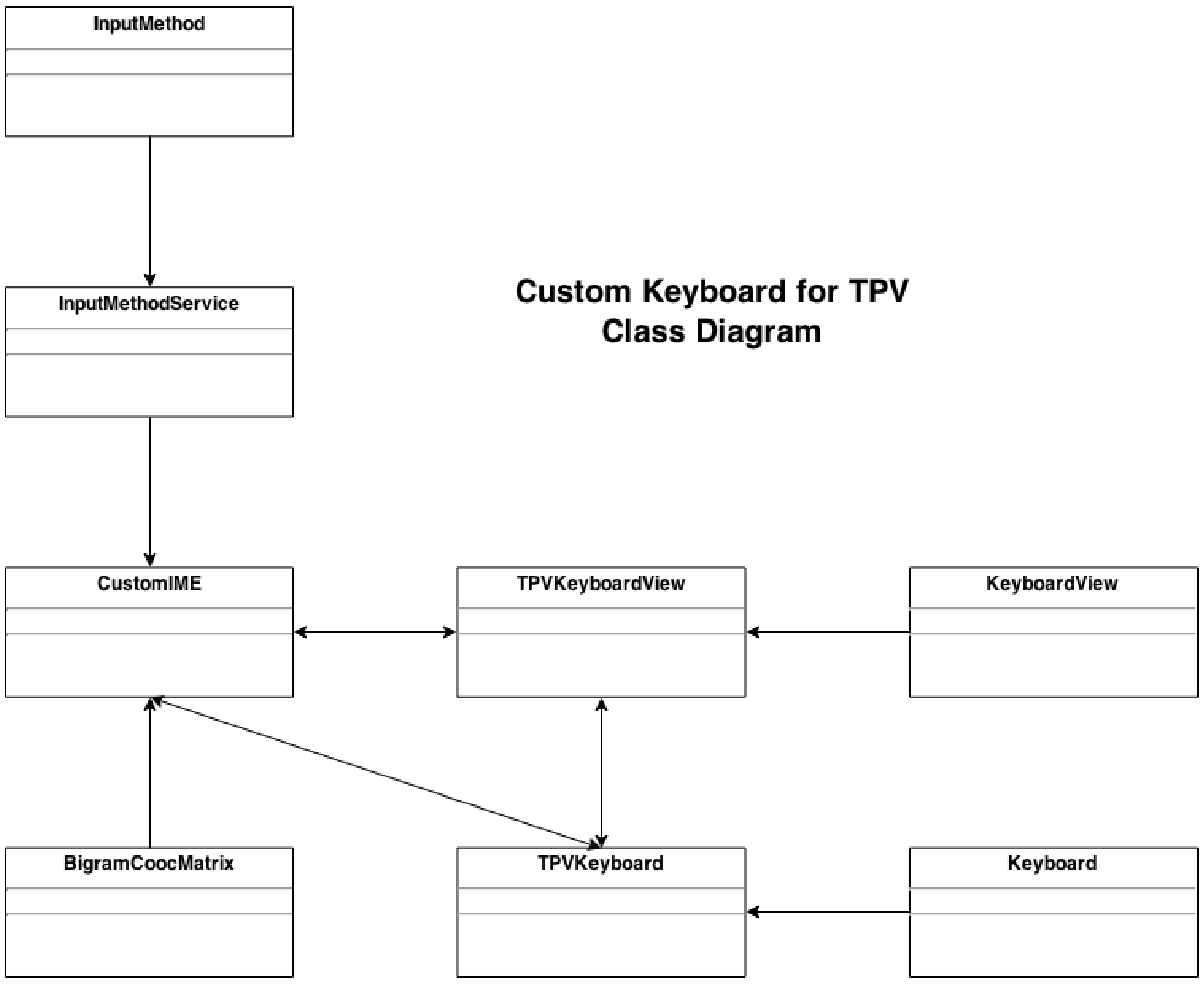


Figure A.3: TPV Class Diagram for TPV Keyboard

