KEYBOARD MAPPING AND FONT RENDERING TECHNIQUES FOR NON-LATIN LANGUAGES CASE OF ANDROID MOBILE PHONES

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Dhruv Bhasin
Spring 2011
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the
Thesis of Dhruv Bhasin:

Keyboard Mapping and Font Rendering Techniques for Non-Latin Languages
Case of Android Mobile Phones

Joseph Lewis, Chair
Department of Computer Science

Tao Xie
Department of Computer Science

Samuel Kassegne
Department of Mechanical Engineering

11/16/2010
Approval Date
DEDICATION

I dedicate my thesis work to my Dad, it won’t have been possible without his support and faith in me whenever I needed it. Thank you Dad.
ABSTRACT OF THE THESIS

Keyboard Mapping and Font Rendering Techniques for Non-Latin Languages Case of Android Mobile Phones

by

Dhruv Bhasin
Master of Science in Computer Science
San Diego State University, 2011

The use of native language for sharing views in writing has experienced explosive growth around the world in the past several years. This has been the case because of the rapid growth of Internet services round the globe. In developing countries in particular, internet and wireless mobile services have proved to offer a unique opportunity to service a wider subscriber base in the shortest possible time previously deemed unattainable. In addition to wireless voice, wireless data services have proved time and again to be of immense value to subscribers as well as wireless operators. Among these valuable wireless data services, now-a-days social networking based websites have proved to be an asset to the users for sharing their thoughts and ideas. There has been a growth in the software for offering the users to share content based on their native language. Now most of the Mobile operating systems offer Latin language support through which the user can input text in Latin languages, these include recently added Japanese, Chinese, Korean, etc. For the users of Non-Latin languages like Devanagari script and writing system, no input method solution has existed so far in Android Mobiles.

Therefore, there is a very visible and pressing need to introduce text editor in either Devanagari script or in a multi-script (Latin and Non-Latin both) environment specifically for the mobile smart phone platforms. Developing such an application, however, requires the scholarly investigation and eventual solution of a number of technical challenges. These challenges vary from developing an innovative keyboard board mapping for the more than 110 characters of the Devanagari writing system onto a 70-key virtual keypad to writing, a completely new input method editor (IME) for Devanagari script in a Java environment. Further, this will also involve addressing issues such as Unicode compatible standards and efficient character encoding schemes. It involves the study of Non-Latin script rendering techniques used in conjunction with the IME. It also includes the script sharing engine which detects the use of Devanagari script characters and read them correctly on the mobile platform.

This thesis, therefore, is an original and formal scholarly investigation of these technical and script challenges to develop a working and testable Input Method Editor and Non-Latin script sharing applications in the Devanagari writing system. In this thesis, the core subject matter is addressed by providing additional detailed discussion to relevant technologies such as the Android application development environment (purely based on Java) and Unicode standards.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT .................................................................................................................. v</td>
</tr>
<tr>
<td>LIST OF FIGURES........................................................................................................ viii</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS............................................................................................... ix</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS..................................................................................................... x</td>
</tr>
<tr>
<td>CHAPTER</td>
</tr>
<tr>
<td>1 INTRODUCTION ........................................................................................................... 1</td>
</tr>
<tr>
<td>1.1 Need for Non-Latin Script Support on Mobile Platforms.................................... 2</td>
</tr>
<tr>
<td>1.2 Android Application Framework ........................................................................... 3</td>
</tr>
<tr>
<td>1.3 Non-Latin Language Support on Android Platform ............................................. 5</td>
</tr>
<tr>
<td>2 IMPLEMENTATION ..................................................................................................... 8</td>
</tr>
<tr>
<td>2.1 Input Method Editor ............................................................................................... 8</td>
</tr>
<tr>
<td>2.1.1 Editor Control ................................................................................................... 11</td>
</tr>
<tr>
<td>2.1.2 CandidateView ................................................................................................. 12</td>
</tr>
<tr>
<td>2.1.3 InputMethodService ......................................................................................... 13</td>
</tr>
<tr>
<td>2.1.4 KeyEvents ....................................................................................................... 13</td>
</tr>
<tr>
<td>2.1.5 Keyboard Layout .............................................................................................. 14</td>
</tr>
<tr>
<td>2.1.6 Decimal Code Points ....................................................................................... 16</td>
</tr>
<tr>
<td>2.1.7 Typeface Font Support .................................................................................... 17</td>
</tr>
<tr>
<td>2.2 Font Rendering Engine ......................................................................................... 18</td>
</tr>
<tr>
<td>2.2.1 Key Event Rendering ....................................................................................... 18</td>
</tr>
<tr>
<td>2.2.2 Half Character Rendering ............................................................................... 22</td>
</tr>
<tr>
<td>2.2.3 Double-Half Character Rendering ................................................................ 23</td>
</tr>
<tr>
<td>2.2.4 Devanagari Script Rules ................................................................................ 24</td>
</tr>
<tr>
<td>2.3 Font to Unicode Rendering Engine .................................................................... 28</td>
</tr>
<tr>
<td>2.4 REST API ............................................................................................................. 30</td>
</tr>
<tr>
<td>2.5 Script Detection Engine ..................................................................................... 31</td>
</tr>
</tbody>
</table>
3 RESULTS DEMONSTRATION ........................................................................................................... 33

3.1 Importance of Hindi Language Android Mobile Application .................................................. 33
3.2 Use of Facebook API and Facebook Android sdk ................................................................. 34
3.3 Change of Facebook’s Profile Status Message in Hindi Language ........................................ 35
3.4 News Feed and Wall Feed from Facebook ............................................................................ 36
3.5 Future Work in the Field ....................................................................................................... 37
3.6 Limitations ............................................................................................................................ 37
3.7 Screenshots for Result Demonstration .................................................................................. 38

4 CONCLUSION ................................................................................................................................. 40

BIBLIOGRAPHY .............................................................................................................................. 42
LIST OF FIGURES

PAGE

Figure 2.1. Standard ITU telephone keypad................................................................. 9
Figure 2.2. iPhone IME – standard English virtual keypad. ........................................ 9
Figure 2.3. Android IME – standard English virtual keypad. .................................... 10
Figure 2.4. Class diagram for input method editor for Devanagari.............................. 11
Figure 2.5. Devanagari Inscript bilingual Keyboard.................................................. 15
Figure 2.6. Hindi keyboard IME - Android mobile. ......................................................... 15
Figure 2.7. Hindi keyboard IME second page - Android mobile. ............................... 16
Figure 2.8. Functions to be performed by Font Rendering Engine.............................. 19
Figure 2.9. Main Font Rendering Engine is composed of three sub-rendering engines.. 19
Figure 2.10. Vowel letters in Devanagari. S............................................................... 25
Figure 3.1. Hindi Facebook application login page on the application......................... 38
Figure 3.2. Hindi Facebook application main page and profile page. .......................... 39
Figure 3.3. Hindi Facebook login news feed page and typing status message............. 39
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>DCP</td>
<td>Decimal Code Points</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>IME</td>
<td>Input Method Editor</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PCE</td>
<td>Percentage Character Encoding</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>ZWJ</td>
<td>Zero Width Joiner</td>
</tr>
<tr>
<td>ZWNJ</td>
<td>Zero Width Non-Joiner</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

This thesis study could not have been completed without the help of my advisors, Dr. Samuel Kassegne and Dr. Joseph Lewis who offered me a lot of encouragement, guidance, and continued supervision. Dr Samuel and I have worked on various different strategies to deal with various problems that came by my way in designing the rendering engine. He was constantly encouraging and discussing with me the contents of my thesis even by making telephone calls and sharing views through SMS, in addition to the response and advises he was giving me within unexpectedly short time intervals. In addition to the numerous advices he offered me for this thesis, Dr. Joseph Lewis has also been helping me a lot in my academic program. Therefore, I would like to thank both of my advisors a lot.
CHAPTER 1

INTRODUCTION

The main objective of this thesis is to investigate the development of a framework for the development of a generic Mobile Input method Editor (IME) that provides support for input of Non-Latin scripts in mobile platforms like Devanagari script and to develop a rendering engine that renders the characters from the IME following the rules of Devanagari Unicode character formation. Another aim of the thesis is to investigate a method of recognition of Devanagari script characters when they are received from other third party applications and render them back on the mobile phones using their native font. The purpose of the whole thesis is to build a rendering engine that supports and renders Devanagari characters on mobile phone platform. The representation of the same would be based on the Android Mobile Platform. The algorithm for the rendering engine is platform independent and can be used on any mobile platform that supports Java.

