MULTI-LINGUAL SCREEN READER AND PROCESSING OF FONT-DATA IN INDIAN LANGUAGES

A THESIS

submitted by

A. ANAND AROKIA RAJ

for the award of the degree

of

Master of Science (by Research)

in

Computer Science & Engineering

LANGUAGE TECHNOLOGIES RESEARCH CENTER

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD - 500 032, INDIA

February 2008

Dedication

I dedicate this work to my loving parents ANTONI MUTHU RAYAR and MUTHU MARY who are always behind my growth.

THESIS CERTIFICATE

This is to certify that the thesis entitled **Multi-Lingual Screen Reader and Processing of Font-Data in Indian Languages** submitted by **A. Anand Arokia Raj** to the International Institute of Information Technology, Hyderabad for the award of the degree of Master of Science (by Research) in Computer Science & Engineering is a bonafide record of research work carried out by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Hyderabad - 500 032 (Mr. S. Kishore Prahallad)

Date:

ACKNOWLEDGMENTS

I thank my adviser *Mr.Kishore Prahallad* for his guidance during my thesis work. Because of him now I am turned into not only a researcher but also a developer.

I thank *Prof.Rajeev Sangal* and *Dr.Vasudeva Varma* for their valuable advices and inputs for improving the screen reading software.

I am also thankful to my speech lab friends and other IIIT friends who helped me in many ways. They were with me when I needed a help and when I needed a relaxation.

I am very grateful to my colleagues *Mr.Diwakar, Ms.Susmitha, Mr.Rohit and Ms.Bhuvaneswari* in a special way for their advices, helps and contributions in my work.

The Lab *(Speech Lab, LTRC)* and the Institute *(IIIT, Hyderabad)* provided me a wonderful environment to work.

Last but not the least; I thank *God* for all his grace on me.

A.Anand Arokia Raj

ABSTRACT

Keywords: *Screen Reader; Font Encoding; Font Data; Text to Speech Systems; TF-IDF Weight; Glyph; Glyph Assimilation Process; Font Converter.*

Screen reader is a form of assistive technology to help visually impaired people to use or access the computer and Internet. So far, it has remained expensive and within the domain of English (and some foreign) language computing. For Indian languages this development is limited by 1) availability of Text-to-Speech systems in the language and 2) support for reading glyph based font encoded text. Because there is no consistency within font definitions, number of glyphs per font and glyph code mapping for such glyphs for Indian languages. So there are many fonts per language and huge amount of electronic data is available in those fonts. So most of the existing Indian Language screen readers do not support many fonts, consequently not many languages.

Most of the Indian language electronic data is either Unicode encoded or glyph based font encoded. Processing Unicode data is quite straight forward because it follows distinguished code ranges for each language and there is a one-to-one correspondence between glyphs (shapes) and characters. This is not true in the case of glyph based font encoded data. Hence it becomes necessary to identify the font encoding and convert the font-data into a phonetic notation suitable for the TTS system.

This thesis proposes an approach for identifying the font-type (font encoding name) of a font-data using TF-IDF (Term Frequency - Inverse Document Frequency) weights. Here the *term* refers to a glyph of a font-type and the *document* refers to a collection of words or sentences in a specific font-type. The TF-IDF weight is a statistical measure used to evaluate how relevant an N-glyph (‘N’ sequential glyphs like uniglyph, biglyph, triglyph etc.,) to a specific font-data in a collection of textual data. We have built models of various font-data for *uniglyph, biglyph and triglyph*. When these font models were evaluated for font-type identification, triglyph models performed best and biglyph models performed better than uniglyph models. The results show the performance increases when longer context (neighboring glyphs) taken for building font models, subject to reasonable coverage in the training data.

This thesis also proposes a generic framework to build font converters for conversion of font-data into a phonetic transliteration scheme in Indian languages. It consists of two phases: (i) building the glyph-map table for each font-type and (ii) defining glyph assimilation rules for each language. A font-type has a set of glyphs which may be a character or part of a character (merely a shape) and they have an 8-bit code value in the range 0 to 255. Conventional approaches map glyphs to a closest phonetic notation while building the glyph-map table. This technique fails to uniquely identify the duplicates of a glyph. As a novelty, we handle this by taking the place/positional information of the glyph into account. The position, denoted by here numbers, refers to the position of glyph with respect to a pivotal character. Glyph assimilation, which is the second phase of font conversion is a process of merging two or more glyphs to produce a single valid character/Akshara. A set of glyph assimilation rules are defined under each level for each Indian language. Promising results were obtained by following this approach for 10 languages like Hindi, Marathi, Telugu, Tamil, Kannada, Malayalam, Gujarati, Punjabi, Bengali and Oriya.

As a major contribution of this thesis, we have successfully developed a multi-lingual screen reader (RAVI) for Indian languages and have incorporated the modules for font-type identification and font-data conversion.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

MSAA	- Microsoft Active Accessibility
COM	- Component Object Model
DLL	- Dynamic Link Library
INI	- Initialization files
OCR	- Optical Character Recognition
CART	- Classification And Regression Trees
TF	- Term Frequency
IDF	- Inverse Document Frequency
JAWS	- Job Access With Speech
HAL	- Home Automated Living
SAFA	- Screen Access For All
RAVI	- Reading Aid for Visually Impaired
NAB	- National Association for Blind
IIIT	- International Institute of Information Technology
IIT	- Indian Institute of Technology
IISc	- Indian Institute of Science
ATA	- Assistive Technology Act
ATM	- Automatic Teller Machine
API	- Application Program Interface
ISO	- International Organization for Standards
TTS	- Text-to-Speech System
ASR	- Automatic Speech Recognition

ASCII	- American Standard Code for Information Interchange
ACII	- Alphabetic Code for Information Interchange
ISCII	- Indian Standard Code for Information Interchange
UCS	- Universal Character Set
UTF	- Unicode Transformation Formats
ITRANS	- Indian languages TRANSliteration
IT3	- Indian Transliteration 3
OSM	- Off Screen Models
SAPI	- Speech Application Program Interface
GUI	- Graphical User Interface
MBROLA	- Multi-Band Re-synthesis OverLap Add
NLP	- Natural Language Processing

CHAPTER 1

INTRODUCTION

1.1 SCREEN READER

A screen reader is a software designed for visually impaired people, integrated with speech synthesizer to speak aloud everything on computer screen, such as icons, menus, text, punctuations and control buttons. Screen readers (or speech enabled systems) are designed for blind people to use/access standard computer softwares, such as Word Processors, Spreadsheets, Email and the Internet. Typical computers like desktop or laptop can be adapted to talk by adding a screen reader to it. The screen reader allows navigation around the computer screen and presents what is happening on the screen to the user in a meaningful way using speech output.

Today, in the world of Human Computer Interaction (HCI) the visually challenged community in India and other developing countries are deprived of technologies that could help them to interact with the sighted world especially with gadgets which are not user friendly for blind. Screen readers are a form of assistive technology for the blind people. It is a technology that promise the visually challenged natural access to computers, gadgets and the Internet.

Assistive Technology: A general term used to describe technology that assists people with a disability such as screen reading software or alternate keyboards is "assistive technology". The Assistive Technology Act (ATA) of 1998 defines assistive technology as *"any item, piece of equipment, or product system, whether acquired commercially, modified, or customized, that is used to increase, maintain, or improve the functional capabilities of individuals with disabilities."* Some of this assistive technology includes screen readers, talking web browsers, printed text readers, Braille translators (Text-to-Braille and Braille-to-Text), screen magnifiers, special computer keyboards and technology that allow control of a computer through head movements or eye movements.

Blindness and Low Vision: Blindness poses a threat to the way of life of people not only in India but also world wide. There are an estimated 180 million visually-impaired people [1] in the world. Approximately 10 to 20 million persons [2] in India are unable to read or write due to the vision problems and 95 percent of them do not know English. While there are many amongst us who share and concern the challenges faced by visually impaired, studies/research reveals that many visually challenged persons naturally possess high degree of senses and IQs than the normal persons. Most people who are visually impaired keenly uses sensory touch and hearing, such senses are primary asset in operating fine controls, working in voice based processes in BPO operations, or even doing complex operations which require monitoring and resultant operations of audio signals.

While a blind or visually challenged person may or may not be able to read or write effectively, their sharp sense of hearing becoming primary mode of interface. Today's technology can exploit this very mode. Speech technology (that can convert a written/typed text into speech) offers solutions which can enable development of interfaces on the computer transforming documents, information and content accessible to the blind. One such solution is called screen reading software which can be ported on a range of devices via. personal computers, information kiosks, ATMs, handhelds and mobile phone, etc.,

Rendering on-line instruction allows blind and low vision students at a distinct disadvantage when the majority of the information is in visual format. Although blind and low vision students can learn to use keyboards effectively, accessing output is a challenge. With the help of Braille output devices, optical character recognition software, and screen readers, blind students can also equally access to electronic information on par to their counterpart.

Screen readers are categorized into three major types: Command-line text screen reader, GUI based screen reader and web-based screen reader, accordingly

- *CLI (text) Screen Readers*: Early operating systems such as MS-DOS which employed a Command Line Interface (CLI), and the screen display consisted of characters mapping directly to a screen buffer in memory and to a cursor position. Input was keyed from keyboard. All this information could be obtained from the system by hooking the flow of information around the system and reading the screen buffer and communicating the results to the user.

- *GUI based Screen Readers*: Arrival of Graphical User Interfaces (GUIs), added more complication. A GUI is composition of characters and graphics drawn on the screen at particular positions, and as such there is no pure textual representation of the graphical contents on the display. Screen readers were only choice but driven to employ new low-level techniques, gathering messages from the operating system and using these to build up an "off-screen model", model is a representation of the display in which the required text content is stored. Screen readers can also communicate information from menus, controls, and other visual constructs to permit blind users to interact with these constructs. However, maintaining an off-screen model is a significant technical challenge: hooking the low-level messages and maintaining an accurate model are equally difficult tasks. Operating system and application designers have attempted to address these problems by providing ways which screen readers can access the display contents without having to maintain an off-screen model. This involves the provision of alternative and accessible representations of what is being displayed on the screen accessed through an API. Existing APIs include: (1) Apple Accessibility API, (2) IAccessible2, (3) Microsoft Active Accessibility (MSAA), (4) Java Access Bridge and etc.,. Screen readers can query the operating system or application which is currently being displayed and receive updates when the change in the display.

- *Web based Screen Readers*: A relatively new approach in the field is web-based applications like Talklets that use JavaScript for adding Text-to-Speech functionality to web content. The primary audience for such applications are those who have difficulty in reading because of learning disabilities or language. Comparatively functionality for this type remains limited than desktop applications, the major benefit is to increase the accessibility of websites/URLs when accessed on public machines where users do not have permission to install custom software, web-based plug-ins give people greater *‘freedom to roam’*.

1.2 ISSUES IN BUILDING A SCREEN READER

Building a screen reader is a complicated task, considering the technology and usage. There are many hurdles in development of multilingual screen reader, some of them are common for all the screen readers. Let us review and highlight the problems with the existing screen readers. The common issues are speech synthesizer, implementation strategy, user interfaces, supported documents of different format, multi-lingual issues, supporting OS platforms and the cost component. We discuss briefly about each of these points below.

- *Speech Synthesizer*: A speech synthesizer (aka TTS System) is the major component of any screen reader system. A speech synthesizer accepts plain or marked up text as input and generates speech output. In India, there are different organizations involved in development of speech synthesizers for Indian languages. Some of them are private companies and others are institutes. Every one of these has unique interface to the speech synthesizer. The problems are: (i) they provide support for one or few Indian languages, (ii) the quality of the speech output is still not natural and (iii) no effort to integrate with any application and provide a unified model for a common platform. Viable action in windows operating system involves creating a layer on top of all the synthesizers by following Speech Application Programming Interface (SAPI) standard proposed by Microsoft.
- *Screen Reading Application*: A screen reader monitors and extracts information on computer screen through keyboard command/mouse click and sends that information to the speech synthesizer in text form. Handling the keyboard events is relatively easy; one can just capture keystrokes and convert with the help of in built keyboard layouts. Monitoring the screen is a complex job particularly in windows based system. By windows based system we mean any system with windowing capabilities. Problems in windows based systems arise due to the fact that each window is just

an array of pixels on the screen, hence making it mandatory to get that information from the program/application that has written that text on the screen. It is possible only when the program has a well defined interface for obtaining that information or graphics APIs are monitored for text output and then a record is maintained.

In Windows operating system almost all the screen components are well defined and they allow one to get such information in multiple ways. Bare-bone approach is to get such information through Win32 API and preferred approach is to hook into Microsoft Active Accessibility (MSAA) [3]. MSAA provides notification to the interested application and provides standard mechanism to obtain this information. However MSAA happens to be the heart of any screen reading application, it still can not provide information for all the components and all the desired forms of information. So one has to rely on Component Object Model (COM), Win32 API, and at times on display drivers. Then there is the issue of handling each screen component differently than others. For example the “Language Bar” in Windows XP doesn’t expose any interface to retrieve the text.

- *Multi-Lingual Issues*: Discussing to issues specific to multilingual screen readers, we can mainly focus our attention to language representation code or scripts. The preferred coding is Unicode [4] which supports codes for almost any language currently available. So if the world followed Unicode only, it would be easier for programmers to write programs which could understand any language text and handle it appropriately. But unfortunately there are different coding standards used by various vendors and applications. So our approach has to address both forms of text. Our software will internally use IT3 [5] phonetic notation but whenever required it can be converted to Unicode or to other notations (when a speech synthesizer does not understand IT3 notations). We also convert from other font-encodings to IT3 (when a particular application does not provide IT3 text). Our main focus will be on IT3 and Unicode due to the fact that the current Windows and the subsequent versions will use Unicode for multi lingual applications. When required a converter (font converter, ISCII converter etc.) it can be inserted to accommodate non IT3/Unicode applications.

- *User Interfaces*: User interfaces or navigation controls differ from one screen reader to screen reader. A standard set of user interfaces has to be identified and to be defined. Some time the user expects, they should be customizable like JAWS which provides some scripting mechanism to modify the user interfaces. Teaching and Training regarding the new interfaces or navigation controls become mandatory.

- *Document Formats*: Screen reader has to read content from the documents in different formats. They may be simple multi-line edit control (Notepad), word processor (MS Word), spreadsheet (Excel), presentation tool (PowerPoint), web/marked up pages (HTML/ASP/XML) and encrypted documents (PDF/PostScript). Extracting the required text from those documents is a complicated task. For some documents there are built-in libraries and for others one needs to write his own parser to extract the text.

- *Supporting Platforms (OS)*: A screen reader is developed aiming at one platform (Windows/Linux). Even for a single OS, it may work very well with one version and may not with the other versions. So the user has to maintain one screen reader for each platform if he works with multiple platforms. A platform independent architecture has to be designed which is relatively challenging task.

- *Cost*: This is the most exciting question from the user end about a screen reader (especially from Indian community perspective) *"Whether it is available for free or cost?"* Well, if it costs then how much? A well developed screen readers like JAWS, Window-Eyes etc are costly (in the range $500 dollars and above) which can not be affordable by a common men in Indian. Users are expecting the screen reader at affordable or free of cost with quality that of existing solutions.

1.3 RELATED WORK ON SCREEN READER

Advances in synthetic speech, has allowed the development of screen reader software, which can capture text from the computer and transform it to the audio form thereby helping visually impaired people get the information via. the speech modality. Increasingly, screen readers are being bundled with operating system distributions. Recent versions of Microsoft Windows come with the rather basic Narrator, while Apple Mac OS X includes VoiceOver, a more feature-rich screen reader. The console-based Oralux Linux distribution ships it with three screen-reading environments: Emacspeak [6], Yasr and Speakup. The open source GNOME desktop environment long included Gnopernicus also now includes Orca. There are also many open source screen readers, such as the Linux screen reader for GNOME and NonVisual Desktop Access for Windows. The most widely used screen readers are separate commercial products: JAWS from Freedom Scientific, Window-Eyes from GW Micro, and HAL from Dolphin Computer Access being prominent examples in English. With recent initiatives from research organizations and major software vendors, the development of screen reading software has begun in Indian languages. But support to languages and types of content reading is limited due to technology constraint.

Here we review screen readers which are mostly in use. We categorize them as (1) screen readers available for English and other foreign languages and (2) screen reader efforts in Indian languages. The first four screen readers are for English and the later three are the efforts in Indian languages. While English screen readers are widely used, attempts in Indian languages have been met with limited success and popularity.

JAWS [7] : The most popular screen reader used worldwide, JAWS for Windows works on any PC to provide access to latest software applications and the Internet. With its internal software for speech synthesizer and the computer's sound card, the information from the screen is read aloud, providing access to a wide variety of information, education and job related applications. JAWS also outputs to refreshable Braille displays, providing unmatched Braille support of any screen reader on the market. JAWS Professional costs $1095 and JAWS Standard costs $895. It has been released with Unicode (international digital coding standard for representation of languages) enabling users to avail support in Indian languages. It follows OSM (Off Screen Model), a standard developed to read Indian languages (Hindi) based on what is displayed, which sometimes sounds weird from the actual pronunciation of a letter or a word.

Window-Eyes [8] : It is comparable to the most stable screen reader available in the market today. Window-Eyes gives total control over what you hear and how you hear it. Provides an additional feature of enhanced Braille which understands user's feelings. On top of all that, the power and stability of Window-Eyes means that most applications work right out of the box with no need for endless tinkering and juggling configuration in order to get them to function properly. Applications that utilize standard Microsoft controls will be spoken automatically with little or no configuration necessary. Other applications may require simple modifications to Window-Eyes whose options are easily accessible via a well cataloged menu organized in user friendly manner such as speech control panel. Window-Eyes costs $895.00 and Window-Eyes SMA costs $299.00.

HAL [9] : It is a screen reader software that works by reading the screen interactively and communicating through a speech synthesizer or by a refreshable Braille display. One of only a few with rich featured screen readers in the world, HAL gives independence - not only on a desktop PC, but can be carried on a USB stick to use at any campus, Internet cafe, library or work place where PC is available across the globe. The benefit of this is that users can now carry their screen access, with their favorite settings with them at all times. In addition, HAL users can connect remotely on terminal servers or Citrix mainframes.

Kurzweil [10] : Kurzweil (1000 and 3000) reads printed materials such as books, magazines, single sheets and newspapers. Printed materials are first scanned into the computer. Then Kurzweil 1000's optical character recognition capabilities recognize the and convert it into synthetic speech. This gives people who are blind audible access to the printed pages.

SAFA [11] : Screen Access For All (SAFA) is an open source initiative to develop a screen reading software for the vision impaired users to read and write in their language of choice. The goal is to have multi-lingual software on multiple platforms. Based on Vachak [12], a Text-to-Speech (TTS) engine developed by Prologix, Lucknow (India) and

National Association for the Blind (NAB-New Delhi) has developed this screen reader. It currently supports English and Hindi only and the work is in progress to provide support for all official Indian languages.

IITM Software for the Visually Handicapped [13] : Open source community at IIT Madras has realized the importance of the need to incorporate some aspects of Assistive Technologies in the Multilingual software, so as to benefit the disabled, at least in respect of giving them opportunities to get educated in the first place by means of their own mother tongue. The multilingual software, enhanced with speech, has gained wide acceptance in the country on account of its simplicity and ease of use. Their development of systems for the disabled is still at a stage of infancy when compared to solutions available in the west for the English speaking users. MBROLA is a freely available speech engine which has been used in this application and as of now, there is a slight European accent to the speech output. Still packaging the software in any form that would allow visually handicapped persons to download and install the same all by them is part of the future work. What the team has done is to provide a suitable interface program to substitute the computer's sound interface for the external synthesizer so that the application could work on a Windows machine or a Linux system. The JAWS reader interface uses MBROLA as the speech engine. MBROLA is the dependency for the application has to work.

VoiceDot [14] : Analytica's VoiceDot aims at fulfilling the need for a cost effective screen reading solution, for visually impaired people in India and around the world. Analytica envisions VoiceDot as a means to aid the visually impaired in harnessing knowledge from the wired world. VoiceDot will enable the visually impaired user to read and write electronic documents, and communicate with the world through the Internet. VoiceDot is built around a plug-in architecture, this supports third party plug-ins for supporting custom applications. For instance, applications like Lotus Notes, Acrobat reader or Tally can be made accessible using the plug-ins. VoiceDot supports Microsoft's SAPI 5.1 (Speech Application Programming Interface) and MSAA (Microsoft's Active Accessibility). It uses Microsoft's TTS synthesizer to render text as speech output.

1.4 NEED OF SCREEN READER FOR INDIAN LANGUAGES

India is a multi-language, multi-script country with 23 official languages and 11 written script forms. About a billion people in India use these languages as their first language. English, the most common technical language, is the lingua franca of commerce, government, and the court system, but is not widely understood beyond the middle class and those who can afford formal, foreign-language education. About 5% of the population (usually the educated class) can understand English as their second language. Hindi is spoken by about 30% of the population, but it is concentrated in urban areas and north-central India, and is still not only foreign but often unpopular in many other regions. Screen reading technology provides many functionalities using speech technology to help the visually impaired people, but it has hardly reached to the needy population in India other than small fortunate English speaking community only as potential customer. Hence there is a need to have screen reader that can aid other visually impaired people who speak Indian languages.

Some of the limitations observed in existing screen readers include:

- Professional English screen readers fail to provide good support for Indian languages. The English voices have US/UK accent and thus native speakers found it hard to comprehend.
- Screen readers in Indian languages support one or two major spoken languages, and thus lack a generic framework of supporting multiple Indian languages. At the same time, these screen readers often support only Unicode formats and thus ignore best local websites such as Eenadu, Vaartha, Jagran which use glyph based fonts.
- Screen readers in Indian languages have also limitations in terms of support for computer applications or programs. Typically the support is provided at the screen level and to the browsers, and thus major applications such as Email clients, various internet browsers and PDF/Postscript readers are ignored.
- Some screen readers do not make use of recent advances in text-to-speech technologies and thus provide intelligible but not as natural as human speech.