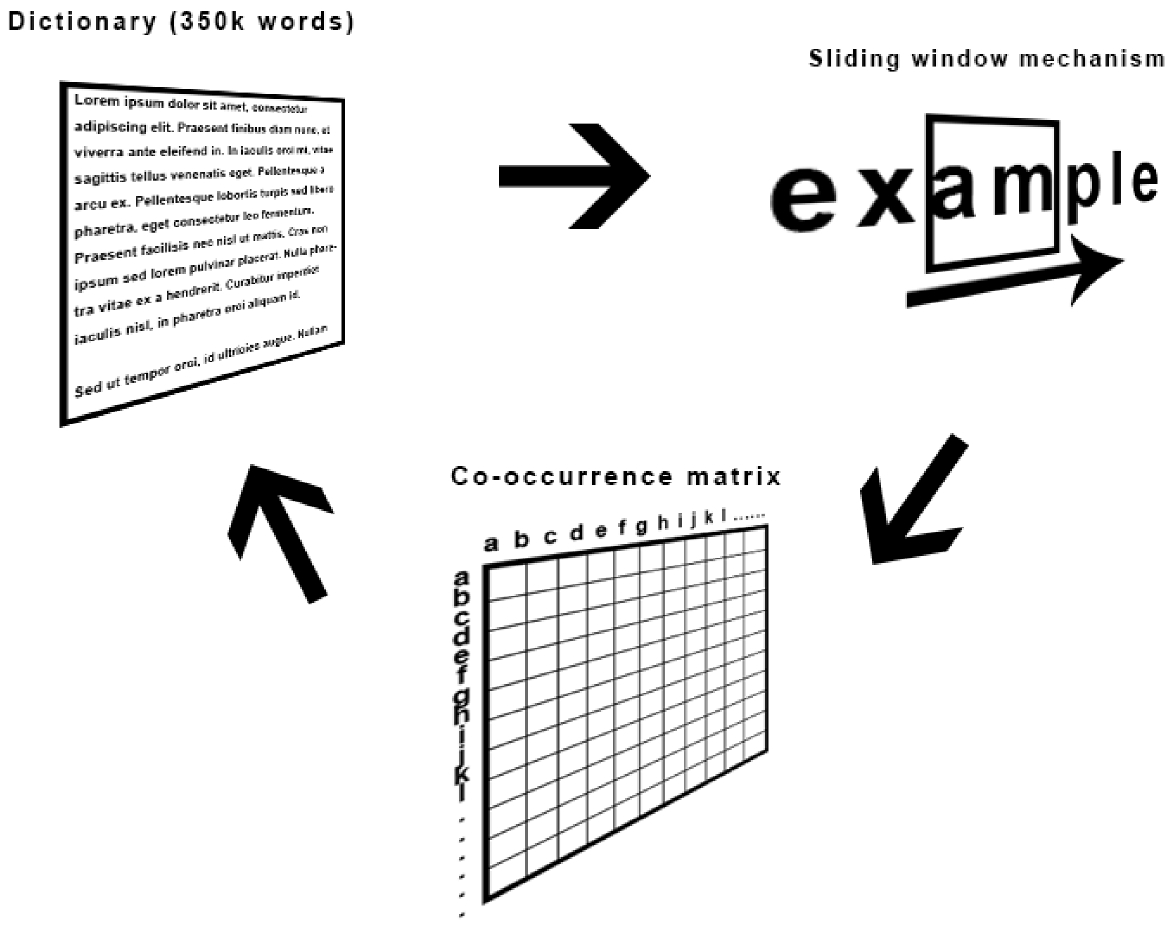


Figure A.4: Language model probability generation process

