PARAMETERIZED ARABIC FONT DEVELOPMENT FOR COMPUTER TYPESETTING SYSTEMS

by

Ameer Mohamed Sherif Mahmoud Hamdy

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
in
Electronics and Electrical Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
January 2008

PARAMETERIZED ARABIC FONT DEVELOPMENT FOR COMPUTER TYPESETTING SYSTEMS

by

Ameer Mohamed Sherif Mahmoud Hamdy

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
in
Electronics and Electrical Communications Engineering

Under the Supervision of

Amin M. Nassar
Professor
Elec. and Com. Dept.

Hossam A. H. Fahmy
Assistant Professor
Elec. and Com. Dept.

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
January 2008

PARAMETERIZED ARABIC FONT DEVELOPMENT FOR COMPUTER TYPESETTING SYSTEMS

by

Ameer Mohamed Sherif Mahmoud Hamdy

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE
in
Electronics and Electrical Communications Engineering

Approved by the
Examining Committee

Prof. Amin M. Nassar, Main Thesis Advisor

Prof. Mohsen A. Rashwan, Member

Prof. Mohamed Y. El Hamalawy, Member

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
January 2008

Acknowledgments

First, I would like to thank Allah for granting me the chance to join this department, in which I have enjoyed learning and teaching for the past eight years. It gave me the opportunity to meet many people who have influenced my life in many ways. I also thank Him for surrounding me with those I mention below.

Many thanks goes to my supervisors, Prof. Amin Nassar for his assistance and habit of making things easy and enjoyable, and to Dr. Hossam Fahmy for his encouragement, patience, and lively discussions. His guidance and support reached far beyond scientific research, and was more of a mentor than a supervisor.

I would also like to express my gratitude to ‘Abdel-Nasser Ghoneim for his assistance on using TEX and his many ideas, Mr. Sameer El-‘Aidy, and Mr. Khaled El-Husseiny for their invaluable addition to my knowledge of calligraphy. Thanks also to Prof. Donald Knuth for the wonderful work he has done in developing TEX and METAFONT without which this work could not have been done. Also thanks for the people behind the Crimson Editor and the King Fahd complex for printing the Holy Qur’an for their freely distributed tools.

Very special thanks goes to Amir El Sherbiny, Sheikh Sa‘id, and my colleagues at the department, Hany Abol-Magd, ‘Amr ‘Essawy, and Mohamed Ismail who have been true friends and very encouraging on many occasions during this work.

Finally, I am thankful to my father, mother, siblings, and my many other family members for the complete support they provided through my entire life. I must also acknowledge my fiancée, Aya, without whose love and encouragement, I would not have finished this thesis.

Abstract

This thesis presents new approaches to Arabic font development for computer typesetting systems. In order to achieve an output quality close to that of Arabic calligraphers, we model the pen nib and the way it is used to draw curves as closely as possible using a font description language – METAFONT. Parameterized fonts are introduced to enable the drawing of whole words as single entities, this results in improved quality since the Arabic script is cursive by nature.

We utilize the true meta-design capability of METAFONT, analogous to the Computer Modern typeface families, and hence our design of Arabic letters includes a number of parameters which used to connect glyphs together, form ligatures, control kerning, and extend character lengths. We divide each letter glyph into smaller primitives that exist in multiple glyphs. Designing a primitive and then reusing it reduces design time.

We compare our method to the basic binding of glyphs using simple box and glue mechanisms that are used in most of today's word processors and typesetting systems, and also to currently existing font design technologies. Our method enables better connectivity of glyphs, hence better calligraphic quality, and more dynamic fonts, enabling more flexible typesetting. This comes at the expense of higher complexity of glyph designs. Meta-design of Arabic letters is discussed in detail through many examples, and methods of connecting glyphs to form words are also presented.

Finally, a subjective test was conducted for evaluating words produced using our parameterized font in comparison to other widely used fonts.

Contents

Acknowledgments **iii**

Abstract **v**

1 Introduction **2**

1.1 Historical Review of the Arabic Script 2

1.2 Differences with Latin Script 6

1.3 Goal of Thesis 9

1.4 Motivation 14

2 Digital Typography **18**

2.1 Typesetting Systems 19

2.2 Basic Typographic Concepts 21

2.3 Arabic Typesetting Requirements 29

3 Font Technologies **32**

3.1 OpenType 33

3.2 Metafont 38

4 Pen Modeling **44**

4.1 Pen Nib Outline 44

4.2 Pen Stroke 49

4.2.1 Describing Curves 49

4.2.2 Drawing with Pens 50

5 Arabic Meta-Design **66**

5.1 Meta-Design Methodology 67

5.1.1 Point Selection 68

5.1.2 Point Dependencies 70

5.2 Primitives 71

5.2.1 Type-1 Primitives 72

5.2.2 Type-2 Primitives 77

5.2.3 Type-3 Primitives 84

5.3 Primitive Substitution 88

5.4 Diacritic Glyphs 92

6 Forming Words **94**

6.1 Joining Glyphs with Kashidas 94

6.2 Vertical Placement of Glyphs 98

6.3 Word Lengths 100

6.4 A Final Example 101

6.5 Design of a Graphical User Interface 103

7 Results and Future Work **106**

7.1 A Subjective Test 106

7.1.1 Testing Methodology 106

7.1.2 Design of the Test 107

7.1.3 Test Results 111

7.1.4 Comments and Conclusion on Results 113

7.2 Future Work 113

7.2.1 Font Technology 113

7.2.2 Typesetting System 115

7.2.3 Output Format 116

List of Figures

1.1 Writing systems of the world. 3

1.2 The writing styles of Arabic 5

1.3 A photo showing Arabic metal type. 10

1.4 Development of Arabic typesetting. 11

1.5 Part of the Holy Qur'an written by a calligrapher. [1] 13

1.6 Sample output of AlQalam system. 15

2.1 Boxes and glue. 22

2.2 A numerical example of justification using the box/glue model. . . 23

2.3 Line-breaking examples. 24

2.4 Ligatures in English. 25

2.5 Kerning examples. 25

2.6 Optical considerations. 26

2.7 Optically uneven spacing. 27

2.8 Example of rivers in a paragraph. 28

2.9 Examples of orphans and widows. 29

2.10 Problems of Arabic justification. 30

3.1 A screenshot from an outline font editor, FontCreator 5.5. 34

3.2 Sample fonts from TrueType and OpenType. 35

3.3 Problems with OpenType fonts. 36

3.4 Part of the character code tables allocated to Arabic ligatures. . . 37

3.5 Optical scaling example. 38

3.6 Different fonts generated by a single METAFONT program. 39
3.7 Sample METAFONT glyphs of the letter ‘y’. 41
3.8 Sample METAFONT program describing the letter ‘y’. 42

4.1 A photo of an Arabic pen. 45
4.2 Various pen nib shapes in METAFONT. 45
4.3 A circular pen digitized after scaling. 46
4.4 Arabic pen nib outline defined. 47
4.5 The letter *alif* drawn with 3 different pen nibs. 48
4.6 A Bézier curve. 50
4.7 One path traced by two different pens. 51
4.8 The effect of pen rotation. 52
4.9 Drawing with the `penstroke` macro. 53
4.10 The first problem with `penstroke` – razor pen 54
4.11 The second problem with `penstroke` – figure-8 shape. 55
4.12 The third problem with `penstroke` – bad pen approximation. . . 56
4.13 A stroke with multiple crossings of right and left paths. 57
4.14 First solution of stroke modeling - `filldraw`. 59
4.15 Second solution of stroke modeling - `astroke`. 60
4.16 Third solution of stroke modeling - `qstroke` 62
4.17 Fourth solution of stroke modeling - `envelope` 64

5.1 Choosing points to define the path of the letter ’alif. 69
5.2 Usage of primitives in roman letters. 72
5.3 The tail primitive. 73
5.4 Four consequent tails in a word. 73
5.5 The base primitive. 76
5.6 The helya primitive. 77
5.7 The bow primitive. 78
5.8 The letters faa’ and qaf as drawn by three calligraphers. 79

5.9 The waw head primitive. 80
5.10 The sād head primitive. 82
5.11 The 'alif primitve. 83
5.12 Approximate directions in calligraphy books. 84
5.13 The kasa primitive. 85
5.14 The letter noon shown with kasa widths of 3, 9, 10 and 13 nuqtas. 87
5.15 The initial form of the letter haa' with two different kashida lengths. 89
5.16 Many primitive glyphs used to produce the different forms of haa'. 90
5.17 Isolated and ending forms of the letter dal. 91
5.18 The shara of kaf and the hamza. 92
5.19 The dot as drawn by different pen orientation. 93

6.1 Changing length of kashida between two letters. 97
6.2 Sample placing of kashidas using TrueType fonts. 97
6.3 The word *yahia* as it is written using four font technologies. . . . 99
6.4 Tracing a word from left-to-right to know starting vertical position. 100
6.5 Multiple instance of a word with different lengths. 102
6.6 Block diagram of GUI. 103
6.7 Screenshot of the designed GUI. 104
6.8 Screenshot of the DVI previewer displaying the output word. . . . 104

7.1 Different forms of the letters baa' and seen. 114
7.2 Example of variable diacritic lengths and their placement. 114
7.3 Dependence of letter's shape on following letters. 116

List of Tables

7.1 MOS results for each font. 111

7.2 MOS results for individual words in the test. 112

List of Listings

4.1 METAFONT code for drawing the letter baa' using `penstroke`. . . 54

4.2 METAFONT code of both the original `penstroke` and its modified version. 58

4.3 METAFONT code showing the modification to `penpos` macro. . . . 61

4.4 METAFONT code of the `qstroke` macro. 62

4.5 METAFONT code of the `envelope` macro. 63

5.1 METAFONT code describing the tail primitive. 74

5.2 METAFONT code describing the base primitive. 75

5.3 METAFONT code describing the helya primitive. 76

5.4 METAFONT code describing the bow primitive. 78

5.5 METAFONT code describing the waw head primitive. 80

5.6 METAFONT code describing the saad head primitive. 81

5.7 METAFONT code describing the alif primitive. 83

5.8 METAFONT code describing the kasa primitive. 86

5.9 METAFONT code describing the letter dal. 91

5.10 METAFONT code describing the shara and the hamza. 92

6.1 METAFONT code showing how effect of the kashida on glyphs. . . 96

Chapter 1

Introduction

The Arabic alphabet is used in writing many languages in many places in Africa and Asia. It is used to write Arabic in the Arab world, Persian in Iran, Urdu and Punjabi in India and Pakistan, Dari and Pashto in Afghanistan, Uyghur in some parts of China, Kashmiri in Kashmir, and Jawi in Malaysia and Indonesia are just some. Millions of people around the world use the alphabet's letters to write their languages. It was first used to write Arabic, and most importantly, the Qur'an. With the spread of Islam, this alphabet was then used to write languages different than Arabic, and in the process additional characters and symbols have been added to the original set in order to accommodate some of these languages. See Fig. 1.1

1.1 Historical Review of the Arabic Script

According to contemporary studies, Arabic writing is a member of the Semitic alphabetical scripts in which mainly the consonants are represented. The Arabic script was developed in a comparatively brief span of time, and its alphabet today is second in use only to the Roman alphabet.

The Arabic alphabet is a script of 28 letters and uses long but not short vowels. The letters are derived from only 17 distinct forms, distinguished from

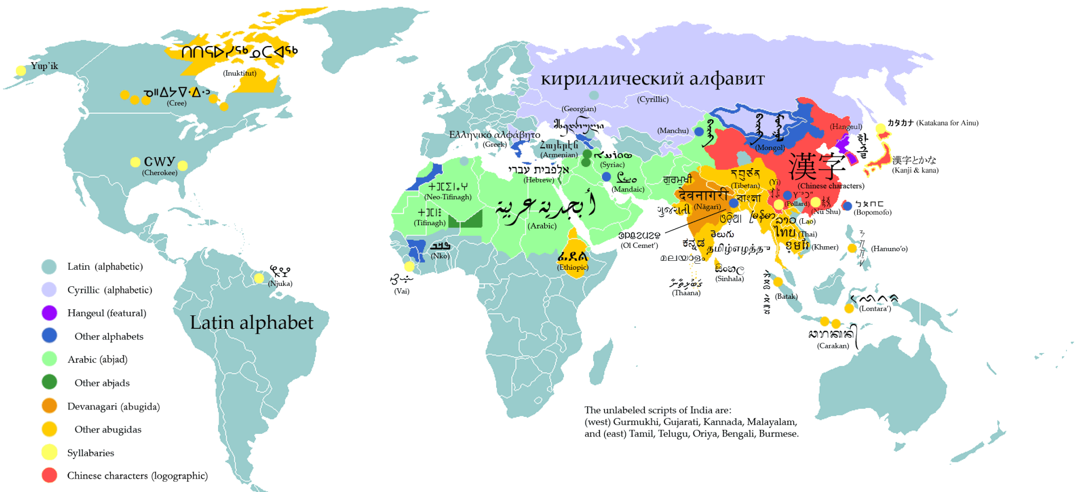

Figure 1.1: Writing systems of the world.

one another by one or more dots placed above or below the letter. Short vowels are indicated by small diagonal strokes above or below letters.

The Arabic script was first introduced during the 5th century in the north-eastern part of Arabia. As the teachings of Islam spread beyond the boundaries of the Arabian Peninsula, an enormous number of people worldwide became Muslims. The new Muslims interpreted the art of writing as an abstract expression of Islam, each according to their own cultural and aesthetic systems. The influx of this cultural diversity led to two major events: the birth of regional calligraphic schools and styles such as Ta'liq in Persia and Diwani in Turkey, and the need to reform the Arabic script. A clear and universal language with legible script was needed if the non-Arab Muslims were to learn Arabic and become part of the Islamic melting pot.

Sophistication of the Arabic Letterforms

While many religions have made use of figural images to convey their core convictions, Islam has instead used the shapes and sizes of words or letters. Because Islamic leaders saw in figural arts a possible implication of idolatry, the artistry of calligraphy was used for religious expression. In Islamic and Arabic cultures,

calligraphy became highly respected as an art – the art of writing.

Anthony Welch in his book "Calligraphy in the Arts of the Muslim World" in 1979 says: "Written from right to left, the Arabic script at its best can be a flowing continuum of ascending verticals, descending curves, and temperate horizontals, achieving a measured balance between static perfection of individual form and paced and rhythmic movement. There is great variability in form: words and letters can be compacted to a dense knot or drawn out to great length; they can be angular or curving; they can be small or large. The range of possibilities is almost infinite, and the scribes of Islam labored with passion to unfold the promise of the script."

Arabic lettering has achieved a high level of sophistication, and Arabic scripts can vary from flowing cursive styles like Naskh and Thuluth to the angular Kufi. On a traditional Islamic building, a number of different writing styles may appear on, for example, the walls, windows, or minarets. Most of the inscriptions are not only from the Qur'an but also the Hadith and are in harmony with the religious purposes of the building.

Arabic calligraphy is a symbol representing power and beauty. Its history is the integration of artistry and scholarship. Through the abstract beauty of the lines, energy flows in between the letters and words. All the parts are integrated into a whole. These parts include positive spacing, negative spacing, and the flow of energy that weaves together the calligrapher's rendering. The abstract beauty of Arabic calligraphy is not always easily comprehended – but this beauty will slowly reveal itself to the discerning eye.

Arabic Writing Styles

There are six major writing styles for Arabic: Kufi, Thuluth, Naskh, Riq'aa, Deewani, and Ta'liq. See Fig. 1.2 showing different Arabic writing styles. As the figure shows there are great differences between the different styles. These styles

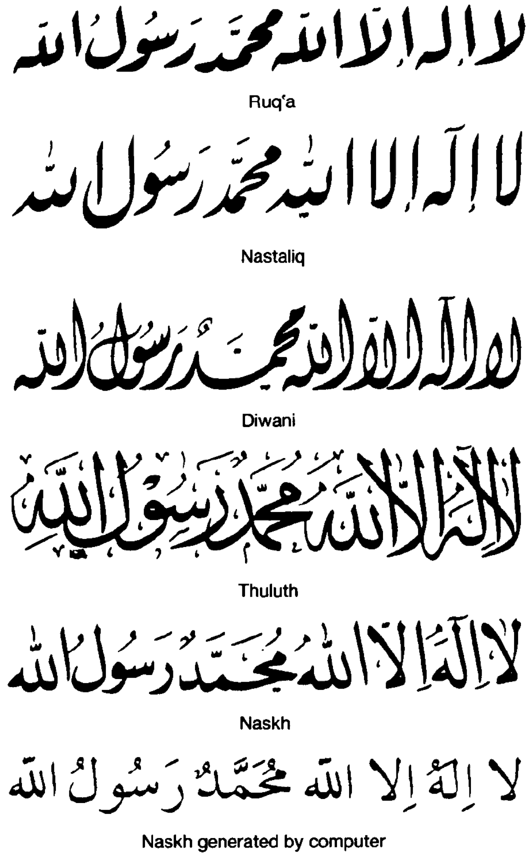

Figure 1.2: The writing styles of Arabic

as we write them today had undergone development through a long period of time, and each was developed by different people in different parts of the Arabic and Islamic worlds.

In most Arab countries in the Middle East, Naskh and Riq'aa, are the two most common styles of writing. Riq'aa is usually used in handwriting since it is simpler to write than other styles. Naskh is standard for printing books since a long time. This thesis focuses on implementing the rules of writing Naskh on computers. Other styles will need considerable modifications on the work developed in this thesis in order to be implemented in the same way.

1.2 Differences with Latin Script

There are many differences between the Arabic and Latin scripts. Most important is the fact that Arabic is cursive. Many other differences evolve from this fact. When a script is cursive each letter in a word is affected by the preceding and succeeding letters. In Latin, the look and shape of each letter normally does not depend on neighboring letters, and hence has a fixed shape.

Another consequence of being cursive is the existence of ligatures. Ligatures are special forms of characters joined together. In Latin, there are very few ligatures in each language. For example in English, combinations of the letter ‘f’ followed by ‘i’ or ‘l’ should take a special form, different from just bring shapes of both letter together, this is done according aesthetic considerations. In Arabic, the situation is different. Almost all letters join together forming ligatures, and to make things more complicated, ligatures may result from combinations of more than two letters, نَجْمَعُ instead of نجمع. Taking this further will show us that any word in Arabic, may be considered as a single ligature, hence we may conclude that ligatures in Arabic are infinite, compared to only few in Latin scripts.

Still more differences due to being cursive, is that in Arabic, the baseline of a word is that of the ending letters. Letters at the start of a word may be raised above their default baseline when a ligatures is formed, as in with the letter meem of the word Muhammad in Fig. 1.2. But note that ligatures are in many cases optional, and it is possible to write whole words in Naskh with all letters having the same baseline.

In its early development stages, the Arabic script did not contain any dots or diacritics. As time passed, it was seen a necessity to add suck additional marks for the purpose of simplifying the reading process especially for Muslims who were not originally Arabs. Without these marks it was difficult to identify a lot of letters, since many letters had similar shapes and were only identified from the context of the text. Although these marks did in fact make it easier to read and

comprehend what was written, it made the writing more complicated, especially for computer systems. Diacritics other than dots are also called 'tashkeel', and indicated vowel sounds, to be added to the word.

Yet another result of cursive script is the extensibility of Arabic letters. An Arabic letter has no fixed dimensions like Latin letters, and may spread in length for different reasons. One reason may be to fill up the space left in a line, another use is in mathematical writings to describe limits is an example. Calligraphers may also use this extension for aesthetic reasons.

In Latin scripts, there is a difference between printed and handwritten text, in Arabic, this difference is not supposed to exist. Latin letters were originally used in writing through carving on stones, while Arabic letters were first used to write on animal skin using pens. These different methods of writing affected the shapes of letters in both cases. Printed Latin letters are difficult to draw using normal pen strokes. Due to its cursive nature, Arabic words are written using pens, and continuous strokes, that may span the whole word, to the extent that some words may be written without removing the pen nib from the paper.

Another difference is not in the shape of the script but in the unit of measurement used. Latin calligraphers and punch cutters use the 'point' as the unit of measurement for glyph dimensions. A point is approximately equivalent to 1/72 of an inch. Arabic calligraphers on the other hand use the 'nuqta' or dot as unit of measurement. The dot is the length of the diagonal of the rectangular dot drawn by a specific pen.

It is acceptable to vary the space between adjacent words in order to justify a line to the margins of a page in Latin. In Arabic, however, it is not. Inter-word spacing in Arabic is fixed and minimal. Line justification if required, is achieved by extending some letters, or breaking and forming ligatures. Even the direction of text is different, while most languages are written from left-to-right, very few languages like Arabic and Hebrew are written the other way round, right-to-left.

The Arabic alphabet, written from right to left, is composed of 28 basic letters. Adaptations of the script for other languages such as Malay, Persian and Urdu have additional letters. There is no such thing as upper and lower case letters nor is there a difference between written and printed letters. Most of the letters connect directly to the letter which immediately follow, which gives written text an overall cursive appearance. Each individual letter can have up to four distinct forms, based upon where the letter appears in a word.

Considering all the above mentioned differences, it is very easy to understand why Arabic is classified in most computing classifications as a 'Complex Script'. And as a further conclusion, it is easy to understand why most computer typesetting systems still lack in supporting Arabic, while their support to Latin is very advanced.

How the Problem Came to Existence?

In the light of the previously mentioned and vast differences between Arabic and Latin text, we shall now see the situation of current desktop publishing software that is most used to write Arabic, and how it has developed in such a way.