- Some are freely available and others are costly.

Hence we need good professional screen readers for Indian languages providing support for many or all of the Indian languages, different computer applications and intelligible, human-sounding synthetic speech. Such screen readers should be available freely or in some reasonable cost so that the common Indian user can reap the benefit of such technologies.

1.5 THESIS STATEMENT

Developing a full-fledged screen reader for Indian languages is a challenging task considering the computer-science, language and speech processing aspects involved in the implementation. The problem of developing a screen reading software for Indian languages could be stated as: Given a text in any Indian language encoded in any format the objective would be to process them into the required format and speak it out in the respective language. We wish to develop a screen reader for Indian languages and provide it for freely which would help Indian community, especially blind people to a great extent.

1.6 CONTRIBUTIONS

The scope of the research includes 1) understanding the scripts of Indian languages and their storage formats, 2) identifying the font-type (font encoding name) of the font-data, 3) designing a generic framework for building font converters for scripts in Indian languages and 4) developing a screen reading software for Indian languages.

This research documents three main contributions. They are:

(a) *An approach for identifying the font-type (font encoding name)*: TF-IDF weight based vector space models for fonts, constituting uniglyph, biglyph and triglyph to classify or identify the font encoding name of the text (Chapter 3).

(b) *A generic framework for building font converters in Indian languages*: Helps building rapidly a font converter by providing only the *glyph-map table* of a font. This glyph-map table provides mapping information between glyph and its closest phonetic notation with position information. A set of *glyph assimilation rules* are already identified and defined per language (Chapter 4).

(c) *A screen reader for Indian languages*: Reads content in different Indian languages and helps blind people to use the computer and Internet (Chapter 5).

1.7 ORGANIZATION OF THE THESIS

This thesis is organized into six chapters.

- First chapter talks about general introduction to screen readers and common issues involved in building screen readers.
- Second chapter discusses the nature of Indian language scripts, their digital storage formats and need for handling the font-data especially. It describes a transliteration scheme used for our research.
- Third chapter presents font-type (font encoding name) [1] identification in Indian languages. In detail it covers the TF-IDF weight, modeling the font-data using those weights and classifying the font-type. Lastly it analyzes the performance for different cases.
- Fourth chapter presents a generic framework for building font converters for font-data conversion in Indian languages. It describes exploiting glyph shape and place information and building glyph-map table for fonts. It explains glyph assimilation process and rules for glyph assimilation in each language. In the end it analyzes the conversion performance for different Indian languages.
- Fifth chapter presents a multi-lingual screen reader for Indian languages and issues in development. It covers in detail the proposed architecture, user interfaces, functionalities, evaluation and limitations.

[1] This thesis uses the terms font, font-encoding and font-type interchangeably

- Sixth chapter summarizes the thesis work and concludes then provides the road map for the future work.

CHAPTER 2

A Review of Nature of Scripts in Indian Languages

This chapter presents a brief introduction to nature of scripts of Indian languages. Such introduction is necessary as in Indian languages the basic alphabets take multiple shapes and positioned in different places in the script (top, down, left and right). Most importantly we have utilized these position and shape characteristics of Indian language scripts in our approaches for identification of font-type and conversion of font-data. This chapter also describes various digital storage formats used for the Indian language electronic content and presents the IT3 transliteration scheme employed in this thesis.

2.1 AKSHARAS OF INDIAN LANGUAGE SCRIPTS

Indian language scripts originated from the ancient ‘Brahmi’ script. As Indian languages are phonetic based, the minimal sound units called phonemes, are identified as root case. The phonemes are divided into two groups: vowels and consonants and into further classifications. Systematic combinations of elements of these groups resulted into the basic units of the writing system are referred to as “Aksharas” [1]. The properties of Aksharas are as follows:

1. An Akshara is an orthographic representation of a speech sound in an Indian language.
2. Aksharas are syllabic in nature.
3. The typical forms of Akshara are V, CV, CCV and CCCV, thus have a generalized form of C*V where ‘C’ stands for consonant and ‘V’ stands for vowel.

[1] Akshara may mean ”syllable” of C*V form (in Sanskrit grammar)

4. An Akshara always ends with a vowel or nasalized vowel.
5. White space is used as word boundary thus separating Aksharas present in two successive words.
6. The scripts are written from left to right.
7. Roman digits (0...9) are used as numerals. Some of the languages have their own numeric symbols which are rarely used.
8. English Punctuations marks such as comma, full stops etc.,. are mostly used in writing. Languages such as Hindi have a set of their own punctuation marks which are often used. Some languages like Tamil have special symbols for date, month, year, dept and credit etc.,.

2.1.1 Convergence and Divergence

India is a multi-lingual nation with 23 recognized official languages. These languages are: Assamese, Bengali, Gujarati, Kannada, Kashmiri, Konkani, Malayalam, Manipuri, Marathi, Nepali, Oriya, Punjabi, Sanskrit, Sindhi, Tamil, Telugu and Urdu. Except Urdu and English, the remaining official languages have a common phonetic base, i.e., they share a common set of speech sounds. While these languages share a common phonetic base, languages such as Hindi, Marathi and Nepali also share a common script known as Devanagari. Languages such as Telugu, Kannada and Tamil have their own scripts. The property that makes these languages distinct is the phonotactics in each of these languages rather than the scripts and speech sounds. Phonotactics is the permissible combinations of phones that can co-occur in a language. The shape of an Akshara depends on its composition of consonants and the vowel, and sequence of the consonants. In defining the shape of an Akshara, one of the consonant symbols acts as pivotal symbol. Depending on the context, an Akshara can have a complex shape with other consonant and vowel symbols being placed on top, below, before, after or sometimes surrounding the pivotal symbol. Ideally, the basic rendering unit for Indian language scripts should be Aksharas themselves. However Telugu, for instance has around 15 vowels and 36 consonants. To render Aksharas as a whole unit, it requires 540

CV units, 19440 CCV units and 699840 CCCV units [5]. It is reasonable to assume that not all combinations of consonant clusters are allowed, but even with that assumption, 10000 Aksharas are needed as rendering units. Due to this large number of units, an Akshara is rendered by concatenating the constituent consonant and vowel symbols. The following are the symbols used to render Aksharas by a Unicode rendering engine.

2.1.1.1 Consonant Symbol

A consonant symbol in an Indian language represents a single consonant sound with the inherent vowel (short /a/). Akshara is syllabic so each consonant symbol represents a consonant and the inherent vowel. Fig. 2.1 shows the consonant symbols in Hindi and Telugu.

जानकारी
jaankaarii
दिल्ली
dillii
సర్కారు
sarkaaru
అల్లం
allan:

Fig. 2.1: Full consonant symbols in Hindi and Telugu.

2.1.1.2 Half forms of the consonant

Aksharas of the type CCV or CCCV, have more than one consonant. In these cases, one of the consonant(s) has full-form shape where others in half-form shape. These half-forms do not have an inherent vowel. An half-form consonant occupies a place on top, below, before or after the full-form consonant in the script. Fig. 2.2 shows the the half form of the consonant symbols in Hindi and Telugu.

प्रवासी
pravaasii
दिल्ली
dillii
ప్రజల
prajala
అల్లం
allan:

Fig. 2.2: Half consonant symbols in Hindi and Telugu.

2.1.1.3 Vowel symbols - independent vowels

A vowel symbol represents a vowel sound and is used to render a syllable of type V which has no consonants before or after it. These vowel symbols are also referred to as independent vowels. Fig. 2.3 shows the the vowel symbols in Hindi and Telugu.

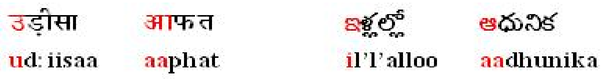


Fig. 2.3: Vowel symbols in Hindi and Telugu.

2.1.1.4 Maatra - dependent vowels

Consonants can associate with vowels other than inherent vowel. If a non-inherent vowel is needed, then a diacritical mark corresponding to the non-inherent vowel is added to the consonant symbol. These vowels with their attached diacritical marks are referred to as Maatras or dependent vowels. A Maatra can occupy a place on top, below, before, after, or sometimes surrounding the consonant symbol and thus is often referred to as dependent vowel. Fig. 2.4 shows the the Maatra symbols in Hindi and Telugu.

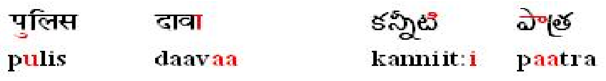


Fig. 2.4: Maatra symbols in Hindi and Telugu.

2.1.1.5 Halant

Sometimes it is necessary to write consonants without inherent vowels. To remove the inherent vowel from a consonant, a symbol called Halant is used. The symbol may be an oblique stroke under the consonant symbol, or can change the shape of the consonant if it is on top of the consonant.

2.2 DIGITAL STORAGE FORMAT

Another aspect of diversion in electronic content in Indian languages is their digital storage format. Formats like ASCII (American Standard Code for Information Interchange), ISCII (Indian Standard code for Information Interchange) and Unicode are often used to store the digital text data in Indian languages. The text is rendered using fonts of these formats. This section describes briefly about each storage format and also about fonts and glyphs.

2.2.1 ASCII Format

ASCII is a character encoding based on the English alphabets. Digital computers and operating systems in the early 90s supported only ASCII based encoding and hence many electronic news papers in Indian languages used glyph based fonts to store and render scripts of Indian languages. A font encoding stored in ASCII format specifies a correspondence between digital bit patterns and the symbols/glyphs of a written language. ASCII is strictly an eight bit code and ranges from 0 to 255.

2.2.2 Unicode Format

To allow computers to represent any character in any language, the international standard ISO 10646 defines the Universal Character Set (UCS) [4]. UCS contains the characters to practically represent all known languages in the world. ISO 10646 originally defined a 32-bit character set. Each character is assigned a 32 bit code. However, these codes vary only in the least-significant 16 bits. Table 2.1 shows the Unicode ranges for Indian languages.

UTF: A Universal Transformation Format (UTF) [15] is an algorithmic mapping from every Unicode code point (except surrogate code points) to a unique byte sequence. Actual implementations in computer systems represent integers in specific code units of particular size (8 bit, 16 bit, or 32 bit). Encoding forms specify how each integer (code point) for a Unicode character is to be expressed as a sequence of one or more code units. There are many Unicode Transformation Formats for encoding Unicode like UTF-8, UTF-16 and

Table 2.1: Unicode ranges for Indian languages.

Indian Language	**Unicode Range (Hexadecimal)**	**Unicode Range (Decimal)**
Devanagari	0900 - 097F	2304 - 2431
Bengali	0980 - 09FF	2432 - 2559
Oriya	0B00 - 0B7F	2816 - 2943
Gurmukhi (Punjabi)	0A00 - 0A7F	2560 - 2687
Gujarati	0A80 - 0AFF	2688 - 2815
Tamil	0B80 - 0BFF	2944 - 3071
Telugu	0C00 - 0C7F	3072 - 3199
Kannada	0C80 - 0CFF	3200 - 3327
Malayalam	0D00 - 0D7F	3328 - 3455
Urdu	0900 - 097F	1536 - 1791

UTF-32. Both UTF-8 and UTF-16 are substantially more compact than UTF-32, when averaging over the world's text in computers. With the advent of Unicode, UTF-8 and their support in operating systems, most of the current electronic documents are being published in Unicode specifically in UTF-8 formats. Some of the news websites which produce Indian language content in Unicode format are: BBC news, Yahoo, MSN and Google.

2.2.3 ISCII Format

In India since 1970s, different committees of the Department of Official Languages and the Department of Electronics (DOE) have been developing different character encodings schemes, which would cater to all the Indian scripts. In 1983, the DOE announced the 7-bit ISCII-83 code , which complied with the ISO 8-bit recommendations [16]. ISCII (Indian Script Code for Information Interchange) is a fixed-length 8-bit encoding. The lower 128(0-127) code points are plain ASCII and the upper 95(160-255) code points are ISCII-specific, which is used for all Indian Script based on Brahmi script. This makes it possible to use an Indian script along with Latin script in an 8-bit environment. This facilitates 8-bit bi-lingual representation with Indic Script selection code. The ISCII code contains the basic alphabet required by the Indian Scripts. All composite characters are formed by combining these basic characters. Unicode is based on ISCII-1988 and incorporates minor revisions of ISCII-1991, thus conversion between one to another is possible without loss of information.

2.2.4 Fonts and Glyphs

People interpret the meaning of a sentence by the shapes of the characters contained in it. Reduced to the character level, people consider the information content of a character inseparable from its printed image. Information technology, in contrast, makes a distinction between the concepts of a character's meaning (the information content) and its shape (the presentation image). Information technology uses the term "character" (or "coded character") for the information content; and the term "glyph" for the presentation image. A conflict exists because people consider "characters" and "glyphs" equivalent. Moreover, this conflict has led to misunderstanding and confusion. [17] explains a framework for relating "characters" and "glyphs" to resolve the conflict because successful processing and printing of character information on computers requires an understanding of the appropriate use of "characters" and "glyphs".

Historically, ISO/IEC JTC 1/SC 2 (SC 2) [17] has taken responsibility for the development of coded character set standards such as ISO/IEC 10646 for the digital representation of letters, ideographs, digits, symbols, etc.,. ISO/IEC JTC 1/SC 18 (SC 18) has developed the standards for document processing, which presents the characters coded by SC 2. SC 18 standards include the font standard, ISO/IEC 9541, and the glyph registration standard, ISO/IEC 10036. The Association for Font Information Interchange (AFII) maintains the 10036 glyph registry on behalf of ISO.

- *Character*: A member of a set of elements used for the organization, control, or representation of data. (ISO/IEC 10646-1: 1993)
- *Coded Character Set*: A set of unambiguous rules that establishes a character set and the relationship between the characters of the set and their coded representation. (ISO/IEC 10646-1: 1993)

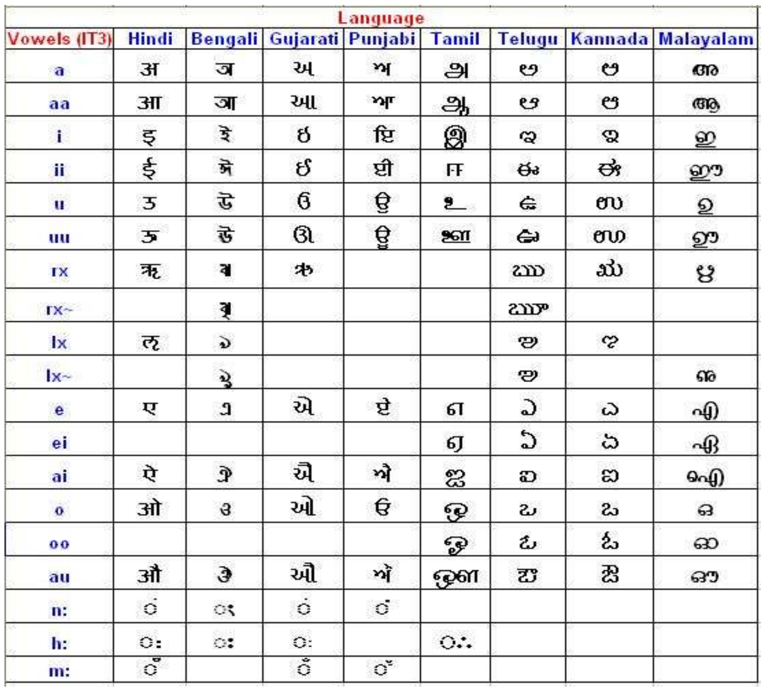
- *Font*: A collection of glyph images having the same basic design, e.g., Courier Bold Oblique. (ISO/IEC 9541-1: 1991)
- *Glyph*: A recognizable abstract graphic symbol which is independent of any specific design. (ISO/IEC 9541-1: 1991).

Indian language electronic contents is scripted digitally using fonts. A font is a set of glyphs (images or shapes) representing the characters from a particular character set in a particular typeface. Glyphs are generated by font developers by considering the frequency it appears in the script and the place/position where they appear. In typography, a typeface is a coordinated set of glyphs designed with stylistic unity. A typeface usually comprises an alphabet of letters, numerals, and punctuation marks; it may also include ideograms and symbols, or consist entirely of them. A glyph is a particular image that represents a character or part of a character. Glyphs do not correspond one-to-one with characters. A font family is typically a group of related fonts which vary only in weight, orientation, width, etc, but not design. Common font formats are META-FONT, "PostScript" Type 1, TrueType and OpenType. A font may be a discrete commodity with legal restrictions. For more details on Indian fonts, please refer to Appendix B.2 to B.13.

2.3 A PHONETIC TRANSLITERATION SCHEME

Transliteration is the method of writing script of one language using the characters of another language (usually English). To conveniently process the text by computer most of the time the Indian text content is converted and stored in these formats. Even they occupy less space on the computer. Hence they could be considered as one of the storage formats. There exist many such transliteration schemes for Indian languages like Roman WX, ITRANS etc., to key-in the Indian language scripts. Using these schemes, text of Indian languages can be written using English characters. This is a tremendous feature, which lets you use your standard English keyboard with no changes whatsoever, to create documents in Indian languages. The focus of these schemes was mainly to represent the Indian language scripts and paid less attention on the importance of user readability aspect. But the user is concerned with how to key-in the Indian language scripts. IT3 [5] is a transliteration scheme developed by IISc Bangalore, India and Carnegie Mellon University with the primary focus on user readability of the transliteration scheme [18].

Following figures shows some IT3 phones [2] for Indian languages. Fig. 2.5 shows IT3 vowel phones for Indian languages whereas Fig. 2.6 shows part of consonant phones for Indian languages. For remaining list of IT3 phones, please refer to Appendix A.1.



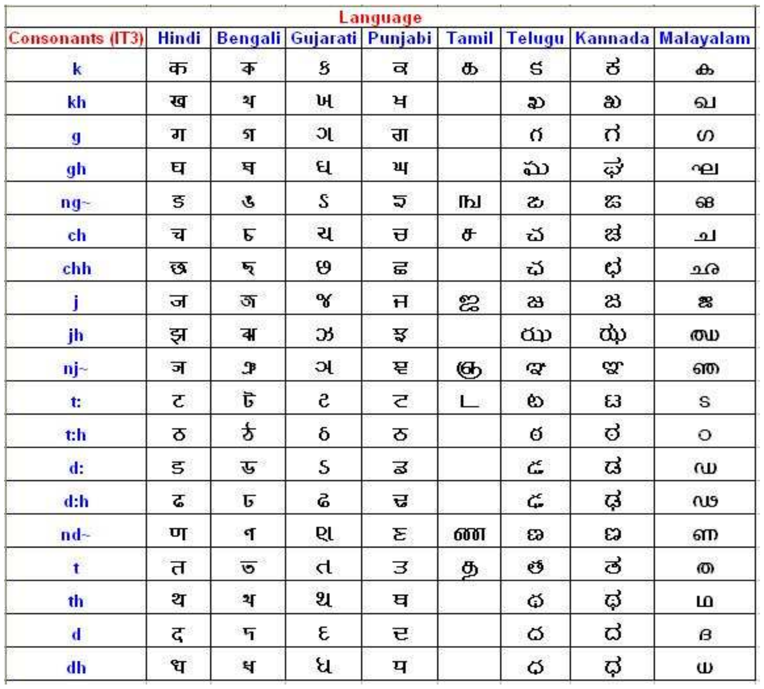
	Language							
Vowels (IT3)	Hindi	Bengali	Gujarati	Punjabi	Tamil	Telugu	Kannada	Malayalam
a	अ	অ	અ	ਅ	அ	అ	ಅ	അ
aa	आ	আ	આ	ਆ	ஆ	ఆ	ಆ	ആ
i	इ	ই	ઇ	ਇ	இ	ఇ	ಇ	ഇ
ii	ई	ঈ	ઈ	ਈ	ஈ	ఈ	ಈ	ഈ
u	उ	উ	ઉ	ਉ	உ	ఉ	ಉ	ഉ
uu	ऊ	ঊ	ઊ	ਊ	ஊ	ఊ	ಊ	ഊ
rx	ऋ	ঋ	ઋ			ఋ	ಋ	ഋ
rx~		ৠ				ౠ		
lx	ऌ	ঌ				ఌ	ಌ	
lx~		ৡ				ౡ		ഌ
e	ए	এ	એ	ਏ	எ	ఎ	ಎ	എ
ei					ஏ	ఏ	ಏ	ഏ
ai	ऐ	ঐ	ઐ	ਐ	ஐ	ఐ	ಐ	ഐ
o	ओ	ও	ઓ	ਓ	ஒ	ఒ	ಒ	ഒ
oo					ஓ	ఓ	ಓ	ഓ
au	औ	ঔ	ઔ	ਔ	ஔ	ఔ	ಔ	ഔ
n:	◌ं	◌ং	◌ં	◌ਂ				
h:	◌ः	◌ঃ	◌ઃ		◌ஃ			
m:	◌ँ		◌ઁ	◌ਁ				

Fig. 2.5: IT3 vowel phones for Indian languages.

The following are the salient points of this transliteration scheme.

1. It is case-insensitive.
2. This scheme is phonetic in nature, the characters correspond to the actual sound that is being spoken. Thus a single transliteration scheme is used for all the Indian languages, as they share the same set of sounds.
3. Each character (corresponding to a phone/sound) should be not more than three letters length.

[2]IT3 scheme uses the symbols /'/ with /:/ and /-/ with /~/ interchangeably.