The development of such an application requires considerations and original scholarly contributions in the following major areas:

1. Designing and implementing a virtual keyboard mapping for Devanagari script based on Unicode standard,
2. Implementing the best efficient mobile text inputting method for Devanagari (i.e. Input Method Editors – IME),
3. Implementing character encoding standards suitable for binary data transmission in wireless network,
4. Addressing multi-script functionality for the application,
5. Creation of an Algorithm to follow the rules in Devanagari script as per the Unicode standards for character formation,
6. Solving the message sending and receiving functionality for the application such as calling the REST API for social networking platforms for sending and receiving Devanagari characters,
7. Formulation of a Script Recognition Engine that checks for the Devanagari characters when received from the third party applications and render them correctly on the mobile phones.
In subsequent chapters, these topics will be thoroughly and critically investigated as the case for a functional multi-script text sharing application is developed.

In this introductory chapter, we consider the historical development of applications that support Non-Latin script on mobile platforms. Wherever applicable, we support the discussion on the developments in these areas with published data collected over the past several years.

### 1.1 Need for Non-Latin Script Support on Mobile Platforms

Now-a-days mobile applications serve an integral part of the huge mobile market. These mobile applications now serve and perform some common user common tasks such as viewing emails, checking up social network updates, financial applications, games, movies, entertainment, search, maps, local business, etc. People want to share their common information like what’s the thought on their mind at the moment, what’s the current place they are checked in right now, what kind of entertainment they like, etc. They want to remain connected based on their likes and dislikes. Everything they ever shared in any forum or any community now everything is on the go. People interact with each other based on the communities they share or are a part of. People love to share that information with each other in the native languages they speak. These languages could be Latin as well as Non-Latin based on the part of the world they belong to.

Now all the smart phone users connect with one another using their mobile phones on the go. Everyone is registered with some social network with which they connect with their friends. They share audio, video, text, articles, links, etc. over that network. Some of the shared audio or video or other form of multimedia is in their native language also. They also share links and articles over the internet that has other forms of multimedia in their native language. The only thing that is lacking is most of the times they want to share text written in their native languages and they are unable to do that. Some of the mobile platforms like iPhone does provide support for using a different keyboard and enter text in other Latin languages like Chinese, Spanish, and Korean, etc. But still it lacks support for Non-Latin language keyboards as an input method. Some mobile platforms like Android do not even support the Non-Latin font. Either people have to supply the font with the application or have
to hack the OS to add a different version of the true type font that is already included in the android OS.

Some of the Non-Latin languages rank among the top 10 languages being spoken worldwide. These languages include Hindi/Urdu which rank 4th amongst the spoken languages across the world in terms of the native speakers. There are about 182 million speakers of Hindi language and 60 million Urdu language speakers as compared to 328 million native speakers of English [1]. Another prospective of that is Niche Marketing of the mobile products. “Niche marketing is a common path to small business success. No large retailer can be all things to all people, and there are always going to be segments of the population whose needs for particular products and/or services are going unmet. Learn how your small business can capitalize on these unmet needs and find and master your own niche market” [2]. Businesses can get their products closer and useful to the locals by having support for native languages.

Another aspect of it is the brand loyalty. The mobile company that has support for native languages will definitely make people feel more connected to their demography and race. For example, India being a 700 million mobile market is already having a stiff completion among the various mobile products. This shows that it has a very mature mobile market and which has very limited success rate for new products. If any mobile product supports their native language (like Hindi which is their mother tongue) will definitely occupy a good place in the market. For new mobile operating systems such as Android, it’s a good opportunity to incorporate the support for input method for native languages to have some heads up in the market. For entering in such a market Android has to have an incentive that it provides to the users that they shift to their brand. And this could be their first mover advantage over the other wide range of products.

1.2 Android Application Framework

Android is an open application development mobile platform from Google. It supports Java programming language as the base language for the developers to build their applications on top of the Android operating system. The developers have free access to the mobile phones resources through the OS and can perform and create any top level applications that make use of the underlying hardware. It also provides very good and
interactive access to various Google technologies like Location based services, GPS and search facilities.

The Android platform is very robust in nature. The developers are free to build top-notch high end graphic applications for the Android mobile phones. Android has a solid Memory management foundation over which it has laid all its libraries for the use of developers. Its kernel is actually linux based with a lot more features for memory management added to it. Its runtime environment is known as Dalvik Virtual Management.

Any Android mobile application has various activities. Each android application runs in its own process. It makes use of Intents to transfer from one activity to the other. Intents are also used for making use of or calling any Android OS functionalities. For all these activities, each Android application has to have an Android Manifest XML file which contains information about the activities, the intent filters and the permissions allowed to the application.

Android activity has a lifecycle through which it starts processing. At first, the onCreate() method form any activity is called followed by onStart() method. Each of these functions can be overridden and the application is set to perform the tasks according to the developer requirements. It also has methods such as onPause() and onResume() which handles the tasks when the control from that particular activity is gone and when the control to that activity is returned.

There are four types of components of an Android operating system which are explained as under:

- Activities – An Activity in android is a single page in the application which is visible to the user and the user can traverse from one Activity to another.
- Services – A Service in Android is a part or as a whole an application that runs in the background and provide services to the other applications.
- Broadcast Receivers – A Broadcast receiver as the name suggests is a receiver which waits for some broadcast signal sent by some service, some widget or some other application.
- Content Providers – A content Provider is responsible to access the internal resources of the mobile phone. One can make use of the content providers to store and retrieve data from the Sqlite database.
- SQLite – SQLite serves as the database for the Android mobile phones.
### 1.3 Non-Latin Language Support on Android Platform

Although android mobile phones support most of the Latin languages like Spanish, Korean, etc., it does not have support for the Non-Latin languages. Android mobile phones do not have the font support for any Non-Latin language except for the Chinese. Whenever there is a use of Non-Latin language glyphs like when anyone is browsing and just came across a web page displaying the characters that the Android mobile do not have, a square box is displayed instead of that glyph. So, there is a need to add multi-language support on the Android platform. This thesis represents a way of introducing Hindi language support on the Android mobile phones which is based on Devanagari scripts.

One of the ways of adding the support for Indic scripts on the Android mobile phone is to hack the root and change the system font true type file of the Android system. This can be done by using the adb tool from the Android sdk and connecting your mobile to the computer system with a data cable. This way although is a lot more efficient because it change the system font completely and the users can view the correct font glyphs on their browsers instead of the square boxes. This rooting for example, can be done on a Droid phone (an Android phone) by replacing the DroidSans.ttf file of the Android OS. This file can be found in the /System/Fonts section of the phone. The file that replaces it should also include that particular language glyphs that you want to entertain. These glyphs have to Unicode based so that one doesn’t have to deal much with the transformation of Unicode glyphs shown on the browsers than that on the mobile phone itself. But the system still doesn’t have an IME to push in those characters as an input to the system. Another thing that would still be missing in this manner would be the rendering engine which is the core requirement for any language input system. Since both these requirements are still absent and is illogical for a general user to root the phone to add the fonts, we cannot use it anyway.

Another way to do it could be to use bitmaps as a part of the Android mobile application. These bitmaps should include all the glyphs for a specific non-Latin language or may be multiple-languages. Then a specific algorithm parses through the bitmap to find the exact location of the glyph on the bitmap and crop the bitmap to get the glyph. This glyph can then further be added instead of those squares that are displayed on the browser or wherever those scripts are found. There also has to be some IME to parse through the
bitmaps and get the glyph to present it as a part of the system and display the glyphs on a canvas as an input system for the mobile phone. This method also has its own limitations like it is very hard to debug any fault in the way it is parsing through the bitmaps. There has to be a different glyph as a part of the bitmap with a different font size instead of having it ready as true type font. Now these glyphs that are parsed through a bitmap will also not be in Unicode format so there has to be certain algorithms that parse through the Unicode glyphs available on the internet than that on the phone. Also while sharing anything it has to have to be converted to Unicode for the users to view it.