The movable type was invented in the early 1400s by Johann Gutenberg in Germany. This invention made book production far more cost effective than before. It took him 10 years of experimentation. His books had to look handwritten because that was the market. For hundreds of years printing was only carried out on Latin text, and as mentioned before, Latin text is far simpler than Arabic, and each letter can be isolated in a separate box. When this technology was applied to Arabic, some sort of simplification was needed in order to isolate each glyph alone in analogy to Latin. Important or obligatory ligatures like lam-alef were cast on separate boxes. The mechanical process had imposed limitations on Arabic. Fig. 1.3 shows an example with the problem of metal type. The shadda diacritic is placed on a separate box, hence it will appear beside the word instead

of above it.

One of those limitations beside totally isolated letters was apparent in adding dots and tashkeel. Although the skeleton of letters like haa, jeem, and khaa is identical, a dot is what separates them. A metal cast had to be made for each of them since it was not feasible to arrange boxes above each other. Each letter had a cast of a box's shape and reserved all the vertical space. In order to add tashkeel then each of these letters can have a fatha, damma, kasra or shadda or even tanween, and many more. In order create a metal cast for each one, add to that the combinations of the dots would require an enormous amount of casts. And finally it this letter may occur more than once in a page. The practical solution was to add tashkeel on additional kashidas. This made the diacritics not on top of the corresponding letter but it was the only solution possible. See the top right word in Fig. 1.4.

When computers were invented and desktop publishing software developed, this limitation of the metal type mechanism migrated with the letters as if it was indeed part of the language. In computers nowadays, the boxes containing the letters are virtual and it is very possible to overlap them or stack them on top of each other. This fact was in fact exploited to improve the placement of dots and diacritics. Only lately did the original curves of cursive Arabic had been added to computerized Arabic fonts, but still the limitation of boxes themselves was not relieved. See Fig. 1.4 for a comparison of the printed word using metal and modern word processors versus that written by a calligrapher.

1.3 Goal of Thesis

Now, that we have identified the deficiencies in current Arabic typesetting systems, and mentioned the importance of filling the gap between the works of calligraphers, and the outputs of computer systems, we present our view of the solution and what is thought to be achieved through this thesis.

Figure 1.3: Metal type showing a word consisting of two ligatures and diacritics.

The goal of this thesis is to produce high quality Arabic text both for display on computers and for printing on paper. Our approach is to model how calligraphers write Arabic as accurately as possible to provide, to provide the required calligraphic quality, and the typesetting flexibility of the Arabic script. To provide better quality, we shall design better letterforms that truly follow the rules of Arabic calligraphy, and model the pen [2] and the way it is used to trace paths as accurately as possible.

As we mentioned before, the Arabic script being cursive, can be dynamic and hence should allow more flexibility in typesetting, like in line breaking and justification, which is limited in Latin text to just inter-word spacing and hyphenation. In order to achieve this flexibility on computers we will enable the decisions to add tatweel and form ligatures in a natural way.

Our proposed solution is to model as closely as possible how people, in general, handwrite the Arabic script, and more specifically, how calligraphers do it. Hence letters have to be completely interactive with neighboring ones and like in the real world, an Arabic writer in fact looks at a single word as one entity and all letters in it are drawn accordingly, hence it is like one large ligature.

The calligrapher also decides the positioning of the word above the reference line as a single entity not for each letter alone. Moreover, if the line has a limited horizontal space left for one word, the calligrapher will make use of additional

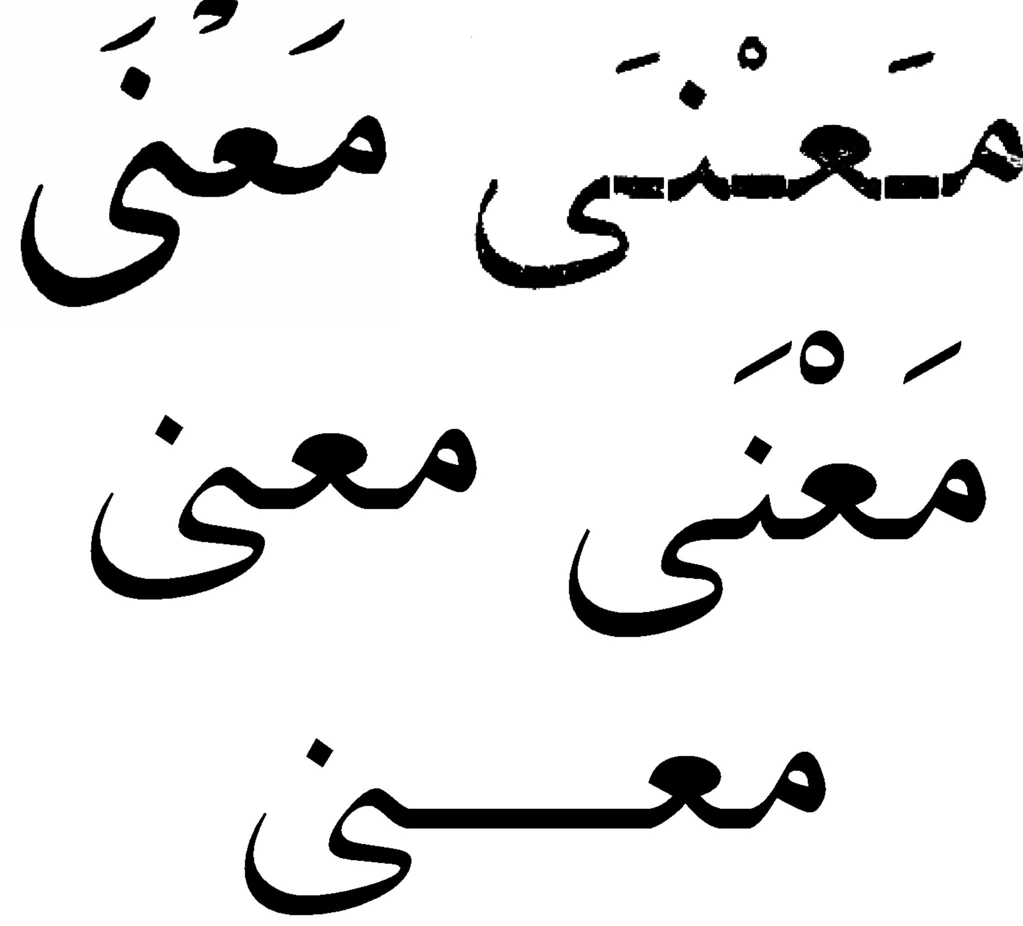

Figure 1.4: Develeopment of Arabic typesetting. The top left shows the word as it is written by a calligrapher. The top right figure shows the same word as printed using metal type, with the diacritics places on separate kashidas. The second row shows the word as it is written using a modern word processor – MS Word 2003. Note how the placement of the diacritics is all at one horizontal line, unlike when written by a calligrapher, and also how they are not well spaced from each other. Adding the diacritics also forced the word processor to break the noon-yaa' ligature. The final row shows the word extended by adding kashidas, which are purely straight lines.

ligatures and compress letters together if space is small, or break some ligatures and extend some letters if the space is large. Of course there are rules for breaking and forming ligatures and also for extending or compressing letters. Some of these rules have been documented in recent papers written in English [3, 4, 5]. Moreover, it is not acceptable to justify the lines only by varying the width of the spaces between words in the Arabic script.

We illustrate the previous ideas with examples obtained from al-Madinah Qur'an [1], one of the most popular printed editions in use. Breaking and forming ligatures is evident in words as أَصْحَٰبُ becoming أَصْحَٰبُ, and also ٱلْحَجِّ becoming ٱلْحَجَّ. Other examples show how the kashida or tatweel (elongation stroke) is used to give words extra length like in أَحَد, أَحَـد, and أَحَـــد. Note that in the latter example, the letter haa' can have different lengths of tatweel, hence it does not make sense to store all these different lengths as glyphs to be substituted when needed.

In some cases, the calligrapher may need to extend more than one letter in a word, for example أَكُونَ extended to أَكُـونَ or even أَكُـــونَ. Notice how the second and third forms are almost 1.5 and 2 times as wide as the first. This property of cursive Arabic script, if made possible in computer typesetting, would make the flexibility of justification in Arabic text more than the previous method of using spaces, which is unacceptable in the first place. We aim to produce a final output comparable to the works of calligraphers, see Fig. 1.5 for a scan of a page from the Holy Qur'an written by a calligrapher [1]. Note how different forms of the letters are used, with sometimes extended or compressed versions.

This thesis is among other works within AlQalam project. A project that was initiated 3 years ago with the intention of typesetting Qur'anic text, hence it is our goal to produce output of similar quality to a calligrapher written Qur'an, which happens to be the majority of Qur'ans in print today. In other words we are targeting the maximum achievable quality and typesetting flexibility. In the past two decades, the approach to typesetting Arabic on computers was through

وَإِذْ قُلْنَا ٱدْخُلُوا۟ هَـٰذِهِ ٱلْقَرْيَةَ فَكُلُوا۟ مِنْهَا حَيْثُ شِئْتُمْ رَغَدًا
وَٱدْخُلُوا۟ ٱلْبَابَ سُجَّدًا وَقُولُوا۟ حِطَّةٌ نَّغْفِرْ لَكُمْ خَطَـٰيَـٰكُمْ ۚ
وَسَنَزِيدُ ٱلْمُحْسِنِينَ ﴿٥٨﴾ فَبَدَّلَ ٱلَّذِينَ ظَلَمُوا۟ قَوْلًا
غَيْرَ ٱلَّذِى قِيلَ لَهُمْ فَأَنزَلْنَا عَلَى ٱلَّذِينَ ظَلَمُوا۟ رِجْزًا مِّنَ
ٱلسَّمَآءِ بِمَا كَانُوا۟ يَفْسُقُونَ ﴿٥٩﴾ ۞ وَإِذِ ٱسْتَسْقَىٰ مُوسَىٰ
لِقَوْمِهِۦ فَقُلْنَا ٱضْرِب بِّعَصَاكَ ٱلْحَجَرَ ۖ فَٱنفَجَرَتْ مِنْهُ
ٱثْنَتَا عَشْرَةَ عَيْنًا ۖ قَدْ عَلِمَ كُلُّ أُنَاسٍ مَّشْرَبَهُمْ ۖ كُلُوا۟
وَٱشْرَبُوا۟ مِن رِّزْقِ ٱللَّهِ وَلَا تَعْثَوْا۟ فِى ٱلْأَرْضِ مُفْسِدِينَ ﴿٦٠﴾

Figure 1.5: Part of the Holy Qur'an written by a calligrapher. [1]

simplifying the Arabic script for easier modeling. Haralambous [6] discusses the typographical simplifications that have been applied to the Arabic script in these past years. Nowadays, with the existence of computers with more powerful processing ability, it makes sense to try and model the Arabic writing more accurately. The work done in AlQalam project until now, was in line justification and in adding Qur'anic marks to text. Fig. 1.6 shows a sample output from AlQalam.

1.4 Motivation

A final note before moving on to the next chapters is to indicate the importance of such a thesis and its possible applications. First of all, the Arabic language is the language of the Qur'an, and due to this fact we see that it deserves to be given the attention and more care in order to adapt computers to correctly view it and present it.

Secondly, we hope that such a work will help in preserving even on a very small scale the Arabic language and how we write it. We see that the current situation of typing Arabic on computers is very deficient and with time people will get used to its output and forget how the language should have be written. So called modern Arabic fonts, are slowly departing from the original letterforms, and are slowly transforming into non-cursive letters, to the extent that some of these fonts use shapes derived from the Latin alphabet.

The heritage of Islamic arts is largely based on calligraphy. This heritage was highly valued in the past, and is in fact decreasing. It is imperative that we try and preserve the beautiful art of calligraphy by bringing Arabic desktop publishing a little closer to the quality and beauty of the past.

Corporations worldwide are trying to build systems that mimics closely the way Arabic is written, and have made advancements, but still it is not satisfactory as we shall see in the fore coming chapters. Microsoft, Adobe, Tradigital, and many more, none of them is based in the Arab world and we see that it is time

مَدَنيَّة سُورَةُ الرَّعْدِ ٤٣ ءَايَة

بِسْمِ ٱللَّهِ ٱلرَّحْمَـٰنِ ٱلرَّحِيمِ

الٓمٓر ۚ تِلْكَ ءَايَـٰتُ ٱلْكِتَـٰبِ ۗ وَٱلَّذِىٓ أُنزِلَ إِلَيْكَ مِن رَّبِّكَ ٱلْحَقُّ
وَلَـٰكِنَّ أَكْثَرَ ٱلنَّاسِ لَا يُؤْمِنُونَ ﴿١﴾ ٱللَّهُ ٱلَّذِى رَفَعَ ٱلسَّمَـٰوَٰتِ
بِغَيْرِ عَمَدٍ تَرَوْنَهَا ۖ ثُمَّ ٱسْتَوَىٰ عَلَى ٱلْعَرْشِ ۖ وَسَخَّرَ ٱلشَّمْسَ وَ
ٱلْقَمَرَ ۖ كُلٌّ يَجْرِى لِأَجَلٍ مُّسَمًّى ۚ يُدَبِّرُ ٱلْأَمْرَ يُفَصِّلُ ٱلْـَٔايَـٰتِ
لَعَلَّكُم بِلِقَآءِ رَبِّكُمْ تُوقِنُونَ ﴿٢﴾ وَهُوَ ٱلَّذِى مَدَّ ٱلْأَرْضَ وَجَعَلَ
فِيهَا رَوَٰسِىَ وَأَنْهَـٰرًا ۖ وَمِن كُلِّ ٱلثَّمَرَٰتِ جَعَلَ فِيهَا زَوْجَيْنِ ٱثْنَيْنِ ۖ
يُغْشِى ٱلَّيْلَ ٱلنَّهَارَ ۚ إِنَّ فِى ذَٰلِكَ لَـَٔايَـٰتٍ لِّقَوْمٍ يَتَفَكَّرُونَ ﴿٣﴾ وَفِى
ٱلْأَرْضِ قِطَعٌ مُّتَجَـٰوِرَٰتٌ وَجَنَّـٰتٌ مِّنْ أَعْنَـٰبٍ وَزَرْعٌ وَنَخِيلٌ
صِنْوَانٌ وَغَيْرُ صِنْوَانٍ يُسْقَىٰ بِمَآءٍ وَٰحِدٍ وَنُفَضِّلُ بَعْضَهَا عَلَىٰ
بَعْضٍ فِى ٱلْأُكُلِ ۚ إِنَّ فِى ذَٰلِكَ لَـَٔايَـٰتٍ لِّقَوْمٍ يَعْقِلُونَ ﴿٤﴾ ۞ وَإِن
تَعْجَبْ فَعَجَبٌ قَوْلُهُمْ أَءِذَا كُنَّا تُرَٰبًا أَءِنَّا لَفِى خَلْقٍ جَدِيدٍ ۗ
أُو۟لَـٰٓئِكَ ٱلَّذِينَ كَفَرُوا۟ بِرَبِّهِمْ ۖ وَأُو۟لَـٰٓئِكَ ٱلْأَغْلَـٰلُ فِىٓ أَعْنَاقِهِمْ ۖ
وَأُو۟لَـٰٓئِكَ أَصْحَـٰبُ ٱلنَّارِ ۖ هُمْ فِيهَا خَـٰلِدُونَ ﴿٥﴾

ربع الحزب ٢٥

Figure 1.6: Sample output of AlQalam system.

that we, the Arabs, take a step to help the computer write Arabic the way it should be.

Most of these companies also sell their products as closed systems, and at high costs. Our work in AlQalam project is aimed at producing an open source system that will give all people the ability to write Arabic on computers in the best possible output for free.

Finally, with a base of more than 500 million people using the Arabic alphabet in their everyday writing, and 27 countries having Arabic as an official language, it is evident that it is worthwhile to develop such a system, and that its consumer base is huge.

Chapter 2

Digital Typography

In order to realize our goal of producing calligraphic output on computers, we will first need to understand how computers are now used to create digital documents, and how word processors and typesetting systems are used in the process, and what do these names mean in the first place. In this chapter we will define what typography is, and then we will review the currently available computer tools that enable us to apply digital typography.

Digital typography is defined by Richard Rubinstein [7] as "the technology of using computers for the design, preparation, and presentation of documents, in which the graphical elements are organized, positioned, and themselves created under digital control." This is the digital part of it, but typography itself is an old and conservative field, and so is calligraphy, which is the precise aim of this thesis. Hundreds of years of development have gone into creating our methods of presenting written information, and for developing the Arabic script as we write it today.

The means of producing printed material has changed several times in the past decades, with the newest method being electronic publishing. The problem for building electronic publishing systems is retaining high quality while replacing old printing methods. Understanding what is good about traditional printing and calligraphy is surprisingly hard. Typography is a design specialty, an art, and a

truly multidisciplinary field. George Bernard Shaw once stated in an article on typography that “well-printed books are just as scarce as well-written ones”.

Rubinstein [7] also mentions that the goal of digital typography lies within three questions: “Will books be easy to read or tiring? Will they communicate efficiently, or slow the reader and provoke errors? Will they give pleasure to their owners?”. The judgment of a good composed text is often a matter of human perception and thus cannot be defined very precisely by some formulae, but still it will be noticed by the trained eye if a certain text is well written or not, and even the untrained eye will feel at ease reading from a well composed text. The main objective of text composition is to place letterforms together to form words and lines. While the objective sounds rather simple and straightforward, its implementation is not trivial at all. Composition is a difficult task, because it has to obey many rules. The most important requirement of good composition is the harmony of the typeset text. It is often said that typography should be invisible, i.e. well typeset text should not call attention to itself and distract the reader from the textual contents.

In the next section we will review currently existing typesetting systems, then we will follow it by explaining some of the basic rules of typography which will be important in understanding the thesis, and finally we will talk about specific requirements of Arabic typesetting in the light of these rules.

2.1 Typesetting Systems

Typesetting systems are computer programs that simply apply the definition of digital typography just mentioned in this chapter. They can be classified into many categories, most commonly is the word processors category, which includes programs that are interactive, and show the users the expected output as they are typing. Microsoft Word[1] is the most common word processor in use today. Its

[1]We are here talking about the version included in the 2003 Microsoft Office suite.

advantages are simplicity of use, good user interface, and interactivity — always previewing the current state of the document at hand.

Yet it has disadvantages, and probably too many which ironically spur from the above mentioned advantages. Being very interactive results in the users being forced to take care of their documents' layout manually, and thus wasting a lot of time on the looks instead of the message they are trying to communicate through the document. Add to that, the fact that most users are not aware of the rules of good typography and layout, and therefore often produce unprofessional output. It is also not easy to maintain a consistent formatting style in a very large document using MS Word. In order to be interactive and show results as the user is typing, Word skips advanced typographic rules and issues for the sake of reducing complexity. But in the end, it is mainly targeted at personal and small office use.

Hence the existence of more professional layout programs like Adobe FrameMaker, QuarkXPress, Corel Ventura, and LaTeX among others that target the more professional printing and publishing industries. Of these systems, LaTeX is the only open source and free program. LaTeX is a typesetting system based on TeX, a very powerful typesetting engine developed in the late 70's by Professor Donald E. Knuth [8] at Stanford University. The main objective of TeX was to get high quality typesetting, and special care was paid to very fine details of composing that had been secrets of typographers. TeX was among the first computer typesetting systems with support for advanced typography like ligatures, kerning, control of spacing around punctuations and hyphenation, we have already mentioned some of them in the previous chapter, but we will describe them in detail in the next section.

One of TeX's most powerful capabilities is the ability to typeset complex mathematical expressions rather easily than most other tools. This ability is what makes it a very favourable technical writing tool in academia.

Another very important advantage of TEX over other typesetting systems is that it is open source and free, hence developing new systems can be based on it, and indeed over the years tens of systems were developed on top of it. In this thesis, we chose TEX as the platform for creating our target system. It provides unsurpassed flexibility to try new typesetting approaches, and over the years many researchers tried to build systems that typeset Arabic with more quality using TEX and came up with systems like Omega ([9] and [10]), ArabTEX ([11] and [12]), RyDArab [13] and Arabi. In the next section we will explain briefly basic typographic rules, and will mention how TEX handles them.

2.2 Basic Typographic Concepts

Now we will discuss basic concepts that were used by typographers and printers since long years ago and how these concepts are now being digitized (more details can be found in [7], [8], and [14]), especially through TEX on which our work is based. This will clarify what we are trying to do for typesetting Arabic in analogy to what was performed for Latin script typesetting.

Boxes and Glues

In the early years of printing punch-cutter created small metal boxes, each box with a letter carved on it. The printers aligned those boxed letters beside each other to create the lines. This is what is still done today in the most advanced typesetting systems, but the only difference is that the computer is the one who aligned them together and we can instruct it to produce sometime better results than when done by humans.

TEX uses a model for typesetting, based on three elements, namely box, glue and penalty. This model is used by TEX to perform line-breaking, the process by which a line is broken and the rest of the words are pushed to the next line.

A box represents some material that should be typeset. A box can be a char-

acter or a sequence of characters from a font, but it can also be a much more complex object, e.g. a mathematical formula or a line or a composition of other boxes. The contents of a box, however, are not important for the line-breaking algorithm. The only relevant information about a box for the line breaking process is its width. We can simply think of a box as a word (or a segment of a hyphenated word), where its width is the sum of the widths of the associated letters.