	Language							
Consonants (IT3)	Hindi	Bengali	Gujarati	Punjabi	Tamil	Telugu	Kannada	Malayalam
k	क	ক	ક	ਕ	க	క	ಕ	ക
kh	ख	খ	ખ	ਖ		ఖ	ಖ	ഖ
g	ग	গ	ગ	ਗ		గ	ಗ	ഗ
gh	घ	ঘ	ઘ	ਘ		ఘ	ಘ	ഘ
ng~	ङ	ঙ	ઙ	ਙ	ங	ఙ	ಙ	ങ
ch	च	চ	ચ	ਚ	ச	చ	ಚ	ച
chh	छ	ছ	છ	ਛ		ఛ	ಛ	ഛ
j	ज	জ	જ	ਜ	ஜ	జ	ಜ	ജ
jh	झ	ঝ	ઝ	ਝ		ఝ	ಝ	ഝ
nj~	ञ	ঞ	ઞ	ਞ	ஞ	ఞ	ಞ	ഞ
t:	ट	ট	ટ	ਟ	ட	ట	ಟ	ട
t:h	ठ	ঠ	ઠ	ਠ		ఠ	ಠ	ഠ
d:	ड	ড	ડ	ਡ		డ	ಡ	ഡ
d:h	ढ	ঢ	ઢ	ਢ		ఢ	ಢ	ഢ
nd~	ण	ণ	ણ	ਣ	ண	ణ	ಣ	ണ
t	त	ত	ત	ਤ	த	త	ತ	ത
th	थ	থ	થ	ਥ		థ	ಥ	ഥ
d	द	দ	દ	ਦ		ద	ದ	ദ
dh	ध	ধ	ધ	ਧ		ధ	ಧ	ധ

Fig. 2.6: IT3 consonant phones for Indian languages (a portion).

4. There should be minimal use of punctuation marks in the composition of a character

However, attention should be paid to the following issue in using IT3 notations. IT3 being phonetic in nature, a character ‘k’ represents a consonant sound /k/ and there is no notion of inherent vowel (short /a/) associated with the consonant sound. Thus to get the syllable /ka/ one has to write a sequence of two characters ‘ka’. The issue is whether the IT3 character ‘k’ should be mapped to the half-form of the consonant /k/ or to the consonant symbol, i.e., the syllable /ka/ which is a CV unit. As IT3 is a sound based scheme, it covers most of the sound units of any new language and any new sound unit can be added to the existing IT3 codes to support a new language. Thus it is generalized and extendable also to a new language.

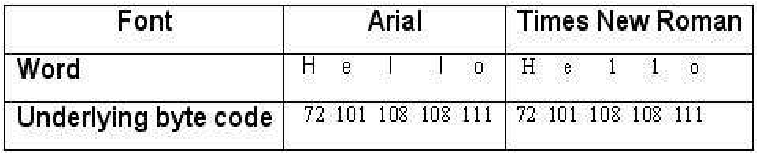
2.4 NEED FOR HANDLING FONT-DATA

There is a chaos as far as the text in Indian languages in electronic form is concerned. Neither can one exchange the notes in Indian languages as conveniently as in English language, nor can one perform search on texts in Indian languages available over the web. This is so because the texts are being stored in ASCII (as apposed to Unicode) based font dependent glyph codes. A large collection of text corpus assumes primary role in building many language technologies such as large vocabulary speech recognition or unit selection based speech synthesis system for a new language. In the case of Indian languages, the text which is available in digital format (on the web) is difficult to use as it is because they are available in numerous encoding (fonts) schemes. Applications like screen readers developed for Indian languages and other applications have to read or process such text. Given that there are 23 official Indian languages, and the amount of data available in glyph based font encoding is much larger than the text content available in Unicode and ISCII format. So it is important to have a generic framework for automatic identification of font-type and conversion of font-data to a phonetic transliteration scheme.

The glyphs are shapes, and when 2 or more glyphs are combined together form a character in the scripts of Indian languages. To view the websites hosting the content in a particular font-type then one requires that font to be installed on local machine. As this was the technology existed before the era of Unicode and hence a lot of electronic data in Indian languages were made and available in that form. The sources for these data are news websites (mainly), Universities/Institutes and some other organizations. They are using proprietary fonts to protect their data. Collection of these text corpora, identifying the font-type and conversion of font-data into a phonetically readable transliteration scheme is essential for building many natural language processing systems and applications.

A character of English language has the same code irrespective of the font being used to display it. However, most Indian language fonts assign different codes to the same character. For example (Fig. 2.7) ‘a’ has the same numerical code ‘97’ irrespective of the hardware or software platform.

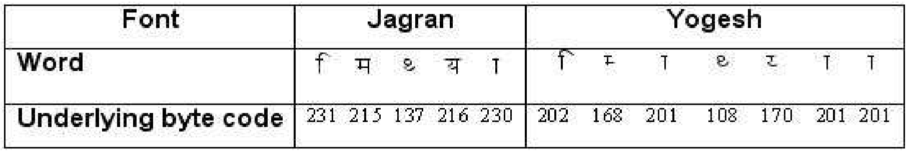
Consider for example the word “hello” written in the Roman Script and the Devanagari Script.



Font	Arial	Times New Roman
Word	H e l l o	H e 1 1 o
Underlying byte code	72 101 108 108 111	72 101 108 108 111

Fig. 2.7: Illustration of glyph code mapping for English fonts.

Arial and Times New Roman are used to display the same word. The underlying codes for the individual characters, however, are the same and according to the ASCII standard.



Font	Jagran	Yogesh
Word	ि म ९ य ा	ि म् ा ९ ट ा ा
Underlying byte code	231 215 137 216 230	202 168 201 108 170 201 201

Fig. 2.8: Illustration of glyph code mapping for Indian (Hindi) fonts.

The same word displayed in two different fonts in Devanagari, Yogesh and Jagran (Fig. 2.8). The underlying codes for the individual characters are according to the glyphs they are broken into. Not only the decomposition of glyphs and the codes assigned to them are both different but even the two fonts have different codes for the same characters. This leads to difficulties in processing or exchanging texts in these formats.

Three major reasons which cause this problem are: (i) There is no standard which defines the number of glyphs per language hence it differs between fonts of a specific language itself. (ii) Also there is no standard which defines the mapping of a glyph to a number (code value) in a language. (iii) There is no standard procedure to align the glyphs while rendering. The common glyph alignment order starts with left glyph followed by the pivotal character and followed by any of the top or right or bottom glyphs. Some font based scripting and rendering scheme may also violate this order and use their own.

As already mentioned, the basic writing unit of an Indian language is an “Akshara” and is in the form C*V. When the consonant becomes half consonant the shape changes and for vowel it becomes a Maatra. The shape changes but the sound remains same. These different shapes become the glyphs for a font-type. Sometimes a broken part of a character becomes a glyph which doesn't have correspondence to any sound/character. It will be combined with other characters to produce a valid character. The rendering engine renders a whole character by combining and positioning a set of glyphs appropriately. Because of these reasons font-data in Indian languages is very hard to process or convert to a consistent form.

2.5 SUMMARY

This chapter elaborated the nature of the Indian language scripts. Also we enumerated the convergence and divergence properties of Indian language scripts. Lensing out the various units/symbols and Aksharas of those scripts. Adopted standards of digital storage formats for scripts like ASCII, ISCII, Unicode and transliteration schemes already available is discussed. Explaining fonts and glyphs in Indian context and compared the glyph/codes for English and Hindi. A implemented transliteration scheme version IT3 which we discussed and used for research purpose. Finally it addresses the need for handling Indian context fonts specially. In the next chapter we are going to discuss font-type identification using TF-IDF weights based method.

CHAPTER 3

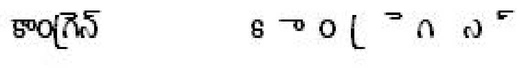
IDENTIFICATION OF FONT ENCODING

The widespread and increasing availability of huge electronic textual data for Indian languages has increased the importance of developing a generic method for font-type identification of various non-standard font encodings. Just as in any other text processing systems, screen readers have to pre-process the raw text data before giving it to TTS for synthesizing. One of the steps in pre-processing is font identification. Some forms of raw data such as HTML contain the font-type but in most cases the font-type has to be identified at the runtime such for data in MS-word or Notepad. This chapter presents our approach for identifying the font-type.

3.1 PROBLEM STATEMENT

The problem of font-type (also referred to as font-encoding) identification could be defined as: Given a set of words or sentences, identify the font-type by finding the minimum distance between the input glyph codes and the models representing font-types.

For example consider a word and its glyph sequence is shown in the figure below.



So the input would be a sequence of ASCII values of the glyph sequence such as.

194 195 162 147 233 146 174 253

The issue is to find out whether this glyph sequence belongs to Eenadu or Vaarttha or Amarujala etc.,. It should be noted that this identification is done across all font-types belonging to different Indian languages.

3.2 RELATED WORK ON FONT-TYPE IDENTIFICATION

Our problem of identification of font-type is a classification problem bearing similarities to other problems such as Language Identification [1] and Font-Type Identification in OCR (Optical Character Recognition) [2]. In language identification, the goal is to identify the natural language of the given input data. Different types of approaches followed in language identification are word n-gram, character n-gram, syllable characteristics, morphology, syntax etc.,. Similarly, there is a lot of work done for font-type identification in OCR development but it is little different from the current problem as here the input is image and the goal is to identify font-type based on font attributes such as size, style etc.,. In order to see the similarity of both the problems and the applicability of those approaches for font-type identification, a brief review of both is reported in this chapter.

3.2.1 Related Work in Language Identification

Ingle [19] used a list of short characteristic words in various languages and matched the words in the test data with this list. Such unique strings based methods were meant for human translators. The earliest approaches used for automatic language identification were based on that same idea of unique strings. They were called "translator approaches".

Beesley's [20] automatic language identifier for on-line texts used mathematical language models originally developed for breaking ciphers. These models basically relied on orthographic features like characteristic letter sequences and frequencies for each language.

Some of the methods were similar to n-gram based text categorization (Cavnar and Trenkle) [21] which calculates and compares profiles of n-gram frequencies. Cavnar also proposed that top 300 or so n-grams are almost always highly correlated with the language, while the lower ranked n-grams give more specific indication about the text, namely the topic. Many approaches similar to Cavnar's have been tried, the main difference being in

[1]Language identifi cation is the process of determining which natural language the given content is in

[2]OCR is the mechanical or electronic translation of images of handwritten, typewritten or printed text (usually captured by a scanner) into machine-editable text

the distance measure used. The similarity measures tried for language identification include mutual information or relative entropy, also called Kullback-Leibler distance (Sibun and Reynar) [22], cross entropy (Teahan and Harper) [23], and mutual or symmetric cross entropy (Singh 2006).

Giguet [24] relied upon grammatically correct words instead of the most common words. He used the knowledge about the alphabet and the word morphology via syllabication. He tried this method for tagging sentences in a document with the language name, i.e., dealing with multilingual documents.

Johnson's method (Stephen 1993) was based on characteristic 'common words' of each language. This method assumes unique words for each language. In practice, the test string might not contain any unique words.

Cavnar's method, combined with some heuristics, was used by Kikui [25] to identify languages as well as language-encodings for a multilingual text. He relied on known mappings between languages and encodings and treated East Asian languages differently from West European languages.

A K Singh [26] used character based n-grams and noted the language identification problem here is that of identifying both language and encoding. This is because (especially for South Asian languages) the same encoding can be used for more than one languages (ISCII for all Indian languages which use Brahmi-origin scripts) and one language can have many encodings (ISCII, Unicode, ISFOC, typewriter, phonetic, and many other proprietary encodings for Hindi). He conducted a study on different distance measures. Most of them are based on either well known or based on well known measures, but the results obtained with them vis-a-vis one-another might help in gaining an insight into how similarity measures work in practice. He has proposed a novel measure using mutual entropy resulting improved performance better than other measures for the current problem. The work describes some experiments on using such similarity measures for language and encoding identification.

There has been no comparable systematic work on multilingual text documents, although there has been some work based on an Optical Character Recognition (OCR) system, such as by Tan et al. (1999). One attempt at multilingual identification was by Prager (1999). His Linguini system uses a vector space based monolingual identifier to also find out the component languages of a document and the relative proportions of each. Artemenko et al. (2006) tried for identifying the languages in a document and have reported accuracy of 97%. But neither of them identified languages of segments.

A.K.Singh and Jagadeesh [27] extended the previous work for identifying the language-encoding pair for multilingual documents. They divided the problem into three parts: monolingual identification, enumeration of languages and identification of the language of every portion. For enumeration, they have been able to get a precision of 96.20%. They also experimented on language identification of each word. Given correct enumeration, they could obtain type precision of 90.91% and token precision of 86.80%. They even showed how precision is affected by language distance.

3.2.2 Related Work in Font Recognition in OCR

Font recognition plays an important role in OCR systems [28]. People tried to achieve it in a simple and effective way with texture analysis regarding fonts as different textures. Gabor filters with traditional parameters were used to extract texture features. Since the Recognition Rate (RR) was low for similar fonts some attempts tried some adjustment to improve it. The other approach is content independent and involves no local feature analysis. Global features are extracted by texture analysis. They applied the well-established 2D Gabor filtering technique to extract such features and a weighted Euclidean distance classifier to fulfill the recognition task. Experiments are made using 6,000 samples of 24 frequently used Chinese fonts (6 typefaces combined with 4 styles) and very promising results were achieved.

In [29], a multi-font classification scheme to help with the recognition of multi-font and multi-size characters. It uses typographical attributes such as ascenders, descenders and serifs obtained from a word image. The attributes are used as an input to a neural network classifier to produce the multi-font classification results. It can classify 7 commonly used fonts for all point sizes from 7 to 18. The approach developed in this scheme can handle a wide range of image quality even with severely touching characters. The detection of the font can improve character segmentation as well as character recognition because the identification of the font provides information on the structure and typographical design of characters. Therefore, this multi-font classification algorithm can be used for maintaining good recognition rates of a machine printed OCR system regardless of fonts and sizes. Experiments have shown that font classification accuracies reach high performance levels of about 95 percent even with severely touching characters.

In [30], simple and fast algorithm to detect Thai and English characters in a document without doing actual characters recognition is described. The document is segmented into strings of letters separated by a blank, then each string is identified using characters features and their writing positions. This method achieves 100% accuracy if the characters have clear head feature. But if this feature is not used 90% of the strings still can be identified. This identification provides more information about the character set so that OCR can recognize faster with better accuracy.

3.3 TF-IDF APPROACH

As discussed in the above sections, the solutions for language identification make use of word n-grams or character n-grams. In using n-gram based approach typically it is assumed that there is sufficient amount of training data to get a better estimate of likelihood scores. However, in our case of identification of font-type, we would expect to detect the font-type from sparse data such as a single word, thus approaches such as word level n-grams are typically not applicable for our approach. The use of character n-grams is another alternative, however, the definition of character is more applicable to a language than to a font-type as the data corresponding to a font-type are a sequence of byte codes. At the same time, the approaches adopted in font-type identification from OCR deals with image data. While there are many efforts to identify the language, but few efforts are made to identify the font-type from sparse text data specifically given the large number of font-types in Indian languages.

Here we propose to use vector space model and Term Frequency - Inverse Document

Frequency (TF-IDF) weights based approach for identification of font-type. This approach is used to weigh each term in the font-data according to how unique it is. In other words, the TF-IDF approach captures the relevancy among term and document. To perform TF-IDF approach, it is essential to define what a "term" is and what a "document" is.

- Term: It refers to a unit of glyph. In this work we have experimented with different units such as single glyph g_i (uniglyph), two consecutive glyphs $g_{i-1}g_i$ (biglyph), three consecutive glyphs $g_{i-1}g_ig_{i+1}$ (triglyph).
- Document: Document refers the 'font-data (words and sentences) in a specific font-type'.

3.3.1 TF-IDF Weights

The TF-IDF (Term Frequency - Inverse Document Frequency) weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a term/word appears in the document.

The term frequency in the given document is simply the number of times a given term appears in that document. This count is usually normalized to prevent a bias towards longer documents (which may have a higher term frequency regardless of the actual importance of that term in the document) to give a measure of the importance of the term t_i within the particular document.

$$tf_i = \frac{n_i}{\sum_k n_k} \tag{3.1}$$

with n_i being the number of occurrences of the considered term, and the denominator is the number of occurrences of all terms.

The document frequency is the number of documents where the considered term has occurred at least once.

$$|\{d : d \ni t_i\}| \tag{3.2}$$

The inverse document frequency is a measure of the general importance of the term (it is the logarithm of the number of all documents divided by the number of documents containing the term).

$$idf_i = log \frac{|D|}{|\{d : d \ni t_i\}|} \tag{3.3}$$

with $|D|$ total number of documents in the corpus

Here ‘log’ is used in this formula to smoothen the values.

$|\{d : d \ni t_i\}|$: Number of documents where the term t_i appears (that is $n_i \neq 0$) Then

$$tfidf = tf.idf \tag{3.4}$$

A high weight in TF-IDF is reached by a high term frequency (in the given document) and a low document frequency of the term in the whole collection of documents; the weights hence tend to filter out common terms.

3.3.2 Usage of TF-IDF Weights for Modeling

The motivation for the use of TF-IDF for modeling font-data comes from the fact that TF-IDF weights give the relevance between a term and collection of such terms. In font modeling, we consider this term can be contextual glyph (like sequence of glyphs) and the document could be a collection of such terms. Because in Indian languages Aksharas are formed by combinations of glyphs and these glyphs have different code values (numbers). So the code combinations change from font to font. Certain glyph code combinations will be unique to a font-type. Such combinations get high TF-IDF weight than other combinations which helps us to identify the font-type. So in this context, the *term* refers a ‘glyph’ or ‘a sequence of glyphs’ and the *document* refers the ‘font-data (words and sentences) in a specific font-type’. Here the glyph-sequence means uniglyph (single glyph), biglyph (current and next glyph) and triglyph (previous, current and next glyph) etc.,. So this approach tries to capture the relevance among glyph sequence, font-type and font-data. This approach also gives different weights to a term

- occurring in one font-type (document)
- occurring in two font-types
- occurring in many font-types
- occurring in all font-types

3.4 MODELING AND IDENTIFICATION

Data Preparation: For font modeling we need sufficiently enough data of that particular font-type. So we have collected manually and used an average of 0.12 million unique words per font-type. We used nearly 37 differently encoded glyph based font data.

Modeling: Generating a vector space model for each font-type using TF-IDF weights is known as modeling the font-data. For modeling we considered three different types of terms. They are (i) uniglyph (single glyph), (ii) biglyph (current and next glyph) and (iii) triglyph (previous, current and next glyph). And the document refers a collection of data (words or sentences) of a specific font-type.

The procedure for building the models is: First we have taken all the provided data at once and also we have considered three different kinds of terms for building models as mentioned above. (i) First step is to calculate the term frequency (Eqn (4.1)) for the term like, the number of time that term has occurred divided by the total number of terms in that specific type of data. So it will be stored in a matrix format of $N * 256$ for uniglyph, $N * 256 * 256$ for biglyph and $N * 256 * 256 * 256$ for triglyph model depending upon the term. Where 'N' denotes the number of different data types and 256 (0 to 255) is the maximum number value for a single glyph. (ii) Second step is to calculate document frequency (Eqn (4.2)) like, in how many different data types that specific term has occurred. (iii) Third step is to calculate inverse document frequency (Eqn (4.3)) like, all data types divided by the document frequency. Logarithm of inverse document frequency is taken for smoothing purpose. (iv) Fourth step is to compute TF-IDF which is calculated like term frequency * inverse document frequency (Eqn (4.4)). Finally that matrix will be updated with these values. The common terms across the different font-type documents get zero values and other terms get non-zero values depending upon their term frequency values. From those values the models for each data type is generated.

Identification: Given a set of vectors generated out of a sequence of glyph codes the objective is to identify the font-type by finding the minimum distance between the query vector and the vector space models. The steps involved in identification of font-type are as follows:

- Extract terms (glyph codes) from the input word or sentence
- Create the query vectors (uniglyph or biglyph etc.,) using these terms
- Compute the distance between the query vectors and all the models of font-type using TF-IDF values
- The input word is said to be originated from the model of font-type which gives a maximum TF-IDF weightage
- Get the TF-IDF weight of each word of sentence from the models of all font-types

3.5 RESULTS OF FONT IDENTIFICATION

Test Data: It is typically observed that TF-IDF weights are more sensitive to the length of query. The accuracy increases with the increase in the length of test data. Thus two types of test data were prepared independently for testing. One is set of unique words and the other one is set of sentences.

Testing Criteria: While testing we are identifying the closest matching models for the given inputs. And we are evaluating the identification accuracy in (%) as given below.

$$Accuracy = \frac{Correct}{Total} \tag{3.5}$$

Where $Correct$: number of correctly identified tokens and $Total$: total number of tokens.

Testing: The accuracy of a font-type identification depends on various factors:

- The number of encodings from which the identifier has to select one
- The inherent confusion of one font encoding with another and
- The type of unit used in modeling.

For a given 'X' number of different inputs we identified the closest models and calculated the accuracy. It is done (repeatedly) for various (uniglyph, biglyph and triglyph) categories. The results are tabulated below.

Font-Type Identification Results: The testing is done for 1000 unique sentences (Set 1) and words obtained from 1000 unique sentences (Set 2) per font-type. The evaluation results are tabulated below. We have added English data as also one of the testing data set, and is referred to as "English-Text". Table 3.1 shows the performance results for uniglyph (current glyph) based models; Table 3.2 shows the performance results for biglyph (current and next glyph) based models and Table 3.3 shows the performance results for triglyph (previous, current and next glyph) based models. From Tables 3.1, 3.2 and 3.3, it is clear that triglyph seems to be an appropriate unit for a term in the identification of font-type. It can also be seen that the performance at word and sentence level is nearly 100% with triglyph. The results also gives the notion that the performance increases when larger the glyph-sequence taken for building font-type models, subject to reasonable coverage in the training data.

3.6 SUMMARY

This chapter answers where and why we need font-type identification. Screen reader application as an example was discussed along with challenges and practical usage. In this chapter we have proposed TF-IDF weights based approach for font-type identification. We have explained the TF-IDF weight calculation and how that is used for modeling using uniglyph, biglyph and triglyph. Identification results from different models show that the triglyph or

Table 3.1: Font-type identification: Uniglyph (current glyph) based font models performance.