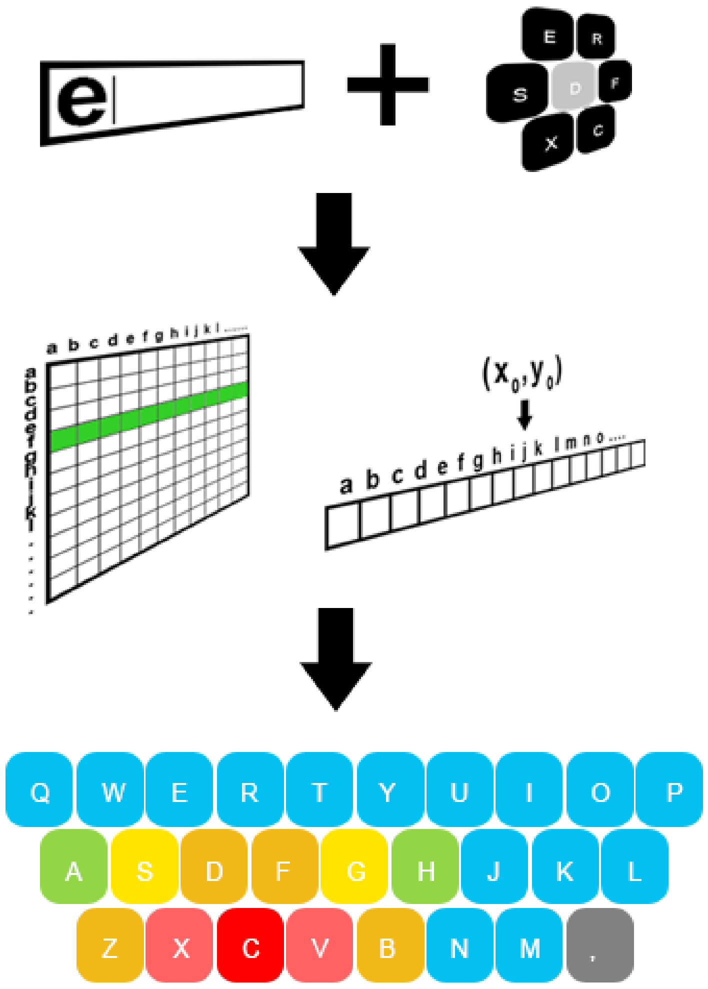
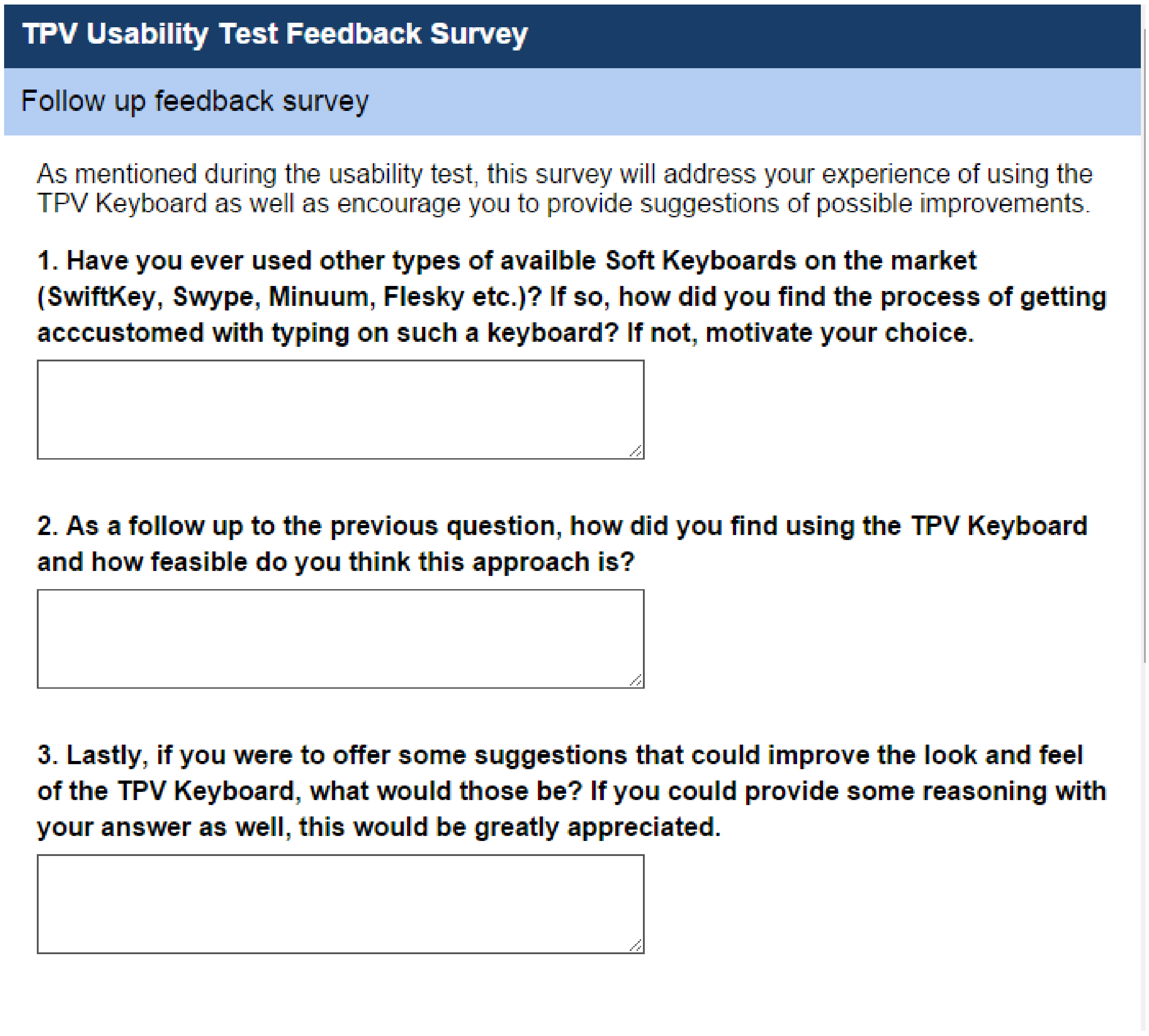