Glue represents a blank space whose width can vary. Glue has natural size, standing for the normal width of the glue. Apart from natural size, glue may have stretchability, which is the maximum extra amount that the glue can be increased by. Similarly, glue shrinkability stands for the maximum amount that the glue can be decreased by. The natural size, stretchability and shrinkability together are called the glue specification. In the context of line breaking, the inter-word space is represented as glue with its specification depending on the selected font. Fig. 2.1 shows how boxes have different dimensions depending on what they contain and how glues can be visualized as the stretchable substance between the boxes.

A penalty is the cost we pay or the reward we gain for breaking a line at a certain place. Using penalties allows the control of line breaking in a flexible way by specifying appropriate values of the penalty at the desired places. For example, when we wish to limit breaking at a certain place, we can insert a high penalty.

Boxes and glue

Figure 2.1: The boxes containing the letters of the sentence “Boxes and glue” have different dimensions, and the glue shown in black is the virtual substance filling the space between them.

An infinite penalty means prohibition of breaking at its location. Similarly, a negative penalty indicates a desirable place for breaking and an infinite negative penalty forces breaking at its location.

Line-breaking and Justification

TEX uses the box/glue/penalty model to decide on where to break the line using a cost function based on the line's width and the values of the widths of boxes and glues [8]. Fig. 2.2 shows an example how this calculation is performed. The process of line-breaking is strongly related to justification, which is based on the results of the line-breaking algorithms.

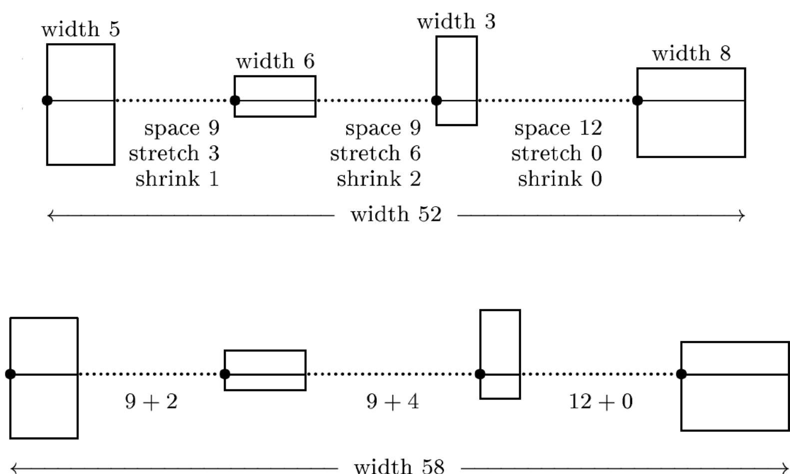

Figure 2.2: A numerical example of justification using the box/glue model. The above figure shows 4 boxes, of different widths, and the specifications of the glue between them. Below it shows the same boxes and glues but adjusted to fill up a different line width.

The performance of TEX in line-breaking and justification surpasses that of simple word processing programs like MS Word, since TEX applies its algorithms on paragraphs as a whole not just on single lines. The concept of considering a paragraph as a whole while breaking it into individual lines, i.e. the total-fit algorithm, in fact has become the fundamental algorithm for paragraph formatting by computers.

Line	Badness
In olden times when wishing still helped one, there lived a king	−.727
whose daughters were all beautiful; and the youngest was so	.821
beautiful that the sun itself, which has seen so much, was aston-	−.455
ished whenever it shone in her face. Close by the king's castle lay	−.870
a great dark forest, and under an old lime-tree in the forest was	−.208
a well, and when the day was very warm, the king's child went	.000
out into the forest and sat down by the side of the cool fountain;	−.577
and when she was bored she took a golden ball, and threw it up	−.231
on high and caught it; and this ball was her favorite plaything.	.000

Line	Badness
In olden times when wishing still helped one, there lived a king	−.727
whose daughters were all beautiful; and the youngest was so	.821
beautiful that the sun itself, which has seen so much, was aston-	−.455
ished whenever it shone in her face. Close by the king's castle	.278
lay a great dark forest, and under an old lime-tree in the forest	.000
was a well, and when the day was very warm, the king's child	.237
went out into the forest and sat down by the side of the cool	.462
fountain; and when she was bored she took a golden ball, and	.343
threw it up on high and caught it; and this ball was her favorite	−.320
plaything.	.004

Figure 2.3: Two different line-breaking and justification algorithms applied on the same paragraph. Small marks between the letters of a word indicate possible break points. The numbers on the right indicate the cost function (badness) of each line, −ve meaning the line is compressed, and +ve meaning the line is stretched beyond normal length. TEX enables the minimization of the cost function on the whole paragraph to be applied. [15]

The tools available in Latin text to provide the flexibility required for line justification are hyphenation, inter-word and inter-letter spacing. Hyphenation is the most flexible, since it enables breaking long words into halves. Almost all typesetting systems also use inter-word spacing for justification, but it does have limitations, beyond which a line will look very crowded or very sparse in space. Inter-letter spacing introduces micro-adjustments to the spaces between letters in order to assist in justification. However, it is rarely used, since changing this inter-letter spacing affects legibility greatly [7]. Fig. 2.3 shows a paragraph typeset using two different line-breaking algorithms utilizing hyphenation and inter-word spacing.

fire flower
fire flower

Figure 2.4: Common ligatures in English, the 'fi' and 'fl' pair, shown in the top line without ligature replacement, and below with correct ligatures used.

Table
Table

Figure 2.5: Kerning example in the letter pair 'Ta', on the above line without kerning, and below with the correct kerning applied by bringing the two letters closer together.

Ligatures and Kerning

Ligatures are combinations of letters that by rule should be typeset differently when following each other. Ligatures in the English script are very few, the most common ones are shown in Fig. 2.4.

Kerning is the process of adjusting spacing between letter pairs, like 'Ta' and 'AV'. Fig. 2.5 shows an example on the effect of kerning.

Optical Considerations

Human perception is a very important aspect affecting typography rules, and should always be put into consideration. Fig. 2.6 on the following page shows the enlargement of a group of letters, indicating how some letter like the letter 'o' do actually extend below the baseline of other letters in order to appear on the same line when used in small size. This is just one example of how optical issues should be considered in designing type faces.

Reading text on computer screens is in general different from reading from printed text on paper. The chief cause is the low resolution of computer screens as compared to resolution of printed text. We need at least two or three times as

tion tion

Figure 2.6: Human perception requires that the letter 'o' extend below the baseline in order to appear of same size as other lowercase letters.

more pixels on screen to begin to approach the quality of the printed page [16]. The most notable disadvantage of such low resolution is that reading on screens is hard on the eyes. It is then important to figure out ways to present text that is both aesthetically pleasing and easy to read and comprehend on computer screens. One such way that is already being applied on most screens is the use of pixels of different shades of gray together with the black and white pixels used to display characters, a process known as anti-aliasing. So work has to be done in order to make both paper and screen viewing as similar as possible.

Color

Color (or greyness) is the term used for the overall darkness of a block of text. Color depends on four factors: interline spacing, inter-word spacing, inter-letter spacing and the font design. The font and inter-line spacing used in a book (or a document) is given by the layout design and therefore is predetermined for composing. On the contrary, inter-word (and sometimes inter-letter) spacing depends on how line breaking is done and how extra spaces are distributed into inter-word spaces. The greyness of a paragraph therefore strongly depends on the uniformity of inter-word spacing. Even greyness of typeset text is difficult to achieve, as to attain even inter-word spacing is always a challenging demand. Moreover, the inter-word spaces in all lines must be optically rather than mechanically equal, which cannot be done without the assistance of human vision yet, see Fig. 2.7 on the next page. The inter-word space can appear wider or narrower than its mechanical width, depending on the shapes of the adjacent characters on either

rear view   letters alike

Figure 2.7: Note the spacing between the words on the left and those on the right. Although spacing between 'rear' and 'view' is mechanically equal (i.e. the boxes containing the spaces are of same width) to that between 'letters' and 'alike', it is optically unequal. This is due to the shapes of different letters, and how they occupy space in their corresponding boxes.

side. For example, a space after a period will look wider than it is, because there is a lot of white space in the shape of the period.

For this reason, sometimes no inter-word space is needed between a period and some capital letters like T, V, etc. All current systems try to make inter-word spaces to be mechanically equal, but there is also some effort to compensate the effect of such optical illusions. Some high quality fonts can contain kerning data with respect to the space character for certain characters that need adjustment when ending up next to an inter-word space. Common cases of such characters are comma, period, quotes, etc. However, there is no known typesetting system that tries to compensate inter-word space according to the shape of characters on both sides. So far this can be only achieved by hand composition.

Rivers

Rivers are vertical alignments of inter-word spaces in typeset text. This phenomenon disturbs the uniformity of typeset text and distracts the readers attention from text contents. Usually rivers appear in typeset text with loose inter-word spacing, as larger inter-word spaces can easily form vertical white strips in typeset text. When text is typeset with tight and uniform inter-word spacing, there is less chance for rivers to appear. Improvement of inter-word spacing therefore helps to avoid an appearance of rivers. However, even in the case that inter-word spaces are tight and uniform, rivers can arise, as there is no guarantee that inter-word spaces will never be aligned vertically. So far, none of automated

As quick as thought is an old mode of expression, used to convey an idea of the greatest rapidity: but no one, until lately, ever dreamed that a thought could be sent hundreds of miles in a few seconds; and that a person standing in London might hold a conversation with another in Edinburgh, put questions and receive answers, just as if they were seated together in one room, ir.stead of being three hundred miles apart.

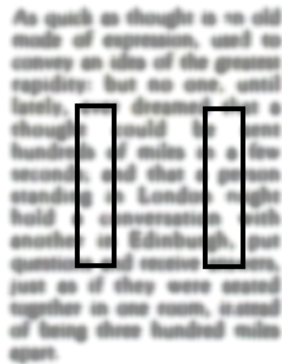

Figure 2.8: Examples of rivers in a paragraph. Looking at the paragraph from a distance makes rivers much more distinguishable. The original paragraph is shown on left, while a more 'hazy' form of the same paragraph, makes rivers clearer to see.

composing systems provides the ability to prevent rivers, as their appearance can be only easily detected by human eyes. This is still a challenge for implementers of typesetting systems to provide a river detector for automated composing. A problem similar to rivers is the alignment of identical or similar words in consecutive lines. This effect is most disturbing at the margins of text, that is, when the first or the last word in consecutive lines are identical. Unlike rivers, this effect happens independently of inter-word spacing. Fig. 2.8 shows a paragraph with two rivers affecting its overall look.

Orphans and Widows

These exist due to bad page-breaking algorithms which does not take whole paragraphs into consideration and instead looks at single lines. As Fig. 2.9 on the following page indicates, an orphan is a line that is written at the bottom of the page, where it should have been pushed to the next page with other text it is related to, while a widow is the same situation but the other way round. Both affect the page layout negatively.

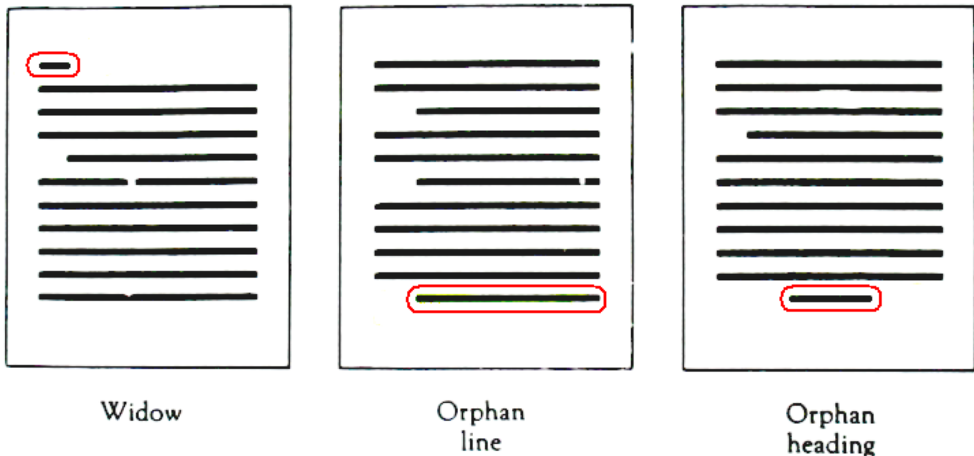

Figure 2.9: Examples of orphans and widows.

2.3 Arabic Typesetting Requirements

Now that typography rules in typesetting Latin texts are understood, we shall now see the corresponding rules in typesetting Arabic. First of all, as mentioned in the introduction chapter, the box/glue model should not be applied to Arabic, since it is cursive, and its letters do have a great deal of interaction between them. Similarly, the tools of applying justification are completely different. There is no such thing as hyphenation in most of the Arabic script languages, and it is not possible to use inter-letter spacing, since letters are already joined together.

Inter-word spacing should rarely be used, as a calligraphic rule, yet some available word processors use them extensively. Instead of these tools, typesetting flexibility in Arabic is possible with the use of ligatures, wider letter forms, *kashīdas*, and finally inter-word and even inter-letter can rarely be used. This thesis aims at providing these two tools in order to achieve better line-breaking, justification, orphans, widows, rivers and color control. The most prominent example on how typesetting Arabic can be a complex process is the Qur'anic text itself, where most chapters are written to occupy exactly 20 pages. This is only violated in few chapters of the Madinah Qur'an [1]. Other prints of the Qur'an achieved this rule for all chapters, except of course for the last chapter (chap-

عند كتابة اللغة العربية
لا يسمح بوضع
مسافات بين الكلمات!

عند كتابة اللغة العربية
لا يسمح بوضع
مسافات بين الكلمات!

عند كتابة اللغة العربية
لا يسمـــــــــح بوضـــع
مسافات بين الكلمات!

Figure 2.10: Arabic justification in MS Word 2003. The leftmost text is without justification, while the middle one uses spaces for justification, and the rightmost used straight kashida segments in the process. All three texts result in non-satisfactory results based on calligraphic rules.

ter 30), which includes lots of small *surahs*, hence many titles that take a large amount of space. This is not possible to achieve on any existing typesetting system. Fig. 2.10 shows the poor performance of the justification performed by MS Word 2003.

Other specific requirements in Arabic are for Ligatures which are unlike Latin, almost infinite and hence need special treatment. Kerning is exists in Arabic, and also in many more letter combinations than in Latin, for example the letter *wāw* followed by an *'alif*. These are the major differences between Arabic and Latin typographic rules. Now that the requirement for Arabic is apparently different, in the next chapter we will review current font technologies to discover if they can be used to provide the flexibility needed for typesetting.

Chapter 3

Font Technologies

All electronic publishing tools consist of two parts, a typesetting system that places letters and images on the page, and the font technology which supplies the letterforms to be used by the typesetter. In this chapter we do a quick review on existing font technologies, and then discuss two of them in detail: OpenType[1] and the METAFONT [17].

A font technology is a standardized file format that specifies font glyphs to be used in a typesetting system. This standard may include information beyond just the glyphs of a specific font, like which glyphs can be substituted instead of others, and which pairs need kerning and by how much and other similar information. Font development changed a lot from the 1980's. The first fonts were simply bitmaps of glyphs drawn pixel by pixel, and this worked just fine when the screen and printing resolutions were very primitive. A strong drawback of bitmap fonts is that scaling them to obtain different font sizes is not possible, and produces very bad results, hence glyphs for different sizes have to be drawn manually.

As output devices advanced with higher resolutions, came the need for a higher quality type of fonts, one that is called vector or outline fonts. This type of fonts has no maximum viewing resolution since they are defined by curves

[1]OpenType specification available at `http://www.microsoft.com/typography/otspec/`

that are reconstructed using vectors at any required resolution. Examples of such outline fonts are the Microsoft TrueType, Adobe Type-1 fonts, and the latest OpenType standard. There is a third category of font tool, that is known as algorithmic font design tools [7]. Typefaces are generated by programs that describe them mathematically, hence the name algorithmic. METAFONT is a kind of this category of tools. It enables the production of different font sizes in different resolutions but with bitmapped output. METAFONT was created by Donald Knuth in order to be used with the TEX he created.

3.1 OpenType

OpenType is currently the de facto font technology. It is replacing older font standards like TrueType and Type-1 fonts. It has a lot of features that support a very wide variety of languages and scripts. It is adopted by Microsoft and Adobe, and thus it is the most supported standard format. It has glyph positioning (GPOS) and glyph substitution (GSUB) tables which allow kerning and ligatures in Arabic. It also has other layout features that help connecting glyphs in cursive scripts like Arabic.

Being the most common font standard lead to the existence of many editors and tools that help design the outline of its glyphs. Designing the outline of a glyph is done by selecting points on the outline interactively using the mouse, like the glyph shown in fig. 3.1 on the next page showing a screenshot of a glyph in the editing screen of FontCreator 5.5. Some tools such as Microsoft's Visual OpenType Layout Tool[2] (VOLT), provide simple graphical interfaces for editing the GPOS and GSUB tables among other features. Fig. 3.2 on page 35 shows a words written using two TrueType fonts then using an OpenType, one of the best Arabic fonts currently existing, *Naskh* by Tradigital[3].

[2]Available for download at `http://www.microsoft.com/typography/developers/volt/`
[3]Company's web page at `http://www.tradigital.de/`

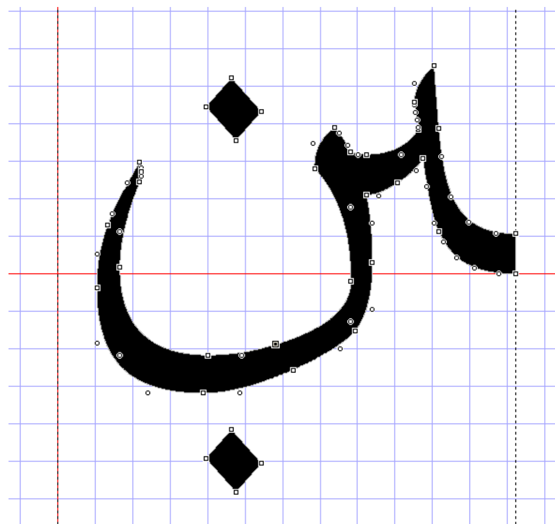

Figure 3.1: A screenshot from an outline font editor, FontCreator 5.5.

Despite the many features provided by OpenType, including those dedicated to the Arabic script they are not sufficient. The whole concept of letter boxes connecting together via other boxes of elongation strokes is not suitable for maximum quality Arabic typesetting as the following examples illustrate.

Outline fonts can be used to draw glyphs of characters in different forms very well when these glyphs are isolated. When connecting glyphs together, the junctions rarely fit perfectly, since adjacent letter glyphs usually have different stroke directions at the starting and ending points. Although this imperfection may not be visible for small font sizes, it is quite clear in large font sizes. An extreme example is the use of these fonts to write large banners or signs. Even for small fonts, when it is required to add a *tatwil*, a ready made *kashīda* of specific length is used to connect the glyphs together. Of course, such a *kashīda* will not match perfectly with all the different glyphs. Fig. 3.3 on page 36 shows examples of problems at junctions. Two of those problems are due to using *kashīdas*. The junction after the *kāf* has no *kashīda* but it shows the stroke width being non-uniform. It would be possible of course to edit the outline of these two glyphs in order to match, but it will certainly make them mismatch with yet other glyphs.

Another limitation is the use of already stored glyphs for different ligatures,

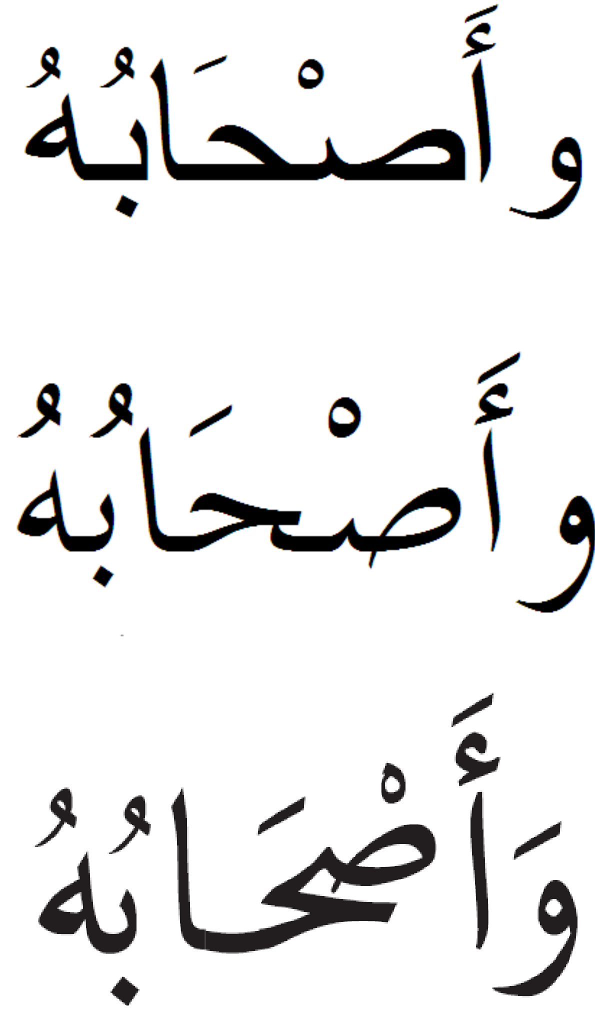

Figure 3.2: A word written with 3 fonts, from top to bottom: TrueType Simplified Arabic, TrueType Traditional Arabic and OpenType Naskh by Tradigital. Note how the OpenType version uses ligatures and natural looking curves in an attempt to satisfy the cursive nature of Arabic.