Font Name	Identifi cation Accuracy (Sentences)	Identifi cation Accuracy (Words)
Amarujala (Hindi)	100%	100%
Jagran (Hindi)	100%	100%
Webdunia (Hindi)	100%	0.1%
SHREE-TEL (Telugu)	100%	7.3%
Eenadu (Telugu)	0%	0.2%
Vaarttha (Telugu)	100%	29.1%
Elango Panchali (Tamil)	100%	93%
Amudham (Tamil)	100%	100%
SHREE-TAM (Tamil)	100%	3.7%
English-Text	0%	0%

Table 3.2: Font-type identification: Biglyph (current and next glyph) based font models performance.

Font Name	Identifi cation Accuracy (Sentences)	Identifi cation Accuracy (Words)
Amarujala (Hindi)	100%	100%
Jagran (Hindi)	100%	100%
Webdunia (Hindi)	100%	100%
SHREE-TEL (Telugu)	100%	100%
Eenadu (Telugu)	100%	100%
Vaarttha (Telugu)	100%	100%
Elango Panchali (Tamil)	100%	100%
Amudham (Tamil)	100%	100%
SHREE-TAM (Tamil)	100%	100%
English-Text	100%	96.3%

Table 3.3: Font-type identification: Triglyph (previous, current and next glyph) based font models performance.

Font Name	Identifi cation Accuracy (Sentences)	Identifi cation Accuracy (Words)
Amarujala (Hindi)	100%	100%
Jagran (Hindi)	100%	100%
Webdunia (Hindi)	100%	100%
SHREE-TEL (Telugu)	100%	100%
Eenadu (Telugu)	100%	100%
Vaarttha (Telugu)	100%	100%
Elango Panchali (Tamil)	100%	100%
Amudham (Tamil)	100%	100%
SHREE-TAM (Tamil)	100%	100%
English-Text	100%	100%

longer glyph sequence based models perform better. The following chapter presents font conversion and a framework to build font converters for Indian languages.

CHAPTER 4

CONVERSION OF FONT-DATA

Chapter 3 presented techniques for font-type identification. This chapter presents the next step in text processing, i.e., font-data conversion into a phonetic transliteration format. Font conversion means conversion from glyph (font unit) to grapheme (*Akshara* - written language unit). In typography [1], a glyph refers to a particular graphical representation of a grapheme, or a combination of several graphemes (a composed glyph), or only a part of a grapheme. Graphemes [2] include letters, Chinese characters, Japanese characters, numerals, punctuation marks and other glyphs. In computing as well as typography, the term character refers to a grapheme or grapheme-like unit of text, as found in natural language scripts. A character or grapheme is a unit of text, whereas a glyph is a graphical unit. Most glyphs originate from the characters of a typeface, where each character typically corresponds to a single glyph. There are exceptions such as a font used for a language with a large alphabet or complex writing system, where one character may correspond to several glyphs, or several characters to one glyph. In graphonomics [3], the term glyph is used for a non-character, (i.e) either a sub-character or multi-character pattern.

This chapter explains the generic framework for building font converters and the process of glyph assimilation for font-data conversion in Indian languages. As we already have discussed in Chapter 2 that the characters (Aksharas) are split up into glyphs in font-data and the solution would be merging up the glyphs to get back the valid character. A generic framework has been designed for building font converters for Indian languages based on this

[1]Typography is the art and techniques of type design, modifying type glyphs, and arranging type

[2]a grapheme is the fundamental unit in written language

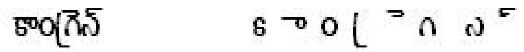
[3]Graphonomics is the interdisciplinary fi eld directed towards the scientifi c analysis of the handwriting process and the handwritten product

idea using glyph assimilation rules. The framework consist of two phases, in the first phase we build the glyph-map table for each font-type and in the second phase define and modify the glyph assimilation rules for each language [4].

4.1 PROBLEM STATEMENT

The problem of font-data conversion is defined as: Given a sequence of glyphs the objective is to rearrange and combine them to form a valid character (Akshara) in the language.

For example consider a word and its glyph sequence in the figure below.



So the input would be a sequence of ASCII values of the glyph sequence such as.

194 195 162 147 233 146 174 253

The issue is to find out how these glyph sequences can be rearranged, combined and mapped unambiguously to Asksharas of a language. Since Aksharas are represented using IT3 transliteration scheme in our work a valid output of the font-conversion process for the above glyph sequence is expected to be *k aa n' g r e s*.

4.2 RELATED WORK ON FONT-DATA CONVERSION

Conversion of Indian language electronic data to a machine processable format has become an essential task in natural language processing. Several attempts were made at font conversion. While many focused on text data few were on font data embedded in images. Many attempts were commercially done where the source code is not released for free use in other applications. Some of such tools are given below for reference. It is to be noted that each of these tools work on one or a few languages but are not generic to all Indian languages. Also support for a new font is not easy.

[4]Assimilation is the process of receiving new facts or of responding to new situations in conformity with what is already available to consciousness

(a) *TBIL Data Converter* [31]: The TBIL Data Converter is a handy tool for the quick and effective transliteration between data in font/ASCII/Roman format in Office documents into a Unicode form in any of 7 Microsoft supported Indian languages. It is developed for the Bhasha project.

(b) *FontSuvidha* [32]: A tool for Devanagari Font Conversion. It allows font conversion within Devanagari fonts like Times New Roman to Arial in English. Now it also supports any Devanagari font to Unicode font and vice versa. It works on any document type like DOC, MDB, XLS, TXT, HTM as well as anything in the clipboard.

(c) *IConverter* [33]: A utility program for various code conversions. The program uses the library 'isciilib' for code conversion. The inputs to the program are: (1) Configuration file (2) Source code file. The code conversion is done according to the rules specified in the configuration file. If, no matching rule(s) is found for any particular code, the code is passed unchanged to the output stream. The configuration files for various code conversions are provided. Configuration files for different types of conversions can easily be written. This program is developed at IIT Kanpur.

The following are freely released attempts in font conversion. Since they released paper or documentation to refer they provide more insight to understand the problem. We explored and extended them to cover all Indian languages. They followed different approaches for different purposes so the output also varies. One thing we would like to mention here is that they worked on few (may be 2 or 3) Indian languages only.

(a) Akshar Bharati et al. [34] have tried to generate converters between fonts semi-automatically. Since Indian language texts do not follow any coding standards, the long term answer might be switch to a standard alphabetic coding scheme (ACII - Alphabetic Code for Information Interchange). Based on this concept they developed a system (converter) for Devanagari fonts. The system takes: (i) a text in an unknown coding scheme (font) and (ii) the same text transliterated in the ACII coding scheme, and generates a converter between the given unknown coding scheme and ACII. The converter can be used to convert a text from the non-standard coding scheme to ACII, and back. It can be progressively refined, manually. For generating the converter, a glyph-grammar for the script of the language is also needed, which specifies what possible glyph sequences make up an Akshara. The grammar is independent of the coding schemes, and is structured. It needs to be developed only once, for a script. The glyph-grammar can have at least three types of rules: Type 1 rules are applicable across all the Indian languages like schwa (‘a’ sound) deletion from an Akshara, Type 2 rules hold for Devanagari, and perhaps for other scripts. Type3 rules pertain to idiosyncratic combination of glyphs for that particular style.

Drawbacks: Some glyphs might remain unconverted because the training data (namely, the given ACII file and the corresponding initial part of glyph file) might not be comprehensive, and the program might not have seen some glyphs as they do not occur in this learning data. It is hoped that once a large part of the file is converted, the user will be able to supply the missing code map and the rules manually, (by looking at the remaining part of the glyph file and making intelligent guesses) using which the converter would become complete.

Result: The overall converter, it is expected, will be able to convert 90 to 95% of the glyphs with a page or two of sample ACII text.

(b) Garg [35] has worked on overcoming font and script barriers among Indian languages as an extension of the work done by Akshar Bharati (et al.). He tried two different approaches for building font converters. (i) First one is *Glyph Grammar* based approach. Here description of each glyph in the font is given using mnemonics in the font glyph description file. If a glyph has equivalent ISCII/Unicode characters the equivalents are given. Otherwise, a mnemonic is used. A sample mnemonic would be like “&TC1@2” where *top, left, right or bottom* and whole character and position in that character. Mnemonics are used to list all the smallest possible logical units that a glyph can be a part of. A logical unit is a glyph or a group of glyphs combined, that can be encoded by the Unicode/ISCII encoding system. Time can be reduced if mnemonics are reused from a font that contains similar glyphs. Some amount of manual editing may be required to make corrections. The user specifies the various logical units a glyph can be a made of. The program uses the context to decide which logical unit the glyph along with its neighbors gets converted. The information about glyphs can be obtained from the glyph table of a font.

Drawbacks: (i) It is difficult to list all possible semantics for a glyph. (ii) Some training in understanding the notation (and the nature of script) is required to be able to describe new glyphs.

Results: Devanagari (Jagran / Kruti / Shusha) 90% and Telugu (Eenadu / Vaarttha) 75%.

(ii) Second one is *Finite State Transducer* based approach. This approach uses example based machine learning techniques to generate Finite State Transducers. Transducer learning algorithms have been used in the EUTRANS project which aims at using example based approaches for the automatic development of machine translation systems for limited domain applications and have been shown to give a low word error rate. These Finite State Transducers can be automatically learned from syllable mappings of font encoded text to Unicode text. The Finite State Transducers encode the correspondence between glyphs and equivalent ISCII/Unicode characters. These can be automatically learned from parallel corpora. The input to these is a parallel list of syllables. The training set consists of a parallel corpus consisting of syllables as a sequence of glyphs and their translation into ISCII. These transducers offer a clear advantage over other statistical machine learning techniques, in that the models that they depict are easily interpretable by humans. As a result they are potentially modifiable to suit one's needs. Moreover domain knowledge about the input or the output can be incorporated to improve their performance.

Drawbacks: (i) finite state transducers cannot learn glyph movement. (ii) the examples required for training must be units smaller than whole words.

Results: Devanagari (Jagran / Kruti / Shusha) 98% and Telugu (Eenadu / Vaarttha) 93%.

(c) Khudanpur and Schafer [36] have followed a new approach in font conversion. They have used two kind of mapping files for a font character (glyph). One is *CHARMAP* file which gives a mapping between the character codes (in ASCII) to a ITRANS character string like given below.

159 = shr

168 = i.n

177 = h

126 = @3

154 = .N

The other is *CLASSMAP* file which gives a mapping between the character codes (in ASCII) to a class defined by them like given below. These classes provide the notion about the place of occurrence and other informations.

255 LFRAG (Left Fragment)

22 RFRAGxV (Right Fragment Vowel)

205 SYL (Syllable or Akshara)

209 RFRAG (Right Fragment)

While converting the font text, they retrieve the character code value (in ASCII) of that character delimited by space and word boundary by ";". Even though this format occupies more space and takes more processing time, compromised in the interest of convenience. In the next step through some well defined rules for syllables in one font they extract the syllables of the word. For each syllable they perform vowel assimilation to reveal the original vowel. Further processing is done to convert to ITRANS. They have reported this on Devanagari fonts. They insist on rapid font converter building than on the accuracy (which was believed to be achieved with some additional modifications).

(d) Kumar et al. [37] have worked on developing semi-automated or automated tools for developing OCRs for Indian scripts. Based on study on the nature or features of

Indian language scripts, they identified four main problems in development of OCRs like (a) the number of letters in the alphabet, the typical number being 50, (b) richness of the structure of basic unit in the language as the combination of up to 3 consonants and 1 vowel forms the basic unit, (c) the variations in the graphical form of the different combinations even within the same script, and (d) non-adherence to any standard structure while designing the fonts. It is to be noted that they have worked on image data. They collected large data (images of printed hard copies) for training as well as for testing. The segmentation algorithm is designed to produce the segments (graphemes) of the images then suitable features are extracted. The boundaries of the segments are tuned based on the output on the train data. The OCR is then tested against a new script's data and tuned. They found it working well but they face problem when the font size is small.

The inference from above discussed works are (i) mapping between the glyph code to some meta format like phonetic notation or some grammar is essential and (ii) on this meta data either rules have to be defined or the machine has to be trained to learn using some parallel corpora. To achieve the maximum performance following requirements are to be met.

- the mapping from glyph codes to a meta notation should be unambiguous.
- in rule based conversion, stage by stage conversion has to happen. And those stages should be in a specific order.
- in machine learning based conversion, the parallel corpora should have covered all possible combinations.
- manual intervention or corrections should be minimal.

In our work, we do an unambiguous glyph code mapping by attaching the position number along with the phonetic notation. We have identified the different stages in the conversion and have ordered them according to their precedence. We have written well defined rules under each stage. Earlier approaches need the effort to be repeated for each font whereas in our approach the effort is for only three different fonts of different font-families of a language to build the converter. To add a new font, only glyph-map table is required and no more repetition of effort. Hence we provide a solution at each language level and a new font could be added easily.

4.3 EXPLOITING SHAPE AND PLACE INFORMATION

As we have seen already in Chapter 3 (could also be observed from Fig. 4.1 and Fig. 4.2), the natural shape of a vowel or consonant changes when they become half-consonant or Maatra respectively. These symbols get attached in different places (top, bottom, left and right) while forming an Akshara. For instance if we consider the character /r/, the shape and place of attachment in the Akshara are different across languages. From the below shown figures, Fig. 4.1 shows how /r/ changes its shape and place in Hindi and Fig. 4.2 shows how /r/ changes its shape and place in Telugu. In this work we intend to capture the information regarding the change of the shape of Akshara depending on the position, and incorporate it in our font-data conversion process.

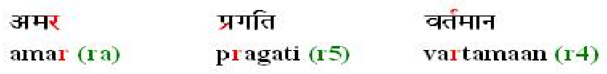


Fig. 4.1: Shape and position change for /r/ in Hindi.

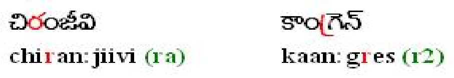


Fig. 4.2: Shape and position change for /r/ in Telugu.

The novelty of our approach lies in exploiting positional information of a glyph in Akshara to perform the conversion. We exploit this information for building the glyph-map table for a font-type. It is possible for a native speaker of the language by looking at the shape of the glyph to say whether that glyph occurs in center, top, left, right, bottom position of an Akshara. Thus we followed a simple number system to indicate the position information of a glyph.

- Glyphs which could be in pivotal (center) position are referred by code 0/1.
- Glyphs which could be in left position of pivotal symbol are referred by code 2.
- Glyphs which could be in right position of pivotal symbol are referred by code 3.
- Glyphs which could be in top position of pivotal symbol are referred by code 4.
- Glyphs which could be in bottom position of pivotal symbol are referred by code 5.

Following figures depict the position number assignment to glyphs. Fig. 4.3 shows the position number assignment for glyphs for Hindi and Fig. 4.4 for Telugu. In the top portion of the figure it shows the word considered here and right below it distinguishes different positions pictorially.

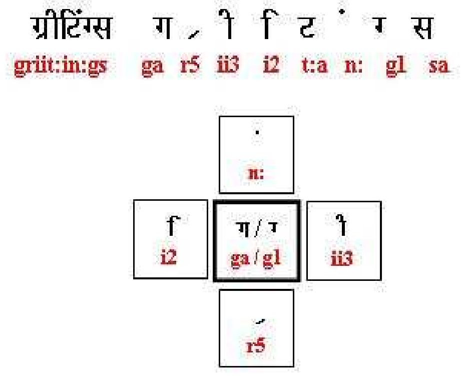


Fig. 4.3: Assigning position numbers for Hindi glyphs.

Independent vowels are assigned ‘0’ or nothing as position number. All the dependent vowels known as “Maatras” occur at left or right or top or bottom side of a pivotal character (vowel). So they are attached with positional numbers like 2 (left), 3 (right), 4 (top) and 5 (bottom). This is common for vowels across all Indian languages. Most of the consonants will be either full or half and occur at center. So accordingly they are assigned 0 or 1 as positional number. Some of the half consonants also occur rarely at the left, right, top and bottom of a full character. They are assigned a positional number like 2, 3, 4 and 5 according

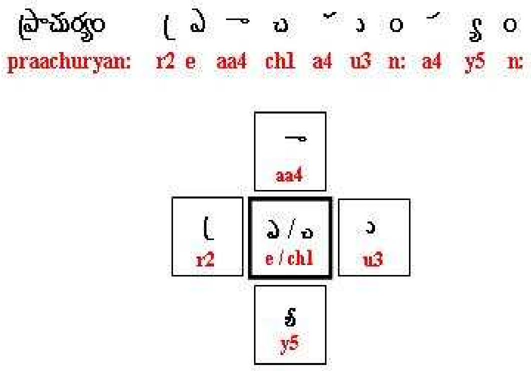


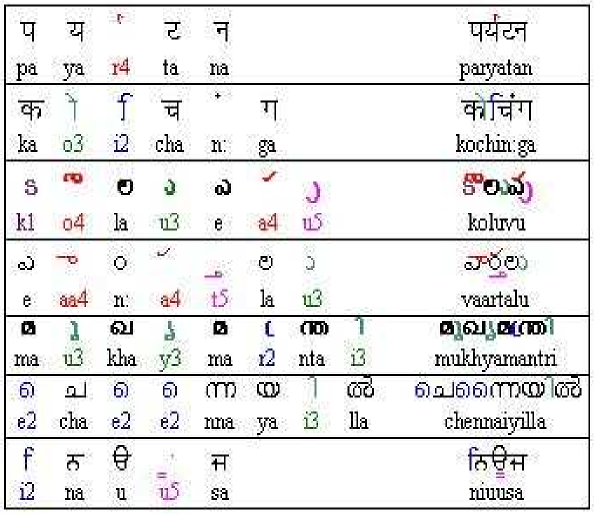
Fig. 4.4: Assigning position numbers for Telugu glyphs.

to their place. But these are specific to some Indian languages but are not common for all Indian languages.

4.4 BUILDING GLYPH-MAP TABLE

Glyph-Map table is a map table which gives a mapping between the glyph code (0 to 255) to a phonetic notation. This table gives us the basic mapping between the glyph coding of the font-type to a notation of a transliteration scheme. As a novelty in our approach we have attached some number along with the phonetic notation to indicate the position of occurrence of that glyph in that Akshara. The various position or places of occurrence of such glyphs are: left, right, top and bottom of a pivotal character. The way the position numbers are assigned is ‘0’ or nothing for a full character (eg: e, ka), ‘1’ for a half consonants (eg: k1, p1), ‘2’ for glyphs occur at left hand side of a pivotal character (eg: i2, r2), ‘3’ for glyphs occur at right hand side of a pivotal character (eg: au3, y3), ‘4’ for glyphs occur at top of a pivotal character (eg: ai4, r4) and ‘5’ for glyphs occur at bottom of a pivotal character (eg: u5, t5).

To illustrate with some real font text we have given below in the Fig. 4.5 the position numbers for glyphs. Glyphs occur in different places or positions are grouped in different colors. We have covered vowel glyphs as well as consonant glyphs here. For more on glyph-mapping for different Indian languages, please refer to Appendix B.2 to B.13.



प	य	र्‍	ट	न				पर्यटन
pa	ya	r4	ta	na				paryatan
क	ो	ि	च	ं	ग			कोचिंग
ka	o3	i2	cha	n:	ga			kochin:ga
క	ొ	ల	ు	వ	✓	ు		కొలువు
k1	o4	la	u3	e	a4	u5		koluvu
వ	ా	ం	✓	్త	ల	ు		వార్తలు
e	aa4	n:	a4	t5	la	u3		vaartalu
മ	ു	ഖ	്യ	മ	്ര	ന്ത	ി	മുഖ്യമന്ത്രി
ma	u3	kha	y3	ma	r2	nta	i3	mukhyamantri
െ	ച	െ	െ	ന്ന	യ	ി	ല്ല	ചെന്നൈയില്ല
e2	cha	e2	e2	nna	ya	i3	lla	chennaiyilla
ਿ	ਨ	ੳ	ੂ	ਸ				ਨਿਊਸ
i2	na	u	u5	sa				niuusa

Fig. 4.5: Examples for assigning position numbers for glyphs.

The position numbers along with the phonetic notation help us to form well distinguished glyph assimilation rules, because when similar glyphs are attached in different positions they form different characters. For each and every font in a language the glyph-map table has to be prepared manually in this way. That means the glyph-map table converts different font encoding to a meta data which is in a phonetic notation. It is observed that when we do such glyph-mapping for three different fonts of a language we have covered almost 96% of possible glyphs in a language.

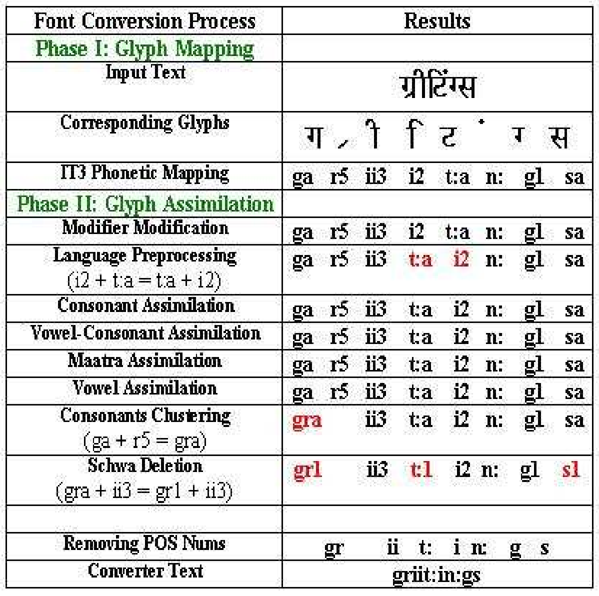
4.5 GLYPH ASSIMILATION

Glyph Assimilation is known as the process of merging two or more glyphs and forming a single valid character. This happens in many levels like consonant assimilation, Maatra assimilation, vowel assimilation and consonant clustering and in that order.