Figure A.5: Large diagram showing the steps of the highlight process

Appendix B

Usability Test Feedback Survey



TPV Usability Test Feedback Survey

Follow up feedback survey

As mentioned during the usability test, this survey will address your experience of using the TPV Keyboard as well as encourage you to provide suggestions of possible improvements.

1. Have you ever used other types of availble Soft Keyboards on the market (SwiftKey, Swype, Minuum, Flesky etc.)? If so, how did you find the process of getting acccustomed with typing on such a keyboard? If not, motivate your choice.

2. As a follow up to the previous question, how did you find using the TPV Keyboard and how feasible do you think this approach is?

3. Lastly, if you were to offer some suggestions that could improve the look and feel of the TPV Keyboard, what would those be? If you could provide some reasoning with your answer as well, this would be greatly appreciated.

Figure B.1: Example of the feedback survey that was presented to the participants

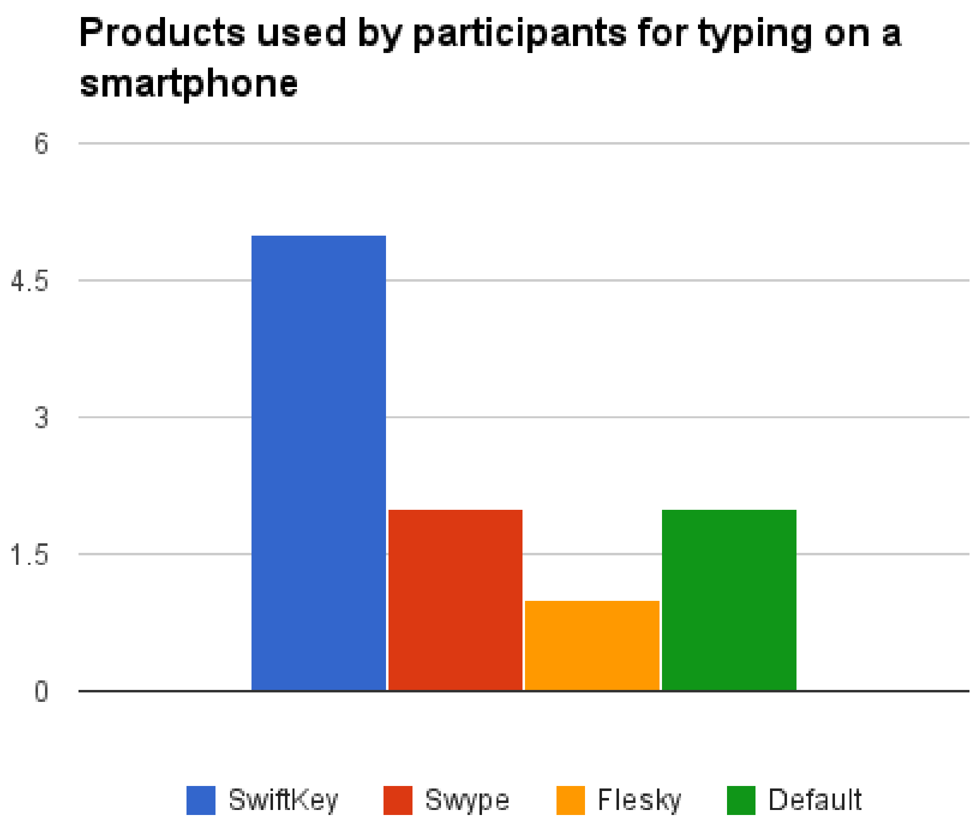


Figure B.2: Question 1 results chart