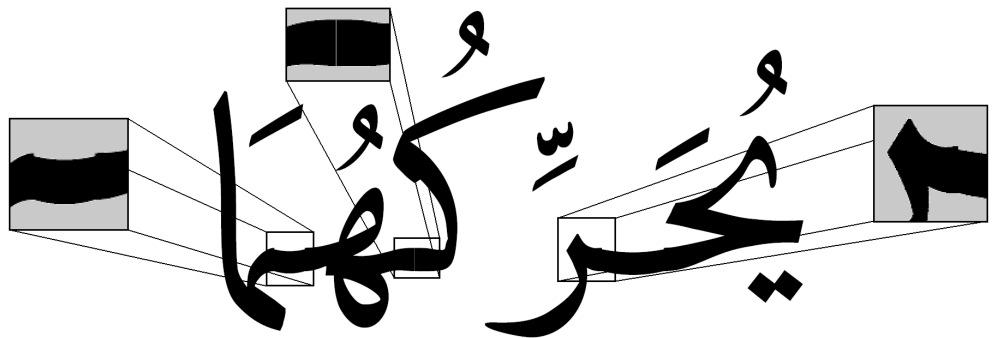

Figure 3.3: An example word typeset with an OpenType font indicating problems with junctions between glyphs. Word obtained from sample products by Tradigital, a sister company of DecoType, with typesetting system developed by Thomas Milo. Note that this font represents the highest quality in OpenType Arabic fonts we have seen, and if viewed at its standard original size, without enlargement, these imperfections are not visible.

but since the number of possible ligatures is very large, only a selected portion can be made ready. If we would like to model the Arabic script more accurately then each word should be considered a ligature and hence we would have an infinite number of ligatures, which is impossible to be made ready for use. The Unicode standard has numerous glyphs called presentation forms, each representing a unique ligature form. The current Unicode version 5 [18], includes around 500 codes for different glyphs, just to describe different forms of only 28 Arabic characters, not including additional codes for short vowels, diacritics, and Qur'anic marks. Fig. 3.4 on the following page shows some of the complex ligatures allocated codes. The provision of a code point for each ligature is an inefficient and non-scalable design in our opinion. As indicated earlier, each Arabic word can in fact be considered one ligature, following this method of code allocation to cover all ligature forms, which would be theoretically infinite in number, would take up all the remaining free codes. The process of selecting a ligature should be left to the typesetting application instead.

A final feature, that is more feasible to implement in METAFONT than in OpenType is the capability to program and embed information regarding the scaling of glyphs to different sizes in the fonts themselves. This enables optical

FD53	FD63	FD73	FD83	FD93	FDA3	FDB3	FDC3
FD54	FD64	FD74	FD84	FD94	FDA4	FDB4	FDC4
FD55	FD65	FD75	FD85	FD95	FDA5	FDB5	FDC5
FD56	FD66	FD76	FD86	FD96	FDA6	FDB6	FDC6

Figure 3.4: Part of the character code tables indicating code allocation to complicated ligatures, combining up to 3 letters.

scaling to be achieved instead of linear scaling. The optical scaling is even more important when two strokes meet as in the medial form of the letter *ṣād* in the left-hand side of Fig. 3.5. At a small scale this stroke crossing produces a black blotch as when it is used in a word. Knuth [17] discussed this problem, and its solution in METAFONT by decreasing the thickness of the strokes as they intersect. This change of thickness makes the words at small sizes appear of uniform darkness, see the 'sad' in the right-hand side of Fig. 3.5 on the next page. This solution can be parameterized such that as the size decreases, the pen width at intersections decreases, thus giving a feel of uniform darkness at all sizes.

Last but not least, OpenType is a proprietary standard, where modifications and developments to the standard's specifications are not possible to all. Add to that the fact that most of fonts created using this technology are commercial and very expensive, and since our target is to develop a system for all to use, and that produces better quality, we chose to use METAFONT in our thesis research instead of OpenType.

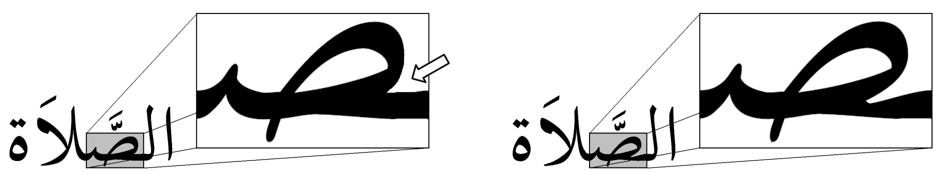

Figure 3.5: Optical scaling requires that stroke widths become thinner at intersections, in order to give an appearance of uniform blackness for a word at smaller scale. Left-hand figure shows a letter 'sad' in its medial form as it should appear correctly. When linearly scaled and used in a word, a black blotch appears at stroke intersections. The right-hand side, shows how the 'sad' should be changed in order to appear properly at smaller size.

3.2 Metafont

These limitations in new font technologies like OpenType led us back to METAFONT which existed in its current form since 1986. It is a font description language to specify how glyphs are drawn under varying conditions, not just drawing them in a special case. What new font technologies are trying to achieve nowadays was mostly feasible by METAFONT since it was created, but it is only due to its complexity that it is not widely used.

One of the most powerful advantages of METAFONT is the notion of pens, and strokes that are very similar to real strokes by real pens. And since this is a requirement for writing Arabic, it is very difficult to draw the strokes of letters with outline font tools as drawing the whole stroke with a pen, means that the width of the stroke at each point along a path should have a specific width. Another powerful advantage is that it enables the mathematical description of the glyphs enabling the creation of parameterized fonts; fonts that are dynamic and can be controlled through parameters.

The Computer Modern (CM) typeface family produced by Donald Knuth [19] using METAFONT is a main source of motivation for this thesis. It is one of the landmarks in producing parameterized fonts. Despite the differences between Latin and Arabic characters, it is possible to apply the same concepts used to design the CM fonts to Arabic ones.

Figure 3.6: Four different lowercase letter 'a' forms generated by a single description program. From left to right: roman, sans serif, typewriter, and bold.

The CM family is a marvelous work of art intermingled with engineering. Its design utilized the concept of meta-design to its maximum. In this thesis we demonstrate that METAFONT which was used to produce fonts like CM and AMS Euler can be utilized to create Arabic fonts of comparable quality and flexibility. The design of CM was such that each character or symbol had a program written to describe it. The font glyphs are defined by spline curves, but unlike current outline fonts, these curves are defined in a clear mathematical way, that can be parameterized allowing them flexibility.

The only aspect of this design which some may see as a drawback is its difficulty. Knuth described his work to produce parameterized CM fonts to be "much, much more difficult than [he] ever imagined" when he started to make them in 1977. He received help from several of the world's finest type designers, and his job, as stated by himself, was "to translate some of their wisdom into precise mathematical expressions". [19]

His final design of CM fonts, uses 62 parameters delivered to the programs describing the characters to produce 75 different standard fonts. These numbers clearly indicate the extent to which these fonts were meta-designed. Fig. 3.6 shows four of the lowercase letter 'a' generated by the CM family. These a's and many more result from a single description program.

We aim to produce Arabic fonts that are as meta-designed as CM. Of course the parameters would be very different, for example many of those used in CM fonts describe the serifs. In Arabic there are no serifs, but instead there are other

parameters to connect the glyphs together to form the ligatures.

Despite METAFONT's advantages and capabilities just mentioned, it has disadvantages. The most important is its lack of editing tools and simple graphical design interface, as with OpenType. Instead, METAFONT programs are written in a text file then are compiled to produce output that can be viewed using a DVI viewer. Another disadvantage is that it is difficult to learn and to use, i.e. it is not for the faint hearted or artists, but still it is not an engineer's job to design letter forms.

Or in other words as described by Knuth [17]: "A top-notch designer of typefaces needs to have an unusually good eye and a highly developed sensitivity to the nuances of shapes. A top-notch user of computer languages needs to have an unusual talent for abstract reasoning and a highly developed ability to express intuitive ideas in formal terms. Very few people have both of these unusual combinations of skills; hence the best products of METAFONT will probably be collaborative efforts between two people who complement each others abilities."

So in order to use METAFONT to produce good quality glyphs that are truly meta-designed it is required to have an artist together with an engineer, and this is what happened as Donald Knuth teamed with Hermann Zapf, a well known type designer in order to produce AMS Euler. Fig. 3.7 on the next page shows three different forms produced by Knuth's CM definition of the letter 'y' and Fig. 3.8 on page 42 shows the program used to meta-design the letter. It is obvious that such a program is very complicated.

Adding to the complexity is that when errors occur in METAFONT programs, it is very difficult to debug. Still, the unmatched abilities of the language attracted us to try and use it to its maximum to develop a truly dynamic and parameterized Arabic font. The rest of this thesis describes how we model Arabic pen strokes accurately, meta-design Arabic glyphs, and join letters together all using METAFONT.

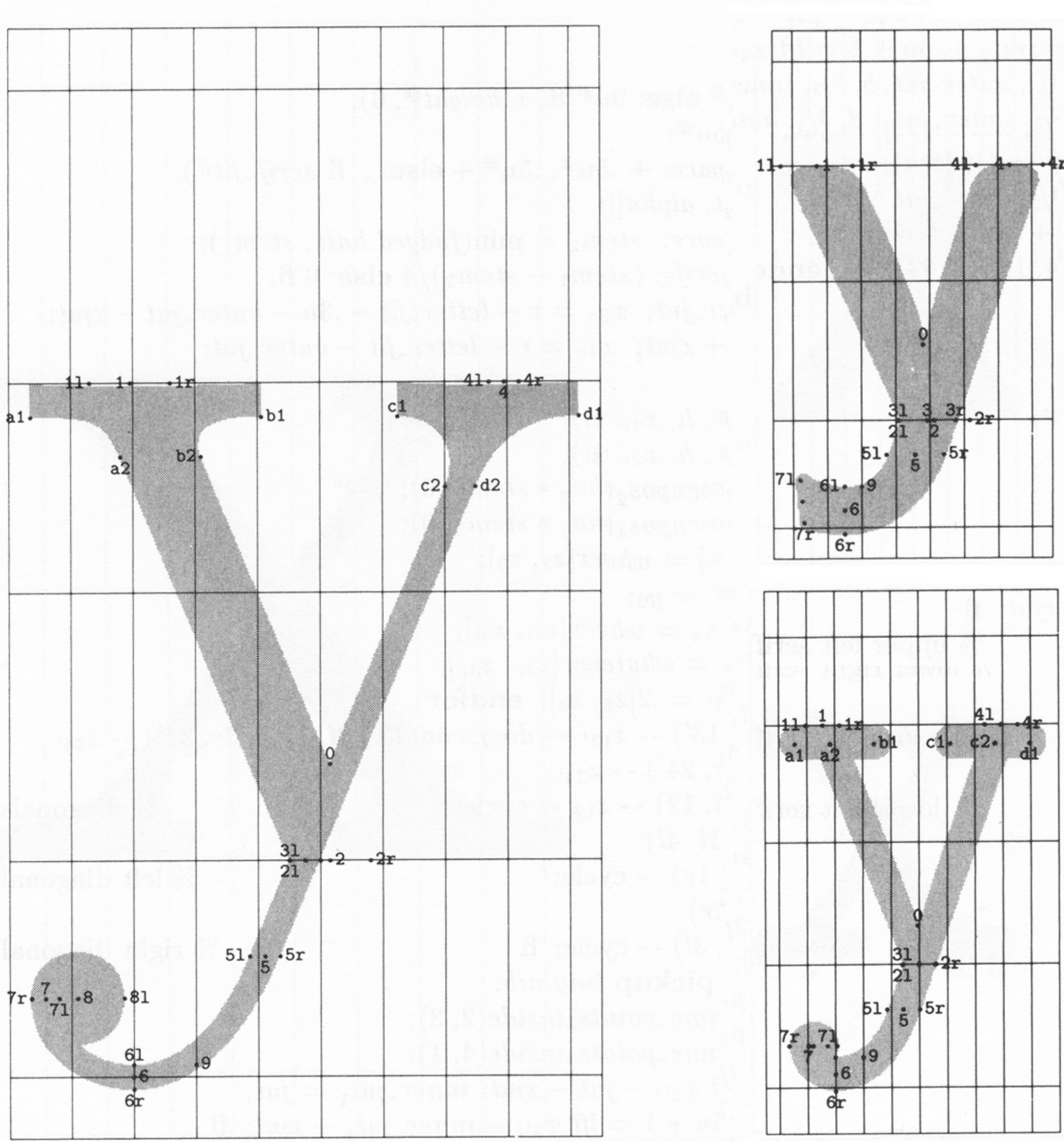

Figure 3.7: Three different forms of the letter ‘y’ as generated by one description program shown on the next page.

```
cmchar "The letter y";
beginchar("y", if serifs: 9.5u# else: 9u# fi, x_height#, desc_depth#);
italcorr x_height# * slant + .25u#;
adjust_fit(serif_fit# if monospace: + .5u#, .5u# + else: , fi serif_fit#);
numeric left_stem, right_stem, bot_stem, bot_vair, outer_jut;
left_stem = fudged.stem - stem_corr;
right_stem = fudged.hair if hefty: -2stem_corr fi;
bot_stem = fudged.hair if hefty: -8stem_corr fi;
bot_vair = Vround(if serifs: vair else: .5[vair, bot_stem] fi);
outer_jut = .75jut;
x1l = w-x4r = l+letter_fit+outer_jut+.25u; y1 = y4r = h; y2 = y3 = 0; x2l = x3l;
numeric alpha, alpha[]; x9 = 3u; y9 = bot_vair - d - oo;
alpha1 = diag_ratio(2, bot_stem, y1 - y3, x4r - x1l - apex_corr);
alpha2 = diag_ratio(1, bot_stem, y1 - y9, x4r - x9);
if alpha1 < alpha2: x2l - x1l = x4r - x3r + apex_corr; alpha = alpha1;
else: alpha = alpha2; z3l = whatever[z9, z4r - (alpha * bot_stem, 0)]; fi
penpos3(alpha * bot_stem, 0); penpos4(alpha * right_stem, 0);
alpha3 = (y1 ++ (x2l - x1l))/y1;
penpos1(alpha3 * left_stem, 0); penpos2(alpha3 * left_stem, 0);
z0 = whatever[z1r, z2r] = z4l + whatever * (z3r - z4r);
if y0 > notch_cut: y0 := notch_cut;
    fill z0 + .5right{up} ... {z4r - z3r}diag_end(0, 4l, 1, 1, 4r, 3r)
        -- z3r -- z2l -- diag_end(2l, 1l, 1, 1, 1r, 2r){z2 - z1}
        ... {down}z0 + .5left -- cycle;              % left and right diagonals
else: fill z0 -- diag_end(0, 4l, 1, 1, 4r, 3r) -- z3r -- z2l
        -- diag_end(2l, 1l, 1, 1, 1r, 0) -- cycle; fi  % left and right diagonals
penpos5(alpha * bot_stem, 0); z5r = whatever[z3r, z4r]; y5 - .5vair = -.5d;
if serifs: numeric light_bulb; light_bulb = hround 7/8[hair, flare]; clearpen;
  penpos6(vair, -90); penpos7(hair, -180); penpos8(light_bulb, -180);
  x6 = 2u; y6r = -d - oo; y8 - .5light_bulb = -.85d; x8r = hround .35u;
  fill stroke z3e --- z5e ... {left}z6e; bulb(6, 7, 8);          % arc and bulb
  numeric inner_jut; pickup tiny.nib;
  prime_points_inside(1, 2); prime_points_inside(4, 3);
  if rt x1'r + jut + .5u + 1 <= lft x4'l - jut: inner_jut = jut;
  else: rt x1'r + inner_jut + .5u + 1 = lft x4'l - inner_jut; fi
  dish_serif(1', 2, a, 1/3, outer_jut, b, 1/2, inner_jut);         % left serif
  dish_serif(4', 3, c, .6, inner_jut, d, 1/2, outer_jut)(dark);    % right serif
else: penpos6(bot_vair, -90); x6 = 2.5u; y6r = -d - oo;
  fill stroke z3e --- z5e ... {left}z6e;                           % arc
  pickup fine.nib; pos6'(bot_vair, -90); z6' = z6;
  pos7(2/3[bot_vair, flare], -85);
  lft x7l = hround u; bot y7r = vround -.96d - oo; y7l := good.y y7l;
  filldraw stroke term.e(6', 7, left, 1, 4); fi             % arc and terminal
penlabels(0, 1, 2, 3, 4, 5, 6, 7, 8, 9); endchar;
```

Figure 3.8: The program used to describe the letter 'y' in CM producing the three different forms of the letter shown on the previous page.

Chapter 4

Pen Modeling

In order to model the writing of a calligrapher, it is common sense to consider modeling the pen first. This chapter deals with pen modeling, both using the plain METAFONT macros then with enhanced macros developed during the work of this thesis. In most books used for teaching the art and technique of Arabic calligraphy, the pens used are the first topic to be explained. The pens used in writing Arabic are of different types and were previously discussed in previous works towards digital Arabic typography as that of Benatia [3]. In order to produce fonts that look very much like a calligrapher's output, we need to model three factors; the pen nib, pen stroke [2], and ink spread. We will discuss the first two in the next sections, and will mention more on the ink spread effect as a future research topic in Section 7.2.1 on page 113.

4.1 Pen Nib Outline

The first factor we need to model is the shape of the pen nib. There are differences between pens used to write different Arabic styles. *Naskh* for example, uses a pen that is cut at an inclined angle of approximately 40 degrees. The pen used can be made of wood, and when sharpened carefully by cutting on one side, a nib is created having a shape that is in the midst of a rectangle and a thin section of

Figure 4.1: A photo of an Arabic pen, notice the shape of the nib: curved on the right side and straight on the left.

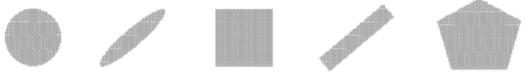

Figure 4.2: Various pen nib shapes in METAFONT. From left to right: circle, inclined ellipse, square, inclined rectangle, and a octagon created by `makepen` macro.

a circle, see Fig. 4.1 for an Arabic pen's nib.

One of METAFONT's powerful features, as discussed previously is having the notion of the pen already available. It also provides predefined pen nibs; `pencircle` and `pensquare` that provide circular and squared pen nib shapes respectively. Each of these can be scaled, with different scaling factors in each direction, to form a multitude of elliptical or rectangular shapes. The nibs can be further transformed by rotation around a specific axis. This is of extreme importance since Arabic is written using the pen nib inclined at an angle. Fig. 4.2 shows a circular nib of `pencircle`, a squared one of `pensquare`, and a transformed version of each.

Other pen nibs defined in METAFONT are the `penrazor` and `nullpen`. The `penrazor` is a rectangular pen having a finite length and zero width, as if we are using a razor to draw, hence the name. The `nullpen` is a theoretical pen with

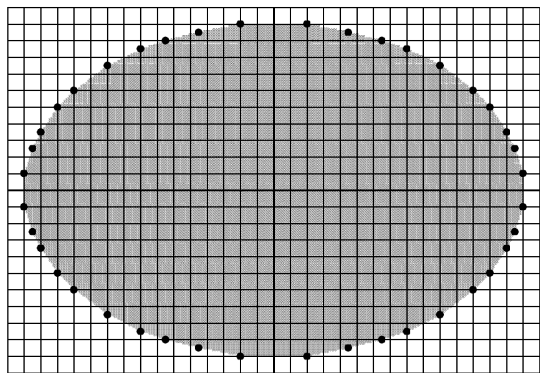

Figure 4.3: A circular pen digitized after scaling. The segments connecting the outline points are straight lines, not curves. It happens that this nib is a 40 sided polygon (the number of sides depends on the transformation applied on pencircle [17]). Note that the actual pens used to draw in METAFONT are all converted to polygons like this one before used in drawing.

zero cross-sectional area, this pen is used internally by METAFONT to outline contours that need to be filled. These two types of pens, are of little use when modeling the Arabic pen.

The most important pen macro in METAFONT is the `makepen`. This macro enables the user to define any other pen nib shape required, as long as it is a convex polygonal shape. The polygonal shape is define by the a number of coordinates connected by straight lines. The rightmost pen nib in Fig. 4.2 on the preceding page shows a polygonal nib produced by `makepen`. The pen nib outline defined this way can not have curves in it. However, even `pencircle` and `pensquare` when used after transformation, they are first digitized before being used to trace a path, and this leads to faster computation of the resulting stroke. When a circular nib is scaled for example, it is digitized after scaling as in Fig. 4.3.

Taking into consideration the shape of the Arabic pen's nib, we chose after long experimentation, to define the Arabic pen to be used in our font by:

```
pickup makepen((−16, −2) -- (16, −2) -- (15, 2) -- (0, 3)
                                        -- (−15, 2) -- cycle);
```

As said previously `makepen` can only make polygons without curves, hence our choice which is approximation to the actual nibs. It is of course possible to modify the outline by defining more points. Adding more points to define pen nib outline more accurately will increase computations and slow drawing down. Fig. 4.4 shows the selected nib.

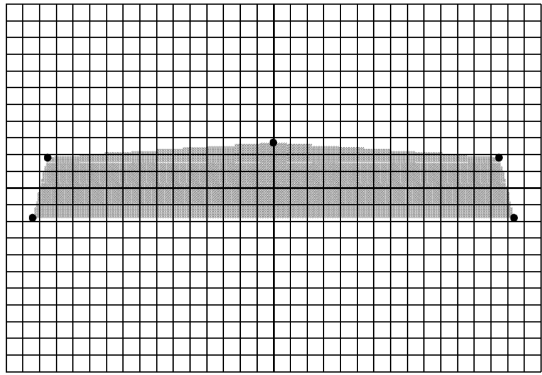

Figure 4.4: Arabic pen nib outline defined.