The identification of different levels and ordering them is another important thing we present here. Broadly they can be classified into four levels. They are language preprocessing, consonant assimilation, vowel assimilation and schwa deletion. Under language preprocessing level, language specific modifications like Halant and Nukta modifications are carried out first. Because after this process only valid character glyphs (and not symbols) are allowed. The next step is to reconstruct the pivotal consonant in an Akshara if it is present. This can be done under three sub-levels like consonant assimilation, consonant and vowel assimilation and consonants clustering. Then we can form the vowels from the left out glyphs. Finally we have to do the schwa deletion because when a consonant and Maatra merge together, the inherent vowel (schwa) has to be removed.

This assimilation process has been observed across many Indian languages and found that they follow certain order. The order is: (i) Modifier Modification, (ii) Language Preprocessing, (iii) Consonant Assimilation, (iv) Consonant-Vowel Assimilation, (v) Maatra Assimilation, (vi) Vowel-Maatra Assimilation, (vii) Consonants Clustering and (viii) Schwa Deletion. The idea is to first reveal the pivotal consonant in an Akshara and then the vowels.

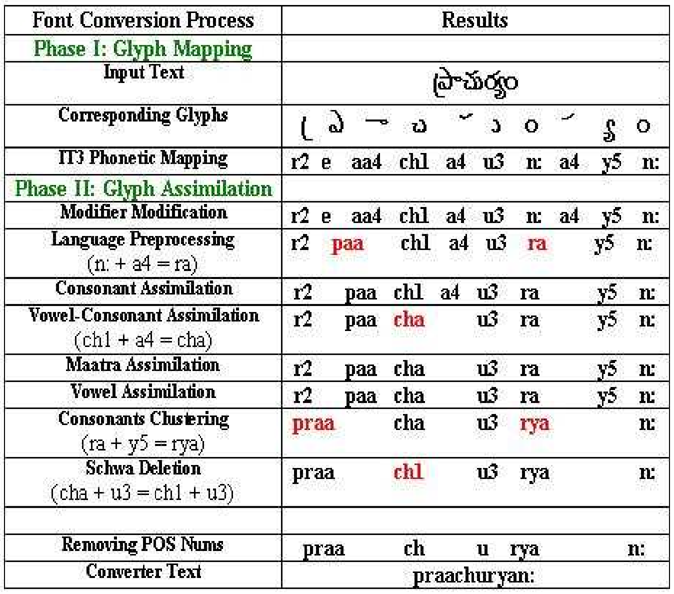
Fig. 4.6 shows how the font conversion is happens in Hindi with an example word. First phase gives the output in meta notation and the second phase re-orders, modifies and assimilates the Aksharas based on rules. Finally the positional numbers get removed to reveal the converted word in a phonetic transliteration form.



Font Conversion Process	Results
Phase I: Glyph Mapping	
Input Text	ग्रीटिंग्स
Corresponding Glyphs	ग ्र ी ि ट ं ग् स
IT3 Phonetic Mapping	ga r5 ii3 i2 t:a n: gl sa
Phase II: Glyph Assimilation	
Modifier Modification	ga r5 ii3 i2 t:a n: gl sa
Language Preprocessing (i2 + t:a = t:a + i2)	ga r5 ii3 t:a i2 n: gl sa
Consonant Assimilation	ga r5 ii3 t:a i2 n: gl sa
Vowel-Consonant Assimilation	ga r5 ii3 t:a i2 n: gl sa
Maatra Assimilation	ga r5 ii3 t:a i2 n: gl sa
Vowel Assimilation	ga r5 ii3 t:a i2 n: gl sa
Consonants Clustering (ga + r5 = gra)	gra ii3 t:a i2 n: gl sa
Schwa Deletion (gra + ii3 = gr1 + ii3)	gr1 ii3 t:1 i2 n: gl s1
Removing POS Nums	gr ii t: i n: g s
Converter Text	griit:in:gs

Fig. 4.6: Glyph Assimilation Process for Hindi.

Fig. 4.7 shows a similar example for Telugu.



Font Conversion Process	Results
Phase I: Glyph Mapping	
Input Text	ప్రాచుర్యం
Corresponding Glyphs	[illegible]
IT3 Phonetic Mapping	r2 e aa4 ch1 a4 u3 n: a4 y5 n:
Phase II: Glyph Assimilation	
Modifier Modification	r2 e aa4 ch1 a4 u3 n: a4 y5 n:
Language Preprocessing (n: + a4 = ra)	r2 paa ch1 a4 u3 ra y5 n:
Consonant Assimilation	r2 paa ch1 a4 u3 ra y5 n:
Vowel-Consonant Assimilation (ch1 + a4 = cha)	r2 paa cha u3 ra y5 n:
Maatra Assimilation	r2 paa cha u3 ra y5 n:
Vowel Assimilation	r2 paa cha u3 ra y5 n:
Consonants Clustering (ra + y5 = rya)	praa cha u3 rya n:
Schwa Deletion (cha + u3 = ch1 + u3)	praa ch1 u3 rya n:
Removing POS Nums	praa ch u rya n:
Converter Text	praachuryan:

Fig. 4.7: Glyph Assimilation Process for Telugu.

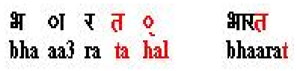
4.6 RULES FOR GLYPH ASSIMILATION

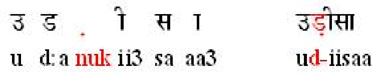
Glyph assimilation rules are defined by observing how the characters are being rendered by the rendering engine. Each rule takes a combination of two or more glyphs in a certain order and produces a valid character. This way we have defined a set of rules for each level and for every language separately. Since they are written in the phonetic transliteration scheme (IT3) it is easily understandable. There may be some common rules across many languages and some specific rules for a language also. The different rules under each and every category are explained with some examples below. These rules can be modified or redefined whenever it is required. For a complete list of glyph assimilation rules per language, please refer to Appendix C.1 to C.9.

(i) *Modifier Modification* is the process where the characters get modified because of the language modifiers like Halant and Nukta. Eg:

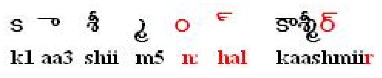
(a) ta + Halant = t1 (Hindi) (Halant based schwa deletion)

(b) d’a + Nukta = d-a (Hindi) (Nukta based modification)



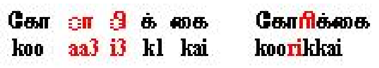


(c) n: + Halant = r1 (Telugu) (Halant based modification)

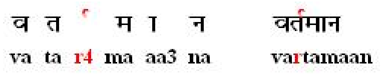


(ii) *Language Preprocessing* steps deal with some language specific processing like Eg:

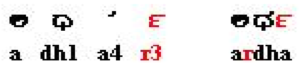
(a) aa3 + i3 = ri (Tamil) (r,ri,rii modifications)



(b) r4 (REF) moves in front of the previous first full consonant (Hindi) (REF movement)

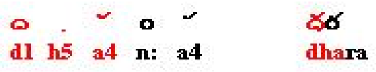


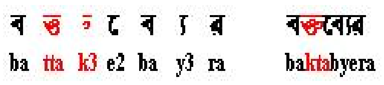
(c) r3 moves in front of the previous first full consonant (Kannada) (Arkka votthu movement)



(iii) *Consonant Assimilation* is known as merging two or more consonant glyphs to form a valid single consonant like Eg:

(a) d1 + h5 + a4 = dha (Telugu) (Consonants da + ha assimilate to dha)

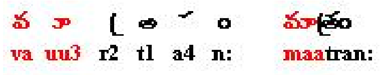




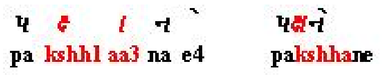
(b) tta + k3 = kta (Bengali) (Consonants tta + k3 assimilate to kta)

(iv) *Consonant-Vowel Assimilation* is known as merging two or more consonant and vowel glyphs to form a valid single consonant like Eg:

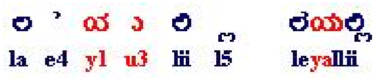
(a) va + uu3 = maa (Telugu) (Consonant + Maatra assimilation)



(b) kshh1 + aa3 = kshha (Gujarati) (Consonant + Maatra assimilation)

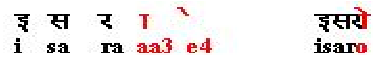


(c) y1 + u3 = ya (Kannada) (Consonant + Maatra assimilation)

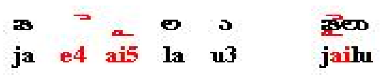


(v) *Maatra Assimilation* is known as merging two or more Maatra glyphs to form a valid single Maatra like Eg:

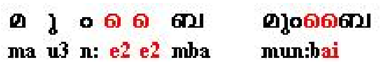
(a) aa3 + e4 = o3 (Hindi) (Maatra assimilation)



(b) e4 + ai5 = ai3 (Telugu) (Maatra assimilation)

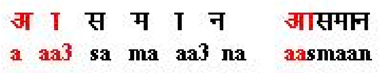


(c) e2 + e2 = ai3 (Malayalam) (Maatra assimilation)

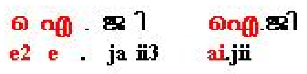


(vi) *Vowel-Maatra Assimilation* is known as merging two or more vowel and Maatra glyphs to form a valid single vowel like Eg:

(a) a + aa3 = aa (Hindi) (Vowel assimilation)

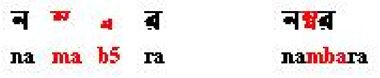


(b) e2 + e = ai (Malayalam) (Vowel assimilation)

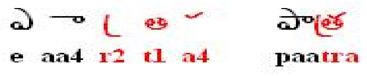


(vii) *Consonant Clustering* in known as merging the half consonant which usually occurs at the left, right and bottom of a pivotal character to the full consonant like Eg:

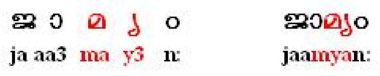
(a) ma + b5 = mba (Bengali) (b5 consonant clustering)



(b) r2 + t1 + a4 = tra (Telugu) (r2 consonant clustering)

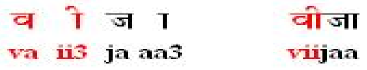


(c) ma + y3 = mya (Malayalam) (y3 consonant clustering)



(viii) The *Schwa Deletion* is deleting the inherent vowel /a/ from a full consonant in necessary places like Eg:

(a) va + ii3 = vii (Hindi) (Maatra based schwa deletion)



4.7 PERFORMANCE ANALYSIS

Data Preparation: In the development phase for each font, ‘N’ different sets of words were prepared for each iteration to define the assimilation rules. In the testing phase, 500 unique words from a new font were used to check the converter performance.

Training: At beginning a simple converter is built with minimal glyph assimilation rules. The converter is tested with the first set of words by generating the output. Then a native speaker (or a linguist) is asked to evaluate the output. He/she gives the evaluation results besides the correction for the wrongly converted words. Based on that input the the existing rules are modified or a new rule is added. Again the updated converter is tested with the next set of words. The feedback is collected and modification or update of the rules is done. This process is repeated until we reach none or minimal conversion errors. This process is repeated for another font of the language. It is found sufficient to train three different fonts of different font-families for each language. At the end of this process we will be having the converter for that language.

Testing: We pass a set of 500 words from a new font of that language to the already built font converter. The output is again evaluated. This is considered as the performance accuracy of that font converter.

Evaluation: We have taken 500 unique words per font-type and generated the conversion output. The evaluations results (Table 4.1) show that the font converter performs consistently even for a new font-type. So it is only sufficient to provide the glyph-map table for a new font-type to get a good conversion results. In the Table 4.1 third column shows either training or testing. Training involves a set of fonts considered for defining and redefining the glyph assimilation rules for a language while building the converter. Testing involves a font is used to test or evaluate the already built font converter of that language. Here for some languages (like Punjabi, Bengali and Oriya) testing wasn’t carried out since we couldn’t find more downloadable fonts. In the case of Telugu, the number of glyphs and their possible combinations are larger than other languages. Also it is common for all Indian languages that the pivotal character glyph comes first and other supporting glyphs come next in the script (Section 2.4). But in Telugu the supporting glyphs may come before the pivotal glyph which creates ambiguity in forming assimilation rules. Hence the converter performed lower than other converters. The conversion results are tabulated below in Table 4.1.

4.8 RAPID FASHION OF BUILDING FONT CONVERTERS

The other aim of this research is to find a method to build font converters rapidly for Indian languages. We have achieved this by following glyph assimilation process to build a single font converter per language. It is to be noted that the converter is for all fonts within that language. These font converters are working based on a set of glyph assimilation rules. And these rules are defined/written on a meta notation known as positional phonetic notations. So the converters are independent of font of a language but the glyph-map tables are dependent on fonts. To add a new font apart from the trained fonts, one needs to prepare only the glyph-map table for the new font. Using the existing font converter of that language, one can get the converted data by passing the glyph-map table and the input text. For building the glyph-map table for a new font a native speaker hardly it takes 1-2 man hours.

4.9 SUMMARY

In this chapter we proposed the generic framework for building font converters for Indian languages, and the related works which provide readily available tools. Two phases in the framework glyph-mapping and glyph assimilation is discussed. Shape and position information of the glyph is exploited to do an unambiguous mapping. Handling the glyph assimilation process and various techniques with rules have been defined under each stage with examples to solve assimilation. Finally font converters are built using the proposed methodology for getting optimum conversion results. In brief, the chapter explains how to rapidly build a converter for a new font without modifying the existing converter but by

Table 4.1: Font-data conversion: Performance results for font converters.

Language	**Font Name**	**Training / Testing**	**Accuracy**
Hindi	Amarujala	Training	99.2%
	Jagran	Training	99.4%
	Naidunia	Training	99.8%
	Webdunia	Training	99.4%
	Chanakya	Testing	99.8%
Marathi	Shree Pudhari	Training	100%
	Shree Dev	Training	99.8%
	TTYogesh	Training	99.6%
	Shusha	Testing	99.6%
Telugu	Eenadu	Training	93%
	Vaarttha	Training	92%
	Hemalatha	Training	93%
	TeluguFont	Testing	94%
Tamil	Elango Valluvan	Training	100%
	Shree Tam	Training	99.6%
	Elango Panchali	Training	99.8%
	Tboomis	Testing	100%

Kannada	Shree Kan	Training	99.8%
	TTNandi	Training	99.4%
	BRH Kannada	Training	99.6%
	BRH Vijay	Testing	99.6%
Malayalam	Revathi	Training	100%
	Karthika	Training	99.4%
	Thoolika	Training	99.8%
	Shree Mal	Testing	99.6%
Gujarati	Krishna	Training	99.6%
	Krishnaweb	Training	99.4%
	Gopika	Training	99.2%
	Divaya	Testing	99.4%
Punjabi	DrChatrikWeb	Training	99.8%
	Satluj	Training	100%
Bengali	ShreeBan	Training	97.5%
	hPrPfPO1	Training	98%
	Aajkaal	Training	96.5%
Oriya	Dharitri	Training	95%
	Sambad	Training	97%
	AkrutiOri2	Training	96%

minimally providing only the glyph-map table. The next chapter deals with how these font identification and font conversion modules are integrated with a screen reader for Indian languages.

CHAPTER 5

RAVI: A MULTI-LINGUAL SCREEN READER

This chapter presents a screen reader (viz. RAVI) for Indian languages which implements the text processing techniques described in the earlier chapters. RAVI (Reading Aid for Visually Impaired) [38] is a GUI based screen reading software for Indian languages. It is developed based on a comprehensive and intuitive design methodology. This system assists the user for navigating through the computer and reads help information based on event triggered or message popped. It reads menu, text boxes, dialogs and all other widgets on the computer screen. It guides the user to navigate and to open a program or an application which user wants to use without others help. Right from the starting of the computer or execution of the software, the user is always provided with adequate voice over and verbosity to operate it.

This software seamlessly integrates with almost all the variants of the Windows operating systems. To enable the end-user to work independently using this software, we provide appropriate audio/speech support for each letter typed/keyed with audio alerts in case of any unsupported key is pressed. However, the most vital feature of this software is its support for handling Indian languages and compatibility to use with MS Office suite, Internet Explorer and many other native windows applications. A laboratory developed Text-to-Speech System is integrated within RAVI to synthesize in Indian languages. Two independent speech engines are internally used by RAVI. One for synthesizing English voice using “Microsoft TTS” [39] system and other one for Indian languages “Indian language TTS” [40] [41] system developed at IIIT Hyderabad. Our screen reader identifies the language specific contexts and automatically selects to the respective synthesizer. The current version of the software supports five Indian languages namely Hindi, Telugu, Tamil, Kannada and Marathi.

5.1 PROPOSED RAVI ARCHITECTURE

The proposed RAVI system architecture in Fig. 5.1 is modular based and it consists of three sub-systems. They are (i) Text Extractor sub-system, (ii) Text Preprocessing sub-system and (iii) Text-to-Speech systems sub-system. The sub-systems are further divided into many modules as given below.

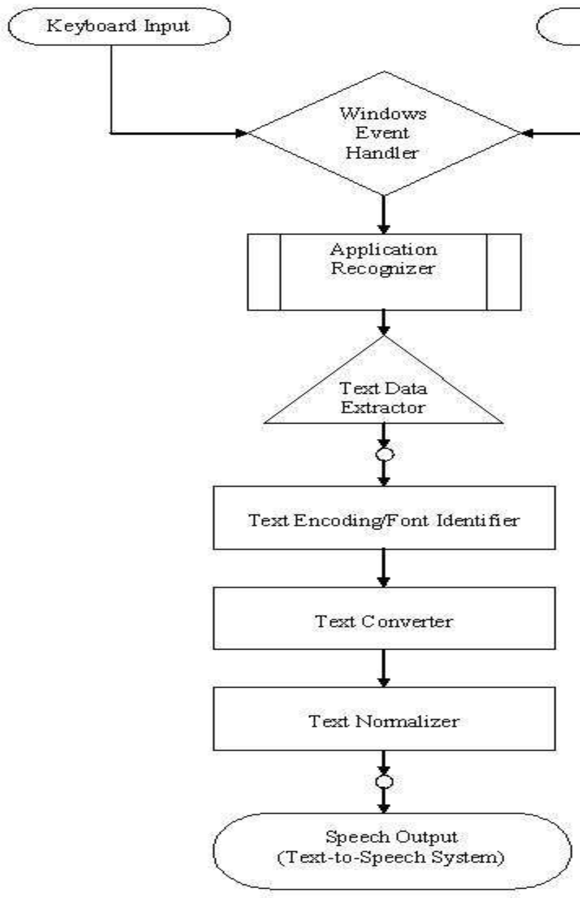


Fig. 5.1: RAVI System Architecture.

- The foremost is *text extraction* sub-system, functionally classified into:
  - (i) *Windows Event Handler* for capturing the system wide events and maintaining the related messages in stack

  - (ii) *Keyboard or Mouse Input* for interfacing with the user to get the keyboard command / mouse click input
  - (iii) *Application recognizer* for identifying events raised by which active application or by which object such as button, check box etc., and
  - (iv) *Text extractor* for extracting the relevant text and some text features (especially font name) from the active application or object based upon the current cursor position.

- Secondly *text preprocessing* sub-system is functionally classified into:
  - (i) *Text encoding or font-type identifier* for determining whether the extracted text is encoded in Unicode or ISCII or ASCII type font
  - (ii) *Text converter* for converting these different encoded text into a phonetic transliteration notation IT3
  - (iii) *Text normalizer* for converting the Non-Standard Word (NSW) (like currency, time, date etc.,.) into a standard pronounceable/readable words, and
  - (iv) *Language identifier* for identifying the language constituent for selecting appropriate TTS system.

- Lastly *text-to-speech systems* sub-system is functionally classified into:
  - (i) *TTS selector* for switching between different language TTS systems based on language
  - (ii) *Voice loader* for loading dynamically available voice for current language
  - (iii) *Voice inventory* for maintaining prerecorded speech segments and language related features
  - (iv) *Unit selection* algorithm for selecting the optimal speech segment for concatenation and
  - (v) *Speech segment concatenation* function for concatenating and smoothing the selected segments to generate the speech corresponding to IT3 text.

5.1.1 Components of RAVI Screen Reader

Developing a full-fledged screen reader for Indian languages is not only a challenging but a daunting task considering the eventualities aspects involved in the design and implementation. As we have addressed the common issues in developing a screen reader in the Chapter 1, we now look into some intrinsic application such as internal text storage, identifying the font type name, identifying the language, converting and normalizing the text as TTS renderable form (IT3 data) and invoking the TTS system which is capable of synthesizing the voice for the identified language text on the screen. Modules required for these addressing issues form integral components of RAVI, functionally they are:

- *Internal Storage Format of Text*: The text extractor module extracts the relevant text which a is complicated task for Indian languages, because the text can be represented in different formats, like mark-up (tagged) formats, encrypted, proprietary formats etc. Relevant text extraction means part or position of text such as word, cell, title, cursor position etc., such fields explicitly restores the text for retrieving in future. Even sometime it is helpful if we are able to identify the corresponding font name also. More over a suitable data type has to be identified to store internally the extracted text. In this regard we faced problem on trying to extract and store Unicode data. Later we found that we should use a suitable data type to retrieve it. So we identified and used data types like ANSI C string and character pointer for ASCII based text and binary string (bstr) for Unicode (UTF-8) data to store internally. Extracting text from Notepad and WordPad is complicated since they don't have any Object Library files. So we were restricted to use cursor position and Windows messages to extract the relevant text.