While researching on the methods that could be used for the Android mobile phones to support such glyphs I came across the method of including the font file in the application itself on the Android mobile phones. Android mobile phones do support any font file to be used by the developer as a part of the application. Now, one way of doing it could be possibly to include the font file in the application that presents the user with the correct glyphs. Another thing that needs to be done with this theory is to have an IME that can be used with that application to input text in any particular language. There is a rendering engine that runs in the background which follows the rules of the Unicode font rendering and changes the inputted text accordingly. Now when the whole of the text has been entered that the user would want to share over the internet with any third party application, since the inputted text is font based as a part of the application on the Android phones, it has to be mapped to the Unicode to share it with the outside world. Then there has to be a rendering engine to convert that text to the Unicode format. Next, whenever any such text is detected when receiving form any third party application that has Unicode support, it has to render it back to the application font for the user to view it. Though it is a lengthy process of getting through to the required output, it offers a complete solution which provides full language support on the mobile phone.

Now, In this thesis I had to chose one of the ways to provide a solution that would provide the users with full support for Hindi Language with Devanagari script as a presentation which then could be followed by other languages as well on other mobile platforms as well. For the reason of providing full support for the language, I chose to go with the third option of using a particular font as per the application and building up a whole rendering engine over it. The solution is language and mobile platform independent. It can
also be used with any third party applications for data sharing purposes. In this solution, I have introduced with a rendering engine that does all of the following three tasks:

- It has an IME that is an editor which is a virtual keypad for entering Hindi characters onto the application.
- A rendering engine that renders the text entered in accordance with the Unicode rules.
- It maps the font specific character glyphs to Unicode to be sent to outside applications.
- It checks to the presence of those particular Unicode glyphs in any third party applications.
- It also converts back the detected Unicode glyphs from the third party applications and converts it to the application specific font for the user to view on the mobile phone.

Following all these steps, the thesis offer a complete solution for introducing non-Latin script support on the mobile phones.
CHAPTER 2

IMPLEMENTATION

2.1 INPUT METHOD EDITOR

Input Method editor also known as the IME is the method by which the users input text on the mobile phones. Each mobile device has a different perspective of layouts for the IME they use. For a general mobile IME we are considering a general English language IME. The input method for mobile phones vary a lot based on the mobile manufacturer, technology used, cost of the handset, and screen size. These input methods may also be differentiated based on the type of editor, which is hardware or software.

The hardware based input method is the one which has physical keys present on the mobile phone which is mapped to the mobile operating system to get the particular character in accordance to the key that is pressed. The other one is Software based input method; it is also termed as a virtual keyboard. It is an input method that pops up on the screen of the mobile phones and the user has the option to punch in the characters. Now this software keyboard can be distinguished in regards to the screen type of the mobile phones. If the screen is a touch screen, the user simply has an option of punching the keys on the virtual keyboard directly on the mobile screen. In the other case, it becomes difficult for the user to use a virtual keyboard which they have to operate by using the physical arrow keys on the mobile phone to reach out any character on the virtual keyboard to select it. Some of the examples of such IMEs on different mobile phones are shown Figure 2.1 [3].

This one is the most common IME hardware input method used in most number of mobile handsets across the word. It gives a 12 key mapping to the 101/112 key QUERTY keyboard. Most of the mobile handsets with this mapping do not support the use of applications or any change to the current mobile OS. Some of the handsets use J2ME jar applications that could be incorporated as any third party applications to serve specific requirements for the users.

Figure 2.2 shows the iPhone IME which is a virtual keypad. For selection of a different language keypad on the iPhone one has to go to Settings-> General-> Keyboard->
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>abc</td>
<td>def</td>
</tr>
<tr>
<td>ghi</td>
<td>jkl</td>
<td>mno</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>pqr</td>
<td>tuv</td>
<td>wxyz</td>
</tr>
<tr>
<td>*</td>
<td>0</td>
<td>#</td>
</tr>
<tr>
<td>_</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>


Figure 2.2. iPhone IME – standard English virtual keypad.
International Keyboards to select the keyboard the user want to use. After selection of the appropriate keyboard the users can switch between the keyboards by selecting the globe sign on the keyboard as shown in Figure 2.2.

Figure 2.3 shows the Android IME which is also a virtual keypad for Android based smart phones. For selection of a different language or a different format custom keypad in android one can go to Settings-> Locale & Text-> Keyboard selection. After selection of the appropriate keyboard the users can use the selected keyboard by long press on any text box-> Input Method-> Keyboard selection. And then the newly selected keypad will pop up to be used with the text box. Android is also one of the smart phones that offer touch screens for the use of any virtual keypad.

![Android IME – standard English virtual keypad.](image)

Since we have chosen to demonstrate the use of the rendering engine with the Android Mobile phones, we’ll discuss the in depth input method architecture on the Android mobile phones in more detail. The basic Android Input Method Architecture can be represented by the Figure 2.4.
Figure 2.4. Class diagram for input method editor for Devanagari.

2.1.1 Editor Control

The editor control represents the main class for the keyboard layout and functioning that controls all the functionality and GUI for the virtual keyboard. It calls the super class Keyboard from the Android library which keeps the track of Keyboard layout XML file. It is responsible for loading the XML data of the keyboard which shows how the keyboard is displayed on the screen. It is also responsible for storing the various attributes of the keys on the keyboard. It gives the number of columns, key padding, and character sequences, etc.

The createKeyFromXml() overrides the function of the Keyboard library to customize the key options from the xml. For example, loading the different text and images displayed for the backspace, return and delete keys. It checks the IME options of the current editor, the text box in which we are using the keyboard and in accordance to that modifies the label and text displayed on some of the keys on the keyboard.

The main keyboard layout xml file is something like:

```xml
<Keyboard xmlns:android="http://schemas.android.com/apk/res/android"
```
android:keyWidth="8.333333333%p"
android:horizontalGap="0px"
android:verticalGap="0px"
android:keyHeight="@dimen/key_height"
>

<Row>
<Key android:codes="221" android:keyIcon="@drawable/q2"
android:keyEdgeFlags="left"/>
<Key android:codes="79" android:keyIcon="@drawable/w2"/>

The main Non-Latin keyboard View class looks like:

```java
public class NonLatinKeyboardView extends KeyboardView {
    static final int KEYCODE_OPTIONS = -100;

    public NonLatinKeyboardView(Context context, AttributeSet attrs) {
        super(context, attrs);
    }

    public NonLatinKeyboardView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
    }

    @Override
    protected boolean onLongPress(Key key) {
        if (key.codes[0] == Keyboard.KEYCODE_CANCEL) {
            getOnKeyboardActionListener().onKey(KEYCODE_OPTIONS, null);
            return true;
        } else {
            return super.onLongPress(key);
        }
    }
}
```

2.1.2 CandidateView

The CandidateView class of the editor extends the super class View of the Android library. It inflates the current view of the keyboard layout to customize it by adding certain functionality that includes the behavior of keyboard on scroll events, using the gestures to
print the characters, and other functionality that is currently included for the English keyboard and have future scope with other languages as well. One such customization is addition of suggested word and auto completion of text using Hindi dictionary.

2.1.3 InputMethodService

It includes the standard implementations of an input method. It has functionality for detection of key events and certain key functions whenever a key event occurs. One can override the onKeyUp() and onKeyDown() methods for detection of which key was pressed. One can detect certain key events by calling the KeyEvent class for example, KeyEvent.KEYCODE_BACK detects the input of backspace key. Similarly, KeyEvent.KEYCODE_RETURN detects the use of Return key by the user. One can then process the functionality when each of these events occur. The onInitializeInterface() methods gets instantiated whenever the keyboard is set up for use. It gets call before even the UI objects are created.