The more accurate the nib, the better looking the final glyph produced. It was found that using our custom defined nib, produces better letters than circular or squared nibs, even with transformations. Fig. 4.5 on the next page shows the letter *'alif* drawn using three different pens. Observing the endings of the main stroke, shows that the custom nib (on the left), does not produce as many sharp edges as the rectangular nib in the middle, nor does it produce excess roundness in edges like the rightmost elliptical nib. The *hamza* on top of the letter is drawn with a pen of half length, same pen nib outline but scaled by half. Although It can be said that the stem of the custom pen is best, the same can't be said for

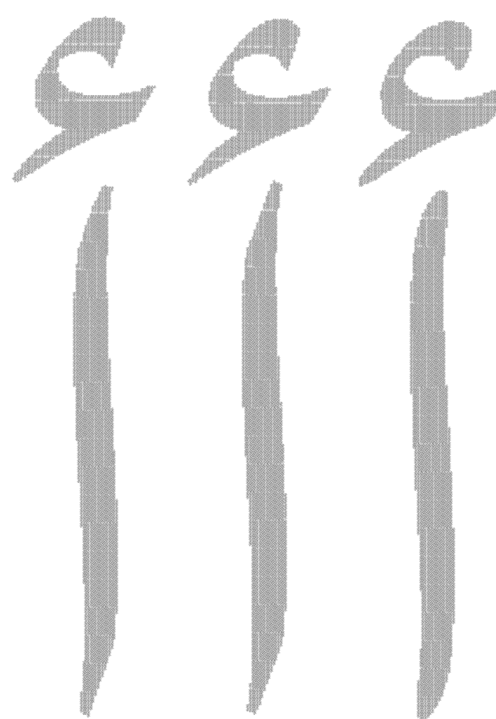

Figure 4.5: The letter alif drawn with 3 different pen nibs. From the left: custom, rectangular, and elliptical pens. Notice the edges of the stem of the letter.

the *hamza*. This is due to the fact that the smaller pens usually have smaller length to width ratio, because if we keep the same length-to-width ratio, then the nib is very thin and hence easy to break. Also the ink spreading effect in reality is more observable for smaller pens, since the ink spreads the same amount on the same paper, but with respect to the smaller diacritic dimensions, it is more noticeable.

4.2 Pen Stroke

After defining the pen nib to be used in drawing, comes the path to trace and the pen's stroke following this path. In the next two sections we mention how METAFONT enables the description of paths and then we discuss methods used to model the pen's stroke on a given path.

4.2.1 Describing Curves

Describing a path, either straight or curved in METAFONT is a very simple mechanism, yet flexible and powerful. In order to define a curve we first define some points that lie on the path using Cartesian coordinates. Then we state the order in which this path goes through the points, and the direction and angle through each one. METAFONT then automatically evaluates the best smooth curve that satisfies the information given. The curves evaluated are mathematically represented using a special curve family called Bézier curves. These are 3rd degree polynomials represented as

$$z(t) = (1-t)^3 z_1 + 3(1-t)^2 t z_2 + 3(1-t) t^2 z_3 + t^3 z_4.$$

As the parameter t varies from 0 to 1, the curve is traced as we move from point z_1 to z_4. Although a Bézier curve is defined using 4 coordinated, it is possible to state only two of them, the starting and ending points, z_1 and z_4. The middle points, z_2 and z_3, are called the control points, and can be calculated from the direction of the curve at z_1 and z_4. For example Fig. 4.6 on the following page shows the curve drawn using the statement:

```
draw z1{dir 70} .. z4{dir -30};
```

The two dots '..' is a primitive in METAFONT that evaluates the Bézier curve, and the `dir` is another primitive that specifies the direction by which the path passes through the points, measured from the x-axis.

Figure 4.6: A Bézier curve. Points 2 and 3 are the control points of the curve, and are calculated by METAFONT from the tangential direction of the curve at the ending points.

There are other options provided by METAFONT to gain more control of the resulting curve. One of those is to specify the `tension` of the curve. The default `tension` of a curve is '1'. If, for example we increase the `tension`, the control points approach closer to the starting and ending points, with the direction at these points constant. A `tension` of infinity will result in a straight line between z_1 and z_4.

4.2.2 Drawing with Pens

Now that a path is defined, the next step is to pick up a pen and use it to draw along the path. As we discussed earlier, many Arabic letters require the pen to change inclination angle while moving across the path in order to achieve different stroke thicknesses. This requirement was not addressed fully in METAFONT since Latin letters can be drawn with a fixed inclination of the pen nib. The next section mentions built-in macros of METAFONT that try to produce real pen-like strokes and discusses their limitations. Then we will explain improvements and modifications to these macros, to better model the stroke of a rotating pen.

Plain METAFONT Drawing Macros

Plain METAFONT is a file that contains a collection of standard METAFONT macros analogous to the standard header files of the C language. Plain METAFONT has 3 main drawing macros that help simulate how pens are used to draw specific paths. These macros are `fill`, `draw`, and `penstroke`.

The `fill` macro is used to fill the inside of a closed contour. It does not model a pen, but it will be demonstrated later how it can be used within other macros to do so. The `draw` macro uses a pen with a defined fixed inclination to trace the path. The algorithm used to define the points that the pen covers with ink was developed by John Hobby [20]. Fig. 4.7 shows a path drawn using the `draw` macro but with different pens.

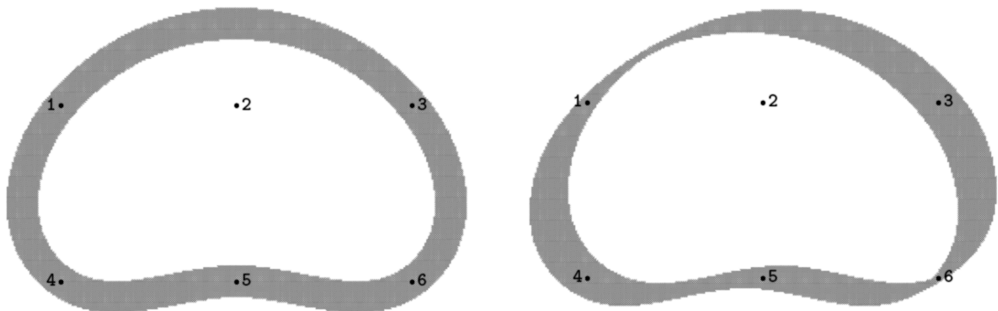

Figure 4.7: One path traced by two different pens [17]. The left-hand path was drawn using a circular pen, and the right-hand path used a elliptical pen inclined at 40 degrees from the x-axis. Both strokes are drawn with the statement:
`draw z5..z4..z1..z3..z6..cycle;`

The limitation of the `draw` macro is that it draws paths in the glyph with a fixed pen inclination. But in Arabic calligraphy, this is not the case. Many letters require that the calligrapher keep on changing the inclination of the pen as he traces the path to draw. An example is the letter *nūn* in its extended form, see Fig. 4.8 on the following page. Its lower segment should be thick at the middle and thin at the tips, and this requires pen rotation while tracing.

The third drawing macro in METAFONT is the `penstroke`. This macro is an

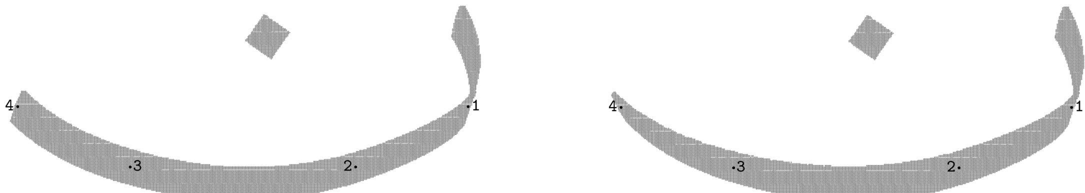

Figure 4.8: The letter noon written with a rotating pen on right, and on left with a fixed inclination pen. Note the segment between points 1 and 4. In the right glyph, the pen inclination measured from the x-axis changes from 70° to 120° as it moves from point 1 to 4 (85° at point 2 and 110° at point 3). In the glyph on the left, the same segment is drawn with a pen of fixed inclination of 70°.

approximation to a razor pen, used to trace a path while rotating. It is used together with another macro called `penpos`, which specifies the inclination angle and the length of the razor pen at every point. An example output of `penstroke` and `penpos` is the stroke in Fig. 4.9 on the next page, which is a result of the following piece of code being executed:

$penpos_1(1.2pt, 30);$
$penpos_2(1.0pt, 45);$
$penpos_3(0.8pt, 90);$
penstroke $z_{1e} \mathrel{..} z_{2e}\{\text{dir } 90\} \mathrel{..} \{\text{dir } 90\}z_{3e};$

Note that the METAFONT code shown above is a formatted version of the actual code, and this is done for better legibility. Whenever METAFONT code is listed in the thesis, different types of reserved words are typeset either in bold or italic and coordinate points have their numbers subscripted. Actually the above piece of code without formatting looks like this:

```
penpos1(1.2pt,30);
penpos2(1.0pt,45);
penpos3(0.8pt,90);
penstroke z1e..z2e{dir 90}..{dir 90}z3e;
```

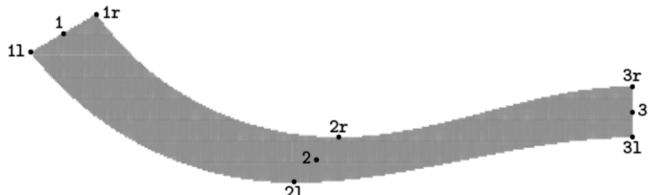

Figure 4.9: A path drawn using `penstroke` macro [17]. Note how the pen inclination changes as it moves across the path, resulting in different path thickness at different parts, an effect that is difficult to model with outline fonts.

Now, back to our discussion. The reason `penstroke` macro uses a razor pen with zero-width instead of a polygonal pen, is that it makes the underlying algorithms simpler. The algorithms used to execute the draw macro were complex enough with the pen not rotating, adding this variable will complicate calculations way further. It was not that much of a problem since the Latin glyphs can be drawn and described perfectly with only `draw` and `penstroke`, but for Arabic these two macros are not sufficient.

Problems with Plain METAFONT Drawing Macros

Due to the fact that the `penstroke` macro uses only a razor pen, three problems jump to the surface when it is used to draw Arabic letters. Although it did solve the problem of pen rotation three problems arise. Most notably, due to the pen used being a razor pen with zero width, some unwanted effects appear as shown in Fig. 4.10 on the following page when we try to draw the letter *bā'*. The upper figure shows the pen inclination at each location, with the pen width kept constant. The lower figure is the same with the pen orientations removed. Close observation of the resulting glyph shows two problems directly resulting from the razor pen used that lend the result unsatisfactory. One problem, is the left tip of the letter being too thin, indicating that the pen used has no width. The second flaw is at the bottom of the rightmost tooth of the letter intersecting with the

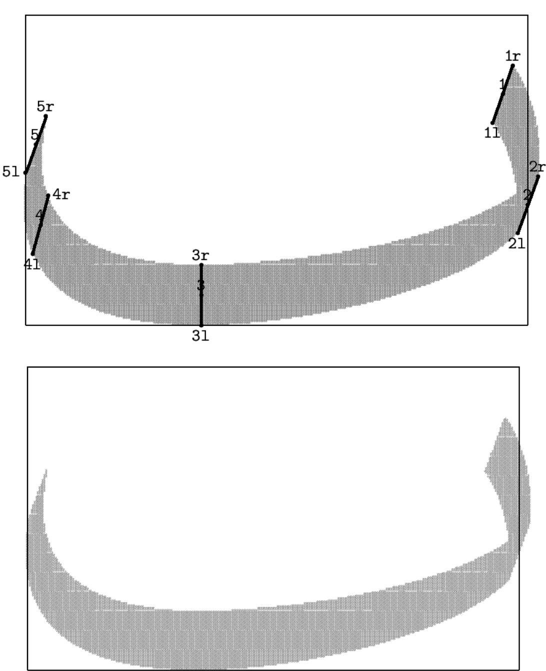

Figure 4.10: The first problem with `penstroke` – razor pen

base of the letter. This intersection is thinner than usual due to the same reason related to the pen. The code used to draw the letter *bā'* using `penstroke` is shown in Listing 4.1:

$bot\ z_3 = (0.35w, 0);$
$z_2 = (w, y_3 + 0.18w);$
$z_1 = (0.95w, y_3 + 0.4w);$
$lft\ z_4 = (0, y_3 + 0.14w);$
$z_5 = (0.02w, y_3 + 0.3w);$
$penpos_1(1.8dy, 70);$
$penpos_2(1.8dy, 70);$
$penpos_3(1.8dy, 90);$
$penpos_4(1.8dy, 75);$
$penpos_5(1.8dy, 70);$
penstroke $z_{1e}\{\text{dir } -55\} \mathinner{..} z_{2e}\{\text{dir } -95\};$
penstroke $z_{2e}\{\text{dir } -135\} \mathinner{..} \text{tension } 1.3 \mathinner{..} z_{3e}\{left\} \mathinner{..} z_{4e} \mathinner{..} z_{5e}\{\text{dir } 85\};$

Listing 4.1: METAFONT code for drawing the letter baa' using `penstroke`.

Yet, this is not the only issue with `penstroke`, and not even the most prominent. There are two more problems with this macro. These problems are due to the way `penstroke` is defined in the plain METAFONT file. First, when `penpos` is used at a coordinate, METAFONT calculates the position of the left and right ends of the razor pen at each coordinate. It then forms 2 paths, right and left ones, connects them at the endpoints with straight lines, then fills the resulting contour. In fact the macro expands to:

fill $path_{_l}$ -- reverse $path_{_r}$ -- cycle;

Where `path_.l` is the path passing through all the left points, and similarly for the right path.

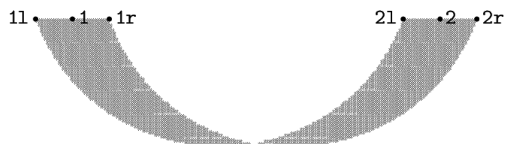

Figure 4.11: The second problem with `penstroke` – figure-8 shape.

This internal description of the macro causes two problems. The first is when we try to draw a shape like that in Fig. 4.11. In this figure both paths intersect, resulting in the contour dividing into two, and sometimes more regions. METAFONT does not have the capability to fill such complex regions that overlap themselves, and hence produces errors. To draw such a shape, a modification was done in our work by detecting the cross points of the paths, then filling each region separately. Such crossings occur frequently when drawing Arabic glyphs.

The definition of `penstroke`, with both left and right paths evaluated independently, results in the macro being a bad approximation of even a razor pen.

Being evaluated independently means that at some points, the distance between both paths may vary in a way that can not result from a fixed length pen. It is not always easily perceptible, but in some cases when there are sharp bends in a path or large amounts of pen rotation, the resulting stroke becomes a very bad approximation of a razor pen as in Fig. 4.12. In drawing Arabic glyphs, such large rotation in pen inclination rarely occurs, but this extreme example shows that `penstroke` is not an accurate model of a razor pen. Fig. 4.11 on the previous page also shows the same problem as a significant chunk of the stroke is missing at the middle of the path.

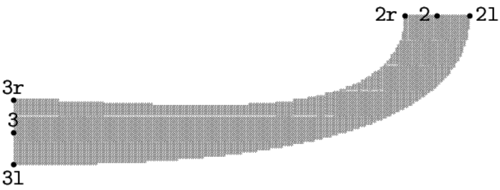

Figure 4.12: The third problem with `penstroke` – bad pen approximation.

Although the `draw` and `penstroke` are good attempts to simulate pens in action, they do not fulfill the needs of Arabic pens. One does not allow pen rotation, while the other uses a pen with zero thickness. It is obvious that both macros are not sufficient, and we need the best of both worlds, of `draw` and `penstroke`; a polygonal pen that traces a path while rotating. The next section discusses in detail a very important part of the work done in this thesis in order to correctly model the Arabic pen strokes. These solutions to the problem use the previously mentioned plain METAFONT macros in various ways.

Enhancements to plain METAFONT drawing macros

In this section we describe four proposed methods, all aiming to provide better modeling of the pen nib tracing a path while rotating. The order in which these methods are presented is of ascending quality. With the last being the most

accurate drawing method. The first two methods make use of the `penstroke` macro. But before we discuss them we will briefly mention how the errors arising from `penstroke`'s left and right paths crossing, discussed in the previous section is solved.

The error as previously mentioned is due to METAFONT not being able to fill a non-simple contour that crosses itself. In order to solve this, we proposed a solution by finding the cross points and then dividing the contour into segments, and filling each one separately. Since we do not know the number of crossings beforehand we do a `forever` loop until there are no more cross points. The original and modified `penstroke` codes are shown in Listing 4.2. The `intersectiontimes` macro used in this code is a plain METAFONT macro that returns the time of the location on the path. Points on any path in METAFONT are defined in time. If a path goes from point 1 to point 2, then time 0 is point 1, time 1 is point 2 and time 0.2 is a point on the path between both points but closer to point 1 and so on. The `intersectiontimes` when returning the times of intersection on right and left paths, these timed points may not have the exact coordinates in the x-y plane, hence the use of the `eps` constant, which is a very small value, that will ensure that the resulting contours have no micro loops at the crossing points. Fig. 4.13 shows an example of a stroke drawn by a pen that rotates 180° from point 1 to 2 and then 180° more from point 2 to 3, hence rotating 360° in total. Such a stroke would result in an error if the plain `penstroke` is used.

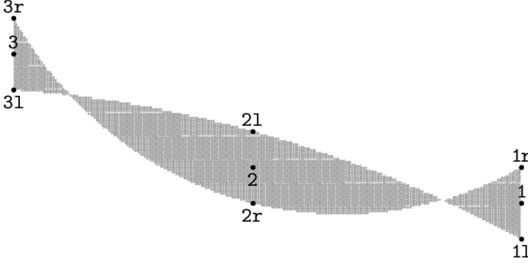

Figure 4.13: The error resulting from left and right paths crossing is resolved. Even in the case of multiple crossings.

```
% Original penstroke
def penstroke text t =
  forsuffixes e = l, r: path_e := t;  endfor
  if cycle path_l: cyclestroke_
  else: fill path_l -- reverse path_r -- cycle fi enddef;

% Modified penstroke
def penstroke text t =
  forsuffixes e = l, r: path_e := t;  endfor
  forever:
    numeric v, u;
    (v, u) = path_l intersectiontimes path_r;
    if v ≠ −1:
      fill subpath(0, v − eps) of path_l
                 -- subpath(u − eps, 0) of path_r -- cycle;
      path_l := subpath(v + eps, infinity) of path_l;
      path_r := subpath(u + eps, infinity) of path_r;
    else:
      fill path_l -- reverse path_r -- cycle;  fi
  exitif v = −1;
  endfor;
enddef;
```

Listing 4.2: METAFONT code of both the original `penstroke` and its modified version.

Now, we will explain each of the four proposed drawing macros. The `penstroke` used is the modified version just explained.

- First Solution (`filldraw stroke`):

This technique is used a lot in Knuth's definition of CM character glyphs. It fixes the problem of `penstroke` having zero width at certain points of a contour. Instead of just filling the `penstroke` contour, with the `fill` macro, another METAFONT macro, `filldraw` is used to fill the contour then trace its outline with a small circular pen nib, thus adding thickness to very thin segments of the "virtual pen stroke". The reason we say it is *virtual*, is because of Knuth's definition of a glyph like the 'e' keeps the pen rotating in a way such that the left and right

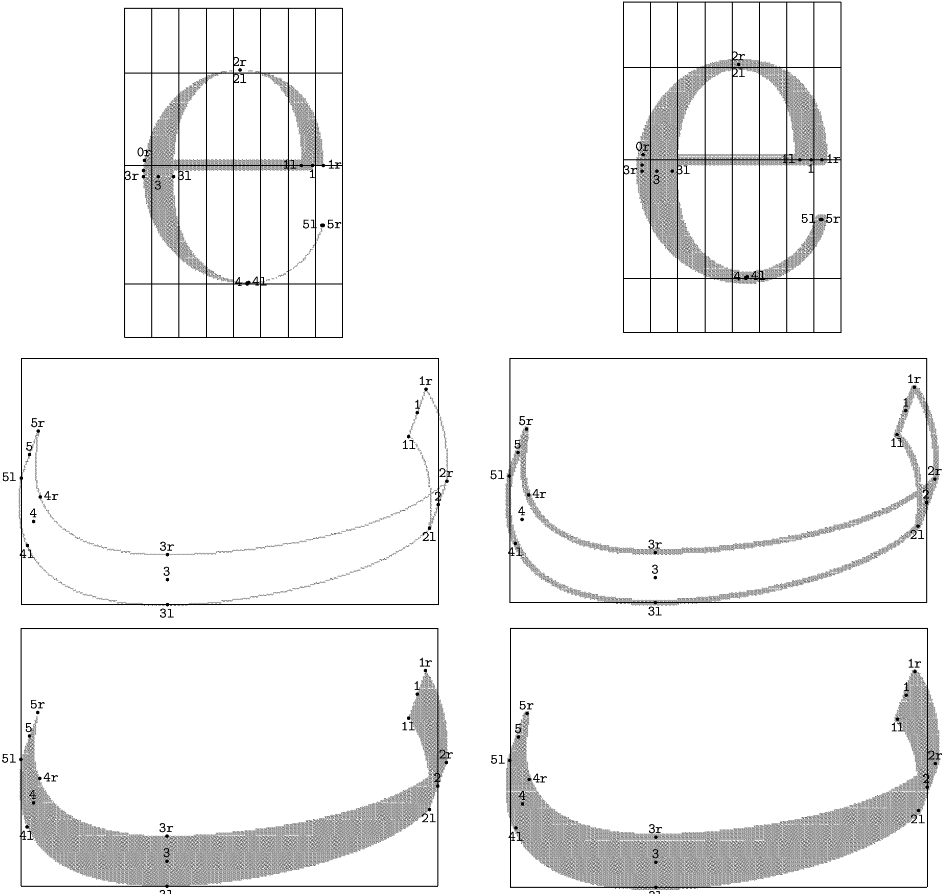

Figure 4.14: First solution - `filldraw`. Letter ‘e’ is shown on the top right as it is used in the CM fonts, and on the left how it would look like if drawn using `penstroke` instead of `filldraw stroke`. The top pair of letter baa’ shows the skeleton of the letter, i.e. the closed contour. The left contour is drawn with a very fine pen nib, while on the right with a thicker nib. The pair of baa’ at the bottom show the same contours after filling.

paths of `penstroke` do not cross, and this is certainly different from what a person would do while drawing the ‘e’, hence not a real pen stroke. The `stroke` macro is provided in the CM base file, and it merely defines the closed contour created by `penstroke` but without filling it. Fig. 4.14 shows the letter ‘e’ and $b\bar{a}$’ with `penstroke` and then with `filldraw stroke`. Note the thickness effect and how Knuth used this method to give letter tips round edges. Letter $b\bar{a}$’ is shown with its contour and after filling with `penstroke` on left and with `filldraw stroke` on right.