- *Font Processing*: As we have learned in the introduction Chapter 2 that most of the Indian language electronic content is available in different font encoding formats. Indian language content processing is a real world problem for a researcher. Here processing includes identifying the font encoding and converting the text into TTS renderable input format. To process the font-data first and foremost job of a screen reader is to identify the underlying encoding or font. There will be no header information like in the case of Unicode (UTF-8) encoding or a distinguished ASCII code ranges for English and Indian languages like in ISCII. All we can get is the sequence of glyph code values (ASCII values). So these can be identified through statistical modeling or machine learning techniques. The statistical models are generated by following some statistical methods using the glyph codes. Converters or models are developed using some machine learning algorithm to learn the glyph code order. And these converters or models are capable of identifying the font using the statistical models. Our font identification module uses vector space models for font-type identification (Chapter 3).

- *Text Preprocessing*: In screen reader perspective text processing means converting the extracted text into some encoded/compatible format for the TTS system, i.e font-to-IT3 conversion, Unicode-to-IT3 conversion, ISCII-to-IT3 conversion etc. Then we need to normalize the converted text. Text normalization is the process of converting non-standard words like numbers, abbreviations, titles, currency etc., to expanded/natural (pronounceable) words. This can be achieved by writing a text Normalizer for each and every language or designing a generic framework which fits for most of the languages. There is one more step which is optional. If we are using some third party TTS systems then we need to convert text in our notation to the notation supported by that TTS system, since the input notation for TTS is vendor/developer specific. If we are using our own TTS system then there is no such overhead. Font Converter module does font-to-IT3 conversion and is covered in detail in (Chapter 4).

- *Language Identification*: The language information of the text is very important to invoke the right TTS system. This can be done in two ways either identify the language of the extracted text (raw text) or identify the language from the converted text (phonetic text). Based on the extracted/identified font name it identifies the language, else it identifies the language based on the phonetic sequences.

5.1.2 Implementation

For developing a screen reader, there are basically four options to get the information from the operating system (Windows). Following approaches are given in the order of complexity to implement. We exploited first three techniques and combinedly used in our implementation, last one considered as part of future work.

(i) *MSAA* [3] : Microsoft Active Accessibility provides access to most of the applications bundled by Microsoft. Since it has been designed specifically to provide interface for assistive devices, it is default to test and use it. This being standard way to retrieve information from the applications, besides this one also gets implicit objects provided for the programmer. All the common controls are supported by MSAA, available for Microsoft applications such as MS Word, MS Excel, and Internet Explorer including Outlook Express etc.

(ii) *COM* [42] : Component Object Model which is widely used in window based applications to expose a lot of functionality through automation components. This is relatively useful as it is a standard way of communication between applications, then MSAA inherits this model for application interface.

(iii) *Win32 API:* Window's 32 application programming interface can be used where MSAA and COM are not supported. The complexity of this approach is limited to structural information.

(iv) Last resort would be to capture *video driver intercept* but harder to implement than said.

5.1.3 Merits of Proposed Architecture

The robustness of this architecture is the Plug-and-Play (PNP) feature, also with ease of allowing user to add a new module or replace the existing one with latest. Since all functionalities are segregated and designed in modular approach, makes it lucid for user to ensure that the configuration details and interfaces are mentioned appropriately. It expects the interfaces and configuration details should correlate independent of internal implementation aspects. For instance, upgrading with much more efficient application for extracting text from Internet Explorer, just one can replace the old one by the latests along ensuring the configuration. And also if we want to add a whole new application to it, we just need update it and provide the interfaces and the configuration. All modules are *Dynamic Link Libraries (dlls)* henceforth the new module name has to be registered in the RAVI configuration file and the interfaces (APIs) have to be listed appropriately. The main configuration file is a *ravi_config.ini (system file)* file which has *hierarchical sections for each module*. Under each section or sub-section the required information is provided in hash table with key, value. So the application loads this system file and retrieves the required module name and its interface pointers and accomplishes the required task. Crux of this architecture lies in ease of augmenting language, font, application and TTS.

(a) Support for a New Language: Providing support for a new language, the components specific to language are vector space models for fonts, font converter for that language and text normalizer for that language then the TTS system of that language. These modules have to be registered into the configuration (ravi_config.ini) file under Section [Language] in *ravi_config.ini* as discussed above. No more changes need to be done in the core software. It will accept and load those modules automatically performing appropriate action.

(b) Giving support for a New Font: To add support for a new font user needs to follow two steps. Firstly the *vector space model* of that font is to be added to the list under Section [FontModels] in *ravi_config.ini*. Secondly, adding the *glyph-map table* of that font under Section [GlyphMapTables] in *ravi_config.ini*. In the phase of font identification, the font identifier will find model of the font and hence we can get the font name. Then from [FontConverter] Section in *ravi_config.ini*, we can get the required font converter module name. The font converter uses this glyph-map table and converts the text into phonetic notations.

(c) Support for a New TTS: Adding new TTS by mentioning it under Section [TTS] in *ravi_config.ini*. Screen reader expects basic commands support from TTS systems such as Start, LoadVoice, Speak and Stop. Under each TTS Section (eg: [HindiTTS]) the four commands should be provided. So the screen reader is totally independent of TTS architecture, used for text synthesis, supported functionalities etc.,. It only expects these basic interface pointers to produce the speech output. When the text and the language information are passed to the TTS sub-system it selects the right synthesizer automatically based on the language. It loads the TTS with the selected voice and synthesizes the text into speech and then plays it out. During the time of toggling of TTS or closing the application, it stops the current running TTS system.

(d) Support for a New Application: Including support for new application is bit complicated. User is restricted to avail this facility as of now but developer has free hand on it to add new application module without disturbing the existing system.

5.2 ASPECTS OF USER INTERFACE

The common user interface adopted in RAVI is through standard/default input devices (keyboard or mouse). It performs the desired action according to pressing the shortcut keys (a combination of keys) or focus of mouse or single click by the user. It automatically announces the error or status message to the user. Providing interface through keyboard we followed two kind of supports; firstly utilizing the default windows shortcut keys and our own defined application specific shortcut keys.

In designing the shortcut keys for applications, we referred the existing commercial screen readers and collected feedback from the users. From this test/benchmarking we observed that the existing screen readers offer a lot of shortcut keys which are practically impossible to remember or too confusing for the users. So we wanted the minimal set of shortcut keys which covers all basic functionalities addressing the ease of remembrance. Accordingly we designed shortcut keys for each and every application supported for reading like previous, current and next word or sentence.

And we found that the users are well familiar with the default windows shortcut keys.

So we utilized those keys as it readily provides support for normal operations. They can access the application menus, buttons, list boxes etc., using the default windows shortcut keys.

Blind users always preferred to use keyboard only, even though sometimes they want mouse support also. Sometimes normal users or students also use screen reader, in that case they prefer mouse instead of keyboard. So we provided mouse support also by playing aloud the object, menu item, button etc., through mouse focus or click.

In the GUI of the software, we also provided menus like file, help, language, options and settings. Using this user can access some sample files or help document. By using language menu user can change language specific TTS support at operating system level.

5.3 FUNCTIONALITIES COVERED

The functionalities offered in RAVI is classified into two. Firstly the general or basic and secondly application specific functionalities. The general functionalities are speaking out time, date and reading the text of simple edit control or rich edit control. It speaks out the menu, combo box and list box items and the selected items. Application specific functionalities are speaking out the cell position in Excel sheet, goto the last slide of the presentation, speaking out the subject of the mail in Outlook etc.,.

They are tabulated in Table 5.1 under categories such as general, office suite, Internet applications and controls. For each category the number of items supported and their functionalities are listed out.

5.4 TESTING AND EVALUATION

Ergonomic systems like RAVI are designed using modularized sub-systems, are composed of procedures and functions. Henceforth testing process should therefore be carried in stages where testing is scaled out incrementally in conjunction to system implementation. Traditional system testing process consist of five stages. Generally, the sequence of testing activ-

Table 5.1: Functionalities supported by RAVI.

Category	**Item**	**Functionalities**
General		reads text as character, word, sentence and paragraph
		informs text attributes like font size, type and style
		echoes the character or word is being typed
		navigating and accessing the desktop
		reading out control panel, start menu and menus of various applications
		reading via keyboard commands and mouse movement/click
		option to change TTS or languages on the fly
Office suite	MS Word	reads previous, next and current character or word or sentence. announces about tables, format and alignment
	MS Excel	reads cell content, informs row or column number and speaks out formula
	WordPad	reads rich edit controls and previous, next and current character or word or sentence
	Notepad	reads previous, next and current character or word or sentence
	PowerPoint	reads slides informations, contents and navigates through slides
	Outlook	reads mail attributes like From, To, Cc, Bcc, subject, mail content and navigates through mails
Internet Applications	Internet Explorer	reads previous and next lines, links informations, headings and images (alt text)
	News Portals	reads headlines, news items (whole content or line by line) and navigates to pages
	Email	supports reading emails in all leading email service providers
Controls	Push Button	reads the caption and its state
	Checkbox	reads the caption and its state
	Radio Button	reads the caption and informs whether it is checked or not
	Group	reads the group name then the items
	Drop-Down Combobox	reads the label and the content of the edit control
	Drop-Down Listbox	reads the label and the selected/focused item
	Simple Combobox	reads the content of the edit control
	Single Line Edit Control	reads previous, next and current character or word
	Multi-Line Edit Control	along single line edit control capabilities it reads previous and next line, sentences and paragraphs
	List Box	reads the selected/focused item in the list
	Static Controls	reads the entire text of the static control since it may be a message or a label

Controls	Calendar	reads the date information
	Progress Bar	reads the percentage of task is done
	Tooltip Control	reads the tool tip text
	Tree View	reads the current item and the level in the tree

ities are component testing, integration testing and user testing. This is an iterative process where the feedback from one testing stage is piped to the next testing stage.

(a) *Unit Testing*: Individual components are tested to ensure that they operate correctly. Each component is tested independently without other system components. We have done an in-house testing of each unit.

(b) *Module Testing*: A module is a collection of dependent components. A module encapsulates related components so can be tested without other system modules. We have done an in-house testing of all modules.

(c) *Sub-system Testing*: This phase involves testing collection of modules which have been integrated into sub-systems. The most common problem which arises in large software systems are sub-system interface mismatches. So the sub-system test process should therefore concentrate on the detection of interface errors by rigorously exercising these interfaces. We have done an in-house testing of sub-system after integrating all modules.

(d) *System Testing*: The sub-systems are integrated to make up the entire system. The testing process is targeted for finding errors which result from unanticipated interaction between sub-systems and system components. It is also deals with validating that the meets it's functional and non-functional requirements. This testing is done by the targeted users other than in-house users and the feedback accounted.

(e) *Acceptance Testing*: This is the final stage of testing process before the system is accepted for operational use. The system is tested with the data supplied by the system procurer rather than simulated test data. This testing reveals requirement problems, where the system does not really meet the users need or the system's performance is unacceptable. This is also called as alpha testing.

When a system is to be marked as a software product, a testing process called beta testing is done. This testing involves delivering a system to a number of potential customers, agreeing to use the system. They report the problems to the developers. After this feedback, the system is modified and either released for further beta testing or for general sale. We have compiled this testing by circulating a copy of the software to outside people and collected feedback for the same task. And, also we have done real-time testing by installing the software at L.V.Prasad Eye Hospital [43], Hyderabad in their teaching lab. They have successfully tested and provided the feedback from the blind students. Some of them are:

- It is easy to instal.
- Feature to define their own shortcut-keys/hot-keys for a command.
- Improve the feature to fill-up on-line application/form.
- Provide more voices per language so that they can choose the suitable one.

All these and other comments collected from many other users will be updated in the next coming version.

5.5 LIMITATIONS

Every system has its own limitations. RAVI is also no exception, listed below are some of them.

- Text extraction: At times RAVI fails to extract the text from some controls (eg: language bar) which do not expose any interface to extract text.
- OS level support: RAVI provides operating system level support for Indian languages using dictionaries. All the basic commands are translated in the interested language. But these dictionaries have limited entries. So it may fail to translate the new/rare words.

- TTS switching: When switching between one language to another, the TTS takes considerable time to load the voice database. And if the switching is so frequent, it may fail to load it and eventually crashes or hang up.
- TTS buffer size: TTS may often have input buffer limitation problems
- Font identification: RAVI comes with models for fonts currently in usage and can identify the font name. For any newly developed font, it is capable of building models by providing the sufficient training data. However it fails to identify the font for which the model is missing.
- Font Conversion: RAVI comes with glyph-map tables for fonts currently in usage and can convert those text. For any newly developed font, it is necessary that either the RAVI developer or the user has to provide the glyph-map table.
- Language identification: Since RAVI is developed to work with only Indian languages it will identify any Indian language but it will fail to identify any foreign language. So it will try to speak out in English.

5.6 SUMMARY

This chapter deals with the design and development of the multi-lingual screen reader for Indian languages. Various components for processing the Indian language specific issues were discussed. A new modular based architecture for the screen reader, and its advantages such as ease to add a new module i.e plug and play (PNP) is highlighted. Modules explained in the previous chapters, font identification and font conversion have been integrated with the software. Customized functionality for various applications and intuitive controls were listed out. Software testing paradigms for testing and benchmarking screen reader is done and feedbacks/results are registered.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Given that there are 23 official languages and 11 scripts in India, the amount of electronic text available in glyph based font is greater than the text available in Unicode format which makes preprocessing the font-data necessary. In this thesis, we have proposed a new method for identification of font-type using TF-IDF approach and have designed a generic framework for building font converters to convert font-data to a phonetic transliteration scheme. We have demonstrated the effectiveness of our conversion to be as high as 99.5% on 37 fonts of 10 different languages. This thesis also explained the nature and difficulties associated with Indian language electronic data. The results show the proposed methodologies are suitable for processing font-data for Indian languages. The advantages and disadvantages of the two proposed approaches are as follows:

(i) Biglyph (current and next glyph) based font-type models yield good classification results and are also time effective. For more accurate classification one can consider triglyph or more glyphs for modeling. In font-type modeling when the number of glyphs in the glyph sequence increases, it consumes more space and the process is computationally slower. In font-type identification it takes more time to load the model into memory and also the model files occupy more space.

(ii) In font-data conversion we found that three different font-type of different font-families data is sufficient to train or to define assimilation rules for a language. A new font-type data can be converted with the same accuracy by providing only the glyph-map table of the new font-type. We have achieved 99.5% overall accuracy. Although not encountered in our observations, there may be a rare font in a language which may require addition of new rules. The effectiveness of the framework has been verified for development of font converters in all Indian Languages.

As an outcome of this thesis, we have developed RAVI (Reading Aid for Visually Impaired) [38] a GUI based screen reading software for Indian languages. It was developed based on a comprehensive and intuitive design methodology with plug-in-play architecture to add new font converters, text-to-speech systems in a easy fashion. This system assists the user for navigating through the computer and reads help information based on event triggered or message popped. It reads menu, text boxes, dialogs and all other widgets on the computer screen. It guides the user to navigate and to open a program or an application which user wants to use without others help. Right from the starting of the computer or execution of the software, the user is always provided with adequate voice over and verbosity to operate it.

However, the most vital feature of this software is its support for handling Indian languages and compatibility to use with MS Office suite, Internet Explorer and many other native windows applications. Two independent speech engines are internally used by RAVI. One for synthesizing English voice using “Microsoft TTS” [39] system and other one for Indian languages “Indian language TTS” [40] [41] system developed at IIIT Hyderabad. Our screen reader identifies the language, appropriate font-type, performs font-conversion and automatically selects the respective synthesizer. The current version of RAVI software supports five Indian languages namely Hindi, Telugu, Tamil, Kannada and Marathi.

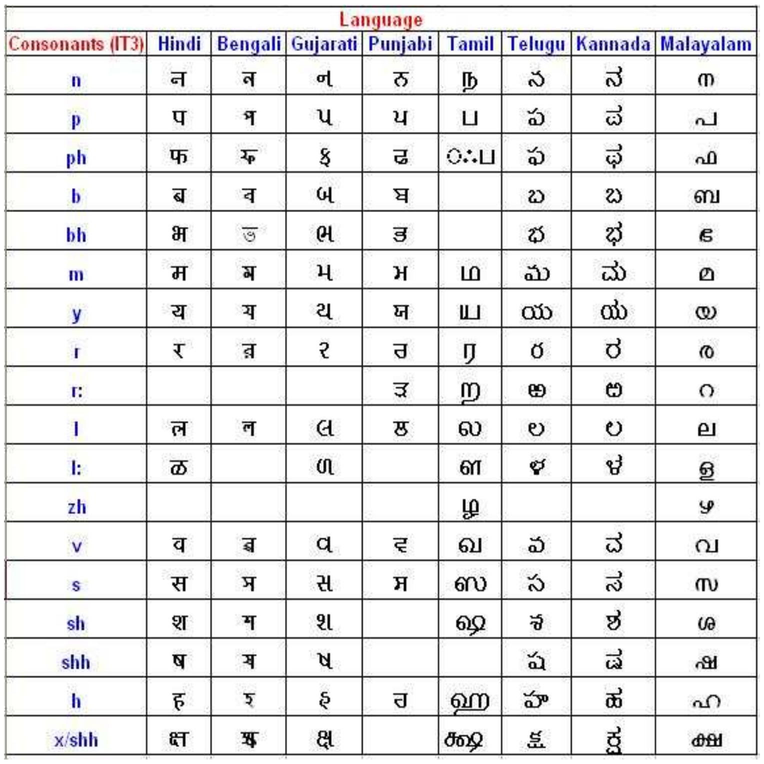
6.1 FUTURE WORK

We have planned to create and add more glyph-map tables for more fonts for all Indian languages. We have planned to improve the quality and increase the support by adding more fonts with the screen reading software. Need to design a new methodology for websites which do not release their font or using some dynamic font rendering mechanism.

APPENDIX A

IT3 PHONES MAP FOR INDIAN LANGUAGE CHARACTERS

In this appendix, we provide the IT3 phones mapping for Indian language characters. One can observe that the IT3 phones are assigned according to the pronunciation of the phones in the native language. This makes it very easy to understand the notation and use it effectively. Fig. A.1 shows the remaining consonant phones list in IT3 scheme. The mapping is showed for 8 different Indian languages.



	Language							
Consonants (IT3)	Hindi	Bengali	Gujarati	Punjabi	Tamil	Telugu	Kannada	Malayalam
n	न	ন	ન	ਨ	ந	న	ನ	ന
p	प	প	પ	ਪ	ப	ప	ಪ	പ
ph	फ	ফ	ફ	ਫ	ஃப	ఫ	ಫ	ഫ
b	ब	ব	બ	ਬ		బ	ಬ	ബ
bh	भ	ভ	ભ	ਭ		భ	ಭ	ഭ
m	म	ম	મ	ਮ	ம	మ	ಮ	മ
y	य	য	ય	ਯ	ய	య	ಯ	യ
r	र	র	ર	ਰ	ர	ర	ರ	ര
r:				ੜ	ற	ఱ	ಱ	റ
l	ल	ল	લ	ਲ	ல	ల	ಲ	ല
l:	ळ		ળ		ள	ళ	ಳ	ള
zh					ழ			ഴ
v	व	ব	વ	ਵ	வ	వ	ವ	വ
s	स	স	સ	ਸ	ஸ	స	ಸ	സ
sh	श	শ	શ		ஶ	శ	ಶ	ശ
shh	ष	ষ	ષ			ష	ಷ	ഷ
h	ह	হ	હ	ਹ	ஹ	హ	ಹ	ഹ
x/shh	क्ष	ক্ষ	ક્ષ		க்ஷ	క్ష	ಕ್ಷ	ക്ഷ

Fig. A.1: IT3 consonant phones for Indian languages - part 2.

APPENDIX B

FONT CHARTS AND FONT GLYPH MAP TABLES

In this appendix, we provide font charts for two fonts namely Eenadu (Telugu) and Jagran (Hindi). It gives an idea how the glyphs are accommodated in the ASCII code range (0 to 255). This kind of chart is necessary to prepare a glyph map table for every font as shown in the figures B.4 to B.13. By looking at this chart a linguist or a native speaker of that language can easily prepare the glyph map table. The rows and columns of this table are in decimal which are used to calculate the the glyph code. The glyph code value for a glyph/cell is calculated like column + row value (80 + 2 = 82).