2.1.4 KeyEvents

It is a class that has all the constant values for all the keys on the keyboard. This can be used for detection of which key has been pressed and then one can assign the things that need to be done on such an event.

We can Override the key events like onKeyUp() and onKeyDown(), an example of which is as shown below:

```java
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
    switch (keyCode) {
    case KeyEvent.KEYCODE_BACK:
        if (event.getRepeatCount() == 0 && mInputView != null) {
            if (mInputView.handleBack()) {
                return true;
            }
        }
        break;
    case KeyEvent.KEYCODE_DEL:
        if (mComposing.length() > 0) {
            onKey(Keyboard.KEYCODE_DELETE, null);
            return true;
        }
    }
```
case KeyEvent.KEYCODE_ENTER:
    return false;

default:
    if (PROCESS_HARD_KEYS) {
        if (keyCode == KeyEvent.KEYCODE_SPACE &&
            (event.getMetaState() & KeyEvent.META_ALT_ON) != 0) {
            InputConnection ic = getCurrentInputConnection();
            if (ic != null) {

                ic.clearMetaKeyStates(KeyEvent.META_ALT_ON);
                keyDownUp(KeyEvent.KEYCODE_A);
                keyDownUp(KeyEvent.KEYCODE_N);
                keyDownUp(KeyEvent.KEYCODE_D);
                keyDownUp(KeyEvent.KEYCODE_R);
                keyDownUp(KeyEvent.KEYCODE_O);
                keyDownUp(KeyEvent.KEYCODE_I);
                keyDownUp(KeyEvent.KEYCODE_D);
                return true;
            }
        }
    }
    if (mPredictionOn && translateKeyDown(keyCode, event)) {
        return true;
    }
}

2.1.5 Keyboard Layout

The keyboard layout for the Latin English keyboard on the android mobile phone is shown in Figure 2.5 [4]. Now in case of Devanagri script keyboard for Hindi language, the number of keys will vary as per the standard Devanagari keyboard layout. This is also known as Devanagari Inscript bilingual keyboard which is the standard Devanagari script keyboard for the hardware keyboard with the desktop systems. The keyboard layout on the desktop version looks like Figure 2.5.

In a very similar fashion, the IME keyboard that needs to be developed for the support on Android mobile phones had to be in the same format. It does not have some of the Marathi language characters because it was being made specifically for Hindi language
following the rules of the Devanagari script. It is divided among four pages, out of which the first two pages represent the Hindi characters, one in normal mode and the other in caps lock mode. The other 2 pages have some common hindi characters that are formed by the combination of two or more other characters and also has other mathematical symbols.

The keyboard developed in the thesis is as shown in Figures 2.6 and 2.7.
2.1.6 Decimal Code Points

There are various schemes for character encoding through which the browsers or any applications are able to render different language scripts. The commonly used encoding schemes are HTML, UTF-8, Decimal Code Points, and Percentage Encoding of URIs, etc. Decimal Code Point scheme is also one of the most common encoding schemes. Android accept DCP as an input encoding scheme for the keyboard characters. Each character on the keyboard layout is mapped to the Decimal Code point value of that corresponding character.

DCPs are mentioned in the Keyboard layout XML file of the keyboard. It is represented by the attribute android:codes of the key tag in the keyboard layout xml file. The value of each characters DCP is read by the InputMethodService class of the IME. This is then passed on to the onKeyUp() and onKeyDown() methods of the class with a tag name of Primary Codes. These primary codes can then be detected on any key event and then can be rendered accordingly. The DCPs are represented in the code as follows:

```
<Row>
  <Key android:codes="221" android:keyCode="@drawable/q2"
       android:keyEdgeFlags="left"/>
  <Key android:codes="79" android:keyCode="@drawable/w2"/>
</Row>
```
<Key android:codes="97" android:keyIcon="@drawable/e2"/>
<Key android:codes="73" android:keyIcon="@drawable/r2"/>
<Key android:codes="85" android:keyIcon="@drawable/t2"/>
<Key android:codes="37373" android:keyIcon="@drawable/y2"/>
<Key android:codes="104" android:keyIcon="@drawable/u2"/>
<Key android:codes="37374" android:keyIcon="@drawable/i2"/>
<Key android:codes="100" android:keyIcon="@drawable/o2"/>
<Key android:codes="37375" android:keyIcon="@drawable/p2"/>
<Key android:codes="68" android:keyIcon="@drawable/sqbrop2"/>
<Key android:codes="44" android:keyIcon="@drawable/sqbrcl2" android:keyEdgeFlags="right"/>

So here, each of the key value is linked to a drawable that will be placed as the character on the keyboard and the DCP value of that character.

These DCP values are different for the application level font and the Unicode. For the keyboard to be compatible with the rendering engine and application, the application must use the same font as given by the keyboard DCPs. These DCPs will later be mapped to the Unicode characters once they have been rendered properly on the IME side.

### 2.1.7 Typeface Font Support

Android support three types of Typeface fonts – Sans, Serif and Monospace. Android also does have way to add your own Typeface font. We will add Hindi Shusha font to the application side later on. The typeface font that is added by the application must match the DCP values of the IME as given in their Keyboard Layout XML file. We can add the Hindi Shusha Font support to the Application by declaring the font Typeface from the assets of the application and then assigning it to the related Text box. For Example,

```java
final EditText e = (EditText) findViewById(R.id.edittext);
final Typeface face = Typeface.createFromAsset(getAssets(), "fonts/shusha.ttf");
e.setTypeface(face);
```

Now, while we assign the typeface to any text box, we first have to pass it on to the Script detection engine which lets us know if the current language of the text is English or not. The result of that engine is then used to set the language for the text box. The same can be done as follows:

```java
if (inEnglish.get(position) == "false") {
    statusView.setVisibility(0);
}
statusView.setTypeface(face);
statusView.setText(message);
englishStatusView.setVisibility(4);
}
else {
    statusView.setVisibility(4);
    englishStatusView.setVisibility(0);
    englishStatusView.setText(message);
}

2.2 FONT RENDERING ENGINE

The font rendering is the task of including all those characters in the language that are not present on the keyboard but are formed by the combination of one or two other characters. It formulates the rules of combining the characters in the language in accordance with the Unicode standards for that particular language. In our case, there are a certain rules that Devanagari script follows to replace certain characters and form matras that can be placed after or before the other consonant characters.

The basic structure of the font rendering engine would be to render the characters in accordance to the rules of Devanagari Unicode standards. It also should take this rendered text as an input and then map it to Unicode characters to be sent to any third party applications on the internet that have basic rendering engine to show Unicode characters. These third party applications could be browsers, or other mobile apps that share data over the internet. Another part of the rendering engine is also responsible to collect the data coming from any third party applications that are sending in Unicode data to our rendering engine. Now, the rendering engine must check the data for the presence of Devanagari based Unicode characters and convert them back to the font based decimal code points to be viewed on the Android mobile’s application side.

The overall architecture of the Font Rendering Engine can be described by the Figures 2.8 and 2.9.

2.2.1 Key Event Rendering

The first and the most important concept for a Non-Latin language rendering engine is to render the characters according to the rules of the Unicode language as per the key event. So there must be an algorithm that stays behind and works on the side of the IME. This rendering engine is a part of the IME itself which keeps record of all the character
inputs from the keyboard. It also keeps track a list of the last five characters that were entered in the form of a character sequence to check with the rules and change the characters as per required.

We have an onKey() function which gets call every time the onKeyDown() event is called. In this function, the primary code of the character entered is matched for specific characters which are called matras. Now, the Devanagari script is composed of three subtypes. These subtypes are Vowels, Consonants and Matras. Now whenever some Matra’s
primary code is encountered, it has to be rendered in accordance to the rules of the Devanagari which will be discussed in more detail in Section 3.1.4.

One of the most important character from script is called ‘Halant’ which has Unicode U+094D which combines with the consonants and render them to half or double half characters. Some consonants also render to some symbol when followed by a ‘Halant’ and another consonant. There are also some Matras which when placed after the consonants need to be replaced to take the place before the consonant.

Some part of the algorithm in which it perform the various Key Event functionalities is as given below:

```java
public void onKey(int primaryCode, int[] keyCodes) {
    if (isWordSeparator(primaryCode)) {
        if (mComposing.length() > 0) {
            commitTyped(getCurrentInputConnection());
        }
        sendKey(primaryCode);
        updateShiftKeyState(getCurrentInputEditorInfo());
    } else if (primaryCode == Keyboard.KEYCODE_DELETE) {
        handleBackspace();
    } else if (primaryCode == Keyboard.KEYCODE_SHIFT) {
        handleShift();
    } else if (primaryCode == Keyboard.KEYCODE_CANCEL) {
        handleClose();
        return;
    } else if (primaryCode ==
        NonLatinKeyboardView.KEYCODE_OPTIONS) {
    } else if (primaryCode == Keyboard.KEYCODE_MODE_CHANGE
        && mInputView != null) {
        Keyboard current = mInputView.getKeyboard();
        if (current == mSymbolsKeyboard
            || current == mSymbolsShiftedKeyboard) {
            current = mQwertyKeyboard;
        } else {
            current = mSymbolsKeyboard;
        }
        mInputView.setKeyboard(current);  
        if (current == mSymbolsKeyboard) {
            current.setShifted(false);
        }
    } else {
        Log.d("KEYCODE", keyCodes + " " + primaryCode);
    }
```
sequence = getCurrentInputConnection().getTextBeforeCursor(5, 0);
Log.d("NewSequence", sequence.toString());
if (primaryCode == 37371) {
    handleCharacter(69, keyCodes);
    handleCharacter(97, keyCodes);
    sequence = getCurrentInputConnection()
        .getTextBeforeCursor(5, 0);
    renderHalfDoubleCharacters(keyCodes);
}
else if (primaryCode == 37372) {
    handleCharacter(120, keyCodes);
    handleCharacter(97, keyCodes);
    sequence = getCurrentInputConnection()
        .getTextBeforeCursor(5, 0);
    renderHalfDoubleCharacters(keyCodes);
}
else if (primaryCode == 37373) {
    handleCharacter(98, keyCodes);
    handleCharacter(97, keyCodes);
    sequence = getCurrentInputConnection()
        .getTextBeforeCursor(5, 0);
    renderHalfDoubleCharacters(keyCodes);
}
else if (primaryCode == 37374) {
    handleCharacter(103, keyCodes);
    handleCharacter(97, keyCodes);
    sequence = getCurrentInputConnection()
        .getTextBeforeCursor(5, 0);
    renderHalfDoubleCharacters(keyCodes);
}
else if (primaryCode == 37375) {
    handleCharacter(106, keyCodes);
    handleCharacter(97, keyCodes);
    sequence = getCurrentInputConnection()
        .getTextBeforeCursor(5, 0);
    renderHalfDoubleCharacters(keyCodes);
}
else if (primaryCode == 37376) {
    handleCharacter(99, keyCodes);
    handleCharacter(97, keyCodes);
    sequence = getCurrentInputConnection()
        .getTextBeforeCursor(5, 0);
    renderHalfDoubleCharacters(keyCodes);
}
else if (primaryCode == 37377) {
    handleCharacter(109, keyCodes);
    handleCharacter(97, keyCodes);
    sequence = getCurrentInputConnection()
        .getTextBeforeCursor(5, 0);
    renderHalfDoubleCharacters(keyCodes);
2.2.2 Half Character Rendering

The IME works in close conjunction with the font on the application side. It takes the Decimal code point values of the font on the application side to render the characters following the script rules on the key events. Now the font that stays on the application side is an open source true type font which is called ‘Shusha.ttf’. The font contains glyphs for half characters in the Devanagari Script. Now, if you use standard Unicode, there are no Unicode characters for the half characters in Devanagari. To render those half characters, the character U+094D is used. There is no such thing for the font on the application side. So, while rendering, whenever U+094D character occurs after a consonant followed by another consonant. The first consonant should automatically convert to half character. The algorithm used for the rendering engine used developed in this thesis does that. It checks for such a combination of characters and convert them to half characters on the key events. So, as soon as the user enters these characters in the text box, he can see that it instantly changes to half characters in the text box.

Some part of the algorithm is as follows:

```java
private void renderHalfCharacters(int[] keyCodes) {
    if (sequence.length() >= 3) {
        Integer last = Integer.valueOf(sequence.charAt(sequence.length() - 1)),
        secondLast = Integer.valueOf(sequence.charAt(sequence.length() - 2)),
        halfChar = null;
        if ((Integer.valueOf(sequence.charAt(sequence.length() - 3)) != 114)
            && secondLast == 92) {
```

```java
if (Integer.valueOf(sequence.charAt(sequence.length() - 3)) != 97) {
    halfChar = mapHalfChars(Integer.valueOf(sequence.charAt(sequence.length() - 3)));
    if (halfChar != null) {
        handleBackspace();
        handleBackspace();
        handleBackspace();
        handleCharacter(halfChar, keyCodes);
        sequence = getCurrentInputConnection().getTextBeforeCursor(5, 0);
        if (last != null) {
            handleCharacter(last, keyCodes);
            sequence = getCurrentInputConnection().getTextBeforeCursor(5, 0);
        }
    }
} else {
    if (sequence.length() >= 4)
        halfChar = mapHalfChars(Integer.valueOf(sequence.charAt(sequence.length() - 4)));
        handleBackspace();
        handleBackspace();
        handleBackspace();
        if (halfChar != null) {
            handleCharacter(halfChar, keyCodes);
            sequence = getCurrentInputConnection().getTextBeforeCursor(5, 0);
        }
}
```

### 2.2.3 Double-Half Character Rendering

Though this rendering is not must on the algorithm perspective but it depends upon the font we use. In our case, we used Shusha.ttf which has to have double half rendering. Some of the Hindi characters from the Shusha font are formed by combining two keys from the font. These characters actually are just a single character on the standard Unicode perspective. So, we also need to render these characters in our rendering engine. We detected such combinations of character inputs from the IME and rendered them accordingly.

The Double Half-Character rendering is also done in conjunction with the Devanagari script rules. Some part of the algorithm is as follows:
2.2.4 Devanagari Script Rules

This section of the thesis discusses in more detail the rules that need to be followed for the rendering of characters in the Devanagari Script.

“The Devanagari block of the Unicode Standard is based on ISCII-1988 (Indian
Script Code for Information Interchange). The ISCII standard of 1988 differs from and is an
update of earlier ISCII standards issued in 1983 and 1986” [5]. Devanagari characters change
form combining with one or more characters following them. The appearance of the
characters is purely based on the ordering of the characters that means their sequence in
which they are input does matter. Some of the Devanagari script characters also change the
ordering or the sequence in which they were input [5].
The Devanagari script has certain Consonant letters. These consonants may also be represented in their half-forms. Some of the Devanagari consonant does not have a half-form but most of them do have one. The half form of these consonants does look alike to their original forms with the vertical stem missing as of most of the cases. Some consonants have a half-form that depends upon one of their neighboring characters or consonants [5].

There are also certain vowels for Devanagari script which usually combines with their neighboring consonants. These vowels being separate characters still reside in the same vertical line of view of the consonants they precede. These are also used for writing syllables which start with one of these letters. The Devanagari Vowel Letters are represented in Figure 2.10 [5].