- Second Solution (`astroke`):

Another solution to the problem of zero-width pen, is to model the pen nib with multiple `penstroke`'s, one for each side of the pen. For a rectangular pen nib, a macro was defined which essentially breaks into 4 `penstroke`'s, each to model the area covered by a side of the rectangle. `penpos` macro also had to be modified in order to evaluate the 4 corner points of the pen nib (l, r, n, and m) instead of only 2 (left and right). Fig. 4.15 shows in the top figure 2 of the 4 `penstroke`'s resulting from the pen nib shown. The pen nib is enlarged for better understanding. The bottom figure shows all 4 `penstroke`'s but with reasonable pen nib dimensions.

Note that it is also possible to produce same output with only 2 `penstroke`'s (l-r) and (n-m) together with 2 nib dots at start and end, since the need for the smaller sides is usually limited to the tips, since it is rare that a pen used to write Arabic is moved in the direction of the smaller side. The modified code for `penpos` is shown in Listing 4.3.

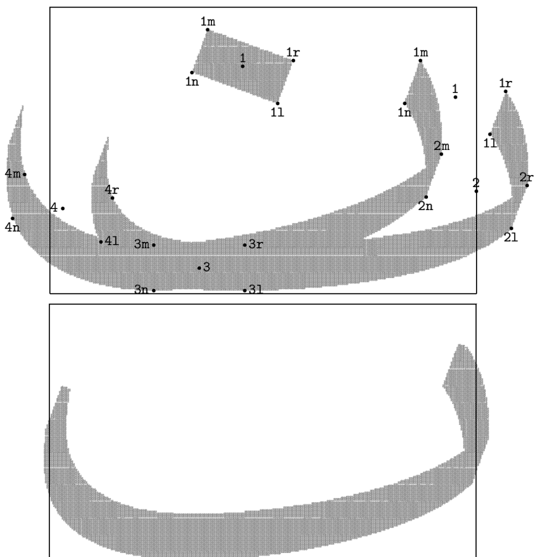

Figure 4.15: Second solution - `astroke`. `penpos` defines 4 points for each coordinate pair, named l, r, n, and m, see dot at top. `astroke` macro then produces 4 `penstroke`'s, top figure shows 2 of these sides (l-r) and (n-m).

```
% Original penstroke
vardef penpos@#(expr b, d) =                          % b is the width of the pen
  (x@#r − x@#l, y@#r − y@#l) = (b, 0) rotated d;      % d is the inclination
  x@# = .5(x@#l + x@#r); y@# = .5(y@#l + y@#r)        % of the pen
enddef;                                               % (x,y) is between
                                                      % left and right pts.

% Modified penstroke
vardef penpos@#(expr b, d) =
  (x@#r − x@#l, y@#r − y@#l) = (b, 0) rotated d;
  transform t[];
  t2 = identity shifted (((py/2), 0) rotated (d + 90));
  (x@#, y@#) = (.5(x@#l + x@#r), .5(y@#l + y@#r)) transformed t2;
  t3 = identity shifted (((py/2), 0) rotated (d − 90));
  (x@#m − x@#n, y@#m − y@#n) = (b, 0) rotated d;
  (x@#, y@#) = (.5(x@#n + x@#m), .5(y@#n + y@#m)) transformed t3;
enddef;
```

Listing 4.3: METAFONT code showing the modification to `penpos` macro.

These two previous modifications, although are improvements to the existing macros, they still have a drawback. That is they do make use of `penstroke`, which is as said earlier, just an approximation to a razor pen. When the rotation of the pen and the bends of the curve are limited in a segment, this approximation is close to the actual pen stroke. However, to see that it is merely an approximation, when tested with sharp bends or large pen rotation, it is possible to see that it is a bad approximation, like in Fig. 4.12 on page 56. The next two approaches do not use `penstroke`, in order to try and model the strokes more accurately.

- Third Solution (`qstroke`):

This macro solves the problem of `penstroke` being just an approximation of a razor pen traced path. The glyph is simply created by drawing footprints of the pen nib with different inclination angles at many consecutive locations along a path. The angle of the pen at each location is an interpolation of segment starting and ending inclination angles. At a given finite resolution, a finite number of pen

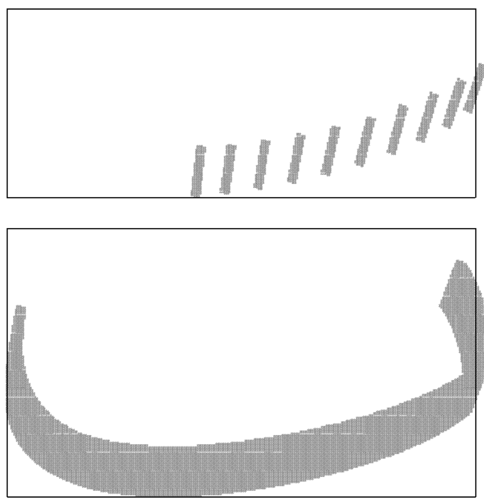

Figure 4.16: Third solution of stroke modeling - `qstroke`

dots will give the effect of a continuous pen stroke. As the distance of path increases, the more pen footprints are needed. Also due to the path times in METAFONT not being linearly distributed, more instances are needed, so is the case when the pen rotation is large in a specific segment. The code describing `qstroke` is shown in Listing 4.4 and a sample letter *bā'* drawn using the macro is shown in Fig. 4.16.

```
vardef qstroke(expr t, a, b, c, d) =
  numeric v, k;
  tilt0(a); tilt1(b); tilt2(c); tilt3(d);
  path qpath; qpath := t;
  for v = 0 step .005 until length qpath:
  k := floor v;
  int_ang := ang[k] + v * (ang[k + 1] − ang[k]);
  pickup pensquare xscaled px yscaled py rotated int_ang;
  drawdot(point v of qpath);
  endfor
enddef;
```

Listing 4.4: METAFONT code of the `qstroke` macro.

- Fourth Solution (`envelope`):

For high resolutions, the previous macro will need to draw even more dots so that the stroke is smooth. A refinement to this idea is to compute the exact envelope of the razor pen and then fill it. This macro moves along the path at small intervals, evaluating at each point two equidistant corresponding points on the left and right paths depending on the pen inclination. The output is a much accurate model of a razor pen than the `penstroke`, see Fig. 4.17 on the next page. Applying 4 of this new stroke as in the `astroke` solution produces the most accurate glyph. The code describing the macro is shown in Listing 4.5.

```
vardef envelope(expr t, a, b, c, d) =
  numeric v, k, j;  j = 0.01;
  tilt0(a);  tilt1(b);  tilt2(c);  tilt3(d);
  path path_l, path_r, path_med;
  path_med := t;
  path_l = (point 0 of path_med) shifted ((px/2, 0) rotated (ang0));
  path_r = (point 0 of path_med) shifted ((px/2, 0) rotated (ang0 - 180));
  for v = 0 step j until (length path_med):
    k := floor v;
    int_ang := ang[k] + (v - k) * (ang[k + 1] - ang[k]);
    pickup pencircle;
    path_l := path_l{direction v of path_med}
          .. {direction(v + j) of path_med}(point v of path_med)
                  shifted ((px/2, 0) rotated (int_ang));
    path_r := path_r{direction v of path_med}
          .. {direction(v + j) of path_med}(point v of path_med)
                  shifted ((px/2, 0) rotated (int_ang - 180));
  endfor
  fill path_l -- reverse path_r -- cycle;
enddef;
```

Listing 4.5: METAFONT code of the `envelope` macro.

It is evident of course that more accurate models require more calculations and hence more computing resources. For nominal resolutions, the `qstroke` macro will produce a final output as good as the more accurate but more complex macro,

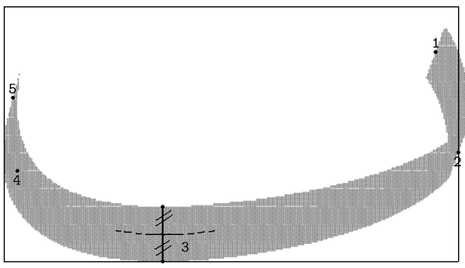

Figure 4.17: Fourth solution of stroke modeling - `envelope`

`envelope`, hence it is preferred.

Now that we have obtained a satisfactory model of the pen nib and the way it is used to draw strokes, we will explore in the next chapter how parameterized glyphs are designed.

Chapter 5

Arabic Meta-Design

In the previous chapter, we discussed the modeling of the Arabic pen nib and its use it to draw strokes like real calligraphers. In this chapter we move on to the next step, which is how does the calligrapher draw the different letters. The design of the letter forms must conform to traditional calligraphic rules. Three books [21], [22], and [23] were used as references along with face-to-face instructions from calligraphers. It is not our aim to evaluate the letterforms created in this thesis aesthetically against other commercial fonts available in the market. The main focus in this chapter is the underlying design using METAFONT.

The main approach used here to model the writings of calligraphers was to make the writing as dynamic as possible. This enables us to simulate the cursive nature of the Arabic script. In order to do that we present a font design that is parameterized in many ways. This parameterization comes in two forms; parameterization of coordinates and of curves.

Parameterization of coordinates means that points in the x-y plane are not given fixed locations. Any point location either depends on parameters or is related to another point in the plane sometimes also through parameters. Parameterization of curves on the other hand, means that either the tangential direction of a curve at some points or the tension on a curve segment is dependent on

parameters, and sometimes both together. This almost complete (since there are more advanced parameters like the pen moving speed when writing which affects the ink spread and hence the look of the stroke, however we did not include it in this thesis but discussed it in the Future Work chapter) parameterization of the glyphs will enable us to join letters better together, extend them easily, and optically scale the font.

This process of designing the glyphs is then better described as meta-designing, since we not only design the shape of the letter, but describe how it is to be drawn which is more difficult. Outline vector fonts like TrueType can be created by merely scanning a handwritten sample then digitizing it by converting it to vectors. The more meta-designed the glyph the more difficult and more time it takes.

5.1 Meta-Design Methodology

The Arabic alphabet, although consisting of 28 different letters, depends on only 17 different skeletons. The dots added above or below these skeletons differentiate one letter from another. For example the letters *ǧīm* and *ḫā'* have the same shape as the letter *ḥā'*, but *ǧīm* has a dot below, and *ḫā'* has a dot above. When we discuss primitives we only mention its use in the groups of letters having same skeleton, not individual letters, and this simplifies further our design. The placements of dots and other diacritics in our work is not parameterized, and their locations are fixed relative to the skeletons they belong to. We discuss their dynamic placement in the Future Work chapter.

Although the placement of dots will not be discussed in this thesis, we will need to mention them for another reason. In the process of designing Latin letters, there are some constants that are used as units of measurement, like the point, *em*, or *ex*. The point is around 1/72 of an inch, while *em* and *ex* are the corresponding widths of a lowercase ‘x’ and an uppercase ‘M’ correspondingly. In

Arabic however, the unit of measurement is different, and it is the dot or *nuqṭā*. The choice of the *nuqṭā* as such, was by Ibn-Muqla. He chose it in order to have some fixed measurements between different letter forms. For example, in the $Naskh$ style, the height of $'alif$ is 4 dots, and the width of an isolated *nūn* is 3 dots. The dot is that made by the pen used, i.e. it is not something fixed like the point used by Latin typographers. It is approximately equal to the diagonal of a dot drawn by the pen, and if the dot is a square then the *nuqṭā* is equal to the pen's width multiplied by the square root of 2, and in our work we take it as $1.4 \times$ *pen width* and in METAFONT programs it is simply abbreviated as `n`.

5.1.1 Point Selection

The first step in the process of meta-designing any primitive or letter, is to select the points through which the pen strokes pass. This is not an easy choice. The solution made when designing outline fonts, is usually to scan a handwritten letterform, digitize its outline, then make any necessary modifications. We did not adopt this approach because Arabic letters do not have fixed forms, and depend on the calligrapher's style, and since we are *meta*-designing, we are more concerned with how the letter is drawn not just its final resulting shape. Hence our choice of the points, should not be dependent on just one calligrapher, our output glyph instead of capturing the fine details of a specific calligrapher, should capture the general features of the letter. To do this we based our design on the works of more than one calligrapher at the same time.

Now that our point selection strategy is made clear, how many points should we select? The answer to this question is very difficult, because there is no definite answer. Very few points will make it very difficult to capture the important features of the letter, and will make curve descriptions very complex. While many points are a sign of a poor meta-design, and will be more like outline fonts designs, where tens of points are used to define just one glyph. Furthermore, the

more points we chose, the more difficult it is to parameterize the glyph. In our design, we look for the minimum possible choice of points that will enable us to correctly capture the main features of a letter and that will also facilitate the parameterization of the glyph.

The minimum number of points to describe any glyph is 2, since a stroke needs at least two points to exist, and a pen stroke at one point only will not be a stroke. The letter *'alif* for example can be designed with just two coordinate points but it could also be designed with 4 or even 5 points which makes path definitions easier, but adding parameters much more difficult. The letter *'alif* is shown in Fig. 5.1 with three different possibilities of point selection. The leftmost glyph has to have the angles at points 1 and 2 explicitly specified, where in the middle glyph, just connecting the points 1-3-4-2 with a Bézier curve can produce the same curve without explicitly specifying any angles. Theoretically speaking, we can specify the path using an infinite number of points, but the less points there are, the better the design, and the easier we can parameterize it. Adding more points that also lie on the same path can be done as in the rightmost glyph, but many points can be redundant like point 5 since the stroke is symmetric, and with it or without it we can produce the same path without explicitly specifying any angles or tension. Hence the decision of using such a point in this case would definitely be a bad design.

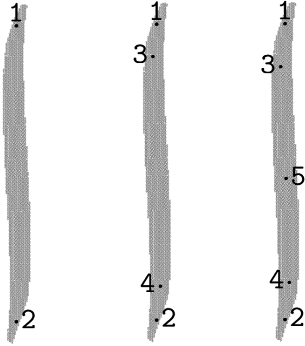

Figure 5.1: Choosing points to define the path of the letter 'alif.

This example, shows that the minimum points to choose for any stroke is 2, and their locations are at the endpoints of the stroke, these are the easiest points to select. Intermediate points are then chosen when curve parameters such as starting and ending directions and tension are not enough to define the curve as needed, capturing the important features. Hence more points are usually needed in stroke segments with sharp bends or at non-symmetric strokes, as shall be discussed in the primitives section, with baa' as an example.

5.1.2 Point Dependencies

After deciding on the number of points needed to describe a glyph and their positions, it is possible to define the curves, then draw the strokes needed. However, part of the meta-design parameterization, is the dynamic location of the coordinate points themselves, which requires that their locations be relative to each other. But then we faced the question, whether to relate all points in a glyph to one point or to many points? For example, if a stroke passes through points 1 to 5. Should all points be defined relative to 1 or to 5?

In our design, we model the direction of the stroke as it is drawn by the calligrapher, i.e. the stroke of the letter *'alif* is drawn from top to bottom, not the other way round. The numbering of points in our designs is based on this natural direction, so for the letter *'alif*, point 1 is the top point, and point 2 is the bottom one. However, a calligrapher chooses his starting point of the stroke depending on the location of the base line, he should write on. This means that point 1 is chosen relative to point 2, so we define 1 depending on 2. METAFONT is a declarative language, not an imperative one. So the two statements: `z1 = z2 + 3;` and `z2 = z1 - 3;` evaluate exactly the same, yet we try to make the dependencies propagate in the natural logical order, which then makes editing the glyph an easier job. We shall see more examples when explaining the different primitives designed.

In the coming pages, we will explain the most commonly used primitives, and will also mention some of their characteristics which are not mentioned explicitly in most calligraphic books. These characteristics were discovered during our meta-designing process, where we try to model the letters as closely as possible. When calligraphers describe their letters in books, their descriptions are approximate, and many detailed features of the letters are embedded implicitly in their curves as they learned them by practice. An example to illustrate this is the use of the *'alif* stroke in a letter like the *lām*. Most calligraphers describe the straight stroke in the *lām* as being identical to the *'alif*. But we discovered that this is not correct, and there are in fact differences between both strokes. (See the *'alif* primitive description in Section 5.2.2 for more details)

5.2 Primitives

Meta-design enables us to break the forms of glyph into smaller parts, or primitives. These primitives can be whole letters or just parts of letters that exist exactly as they are or with little modifications in other letters. With the use of primitives we can save design time, by reusing those primitives already meta-designed. Philippe Coueignoux [24] studied the utilization of primitives to form Latin letters. Fig. 5.2 shows an example of how he classified these primitives.

In Knuth's work on CM fonts [19] primitives were not explicitly defined, as black boxes then reused. This is due to the fact that it was already difficult to add parameters to each letter as a whole, and making use of primitives would mean much more parameterization. For him, the use of primitives was cost effective only in small and limited flexibility shapes like serifs and arcs, for which he wrote subroutines. The difference between his work work and ours, is that he parameterized letters to get a very large variety of outputs, but we parameterize primitives to make the letters more flexible and better connected, not to produce different fonts. In this thesis, only the *Naskh* writing style is of concern.

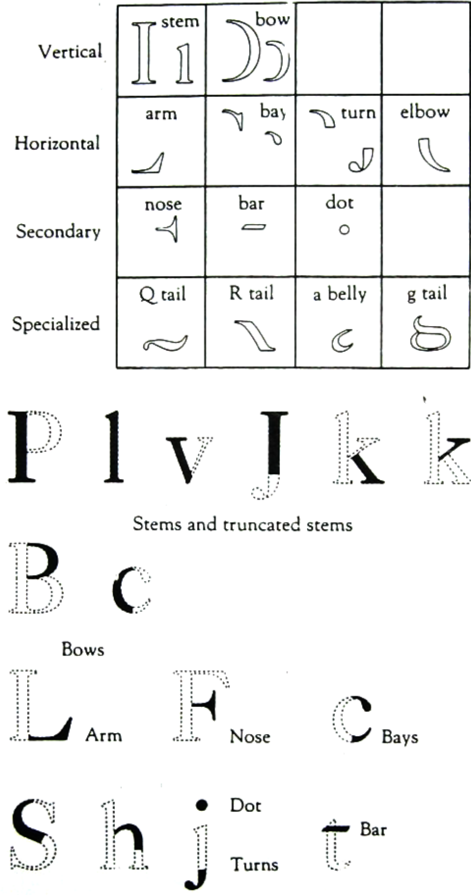

Figure 5.2: Usage of primitives in roman letters. [7]

In this section we explain how we meta-designed some of the most important, used, and dynamic primitives in the Arabic letters. We classified them into 3 categories:

- **Type-1** primitives are used without any modifications in many letters
- **Type-2** primitives are dynamic and change shape slightly in different letters
- **Type-3** primitives are also dynamic but much more flexible

5.2.1 Type-1 Primitives

This is the first class of primitives, and includes primitives that are used as they are without modifications in many letters. We will describe the design of 3 of

such primitives in this section.

Tail Primitive

This primitive is used as the ending tail in letters like *wāw*, *rā'*, and *zāy* in both their isolated and ending forms. It has not flexibility parameters at all. Even in cases where kerning is applied on letters containing this primitive, calligraphers move the letter as a whole in order not to modify the tail's shape. Fig. 5.3 shows the tail designed using METAFONT at the left and its use in *wāw* and *rā'*, and Fig. 5.4 shows an example of kerning applied to letters with tails.

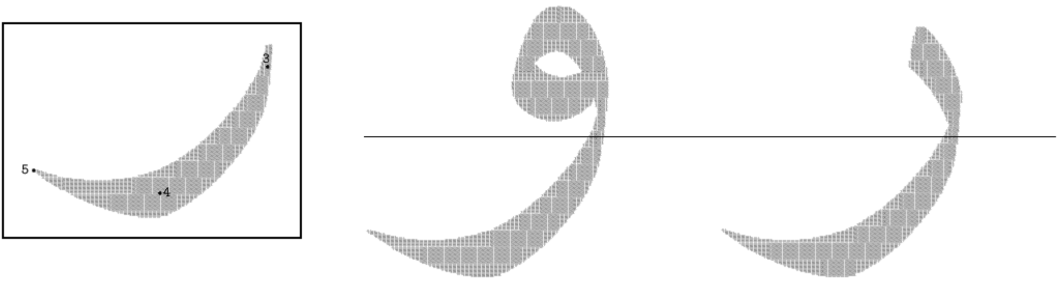

Figure 5.3: The tail primitive.