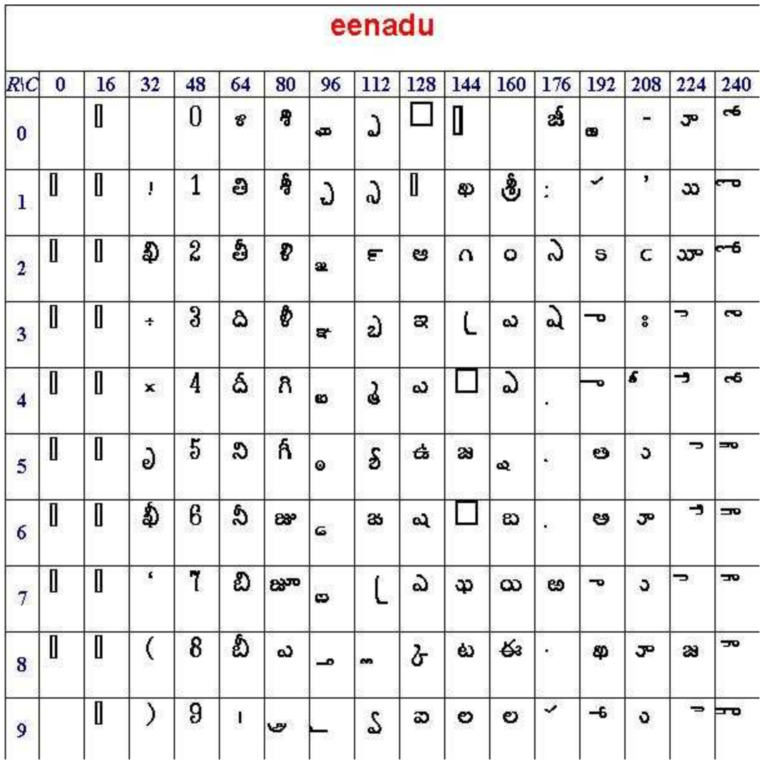
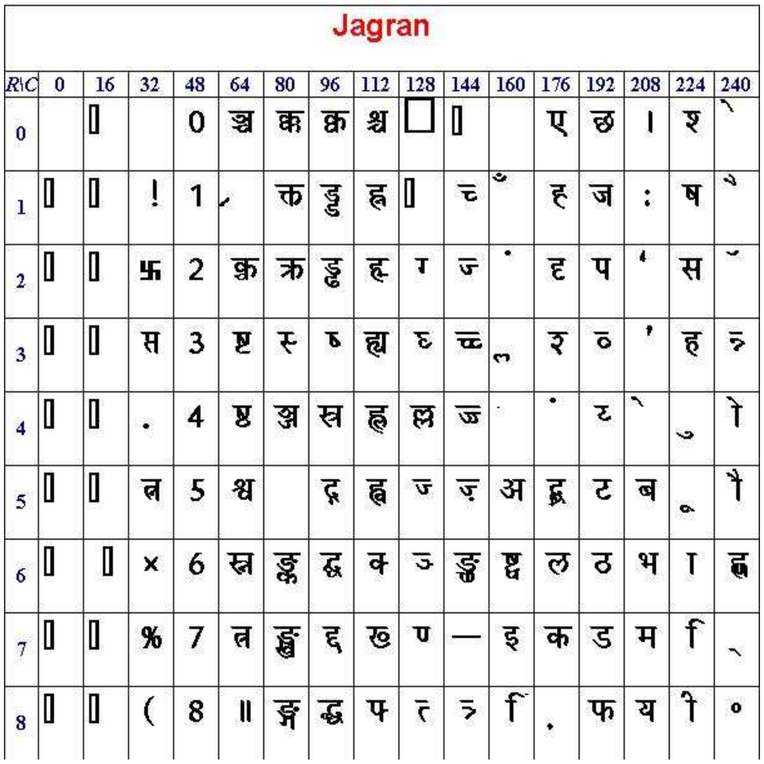

eenadu
R\C
0
16
32
48
64
80
96
112
128
144
160
176
192
208
224
240
0
1
2
3
4
5
6
7
8
9

Fig. B.2: Portion of Eenadu Font Chart - Telugu.

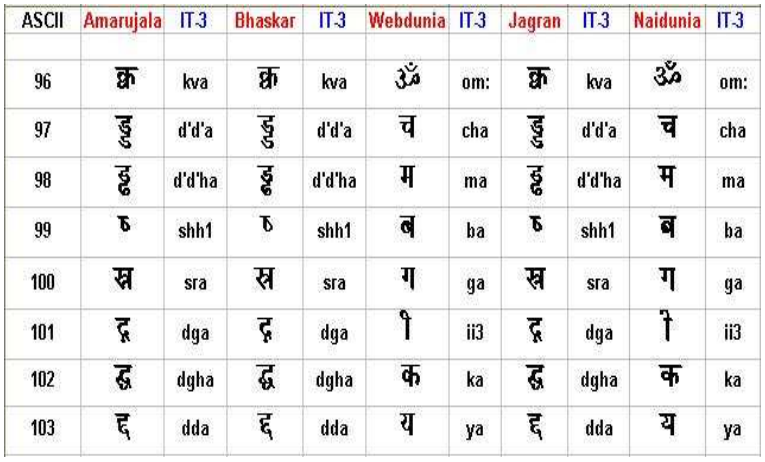


Jagran

R\C	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0		▯		0	ञ्च	क्क	क्क	श्च	□	▯		ए	छ	।	श	ˋ
1	▯	▯	!	1	ˏ	क्त	ड्ड	ह्ल	▯	ट	ँ	ह	ज	:	ष	ˀ
2	▯	▯	卐	2	क्र	क्र	ड्ढ	हृ	ग	ज	˙	द	प	‘	स	˘
3	▯	▯	स	3	ष्ट	स्	ठ	ह्य	ह	च्च	ृ	श	०	’	ह	ॢ
4	▯	▯	.	4	ष्ठ	ञ्ज	स्र	ह्ल	ल्ल	ज्ज	·	˙	ट	ˋ	ु	ो
5	▯	▯	त्र	5	श्व		दृ	ह्व	ज	ज़	अ	द्ध	ट	ब	ृ	ौ
6	▯	▯	×	6	स्न	ङ्क	द्ध	क	उ	ङ्क	द्व	ल	ठ	भ	ा	ह्म
7	▯	▯	%	7	त	ङ्ख	द	ख	प	—	इ	क	ड	म	ि	ˎ
8	▯	▯	(	8	॥	ङ्ग	द्ध	प	त	ॽ	ि	.	फ	य	ी	°

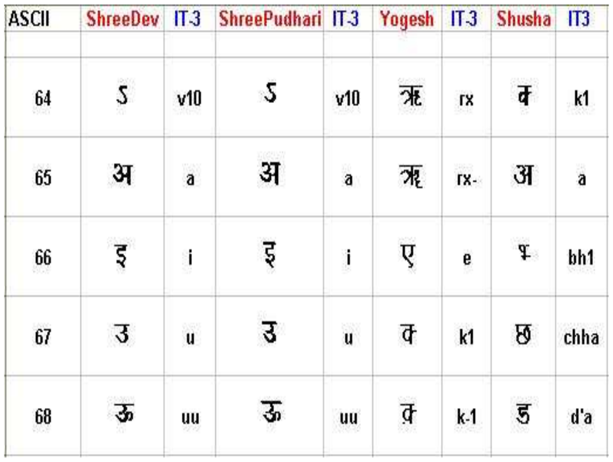
Fig. B.3: Portion of Jagran Font Chart - Hindi.

As stated previously, the glyph-map tables prepared for different languages are provided below. The left most column in the table is the glyph code and on its right are it's corresponding glyph and equivalent IT3 notation with position number attached. It clearly shows that for a single code value different glyphs get assigned for different fonts.



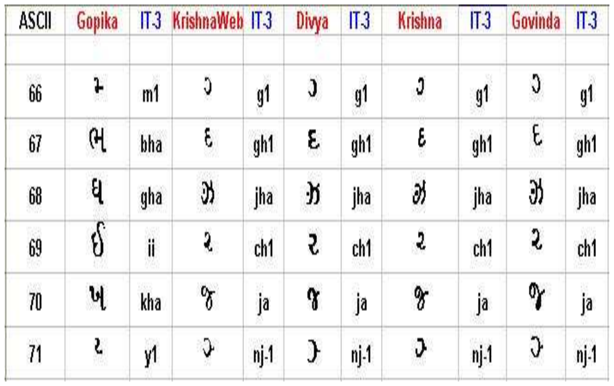
ASCII	Amarujala	IT-3	Bhaskar	IT-3	Webdunia	IT-3	Jagran	IT-3	Naidunia	IT-3
96	क्व	kva	क्व	kva	ॐ	om:	क्व	kva	ॐ	om:
97	ड्ड	d'd'a	ड्ड	d'd'a	च	cha	ड्ड	d'd'a	च	cha
98	ड्ढ	d'd'ha	ड्ढ	d'd'ha	म	ma	ड्ढ	d'd'ha	म	ma
99	ष्	shh1	ष्	shh1	ब	ba	ष्	shh1	ब	ba
100	स्र	sra	स्र	sra	ग	ga	स्र	sra	ग	ga
101	द्ग	dga	द्ग	dga	ी	ii3	द्ग	dga	ी	ii3
102	द्घ	dgha	द्घ	dgha	क	ka	द्घ	dgha	क	ka
103	द्द	dda	द्द	dda	य	ya	द्द	dda	य	ya

Fig. B.4: Portion of Hindi Glyph Map Table.



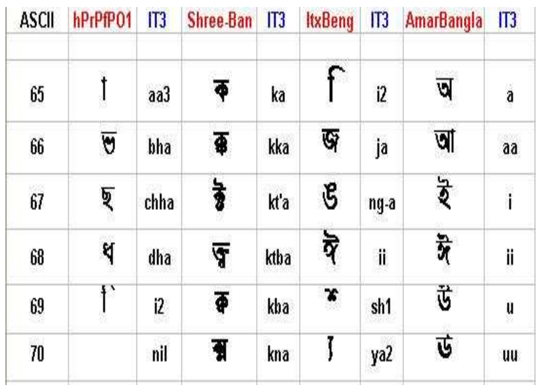
ASCII	ShreeDev	IT-3	ShreePudhari	IT-3	Yogesh	IT-3	Shusha	IT3
64	ऽ	v10	ऽ	v10	ऋ	rx	क्	k1
65	अ	a	अ	a	ॠ	rx-	अ	a
66	इ	i	इ	i	ए	e	भ्	bh1
67	उ	u	उ	u	क्	k1	छ	chha
68	ऊ	uu	ऊ	uu	क्	k-1	ड	d'a

Fig. B.5: Portion of Marathi Glyph Map Table.



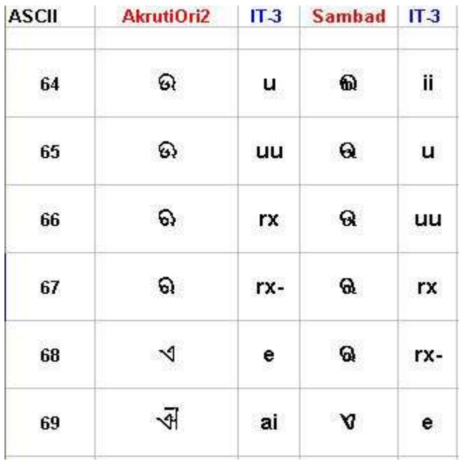
ASCII	Gopika	IT-3	KrishnaWeb	IT-3	Divya	IT-3	Krishna	IT-3	Govinda	IT-3
66	મ્	m1	ગ્	g1	ગ્	g1	ગ્	g1	ગ્	g1
67	ભ	bha	ઘ્	gh1	ઘ્	gh1	ઘ્	gh1	ઘ્	gh1
68	ઘ	gha	ઝ	jha	ઝ	jha	ઝ	jha	ઝ	jha
69	ઈ	ii	ચ્	ch1	ચ્	ch1	ચ્	ch1	ચ્	ch1
70	ખ	kha	જ	ja	જ	ja	જ	ja	જ	ja
71	ય્	y1	ઞ્	nj-1	ઞ્	nj-1	ઞ્	nj-1	ઞ્	nj-1

Fig. B.6: Portion of Gujarati Glyph Map Table.



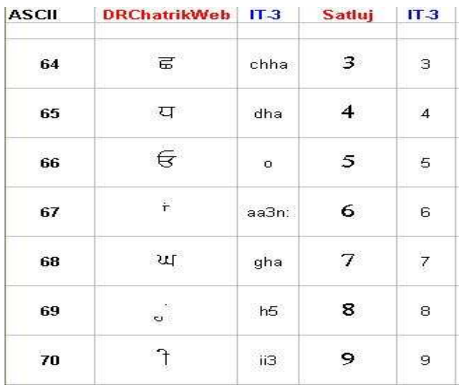
ASCII	hPrPfPO1	IT3	Shree-Ban	IT3	ItxBeng	IT3	AmarBangla	IT3
65	া	aa3	ক	ka	ি	i2	অ	a
66	ভ	bha	ক্ক	kka	জ	ja	আ	aa
67	ছ	chha	ক্ট	kt'a	ঙ	ng-a	ই	i
68	ধ	dha	ক্ত্ব	ktba	ঈ	ii	ঈ	ii
69	ি`	i2	ক্ব	kba	শ	sh1	উ	u
70		nil	ক্ন	kna	্য	ya2	ঊ	uu

Fig. B.7: Portion of Bengali Glyph Map Table.



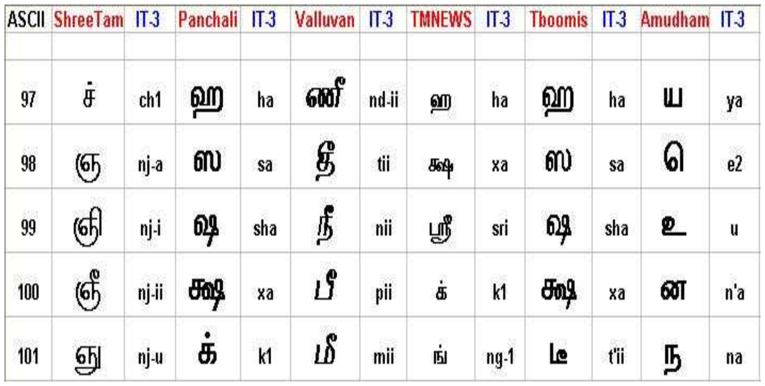
ASCII	AkrutiOri2	IT-3	Sambad	IT-3
64	ଉ	u	ଈ	ii
65	ଊ	uu	ଉ	u
66	ଋ	rx	ଊ	uu
67	ୠ	rx-	ଋ	rx
68	ଏ	e	ୠ	rx-
69	ଐ	ai	ଏ	e

Fig. B.8: Portion of Oriya Glyph Map Table.



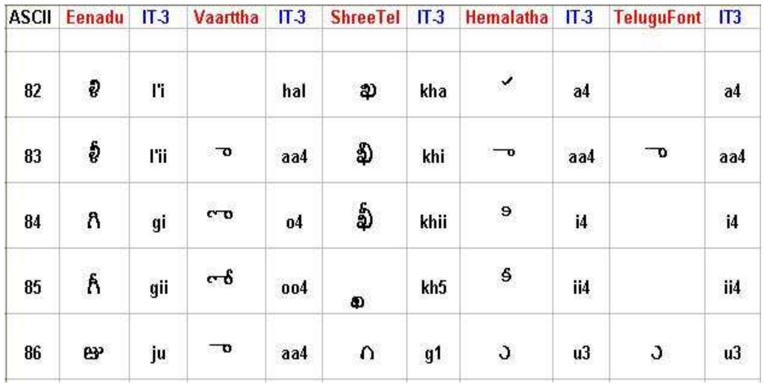
ASCII	DRChatrikWeb	IT-3	Satluj	IT-3
64	ਛ	chha	੩	3
65	ਧ	dha	੪	4
66	ੳ	o	੫	5
67	ਾਂ	aa3n:	੬	6
68	ਘ	gha	੭	7
69	੍ਹ	h5	੮	8
70	ੀ	ii3	੯	9

Fig. B.9: Portion of Punjabi Glyph Map Table.



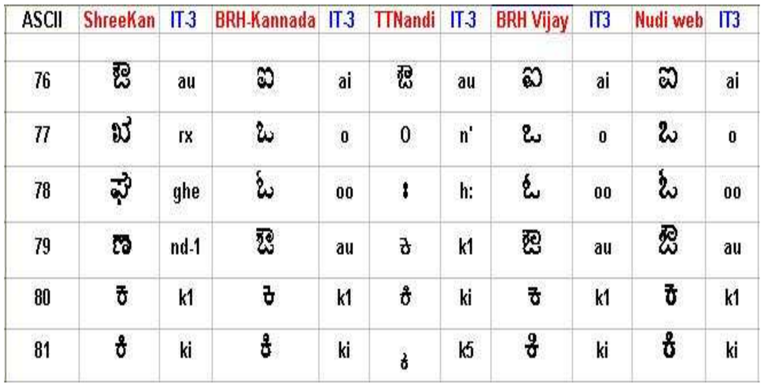
ASCII	ShreeTam	IT-3	Panchali	IT-3	Valluvan	IT-3	TMNEWS	IT-3	Tboomis	IT-3	Amudham	IT-3
97	ச்	ch1	ஹ	ha	ணீ	nd-ii	ஹ	ha	ஹ	ha	ய	ya
98	ஞ	nj-a	ஸ	sa	தீ	tii	க்ஷ	xa	ஸ	sa	ெ	e2
99	ஞி	nj-i	ஷ	sha	நீ	nii	ஸ்ரீ	sri	ஷ	sha	உ	u
100	ஞீ	nj-ii	க்ஷ	xa	பீ	pii	க்	k1	க்ஷ	xa	ன	n'a
101	ஞு	nj-u	க்	k1	மீ	mii	ங்	ng-1	டீ	t'ii	ந	na

Fig. B.10: Portion of Tamil Glyph Map Table.



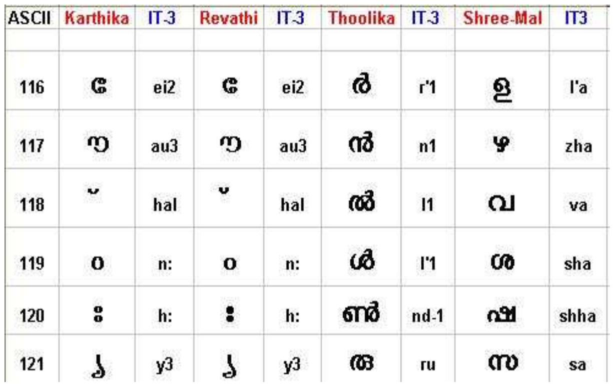
ASCII	Eenadu	IT-3	Vaarttha	IT-3	ShreeTel	IT-3	Hemalatha	IT-3	TeluguFont	IT3
82	ళి	l'i		hal	ఖ	kha	✓	a4		a4
83	ళీ	l'ii	ా	aa4	ఖి	khi	ా	aa4	ా	aa4
84	గి	gi	ొ	o4	ఖీ	khii	ి	i4		i4
85	గీ	gii	ో	oo4	ఖ	kh5	ీ	ii4		ii4
86	జు	ju	ా	aa4	∩	g1	ు	u3	ు	u3

Fig. B.11: Portion of Telugu Glyph Map Table.



ASCII	ShreeKan	IT-3	BRH-Kannada	IT-3	TTNandi	IT-3	BRH Vijay	IT3	Nudi web	IT3
76	ಔ	au	ಐ	ai	ಔ	au	ಐ	ai	ಐ	ai
77	ಋ	rx	ಒ	o	0	n'	ಒ	o	ಒ	o
78	ಘೆ	ghe	ಓ	oo	ಃ	h:	ಓ	oo	ಓ	oo
79	ಣ	nd-1	ಔ	au	ಠ	k1	ಔ	au	ಔ	au
80	ಕ	k1	ಕ	k1	ಕಿ	ki	ಕ	k1	ಕ	k1
81	ಕಿ	ki	ಕಿ	ki	್ಕ	k5	ಕಿ	ki	ಕಿ	ki

Fig. B.12: Portion of Kannada Glyph Map Table.



ASCII	Karthika	IT-3	Revathi	IT-3	Thoolika	IT-3	Shree-Mal	IT3
116	െ	ei2	െ	ei2	ർ	r'1	ള	l'a
117	ൌ	au3	ൌ	au3	ൻ	n1	ഴ	zha
118	്	hal	്	hal	ൽ	l1	വ	va
119	ം	n:	ം	n:	ൾ	l'1	ശ	sha
120	ഃ	h:	ഃ	h:	ൺ	nd-1	ഷ	shha
121	്യ	y3	്യ	y3	ഋ	ru	സ	sa

Fig. B.13: Portion of Malayalam Glyph Map Table.

APPENDIX C

GLYPH ASSIMILATION RULES FOR INDIAN LANGUAGES

In this appendix, we provide the glyph assimilation rules defined by us and used in the above mentioned font converters. For every Indian language a set of glyph assimilation rules are given under respective levels. Rules are written using IT3 scheme 2.3.