```
<table>
<thead>
<tr>
<th>To Represent</th>
<th>Use</th>
<th>Do Not Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>आ</td>
<td>0906</td>
<td>&lt;0905,093E&gt;</td>
</tr>
<tr>
<td>ऋ</td>
<td>090A</td>
<td>&lt;0909,0941&gt;</td>
</tr>
<tr>
<td>एः</td>
<td>090D</td>
<td>&lt;090E,0945&gt;</td>
</tr>
<tr>
<td>एँ</td>
<td>090E</td>
<td>&lt;090E,0946&gt;</td>
</tr>
<tr>
<td>ऐः</td>
<td>0910</td>
<td>&lt;090E,0947&gt;</td>
</tr>
<tr>
<td>ऐँ</td>
<td>0911</td>
<td>&lt;0905,0949&gt;</td>
</tr>
<tr>
<td>ओः</td>
<td>0912</td>
<td>&lt;0905,094A&gt;</td>
</tr>
<tr>
<td>ओँ</td>
<td>0913</td>
<td>&lt;0905,094B&gt;</td>
</tr>
<tr>
<td>ओः</td>
<td>0914</td>
<td>&lt;0905,094C&gt;</td>
</tr>
</tbody>
</table>
```

Figure 2.10. Vowel letters in Devanagari. Source: The Unicode Consortium. The unicode standard, version 5.0. Addison-Wesley Professional, Reading, MA, 2006.

The ‘Virama’ or ‘Halant’ is also an important character of the Devanagari script. It precedes any consonant letter from the devanagari script and can change the consonant to its half form based on the preceding consonant.

The Devanagari Rules are as demonstrated below:
• Rule 1: “When a consonant precedes a VIRAMA, it is considered to be a dead consonant. A consonant that does not precede VIRAMA is considered to be a live consonant” [5].

\[
\text{त + \ ॉ} \rightarrow \text{त}
\]

• Rule 2: “If the dead consonant precedes a consonant, then it is replaced by the superscript nonspacing mark, which is positioned so that it applies to the logically subsequent element in the memory representation” [5].

\[
\text{र + क} \rightarrow \text{क + \ ॉ} \rightarrow \text{क}
\]

• Rule 3: “If the superscript mark is to be applied to a dead consonant and that dead consonant is combined with another consonant to form a conjunct ligature, then the mark is positioned so that it applies to the conjunct ligature form as a whole” [5].

\[
\text{र + ज + ज} \rightarrow \text{ज + \ ॉ} \rightarrow \text{ज}
\]

• Rule 4: “If the superscript mark is to be applied to a dead consonant that is subsequently replaced by its half-consonant form, then the mark is positioned so that it applies to the form that serves as the base of the consonant cluster” [5].

\[
\text{र + ग + घ} \rightarrow \text{र + घ + \ ॉ} \rightarrow \text{गँ}
\]

• Rule 5: “Except for the dead consonant, when a dead consonant precedes the live consonant, then dead consonant is replaced with its nominal form and RA is replaced by the subscript nonspacing mark, which is positioned so that it applies to consonant” [5].

\[
\text{ठ + र} \rightarrow \text{ठ + \ ॉ} \rightarrow \text{ठ}
\]

• Rule 6: “For certain consonants, the mark may graphically combine with the consonant to form a conjunct ligature form. These combinations, such as the one shown here, are further addressed by the ligature rules described shortly” [5].

\[
\text{फ + र} \rightarrow \text{फ + \ ॉ} \rightarrow \text{फ}
\]

• Rule 7: “If a dead consonant (other than itself) precedes it, then the substitution of RA for nonspacing mark is performed as described above; however, the VIRAMA that formed dead consonant remains so as to form a dead consonant conjunct form” [5].

\[
\text{त + र} \rightarrow \text{त + \ ॉ + \ ॉ} \rightarrow \text{त}
\]

“A dead consonant conjunct form that contains an absorbed dead consonant may subsequently combine to form a multipart conjunct form” [5].

\[
\text{व्र + य} \rightarrow \text{व्र}
\]
• Rule 8: “Other modifying marks, in particular bindus and svaras, apply to the 
orthographic syllable as a whole and should follow (in the memory representation) all 
other characters that constitute the syllable. The bindus should follow any vowel 
signs, and the svaras should come last. The relative placement of these marks is 
horizontal rather than vertical; the horizontal rendering order may vary according to 
typographic concerns” [5].

\[ \text{k} + \text{ā} + \text{ō} \rightarrow \text{kō} \]

• Rule 9: “If a dead consonant immediately precedes another dead consonant or a live 
consonant, then the first dead consonant may join the subsequent element to form a 
two-part conjunct ligature form” [5].

\[ \text{jṛ} + \text{c} \rightarrow \text{j} \]

• Rule 10: “A conjunct ligature form can itself behave as a dead consonant and enter 
into further, more complex ligatures” [5].

\[ \text{s} + \text{ṭ} + \text{r} \rightarrow \text{s} + \text{ṭr} \rightarrow \text{str} \]

A conjunct ligature form can also produce a half-form. [5]

\[ \text{kṣ} + \text{y} \rightarrow \text{kṣy} \]

• Rule 11: “If a nominal consonant or conjunct ligature form precedes nonspacial mark 
as a result of the application of rule 5, then the consonant or ligature form may join 
with nonspacial mark to form a multipart conjunct ligature” [5].

\[ \text{k} + \text{ū} \rightarrow \text{kū} \quad \text{f} + \text{ū} \rightarrow \text{fū} \]

• Rule 12: “In some cases, other combining marks will combine with a base consonant, 
either attaching at a nonstandard location or changing shape. In minimal rendering, 
there are only two cases: RAI with Uvs or UUs “ [5].

\[ \text{r} + \text{ū} \rightarrow \text{rū} \quad \text{r} + \text{ū} \rightarrow \text{rū} \]

• Rule 13: “When the dependent vowel is used to override the inherent vowel of a 
syllable, it is always written to the extreme left of the orthographic syllable. If the 
orthographic syllable contains a consonant cluster, then this vowel is always depicted 
to the left of that cluster” [5].

\[ \text{k} + \text{i} \rightarrow \text{ki} \quad \text{ṭ} + \text{r} + \text{i} \rightarrow \text{ṭr} + \text{i} \rightarrow \text{ṭri} \]

• Rule 14: “The presence of an explicit virama (either caused by a ZWNJ or by the 
absence of a conjunct in the font) blocks this reordering, and the dependent vowel Ivs 
is rendered after the rightmost such explicit virama” [5].
2.3 Font to Unicode Rendering Engine

The other part of the rendering engine is the Font to Unicode conversion. This is termed as a Font to Unicode Rendering Engine which serves as a sub-engine to the Font Rendering Engine as a whole. This is the rendering engine which works in conjunction with the rendering engine that stays on the IME side. The Font to Unicode Rendering Engine resides on the application side. It successfully reads all the Primary Codes (Decimal Code Points) from the Keyboard layout XML file and maps them to the respective Unicode forms.

Now, we’ll be using a REST API to transfer these characters to any third party applications. To use a REST API, we’ll establish an HTTP connection with it and will use the POST method to send the Unicode characters to that application that resides on the internet. This third party application is definitely outside the scope the current application which resides on the Android Mobile Phone itself. Both the applications need an internet connection to share data with one another. Now to send the characters via the connection, they have to be included in the URI to which the connection request is being made. We can’t send the Unicode characters directly in the URI as \u0939\u0457\u0934. It has to be encoded in the Percentage Encoding of URI format of character encoding to be sent across the network. This is the reason we directly convert the Font DCP directly to the Percentage Encoding.

Now, to map each of the character with a font, the font DCP needs to be mapped to the Percentage Character Encoding of URIs. For this we set up a rendering engine that maps the already rendered text from the IME to the PCE. This in Java is done with the help of HashMap. Each character is mapped to the respective PCE. Then the already input text from the IME is read character by character and sends to this rendering engine to be converted to their respective PCE. This text is then added as a parameter to the REST API of the third party application and sent through the network.

Another thing with the mapping of characters that need to be kept in mind is that we have both kinds of characters in the font. Some characters are rendered half-characters and some are rendered as Double Half characters. Now in case of Half-Characters, the mapping will simply include the half character from the font and map it to three or more PCE
characters of the Unicode which together make up to form a single character. This is done because the there is no specific Unicode for the Half-Characters in the Devanagari script. These characters are formed by combining three or more characters that have a Unicode value. Whereas in the font, the half characters are represented by a single DCP. Also we need to consider the case of Double Half Characters. This means that some characters that have a single Unicode, may also have two characters on the font forming that single character. So, we also have to map two characters in the font with a single character with a single Unicode value.

The font to PCE is done in the following way:

```java
HashMap<String, String> toUnicode = new HashMap<String, String>();