Figure 5.4: Four consequent tails in a word from the Qur'an [1] - [26:148]

The code for the tail is shown in Listing 5.1. Note the natural direction of the stroke, from point 3 to 4 to 5. Actually, the stroke is only between 3 to 4, the last part of the tail, is called a *shāẓya*, and is usually drawn using the tip of

the pen nib, and then filled, here we use `filldraw` for that purpose. Also note the coordinate points dependencies, `z4` depends on `z3` and `z5` depends on `z4` not `z3`. This makes modification of the glyph much easier, by separating between the definition of of the stroke segment and that of the *shāẓya*. i.e. if we modify the stroke, the *shāẓya* is not affected, unlike if `z5` was function of `z3`.

```
z4 = z3 + (−1.7n, −2n);
z5 = z4 + (−2n, .36n);
path raa_body;
raa_body = z3{dir −95} .. tension 1.3 .. z4{dir −160};
qstroke(raa_body, 85, 100, 0, 0);
path shathya;
shathya = (x4, top y4){dir −160} .. {dir 160}z5{dir −38}
                    .. tension 1.3 .. (rt bot z4){dir 15} -- cycle;
pickup pencircle scaled 1.2;
filldraw shathya;
```

Listing 5.1: METAFONT code describing the tail primitive.

The first and second lines in Listing 5.1 are interchangeable, and order is not important in such a declaration although the second line seems to depend on `z4`, this is due to METAFONT being a declarative language not an imperative one. The primitive is made up of two segments, each having an independent path. The first one, is a stroke and is drawn using the `qstroke` macro, which takes the path to draw and the pen inclination angle at each point on the path. The current definition of the macro needs at least four angles, but here since we have only two, we send the other two as zeros. The second path defines the *shāẓya* outline. The `top` command obtains the coordinate of the top point of the last pen used, and in this case a pen inclined by a 100 degrees, used in the `qstroke` macro. `cycle` on the other hand closes the path in order for us to close the path for filling. The filling is done by tracing the outline with a small `pencircle` and filling the inside. We used `filldraw` instead of `fill` in order to give thickness to

the *shāẓya* edge at point 5.

Base Primitive

The base primitive, is the main body of the letter *bā'*, and is also used in isolated *fā'* and ending *kāf*, *fā'*, and *bā'*. Of course it is also used in other letters of the *bā'* family such as *tā'* and *ṯā'*. The reason the *bā'* is not drawn with a single stroke, is because its starting tip or *senn*– having the shape of the Arabic numeral '١' – can be used in many other letters and hence is considered a separate primitive. Fig. 5.5 shows the base primitive and its use in isolated *bā'* and *fā'* connecting to them through point 3. The extended version of the *bā'* is very similar to the type-3 primitive of the *nūn* to be discussed in the next pages. See Listing 5.2 for the code explaining the primitive.

```
bot z4 = (4n, 0);
z4 = z3 + (−3.6n, −n);
z5 = z3 + (−5.5n, −.15n);
z6 = z5 + (.05n, .8n);
path base;
base = z3{dir −135} .. tension 1.5 .. z4{left} .. z5 .. z6{dir 80};
qstroke(base, 75, 85, 80, 70);
```

Listing 5.2: METAFONT code describing the base primitive.

All points of the base are defined relative to `z3`, except for point 6, which is defined relative to `z5` since this ending part of the glyph is independent from the main stroke. This facilitates glyph editing by isolating the final tip of the glyph. Note that we could not have defined the base without point 4, as the curve between 3 and 5 is not symmetric. If it was, then point 4 would not be needed. Point 4 is almost two thirds the distance from 3 to 5 and is the thickest part of the stroke, hence we input the pen inclination at its location to the `qstroke`

macro as 85 degrees. The path input to the macro is based on 4 points, and this is why we pass four angles representing the pen inclination at each point.

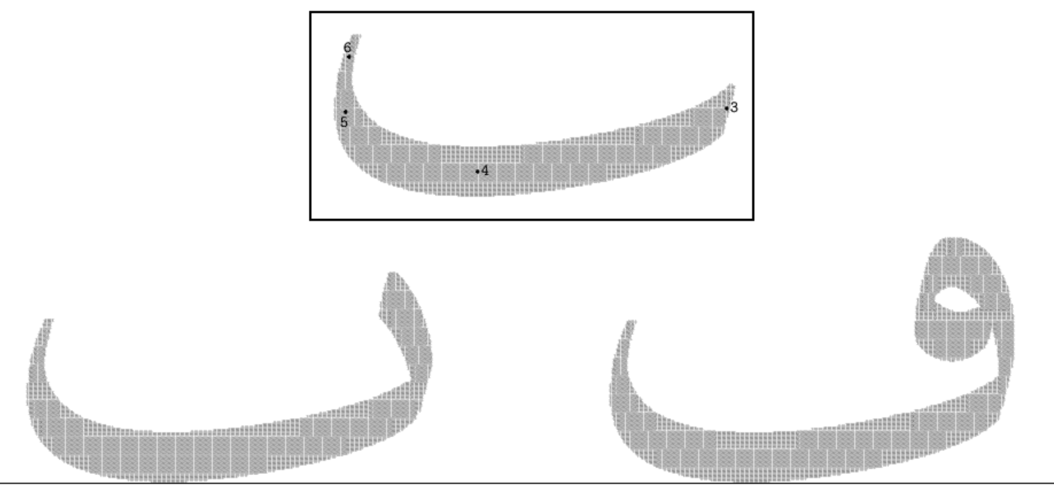

Figure 5.5: The base primitive.

Helya Primitive

The word *ḥelya* is the Arabic for decoration. There are many decorative primitives in the *Naskh* style, Fig. 5.6 shows the most common one and its use in three letters. It is a simple stroke between two points, and is used on top of the *'alif* stroke in *lām* (isolated and initial), *kāf* (isolated), and *ṭā'* (all forms). The code defining the *ḥelya* is shown in Listing 5.3.

```
path helya;
z2 = z1 + (0.43n, −0.35n);
helya = z1{dir −110} .. tension 1.3 .. z2{dir −10};
qstroke(helya, 70, 75, 0, 0);
```

Listing 5.3: METAFONT code describing the helya primitive.

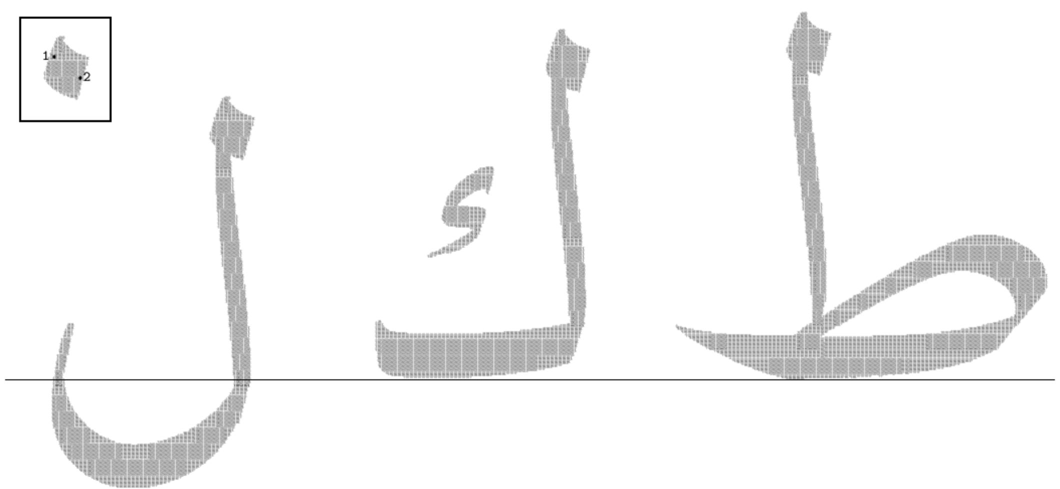

Figure 5.6: The helya primitive.

5.2.2 Type-2 Primitives

This class of primitives has few parameters that enable only slight variations in the primitive's shape in order to be used in different letters. We will discuss 4 primitives of this type.

Bow Primitive

The bow designed here is used in the isolated and final forms of the families of the *ḥā'* and *'ayn*, hence is used in 5 different letters: *ǧīm*, *ḥā'*, *ḫā'*, *'ayn*, and *ġayn*. Fig. 5.7 shows the bow in both families. The code in Listing 5.4, indicates that the 4 points used are fixed, and only the curve and stroke definitions vary between the bows of the two families. The bow in the *'ayn* goes further down and left between 3 and 4 and the starting inclination of the pen is different in both letters in order to merge well with the head of either the *'ayn* or *ḥā'*. The bow in both cases uses a pen tip drawn *shāẓya* at the end.

```
path bow; path shathya;
z3 = z2 + (−3.2n, −4.1n);
z4 = z3 + (3.94n, −1.9n);
z5 = z4 + (1.57n, 0.58n);
shathya = (x4, top y4){dir 0} .. {dir 10}z5{dir −140}
                .. tension 1.5 .. (lft x4, bot y4){dir −180} -- cycle;
fill shathya;
bow = z2{dir −160} .. {dir −85}z3 .. z4{dir 5};              % bow of ‘ayn
qstroke(bow, 70, 60, 75, 0);
bow = z22{dir −175} .. {dir −80}z3 .. z4{dir 5};             % bow of haa’
qstroke(bow, 65, 65, 75, 0);
```

Listing 5.4: METAFONT code describing the bow primitive.

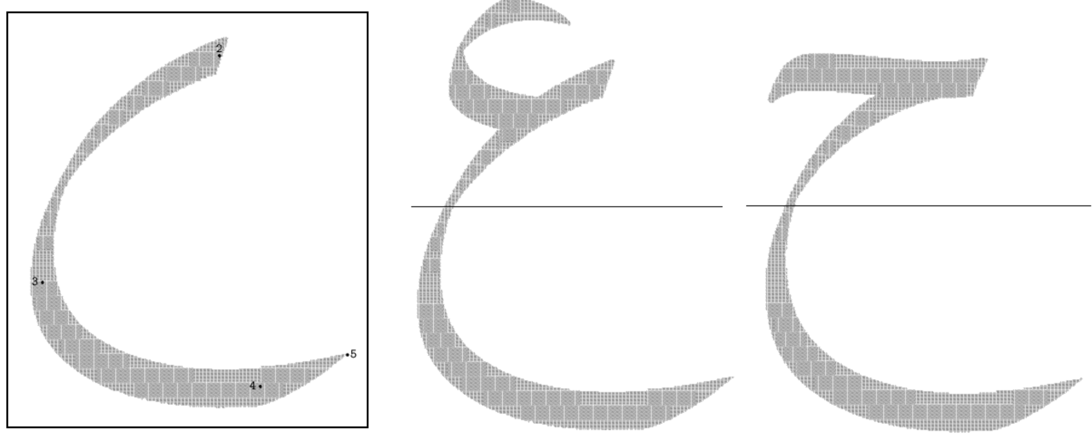

Figure 5.7: The bow primitive.

Waw Head Primitive

This primitive is used in the starting and isolated forms of the letters *wāw*, *fā’*, and *qāf*. It consists of two parts, the head and the neck. In most calligraphy books, while the neck is said to be slightly different, the head is described as being exactly the same in all three letters. However in the meta-design process, it was discovered that there are small differences between the different heads resulting

from the connections with different letter skeletons. Fig. 5.8 shows the letters *fā'* and *qāf* as drawn in three books. Note how the head does in fact look different in both letters, yet none of these book mention that there are variations in the head.

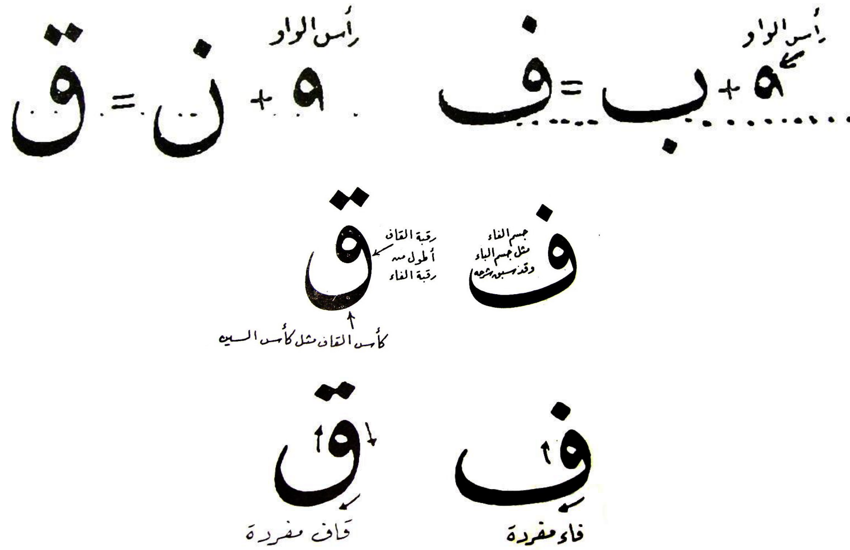

Figure 5.8: The letters faa' and qaf as drawn by three calligraphers, from top to bottom: Afify [21], Mahmoud [22], and Zayed [23].

This *wāw* head primitive consists of 2 strokes, one between points 1-2 the other between points 2-3-4, see the code in Listing 5.5. The largest difference between the different forms is in the neck between points 3-4 see Fig. 5.9. This same primitive can be modified slightly to be used in ending forms as well, by moving point 1 down to the right, to connect to a preceding letter or *kashīda*.

$\textbf{path } head[\,];$
$z_2 = z_1 + (-1.2n, 0.2n);$
$z_3 = z_2 + (0.6n, 0.6n);$
$z_3 = z_4 + (-0.7n, 1.4n);$
$head_1 = z_1\{\mathrm{dir}\ -130\} \mathrel{..} z_2\{\mathrm{dir}\ 90\};$
$\textbf{qstroke}(head_1, 75, 75, 0, 0);$
$head_2 = z_2\{\mathrm{dir}\ 70\} \mathrel{..} \mathrm{tension}\ 1.1 \mathrel{..} z_3 \mathrel{..} \mathrm{tension}\ 1.3 \mathrel{..} z_4\{\mathrm{dir}\ -100\};$
$\textbf{qstroke}(head_2, 75, 85, 85, 0);$

Listing 5.5: METAFONT code describing the waw head primitive.

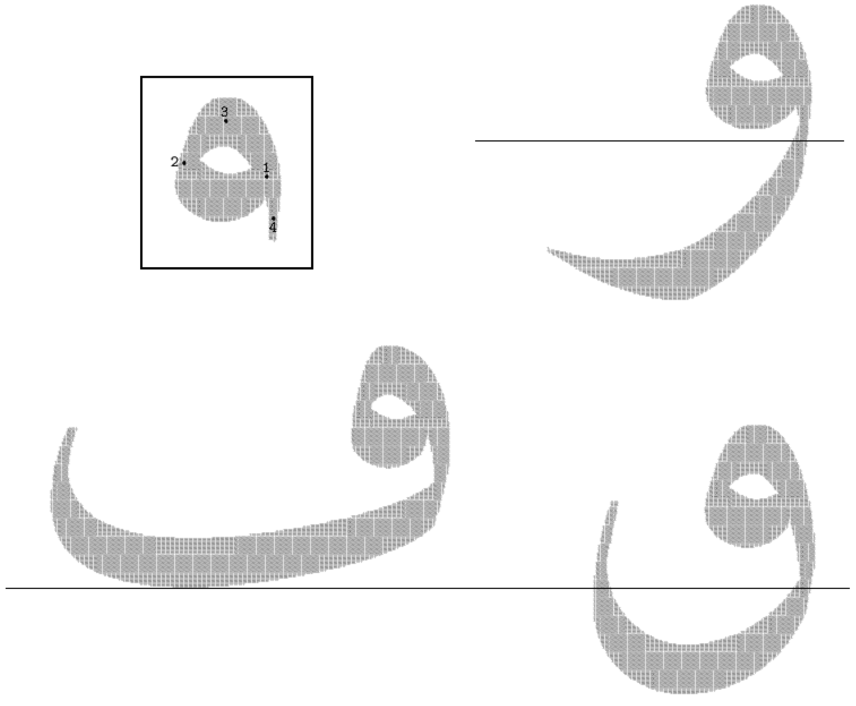

Figure 5.9: The waw head primitive.

Saad Head Primitive

This head is used in the letters *ṣād*, *ḍād*, *ṭā'*, and *ẓā'* in all their forms, see Fig. 5.10 for examples of the primitive in use. There are two differences between that of the *ṣād* and the *ṭā'*. These differences are due to the *'alif* of the *tā'* which makes the impression of applying pressure on the head making it flatter and taller. It is not easily noticeable and is not mentioned in most books, yet is applied. The head is made of two strokes, between 10-11-12 and 13-14.

The difference in the design of the two forms are: the tension in the 10-11 segment, and in the curvature of the lower 13-14 segment (controlled by the location of point 14 and curve tension). The *ṭā'* has a tighter tension in the upper segment, and has less curvature in the lower segment. Although the distance between 10 and 13 is constant, the length of the inside of the head is less in the *ṣād* due to the lower curvature. The code in Listing 5.6 shows the description of the primitive as used in the *ṣād* letter. Note how maximum thickness is obtained at point 14 by setting the pen inclination at 90 degrees to the stroke direction. Of course, when the *ṭā'* is in the final position, we end the lower segment with a *shāẓya*.

```
z11 = z10 + (2.8n, 1.5n);  z12 = z11 + (1.1n, −0.85n);
z13 = z12 + (−0.37n, −0.44n);  z14 = z13 + (−4n, 0.2n);
saad_top = z10{dir 40} .. tension 1.5 .. z11{dir 10}
                          .. tension 1.05 .. z12{dir −110};
qstroke(saad_top, 50, 50, 50, 0);
saad_base = z13{dir −155} .. tension 1.5 .. z14{dir 135};
qstroke(saad_base, 50, 90, 0, 0);
```

Listing 5.6: METAFONT code describing the saad head primitive.

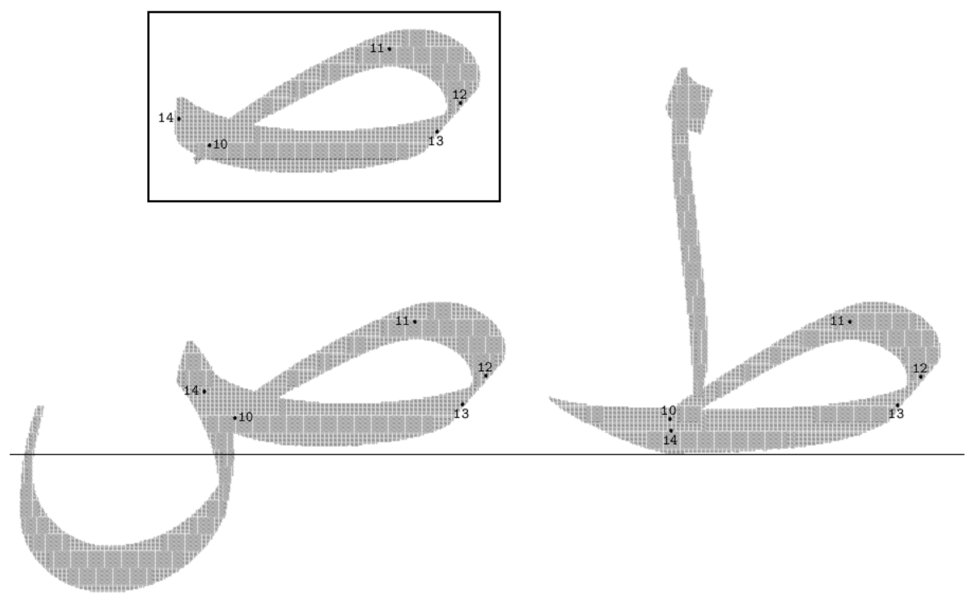

Figure 5.10: The sād head primitive.

'Alif Primitive

The stem of the *'alif* is used in many letters: *lām* (all forms), *kāf* (isolated and final forms), *mīm* (final form), and *ṭā'* (all forms). Fig. 5.11 shows on the far right the isolated *'alif*, and to its left the modified *'alif* that is used in *lām* and *kāf*. It is thinner with less curvature at the middle, or in other words more tension, together with more overall inclination. As the code for both forms shown in Listing 5.7 indicates, they both have the same height, and the thickness of the stem is achieved by increasing pen nib inclination.

```
% Description for isolated 'alif
curve := −100; incline := 70; height := 4.5n;
z2 = z1 + (0, −height);
path saaq;
saaq = z1{dir curve} .. tension 1.4 .. z2{dir curve};
qstroke(saaq, incline, incline, 0, 0);

% Description for 'alif used in lam and kaaf
curve := −95; incline := 79; height := 4.5n;
z2 = z1 + (0.3n, −height);
path saaq;
saaq = z1{dir curve} .. tension 1.4 .. z2{dir curve};
qstroke(saaq, incline, incline − 3, 0, 0);
```

Listing 5.7: METAFONT code describing the alif primitive.