C.1 GLYPH ASSIMILATION RULES FOR HINDI

C.1.1 Modifier Modification

(i) full consonant + Halant = half consonant

(ii) (d’a, d’ha, ra) + Nukta = (d-a, dh-a, r’a)

(iii) (ka, kha, ga, ja, ta, na, la, ya) + Nukta = (k, kh, g, j, t, n, l, y) + “-a”

C.1.2 Language Preprocessing

(i) REF (r4) moves before the first full consonant

(ii) i + r4 = ii

(iii) n: + Maatra = Maatra + n:

(iv) half consonant + aa3 = full consonant

(v) i2 moves to next of the next full consonant

C.1.3 Maatra Assimilation

(i) aa3 + e-4 = o-3

(ii) aa3 + e’4 = o’3

(iii) aa3 + e4 = o3

(iv) aa3 + ai4 = au3

(v) aa3 + ei4 = o3

(vi) aa3 + ein:4 = on:3

C.1.4 Vowel Assimilation

(i) a + o-3 = o-

(ii) a + o’3 = o’

(iii) a + o = o

(iv) a + au3 = au

(v) a + aa3 = aa

(vi) e + e-3 = e-

(vii) e + e’3 = e’

(viii) e + e3 = ai

(ix) ei + ei4 = ei

C.1.5 Consonant Cluster

full consonant + (consonant2/3/5) = consonant cluster

C.2 GLYPH ASSIMILATION RULES FOR TAMIL

C.2.1 Modifier Modification

(i) full consonant + Halant = half consonant

C.2.2 Language Preprocessing

(i) aa3 + Halant = r1

(ii) aa3 + (i4 or ii4) = r1 + (i4 or ii4)

(iii) (e2, ei2, ai2) + consonant = consonant + (e2, ei2, ai2)

C.2.3 Maatra Assimilation

(i) e2 + aa3 = o3

(ii) ei2 + aa3 = oo3

(iii) e2 + l’a = au3

C.2.4 Consonant Cluster

full consonant + (consonant2/3/5) = consonant cluster

C.3 GLYPH ASSIMILATION RULES FOR GUJARATI

C.3.1 Modifier Modification

(i) full consonant + Halant = half consonant

C.3.2 Language Preprocessing

(i) REF (r4) moves before the first full consonant

(ii) i2 moves to next of the next full consonant

(iii) left Maatra + consonant = consonant + left Maatra

(iv) m:1 + n: = m:

(v) full consonant + m:1 + n: = full consonant + e-3

C.3.3 Maatra Assimilation

(i) aa3 + e4 = o3

(ii) aa3 + m:1 = o-3

(iii) aa3 + ai4 = au3

(iv) aa3 + ein:4 = on:3

(v) aa3 + ai4n:4 = aun:3

C.3.4 Vowel Assimilation

(i) a + e4 = e

(ii) a + e-3 = o-

(iii) a + o3 = o

(iv) a + ai4 = ai

(v) a + au3 = au

(vi) a + aa3 = aa

C.3.5 Consonant Assimilation

Assume CONSO = (kh1, g1, gh1, ch1, b1, bh1, p1, t1, th1, tr1, tt1, sh1, shh1, s1, x1, n1, nd-1, nj-1, nn1, m1, y1, v1, l1, l’1, h1, hy1, dy1)

(i) CONSO + aa3 =   + a

(ii) CONSO + o-3 =   + e-3

(iii) CONSO + o3 =   + e3

(iv) ruu + aa3 = stra

(v) jaa + e4 = jo

C.3.6 Consonant Cluster

full consonant + (consonant2/3/5) = consonant cluster

C.4 GLYPH ASSIMILATION RULES FOR MALAYALAM

C.4.1 Modifier Modification

(i) full consonant + Halant = half consonant

(ii) left Maatra + n1 = n1 + left Maatra

C.4.2 Language Preprocessing

(i) e2 + e2 + consonant = consonant + e2 + e2

(ii) left Maatra + consonant = consonant + left Maatra

C.4.3 Maatra Assimilation

(i) e2 + e2 = ai3

(ii) e2 + aa3 = o3

(iii) ei2 + aa3 = oo3

(iv) e2 + au3 = au3

C.4.4 Vowel Assimilation

(i) e + e2 = ai

(ii) o + aa3 = oo

(iii) i + au3 = ii

(iv) u + au3 = uu

(v) o + au3 = au

C.4.5 Consonant Cluster

full consonant + (consonant2/3/5) = consonant cluster

C.5 GLYPH ASSIMILATION RULES FOR TELUGU

C.5.1 Modifier Modification

(i) full consonant + Halant = half consonant

(ii) n: + Halant = r1

C.5.2 Language Preprocessing

(i) a4 + (n:, p1, s1, shh1, h1, ph1, ch1) = (n:, p1, s1, shh1, h1, ph1, ch1) + a4

(ii) left Maatra + consonant = consonant + left Maatra

(iii) ai5 + e4 = e4 + ai5

(iv) (vowel or n:) + (e4 or o4) + consonant = consonant + (e4 or o4)

(v) h1 + aa4 = ha

(vi) Maatra + n: = n: + Maatra

C.5.3 Consonant Assimilation

(i) half consonant + a4 = full consonant

(ii) n: + a4 = ra

(iii) n: + half consonant + Maatra = r1 + half consonant + Maatra

(iv) (h5 + di) or (di + h5) = dhi

(v) (h5 + d1) or (d1 + h5) = dh1

(vi) (h5 + s1) or (s1 + h5) = s1

(vii) (d1 + th1) or (th1 + d1) = dh1

(viii) (h5 + d5) or (d5 + h5) = dh1

(ix) (h5 + d’1) or (d’1 + h5) = d’h1

(x) (ch1, p1) + h5 = (ch1, ph1)

(xi) h1 + a4 + aa4 = ha

(xii) v1 + u3 = m1

(xiii) ri + t’h1 = t’hi

C.5.4 Maatra Assimilation

(i) (a4 + Maatra) or (Maatra + a4) = Maatra

(ii) (e4 + ai5) or (ai5 + e4) = ai3

(iii) e4 + i4 = ei4

(iv) o4 + i4 = oo4

(v) u3 + u3 = u1

C.5.5 Consonant-Vowel Assimilation

(i) e +h5 = ph1

(ii) e + a4 = va

(iii) e + Maatra = p1 + Maatra

(iv) ba + u1 = ru

(v) va + u3 = ma

(vi) va + uu3 = maa

(vii) v1 + a4 + (u1 or uu1) = ma + (u1 or uu1)

(viii) v1 + e4 + u1 = mo

(ix) v1 + ei4 + u3 = moo

(x) v1 + e4 + uu3 = moo

(xi) v1 + a4 + uu3 = maa

(xii) v1 + a4 + u3 = ma

(xiii) vi + u3 = mi

(xiv) vi + i4 + u3 = mii

(xv) vi + uu3 = mii

(xvi) n: + (u1 or uu1) = yi

(xvii) y1 + a4 + u3 = ya

(xviii) y1 + a4 + uu3 = yaa

(xix) y1 + ei4 + u3 = yei

(xx) y1 + e4 + uu3 = yoo

(xxi) y1 + e4 + u3 = ye

(xxii) y1 + u3 = yi

(xxiii) ya + u3 = ya

(xxiv) ya + uu3 = yaa

(xxv) p1 + u3 = ma

(xxvi) p1 + a4 + u3 = ma

(xxvii) p1 +e4 + uu3 = ghoo

(xxviii) p1 + a4 + u3 = ghu

(xxix) p1 + a4 + uu3 = ghoo

(xxx) ph1 + u3 = gh1

(xxxi) ph1 + a4 + u1 = ghu

(xxxii) ph1 + a4 + uu3 = ghu

(xxxiii) pha + u3 = gha

(xxxiv) pha + u1 = ghu

(xxxv) shh1 + u3 = jh1

(xxxvi) shh1 + a4 + u1 = jhu

(xxxvii) shh1 + a4 + u3 = jha

(xxxviii) m1 + e4 + u3 = mo

(xxxix) m1 + e4 + uu3 = moo

C.5.6 Consonant Cluster

full consonant + (consonant2/3/5) = consonant cluster

C.6 GLYPH ASSIMILATION RULES FOR KANNADA

C.6.1 Modifier Modification

(i) full consonant + Halant = half consonant

(ii) u1 + Halant = u3

C.6.2 Language Preprocessing

(i) arkkaa vottu (r3) moves before to the next first full consonant

(ii) left Maatra + consonant = consonant + left Maatra

C.6.3 Consonant Assimilation

(i) half consonant + a4 = full consonant

(ii) e + a4 = va

(iii) n: + u3 = y1

(iv) n: + u3 + a4 = y1

C.6.4 Maatra Assimilation

(i) (a4 + Maatra) or (Maatra + a4) = Maatra

(ii) (e4 + ai5) or (ai5 + e4) = ai3

(iii) e4 + (uu3 or uu5) = o3

(iv) (e4 + ii3) or (ii3 + e4) = ei3

(v) u3 + u1 + aa4 = aa3

(vi) u3 + u3 = u1

C.6.5 Consonant-Vowel Assimilation

(i) e + a4 + u3 = ma

(ii) e + a4 + uu3 = maa

(iii) e + aa4 = vaa

(iv) e + e4 + u1 = mu

(v) e + e4 + u3 = me

(vi) e + e4 + u3 + ii3 = mei

(vii) e + e4 + u3 + ai5 = mai

(viii) e + e4 + u3 + uu3 = mo

(ix) e + e4 + u3 + uu3 + ii3 = moo

(x) ri + jh3 + u3 = jhi

(xi) yi1 + u3 = yi

(xii) y1 + u3 = ya

(xiii) y1 + uu3 = aa

(xiv) y1 + e4 + u3 = ye

(xv) y1 + e4 + u3 + ii3 = yei

(xvi) y1 + e4 + u3 + uu3 = yo

(xvii) y1 + e4 + u3 + uu3 + ii3 = yoo

(xviii) y1 + e4 + uu3 = yuu

(xix) ye1 + u3 = ye

(xx) ye1 + u3 + ii3 = yei

(xxi) ye1 + u3 + ai5 = yai

(xxii) ye1 + u3 + uu3 = yo

(xxiii) ye1 + u3 + uu3 + ii3 = yoo

(xxiv) ye1 + uu3 = yo

(xxv) ye1 + uu3 + ii3 = yoo

(xxvi) ya + (u3 or u1) = ya

(xxvii) ra + jh3 = jha

(xxviii) ra + jh3 + u3 = jha

(xxix) ph1 + u3 = gh1

(xxx) va + aa3 = maa

(xxxi) va + u1 = mu

(xxxii) va + uu3 = muu

(xxxiii) v1 + o3 = mo

(xxxiv) v1 + oo3 = moo

(xxxv) v1 + e4 + u3 = me

(xxxvi) v1 + e4 + u3 + ii3 = mei

(xxxvii) v1 + e4 + u3 + ai5 = mai

(xxxviii) vi + u3 = mi

(xxxix) vi + u3 + ii3 = mii

(xxxx) r1 + e4 + jh3 = jha

(xxxxi) r1 + e4 + jh3 + u3 = jhe

(xxxxii) r1 + e4 + jh3 + u3 + ii3 = jhei

(xxxxiii) r1 + e4 + jh3 + u3 + ai5 = jhai

(xxxxiv) r1 + e4 + jh3 + uu3 = jho

(xxxxv) r1 + e4 + jh3 + uu3 + ii3 = jhoo

(xxxxvi) shi + consonant + ii3 = shi + consonant + i3

(xxxxvii) shi + ii3 = shi + i3

C.6.6 Consonant Cluster

full consonant + (consonant2/3/5) = consonant cluster

C.7 GLYPH ASSIMILATION RULES FOR PUNJABI

C.7.1 Modifier Modification

(i) full consonant + Halant = half consonant

(ii) (la, kha, sa, pha, ja, ga) + Nukta = (l’a, kh-a, sha, ph-a, j-a, g-a)

C.7.2 Language Preprocessing

(i) (n: or m:) + right side Maatra = Maatra + (n: or m:)

(ii) left Maatra + consonant = consonant + left Maatra

(iii) om:1 + m4 = om:

(iv) i2 + consonant = i2 moves next to the next full consonant

C.7.3 Vowel Assimilation

(i) a + aa3 = aa

(ii) a + aa3n: = aan:

(iii) a + a-4 = a-

(iv) a + ai4 = ai

(v) a + au4 = au

(vi) i1 + i2 = i

(vii) i1 + i3 = ii

(viii) u1 + u5 = u

(ix) u1 + e4 = e

C.7.4 Consonant Cluster

full consonant + (consonant2/3/5) = consonant cluster

C.8 GLYPH ASSIMILATION RULES FOR BENGALI

C.8.1 Modifier Modification

(i) full consonant + Halant = half consonant

(ii) (ba, ya, d'a, d'ha, ja) + Nukta = (ra, y-a, d-a, r'a, j-a)

C.8.2 Language Preprocessing

(i) left Maatra + consonant = consonant + left Maatra

(ii) i2 moves to next of the next full consonant

(iii) (m: or n:) + Maatra = Maatara + (m: or n:)

(iv) (sh1, kh1, p1, nd-1, g1, th1, dh1) + aa3 = (sha, kha, pa, nd-a, ga, tha, dha)

C.8.3 Maatra Assimilation

(i) (au4 + e2) or (e2 + au4) = ai3

(ii) e2 + aa3 = o3

(iii) e2 + aa3 + au4 = au3

(iv) e2 + au3 = au3

C.8.4 Vowel Assimilation

(i) a + aa3 = aa

(ii) (e + au4) or (au4 + e) = ai

(iii) (o + au4) or (au4 + o) = au

C.8.5 Consonant Assimilation

(i) tta + k3 = kta

(ii) tra + k3 = kra

(iii) cha + chha = chchha

(iv) tra + r3 = kra

(v) dba + h3 = ddha

(vi) kba + r3 = kka

(vii) x4 + k3 = kta

(vi) dba + uu3 = ddha

(vii) bba + uu3 = bdha

(viii) nba + ii3 = ndha

C.8.6 Consonant-Vowel Assimilation

(i) d’a + au4 = u

(ii) ha + au4 = i

(iii) (d’hs + au4) or (au4 + d’ha) = t’a

(iv) e + r3 = kra

(v) e + nj-3 = nj-a

(vi) dba + uu3 = ddha

(vii) bba + uu3 = bdha

(viii) nba + ii3 = ndha

(ix) x4 + e = kra

(x) x4 + o = tta

C.8.7 Consonant Cluster

full consonant + (consonant2/3/5) = consonant cluster

C.9 GLYPH ASSIMILATION RULES FOR ORIYA

C.9.1 Modifier Modification

(i) full consonant + Halant = half consonant

(ii) (d’a, d’ha) + Nukta = (d-a, dh-a)

C.9.2 Language Preprocessing

(i) left Maatra + consonant = consonant + left Maatra

(ii) REF (r4) moves before the first full consonant

(iii) i2 + consonant = i2 moves next to the next full consonant

C.9.3 Maatra Assimilation

(i) e2 + aa3 = o3

(ii) e2 + au3 = au3

(iii) e2 + ai4 = ai3

(iv) e2 + o4 = ai3

C.9.4 Vowel Assimilation

(i) a + aa3 = aa

C.9.5 Consonant Assimilation

(i) (h + da) or (da + h) = ha

C.9.6 Consonant Cluster

full consonant + (consonant2/3/5) = consonant cluster

REFERENCES

[1] LowVisionCare, Visual Problem Status Globally. http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1705713.

[2] Visual Problem Status in India. http://www.fastcompany.com/magazine/43/drv.html.

[3] MSDN, Microsoft Active Accessibility. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnanchor/html/accessibility.asp.

[4] Unicode Consortium - Universal Code Standard. http://www.unicode.org.

[5] Prahallad Lavanya, Prahallad Kishore and GanapathiRaju Madhavi, "A simple approach for building transliteration editors for indian languages," *Journal of Zhejiang University Science*, vol. 6A, no. 11, pp. 1354–1361, 2005.

[6] EmacSpeak Screen Reader. http://emacspeak.sourceforge.net.

[7] Freedom Scientifi c, Job Access and Work Software (JAWS) for Windows. http://www.freedomscientifi c.com/fs_products/software_jaws.asp.

[8] G W Micro, Windows-Eyes Screen Reader. www.gwmicro.com/Window-Eyes.

[9] Dolphin Computer Access, HAL Screen Reader. http://www.yourdolphin.com/productdetail.asp?id=5.

[10] Kruzweil, Text Reader with TTS. http://www.kurzweiledu.com/support_k3000win.asp.

[11] NAB NewDelhi, Screen Access For All, (SAFA): Screen Reader in Indian languages. http://safa.sourceforge.net.

[12] Prologix, Vaachak: The Indian Language Text to Speech Software. http://www.prologixsoft.com/vaachak.htm.

[13] IIT Madras, Software for the Visually Handicapped. http://acharya.iitm.ac.in/disabilities/software.php.

[14] Analytica, VoiceDot Screen reader for India. http://www.analytica-india.com/products/aries.html.

[15] UTF - Unicode Transformation Form. http://www.unicode.org/versions/Unicode 4.0.1.

[16] ISCII - Indian Standard Code for Information Interchange. http://tdil.mit.gov.in/standards.htm.

[17] ISO/IEC JTC1/SC18/WG8, Document Processing and Related Communication - Document Description and Processing Languages, PDTR15285 An operational model for characters and glyphs, 1997. www1.y12.doe.gov/capabilities/sgml/wg8/document/1909.pdf.

[18] Ganapathiraju M., Balakrishnan M., Balakrishnan N. and Reddy R., "Om: One tool for many (indian) languages," *Journal of Zhejiang University Science*, vol. 6A, no. 11, pp. 1348–1353, 2005.

[19] INGLE N. C., "A language identifi cation table," 1976.

[20] Beesley, K. R., "Language identifi er: A computer program for automatic natural-language identifi cation on on-line text," 1988.

[21] William B. Cavnar and John M. Trenkle, "N-gram-based text categorization," in *Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, (Las Vegas, US), pp. 161–175, 1994.

[22] Penelope Sibun and Jeffrey C. Reynar, "Language identifi cation: Examining the issues," in *5th Symposium on Document Analysis and Information Retrieval*, (Las Vegas, Nevada, U.S.A.), pp. 125–135, 1996.

[23] William J. Teahan and David J. Harper, "Using compression-based language models for text categorization," in *Workshop on Language Modeling and Information Retrieval*, 2001.

[24] Emmanuel Giguet, "Multilingual sentence categorization according to language," in *Proceedings of the European Chapter of the Association for Computational Linguistics SIGDAT Workshop*, (Dublin, Ireland.), pp. 73–76, 1995.

[25] G. Kikui, "Identifying the coding system and language of on-line documents on the internet," 1996.

[26] A. K. Singh, "Study of some distance measures for language and encoding identifi cation," in *Proceedings of the Workshop on Linguistic Distances*, (Sydney, Australia), pp. 63–72, Association for Computational Linguistics, July 2006.

[27] Anil Kumar Singh and Jagadeesh Gorla, "Identifi cation of languages and encodings in a multilingual document," in *Proceedings of the 3rd ACL SIGWAC Workshop on Web As Corpus*, (Louvain-la-Neuve, Belgium.), 2007.

[28] XiuFen Miao, Fang Yang, Xue-Dong Tain and Bao-Lan Guo, "Identifi cation of languages and encodings in a multilingual document," in *Conference on Machine Learning and Cybernetics*, (Beijing, China.), 2002.

[29] M. Jung and Y. Shin and S. Srihari, "Multifont classifi cation using typographical attributes," in *ICDAR '99: Proceedings of the Fifth International Conference on Document Analysis and Recognition*, (Washington, DC, USA), p. 353, IEEE Computer Society, 1999.

[30] Min-Chul Jung Yong-Chul Shin and Srihari S.N., "Identifi cation of languages and encodings in a multilingual document," in *Fifth International Conference on Document Analysis and Recognition, ICDAR*, (Banglore, India), 1999.

[31] TBIL Data Converter. www.bhashaindia.com/Downloadsv2/Category.aspx?ID=4.

[32] FontSuvidha - Devanagari Font Conversion.
http://www.cybershoppee.com/products/fontsuvidha5.htm.

[33] IConverter, A utility program for various code conversions.
http://www.cse.iitk.ac.in/users/isciig/iconverter/main.html.

[34] Akshar Bharati, Nisha Sangal, Vineet Chaitanya, Rajeev Sangal and G Uma Maheshwara Rao, "Generating converters between fonts semi-automatically," in *Proceedings of SAARC conference on Multi-lingual and Multi-media Information Technology*, (CDAC, Pune, India), 1998.

[35] Himanshu Garg, "Overcoming the font and script barriers among indian languages," *MS Thesis at International Institute of Information Technology Hyderabad, India*, 2005.

[36] Khudanpur S. and Schafer C., Devanagari Converters. http://www.cs.jhu.edu/cschafer/jhu devanagari cvt ver2.tar.gz.

[37] M.N.S.S.K. Pavan Kumar, S.S. Ravi Kiran, Abhishek Nayani, C.V. Jawahar and P J Narayanan, "Tools for developing ocrs for indian scripts," in *Proceedings of the Workshop on Document Image Analysis and Retrieval (DIAR:CVPR'03)*, (Madison, WI, USA.), 2003.

[38] IIIT Hyderabad, Reading Aid for Visually Impaired (RAVI) Screen Reader for Indian languages.
http://ltrc.iiit.ac.in/newsletter/trishna-august2005.pdf.

[39] Microsoft TTS. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/SAPI51sr/html/SDKtutorial_tts.asp.

[40] S.P.Kishore, Rohit Kumar and Rajeev Sangal, "A data driven synthesis approach for indian languages using syllable as basic unit," in *Proceedings of International Conference on Natural Language Processing (ICON)*, (Mumbai, India), 2002.

[41] Rohit Kumar, "A genetic algorithm for unit selection based speech synthesis," in *International Conference on Spoken Language Processing (Interspeech - ICSLP)*, (Jeju, Korea.), 2004.

[42] MSDN, Automating Office with VC++: Knowledge Base Articles.
http://msdn.microsoft.com/library/default.asp?url=/archive/en-us/dnaro97ta/html/msoautovc.asp.

[43] L V Prasad Eye Hospital - Vision Rehabilitation Center. http://www.lvpei.org/.

LIST OF PUBLICATIONS

The work done during my masters has been disseminated to the following conferences/journals:

1. Anand Arokia Raj, Rahul.Ch, Susmitha, "A Voice Interface for the Visually Impaired" at Sciences of Electronic, Technologies of Information and Telecommunications (SETIT) - March 2005 Susa, Tunisia.
2. Anand Arokia Raj, A Susmitha, S P Kishore, "Indian Language Text-To-Speech Interface for Visually Impaired" at International Conference on Natural Language Processing (ICON) - December 2005, IIT-Kanpur, India.
3. Anand Arokia Raj, Tanuja Sarkar, Sathish Chandra Pammi, Santhosh Yuvaraj, Mohit Bansal, Kishore Prahallad, Alan W Black, "Text Processing for Text-To-Speech Systems in Indian Languages" at 6th ISCA Workshop on Speech Synthesis (SSW6) August 2007, Bonn, Germany.
4. Anand Arokia Raj, Kishore Prahallad, "Identification and Conversion of Font-Data in Indian Languages" at International Conference on Universal Digital Library (ICUDL2007) November 2007, Pittsburgh, USA.

CURRICULUM VITAE

1. **NAME:** A.Anand Arokia Raj

2. **DATE OF BIRTH:** 10 January 1982

3. **PERMANENT ADDRESS:**
   A.Anand Arokia Raj
   S/O: A.Antoni Muthu Rayar
   South Street, Pattanamkurichy
   Andimadam 621801
   Perambalur Dt
   Tamil Nadu, India

4. **EDUCATIONAL QUALIFICATIONS:**

   - April 2003: Bachelor of Technology (Information Technology, Arunai Engineering College, University of Madras)

   - February 2008: Master of Science (by Research) (Computer Science and Engineering, IIIT Hyderabad)

THESIS COMMITTEE

1. **GUIDE:** Mr. S. Kishore Prahallad

2. **MEMBERS:**

   - Dr. C. V. Jawahar

   - Mrs. Amba P Kulkarni