//Characters for Vowels
//Q
toUnicode.put("Q", "%E0%A4%A7%E0%A5%8D");
//q
toUnicode.put("q", "%E0%A4%A5%E0%A5%8D");
//e
toUnicode.put("e", "%E0%A4%8F");
//R
toUnicode.put("R", "%E0%A5%83");
//r
toUnicode.put("r", "%E0%A4%B0");
//T
toUnicode.put("T", "%E0%A4%9F");
//t
toUnicode.put("t", "%E0%A4%A4");
//Y
toUnicode.put("Y", "%E0%A4%B7%E0%A5%8D");
//y
toUnicode.put("y", "%E0%A4%AF%E0%A5%8D");
//U
toUnicode.put("U", "%E0%A5%82");
//u
toUnicode.put("u", "%E0%A5%81");
//I
toUnicode.put("I", "%E0%A5%80");
//i
toUnicode.put("i", "%E0%A4%BF");
//O
toUnicode.put("O", "%E0%A5%88");
//o
toUnicode.put("o", "%E0%A5%87");
//P
toUnicode.put("P", "%E0%A4%AA%E0%A5%8D");
//p
toUnicode.put("p", "%E0%A4%AA");
//"
toUnicode.put("\", "%E0%A4%87");
//"
toUnicode.put("}", "%E0%A4%8A");
```

Before sending it over the network the text has to be rendered. Some part of the rendering algorithm is as follows:

```java
while (count != status.length()) {
    char myChar = character;
    char nextChar = iterator.next();
    iterator.previous();
    if (myChar == 'Q' || myChar == 'q' || myChar == 'E'
        || myChar == 'Y' || myChar == 'y'
        || myChar == 'S' || myChar == 's'
        || myChar == 'G' || myChar == 'g'
        || myChar == 'H' || myChar == 'J'
```
|| myChar == 'j' || myChar == 'l'
|| myChar == 'X' || myChar == 'x'
|| myChar == 'c' || myChar == 'v'
|| myChar == 'B' || myChar == 'b'
|| myChar == 'N' || myChar == 'n'
|| myChar == 'm' || myChar == '<') {
count++;
if (character == 'a') {
    result.append(toUnicode.get(String.valueOf(myChar) + String.valueOf(character)));
} else {
    character = iterator.previous();
count--;
    result.append(toUnicode.get(String.valueOf(character)));
}

2.4 REST API

The REST API is based on Client-Server architecture in which the client makes request to the server. The server on the other end fulfills that request and responds back to the client. The REST architecture is the one which saves the current instance or the current state of the input resources and present them to the user. Basically it responds back with the data in two formats namely XML and JSON. The client at any time can be in two states, the transition state or at rest state. When the client is sending requests to the server and is in a transition mode to come to a new state, it happens to be in the Transition State. When the client has no user request, and remains in the same state, it is in the rest state.

In the RESTful architecture, there has to be some resources at the back end from which it retrieves its data. Each of the resource is referenced through the REST architecture via a URI. The connection is done via HTTP request and the various components of the network communicate with each other using that protocol. In making calls to the REST API, you can use both GET and POST methods. The GET method is used to retrieve certain data from the REST API whereas the POST method is used to put some data on the server. Both of these methods are implemented by adding the data to be sent or the data to be requested as a parameter to the URI which the backend server understands. The server parses the request for those parameters and performs accordingly.
2.5 Script Detection Engine

Now comes the third sub-engine of the Font Rendering engine as a whole. It is called Script Detection Engine. So far we have considered the case in which the script needs to be rendered in accordance to the Devanagari Script rules, this part was done on the IME side, next we had to match up the application to send data to the third party applications which do not use the font that is included in our application, so the data had to be rendered to the Unicode DCPs. Now, we’ll consider the case in which any third party application has to send data to our application and our application should render those Unicode DCPs correctly to be shown on the application properly.

To have the text sent to the application by any other application through the internet, we have to consider the case of language detection first. In this case, we’ll check the text for the presence of Unicode that belong to the data set of the Devanagari script. Another thing we have to do in this engine is to run a reverse rendering procedure in which we’ll detect the combinations of neighboring Unicode characters for the formation of Half characters of some special form of characters that are formed by the combination of two or more Unicode characters together. In this matter, they also have to be transformed back to the Font DCPs which may be having a single DCP that belongs to a group of Unicode characters together.

In the first case, we detect the presence of Devanagari Script characters in the text; first of all we parse down the response from the REST API which in our case is a JSON Array string. This string when read and converted to the JSON array, the Unicode characters transform to some garbage characters. First of all, to stop them from being turning into garbage characters, we have to change the string to change to Unicode alphabets which are of the form \u0939\u094D\u0934 to some other form so that when turning them on to JSON object, the garbage characters are not produced. For that we decode the whole string character by character and replace each ‘\u’ with ‘Lu’. Now, when transforming the same to JSON object, the garbage characters will not be formed and we’ll be able to check for the presence of Devanagari scripts still as we can still detect the presence of Lu0939, Lu094D and Lu0934 as Devanagari script characters.

Secondly, we’ll parse the JSON array for all the text messages and iterate the strings character by character to locate any combinations of Lu09__ forms to detect for the Devanagari characters. Now, we have to map these characters to Font DCPs. To do that, we
have a Hash Map that maps all the Unicode characters to their respective font DCPs. We check for the presence of a particular Unicode character in the hash map and replace them with its Font DCP. These still not give us a proper rendered text to be displayed on the phone screen. This is because still the users will be able to see the Hindi characters on the screen but still they are not rendered in accordance with the Devanagari script rules.

So, at last, we need an algorithm that works at the backend and works in the reverse rendering procedure as we did in the previous phase. This engine will be responsible to convert all the Unicode character combinations which should form some half form or a special character and replace those combinations of inputs with the half forms and the special characters. These special character glyphs though not found in the Unicode, we have all of those glyphs in the Font to be displayed on the mobile screen.

In this way, we complete the procedure of rendering characters on the application side. This is a separate Engine that works by the applications side. The application makes use of its services for text decoding and reformation to be displayed on the phone screen.
CHAPTER 3

RESULTS DEMONSTRATION

In this section, we’ll discuss an Android based Hindi Facebook Application prototype. In this application we’ll use the Facebook mobile’s News Feed and Profile Wall to share the hindi content across the network. For the creation of such an application, we’ll make use of Facebook’s Old REST API. We also make use of Facebook Android sdk to share the content with Facebook’s API. The Facebook Android sdk is an open source platform for Facebook integration for Android based mobile applications.

In this chapter we’ll discuss mainly the content sharing on Facebook in Hindi language through Android mobile Application. It covers all the areas of the thesis which include

- Importance of Hindi Language based Android Application
- Use of Facebook API and Facebook Android sdk
- Change of Facebook profile status message in Hindi
- News Feed and Wall Feed from Facebook
- Future work that can be done
- Limitations
- Screenshots of Demonstration of the use of Rendering Engine

This chapter displays the results on how the Rendering Engine can be included to perform multi-language content sharing. It can be used on any mobile platform that is Java based. The application on the mobile phone just has to use that specified font and our rendering engine to share multi-language content over the internet.

3.1 IMPORTANCE OF HINDI LANGUAGE ANDROID MOBILE APPLICATION

The major factor that is of importance in creation of a Hindi based content sharing application for Android Mobile Phones is that it does not have Non-Latin script support. It means if you try to enter Hindi language characters on an Android Mobile phone it won’t show you the glyphs as it doesn’t have those glyphs included in the font that is available on
the application side. This was the main reason that led the start of some work on Non-Latin script support on Android mobile phones.

The rendering engine developed in this thesis will work on any java based platform whether or not those glyphs are already present on that mobile phone or not. Those glyphs are provided on the application side to the application itself and the rendering engine renders it for sending to any other application over the network. This thesis led to the first Android Hindi content sharing on the web. On November 5, 2010, First Hindi message for Diwali festival greeting was posted on Facebook through an Android Mobile Phone. Diwali festival is the most popular festival in India which is a Hindi language speaking nation.

There hasn’t been any application as of now on the Android mobile phones that can post a Hindi language message over the web or share Hindi content over the internet. This is because of the lack of support of Non-Latin language support on Android. People had been waiting to create Hindi content sharing applications on Android mobile phones until Google gives them the support to add their own language content in the applications. We had a new view of adding the font support to the applications from our own end and creation of a rendering engine separately to share the content on internet.

### 3.2 Use of Facebook API and Facebook Android SDK

We created a prototype for Android Facebook sdk to share a new profile status message on Facebook. Facebook has recently launched a new Facebook Android sdk and have made it open source for the developers to integrate Facebook with their applications. Using that application in this thesis a platform for News Feed and Profile wall has been developed. It uses the rendering engine for language conversion on the backend.

For user login Authentication, the Facebook Android sdk has tools to connect the Facebook’s OAuth Authentication 2.0 which keeps up the users session and once the user session is valid, we can share the content on their Facebook account. We also need to ask the user for the permissions that he has to allow in the very beginning as soon as he/she logs in. These permissions include all the data that the user has shared in his/her privacy settings on Facebook and the permission to post something on users wall.
The Facebook sdk includes an AsyncRunner class that process the parameters sent as a part of the URI on a UI Thread. Thus as soon as we send the data to the Facebook account as a message in the URI of their REST API, the thread in the backend creates a connection, checks for the user session, post the message on the users wall and finally gives back a positive response when the comment has been posted.

The FbDialog class of the Facebook Android sdk provides an Android WebView to show up the Facebook Login page and the Application permissions page to the user in an Android dialog box. In this way the login information entered by the user is protected on the part of Facebook and the application developer has no control over the page where the user inputs the login information.

3.3 CHANGE OF FACEBOOK’S PROFILE STATUS MESSAGE IN HINDI LANGUAGE

We created a prototype Facebook Hindi application with all the pages like News Feed, Profile, messages, photos, videos, chat client, etc. From these we have implemented the News Feed and profile part implemented to give a demonstration of the rendering engine. On the News Feed activity on the Android Mobile phone, a text box is there on the top of the screen with a share button. The user has an option to click on the text box for the IME to pop up and input text. For the selection of Hindi Keyboard IME that we developed, long click on the text box. A dialog box will appear. Select Input Method and then by selecting Hindi Keyboard one can use the Hindi keyboard on the text box. Once selected the Hindi keyboard will automatically pop up on the screen.

As soon as the user starts pressing the keys on the IME, the IME punch in the characters in the input text box. These characters are first rendered on the IME side algorithm to check for Devanagari script rules and then are pushed on to the text box. The text on the text box also gets automatically replaced with certain characters instantaneously. Finally the user can type in the whole message for that he/she want to post in to their profile. The application side algorithm to convert these application font characters to Unicode hasn’t come to picture as of now. It will be called as soon as the user wants to share that status message.
Now, when the user clicks the share button on the application the text first goes to the application side rendering engine. This engine first converts the text to Unicode PCE to be sent to the Facebook’s Old REST architecture. The Facebook checks for the Access Token that the Facebook sdk generates at the moment, it checks for the permissions – whether or not the application has the wall post permission and it also checks for the current user session. Once everything is verified, the text message sent to the Facebook is shared on the user’s wall.

### 3.4 News Feed and Wall Feed from Facebook

The application also has the facility to show the Facebook’s news feed and the wall feed back to the application. This feature in this prototype application tests the Script Detection Engine and the Reverse Rendering Engine (Unicode to Font). Now it has a get posts button which retrieves the data from the Facebook API. The data in the News feed will be the recent status messages of the user as well as his friends. The profile section of the application will show only the user’s recent status messages.

When the user clicks the Get Posts button on the application, a connection to the Facebook’s REST API is made with the GET request for News Feed in the News Feed section and user’s feed in the user’s profile section. The Facebook responds with a JSON format as a JSON array which needs to be parsed on the application using Android’s JSON library. The parsed message is first sent to the Script Detection Engine. The script detection engine detects the Devanagari script characters in the messages. If not detected, the message is displayed on the application as it is. In case the script contains the Devanagari characters, the same is first sent to the Reverse Rendering Engine.

In case the text has Hindi characters and the same is received by the Reverse Rendering Engine. The engine first converts the received Unicode characters to the font DCPs. These DCPs are then iterated character by character for the formation of special characters which are formed by a few neighboring characters. These special characters have a single DCP in the font so they can be easily mapped to the font DCP.
3.5 Future Work in the Field

The part of rendering the characters is done in this thesis and one can simply call the functions in the rendering engine to be used with any application. The future work in this field could be to provide support for more number of languages. As soon as the same approach is taken for the support of more number of languages, the rendering engine will work fine. It’s like adding support for various different Non-Latin languages to increase its range of functionality. For adding support for other languages as well, one just has to use that language specific font, add the rendering engine for that specific language and it’s good to go.

The future work in this field can also be to add voice to text conversion facilities using accent understanding algorithms. The users can then just speak of the things to be typed or printed in the application. In that case there will be no need for the IME. The IME in that case can just be used for editing the text that has to been entered incorrectly by the voice to text rendering engine.

Another work in this field could be to add complete word suggestions to the application. There could be a dictionary that could be at the backend and works by the side of the emulator and provide suggested words for the Hindi language. This feature is already present in some of the Android Keyboards for English and could be an advanced step in the Hindi language keyboard as well.

3.6 Limitations

Like every task there are a few limitations in the working of IME and the Rendering engine. One of the limitations of the IME is that the number of buttons is limited. All the buttons cannot be represented on just one page. The IME developed in the thesis consists of four pages to display all the characters and symbols. Some of the characters that are present on the Devanagari Inscript Keyboard for the support of Marathi language as well cannot fit in the space leading to the support of just Hindi language. Another limitation is that the font doesn’t have some of the special characters that are formed by combining some Unicode characters. These characters cannot be displayed as the font doesn’t have the glyph for those characters. Though it is not important at all because those characters are the most uncommon to use.
3.7 Screenshots for Result Demonstration

The Facebook login page on the application is derived from Facebook on the Android’s webview. The Facebook controls the actions inside the webview. The webview is shown in Figure 3.1. When the user has logged on to the Facebook, he can use the session id provided by Facebook to move through to the main Facebook Application page. It is as shown in Figure 3.2. It also includes a glance at the user’s profile page. Figure 3.3 shows a screenshot of the News Feed page where the user can type in hindi characters through the Hindi keyboard IME.

Figure 3.1. Hindi Facebook application login page on the application.
Figure 3.2. Hindi Facebook application main page and profile page.

Figure 3.3. Hindi Facebook login news feed page and typing status message.
CHAPTER 4

CONCLUSION

There had been a number of challenges in building a complete solution for the use of Non-Latin scripts with mobile phones. This technology addresses those fundamental and technical challenges and introduced the first time Hindi language support specifically for the Android mobile phones. For the Android mobiles, there had been to technology as of yet to support Hindi language and our application prototype clearly make use of Hindi language on the Android mobile phones. In this technology an Input Method Editor for Hindi language was developed. A fully tested and functional Hindi language keyboard was developed as a part of the Thesis. The Android sdk environment was customized for Hindi language by setting the character encoding to UTF-8 for text sharing which is an appropriate Unicode encoding for the purpose of Internationalization.

There had been various problems while mapping certain characters from font to Unicode and vice-versa. These problems include the formation of garbage characters while reading Hindi language input. All such problems were resolved and a very accurate working rendering engine was developed. The rendering engine has been tested for various inputs from the IME and also while reading any Hindi characters from the Facebook and the application performed very well.

In this Thesis, a vision for more future works in this area has been created and multi-level language platform was created and also the support of this rendering with other Mobile platforms as well that doesn’t support Java as a base programming language. In the previously built technologies the font used to be developed as a part of the application and then sent across the network. This technology gives a method of adding the font as a part of the application instead that provides support for all the font sizes. Another further improvement could also have voice rendering engines and text predictive technologies added to the rendering engine.

The thesis as a whole provides the users a complete solution for Hindi language text sharing over the web. It can interact with other third party applications and share data with
them. These third party applications could be other Mobile based applications or may be a part of web services over the internet.
BIBLIOGRAPHY