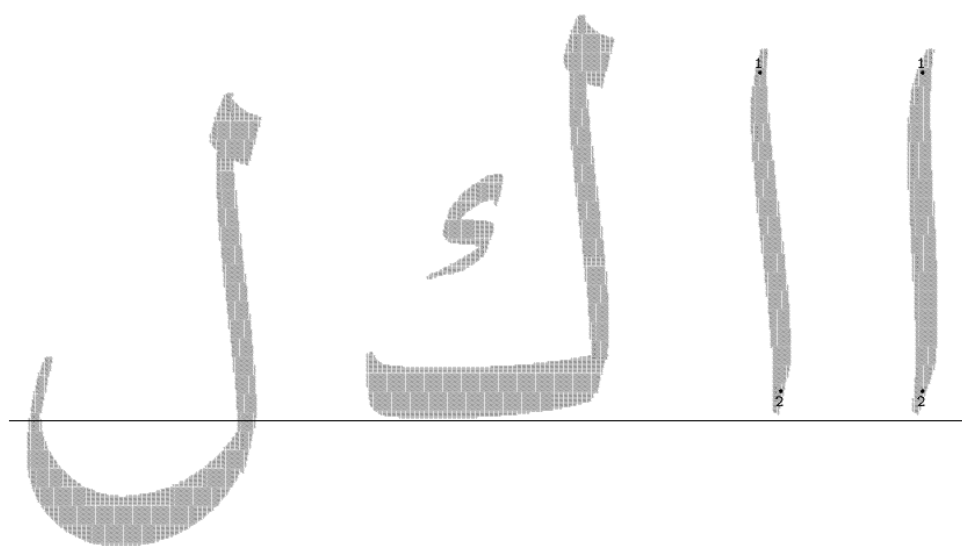

Figure 5.11: The 'alif primitve.

As mentioned before, many characteristics of the primitives were discovered during the meta-deigning process, since they were not mentioned explicitly in most calligraphy books. This is because when calligraphers don't measure their strokes with rulers and their description are only approximate. Detailed features of the letters are embedded implicitly in their curves as they learned them

by practice. But in our design, we represent the stroke mathematically which requires very accurate descriptions. The *'alif* is one of those primitives.

Most calligraphers describe the straight stroke in the *lām*, *kāf*, and *ṭā'* as being identical to the *'alif*. However, as just explained above, this is untrue. There are differences in the thickness, curvature, and height (in case of the *ṭā'*) between the isolated *'alif* and the one used in other letters. Fig. 5.12 shows part of a page from a calligraphy book [22] stating inaccurate directions about the form of the *'alif* stroke being exactly the same in different letters.

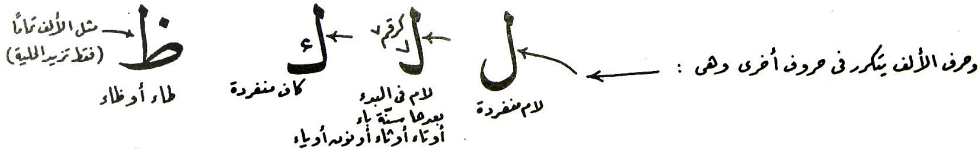

Figure 5.12: Approximate directions in calligraphy books.

5.2.3 Type-3 Primitives

Type-3 primitives are glyphs that have a wider dynamic range, and greater flexibility. We will discuss in this section, the *kashīda* and the skeleton of the letter *nūn*, also called the *kasa*. These primitives can be used in line justification thanks to their flexibility.

Kasa Primitive

The body of the letter *nūn* is used in the isolated and ending forms of *sīn*, *šīn*, *ṣād*, *ḍād*, *lām*, *qāf*, and *yā'*. Fig. 5.13 shows the *kasa* as it is used in five letters. The *kasa* has two forms, short and extended. The short is almost 3 *nuqṭās* in width in the case of *nūn*, one *nuqṭā* longer in *yā'*, and slightly shorter in *lām*. This is difference between the *kasa* of the *lām* and the *nūn* is not well documented in calligraphy books, where most calligraphers mention that both are the same and only few state that that it is slightly smaller.

An important property of the *kasa* is that it can be extended to much larger widths. In its extended form, it can range from 9-13 *nuqṭās*. The code describing the *kasa* in different forms is shown in Listing 5.8, and Fig. 5.14 shows the short form together with three instances of the longer form generated from the same code. Note that its width can take any value between 9-13, not just integer values, depending on line justification requirements. Also note how the starting *senn* of the letter is shorter in extended forms.

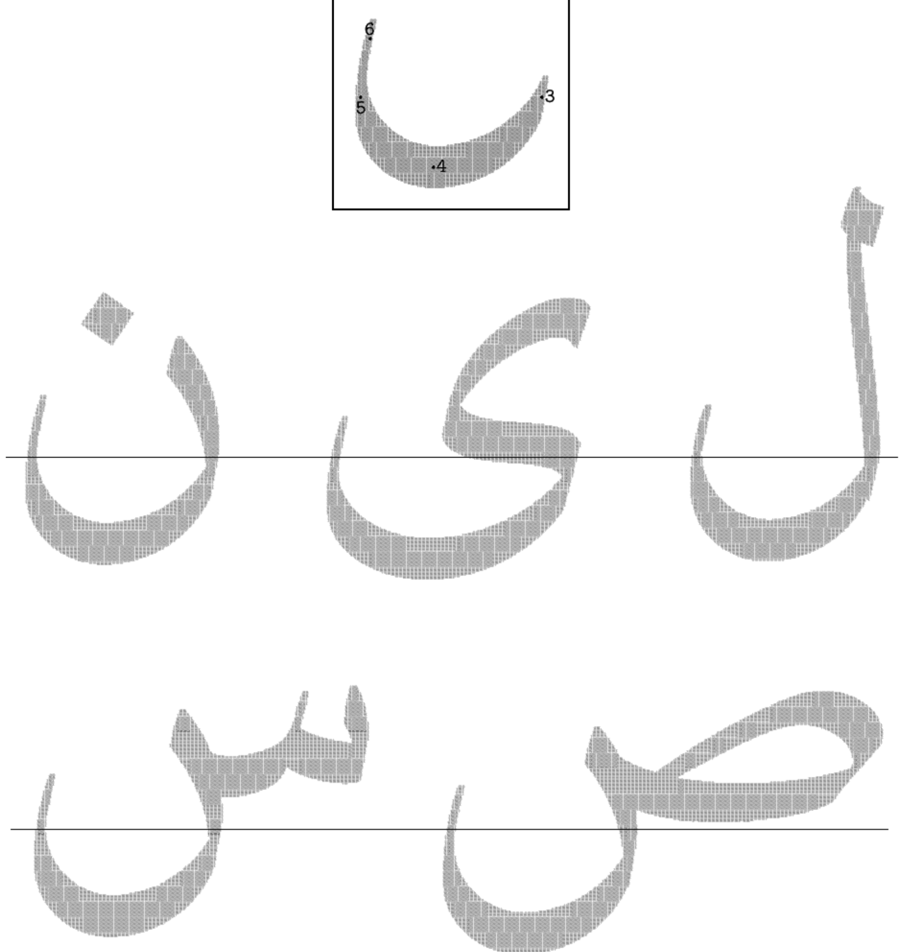

Figure 5.13: The kasa primitive.

```
% noon's kasa

if form = 1: kasa_width := 3.1n;                        % setting kasa width
elseif form = 2: kasa_width := tatwil * n;        % can be passed as parameter
fi

path senn;                                     % defining paths of senn and kasa
path kasa;
if form = 1:
   z1 = z2 + (−0.65n, 2n);
   z5 = (4n, −0.3n);
   z2 = z5 + (kasa_width, 0);
   senn = z1 .. z2{dir −95};
   z3 = z5 + (kasa_width, 0);
   z4 = z5 + (0.4 * kasa_width, −1.2n);
   z6 = z5 + (0.15n, n);
   kasa = z3 .. z4{left} .. z5{up} .. z6;
elseif form = 2:
   z1 = z2 + (−0.35n, 1.2n);
   z3 = z2 + (−0.14n, −0.5n);
   z5 = endpoint + (0, −0.6n);
   z5 = z3 + (−kasa_width, 0);
   senn = z1 .. z2{dir −90};
   z8 = z5 + (0.25 * kasa_width, −1.2n);
   z7 = z5 + (0.75 * kasa_width, −1.2n);
   kasa = z3{dir −(120 + 4 * (tatwil − 9))} .. tension 1.5
                        .. z7 .. z8 .. z5{dir(135 + 2 * (tatwil − 9))};
fi

if form = 1:                                                % drawing the strokes
   qstroke(senn, 75, 80, 0, 0);
   qstroke(kasa, 80, 85, 82, 78);
   z14 = point 0.25 of senn;
elseif form = 2:
   qstroke(senn, 75, 80, 0, 0);
   qstroke(kasa, 70, 85, 110, (120 + tatwil − 9));
   z14 = point 0.25 of senn;
fi

if form = 1: z10 = 0.35[z5, z3] + (0, 2.8n);                 % drawing the dot
else: z10 = 0.5[z5, z3] + (0, 1.6n);
fi
put_nuqta(z10);
```

Listing 5.8: METAFONT code describing the kasa primitive.

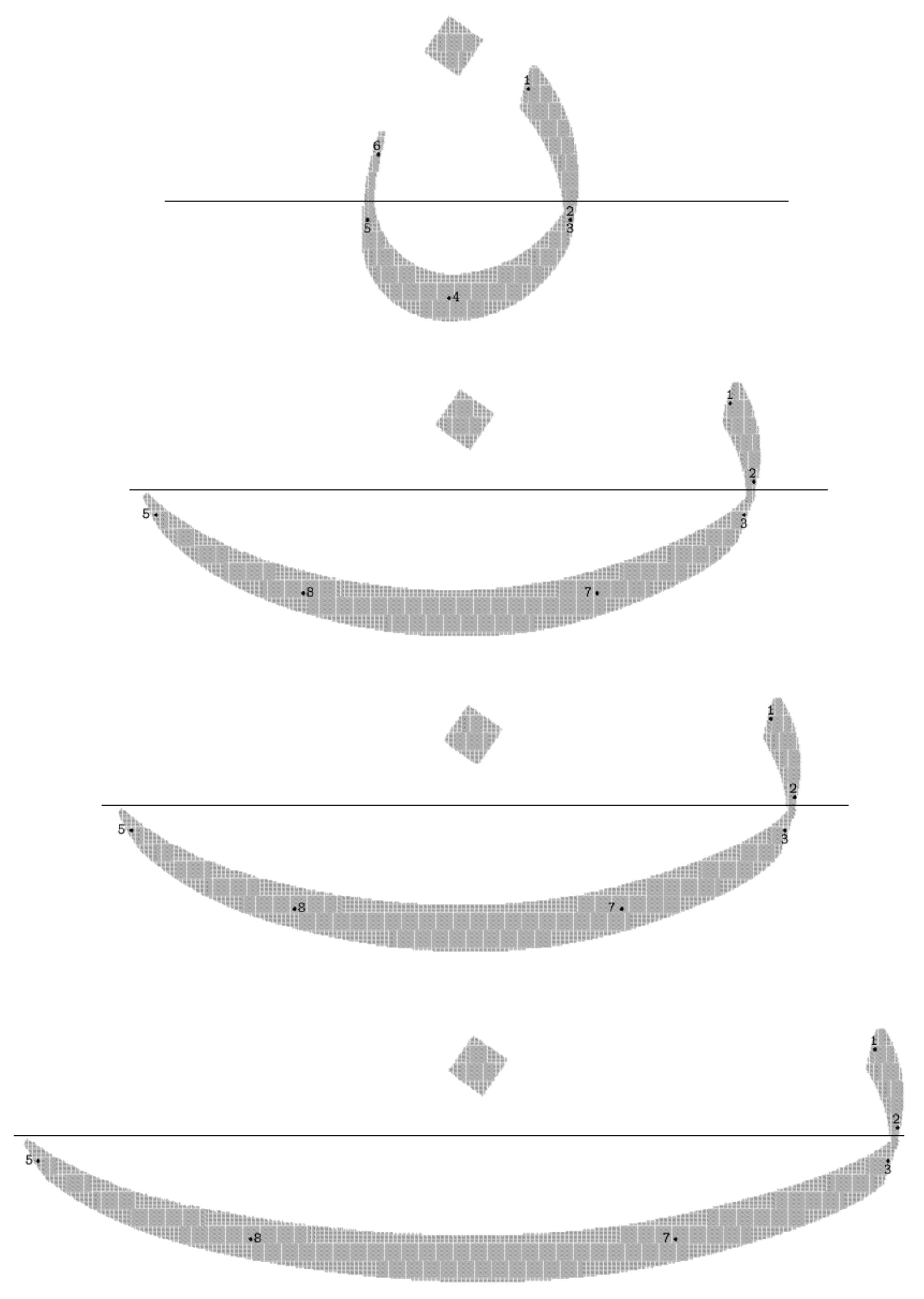

Figure 5.14: The letter noon shown with kasa widths of 3, 9, 10 and 13 nuqtas.

Kashida Primitive

Another very important primitive when it comes to justification, the *kashīda* can be used in almost all connected letters. Here we illustrate the *kashīda* in use with the letter *ḥā'* as an illustrative example. Fig. 5.15 shows the letter *ḥā'* in its initial form with two different *kashīda* lengths, differing by 3 *nuqṭās*. The parameter `tatwil` controls this length by varying the distance between points 9 and 10, both the horizontal and vertical components, as shown in the following line of code:

$$z_9 = z_{10} + (1.74 * n, 0.116n) + (0.5tatwil, 0.025 * tatwil) * n;$$

As `tatwil` increases, point 9 moves far away from point 10 both to the right direction and up. This vertical change helps maintain the curvature in the *kashīda*. If the no vertical adjustment is made, longer *kashīdas* will look more like straight lines, which does not happen in practice since calligraphers like to draw curved lines more than straight ones producing better shapes, aesthetically.

In the definition of the stroke, the tangential direction at point 9 is left free depending on the distance between 9 and 10. We will see in the next chapter, how *kashīdas* are adjusted to join letters together smoothly.

5.3 Primitive Substitution

As already known about the Arabic script, each letter has 4 positional forms depending on its location in a word, either it is initial, medial, ending or isolated. In order to model the writing of calligraphers, we have to go a step further and that is by providing different glyphs for each location within a word. The selection of the appropriate glyph is called *glyph substitution*. The approach we use is

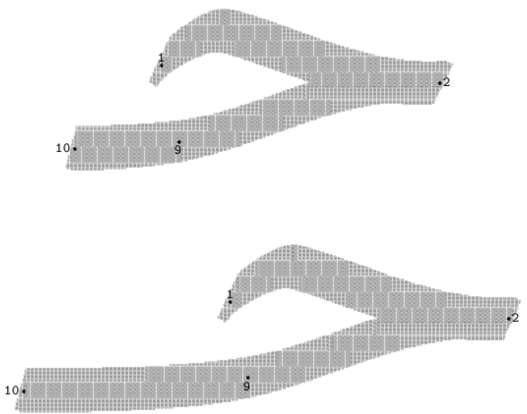

Figure 5.15: The initial form of the letter haa’ with two different kashida lengths.

very different from other existing standards like OpenType, and this substitution is not related only to ligatures as understood in such standards. In our work we utilize primitive glyph substitution instead of whole letters or ligatures glyph substitution. We will illustrate the concept of multiple forms using two examples, one of the letter *ḥā’* and another of the letter *dāl*.

In our design each letter has many forms, not just the 4 positional forms. For example, the *ḥā’* in its initial position alone has more than 4 forms. Our methodology of designing primitive glyphs and combining them enables us to avoid designing tens of different glyphs, then assign different codes for each letter like in the current Unicode standard [18]. Fig. 5.16 shows different primitives that join with the head of the *ḥā’* to produce different shapes. Each of these primitives can have parameters of its own. Note how the head itself changes form when preceded by a *bā’* for example, becoming more flat as in the top right figure. Depending on the form required in a word, we perform primitive substitution instead of whole ligature glyph substitution.

Another example of the multiple forms designed for each letter is for the *dāl*. The code shown below describes the two different forms of the letter: isolated and final. The code is written in a macro form with two input parameters: `form` and `tatwil`. Depending on these two inputs we can get the letter’s different

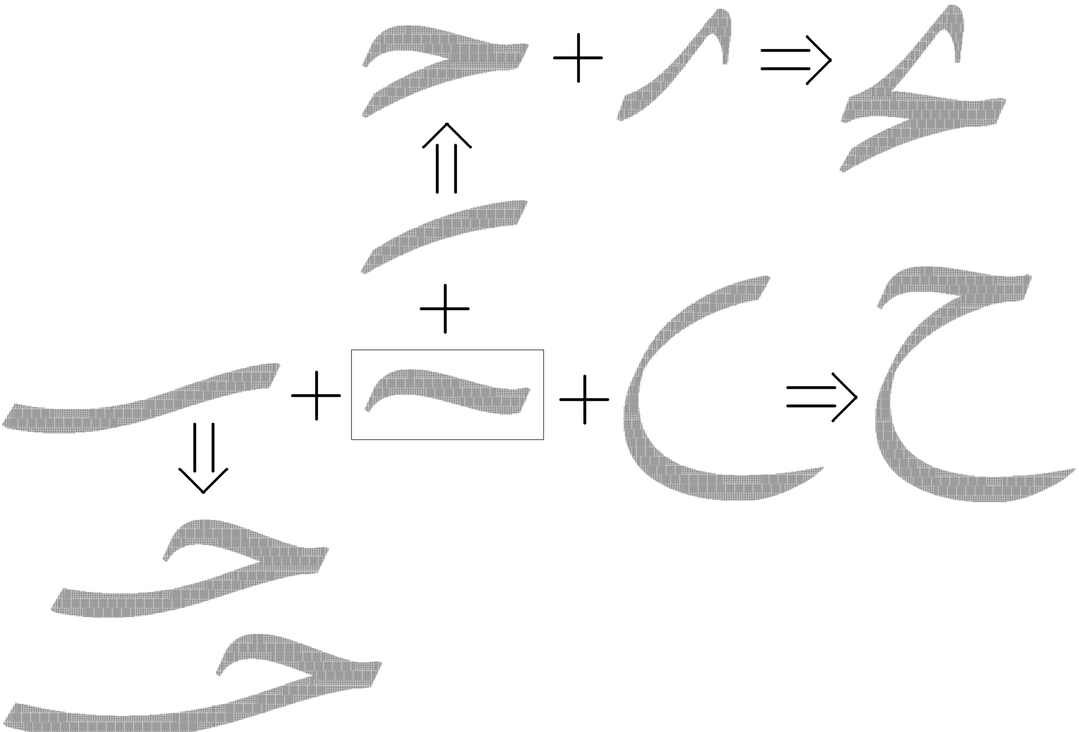

Figure 5.16: Many primitive glyphs used to produce the different forms of haa'.

forms. Fig. 5.17 shows two instances of the final form and one of the isolated form. The difference between the two final forms is very small, and is only in the connecting part called the *stem* of the *dāl*. This part is parameterized in order to join well with preceding letters or *kashīdas*. The *dāl* on the far left of the figure has less curvature in the connecting part - as better illustrated with the curve below it - and hence connects better if preceded by long *kashīdas*. The `tatwil` parameter affects how the the *stem* appears by affecting its tension as the code in Listing 5.9 shows. In the next chapter we will discuss the connections of glyphs using *kashīdas* in more detail.

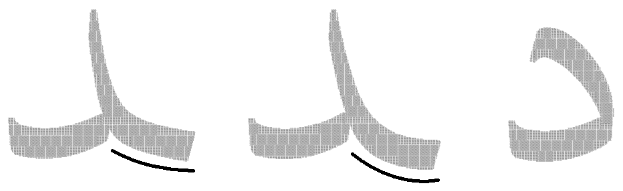

Figure 5.17: Isolated and ending forms of the letter dal.

```
def dal(expr form, tatwil) =                    % dal forms: 1 = isolated
path base;                                      % and 2 = ending
z4 = endpoint;
z3 = z4 + (2.14n, 0.357n);
z5 = z4 + (−0.18n, 0.18n);
base = z3{dir −150} .. z4{dir 165} .. z5;

path bow;
z2 = z3 + (0.05n, 0.05n);
z1 = z2 + (−1.8n, 1.93n);
bow = z1{dir 60} .. tension 1.5 .. z2{dir −90};

path stem;
z6 = z3 + (2.14n, −0.71n) + (0.5tatwil, 0) ∗ n;
z7 = z3 + (0.4n, 0);
z8 = z3 + (−0.36n, 2.5n);
stem = z6{dir 180} .. tension(1 + 0.1 ∗ tatwil)
                 .. z7 .. tension 1.3 .. z8{dir 100};

if form = 1: qstroke(bow, 75, 85, 0, 0);
elseif form = 2: qstroke(stem, 75, 80, 85, 0);
fi
qstroke(base, 80, 90, 95, 0);

if form = 1: endpoint := endpoint + (3.5n, 0);
elseif form = 2: endpoint := z6;
fi
clearxy;
enddef;
```

Listing 5.9: METAFONT code describing the letter dal.

5.4 Diacritic Glyphs

This section discusses in brief, the design of three diacritic glyphs. Fig. 5.18 shows the *shāra* of the letter *kāf* and the *hamza* and below is the code producing them. The `tstroke` macro used in the code in Listing 5.10 is the same as `qstroke` explained in Chapter 4 but it uses a pen of half the width in order to be used in drawing diacritic marks.

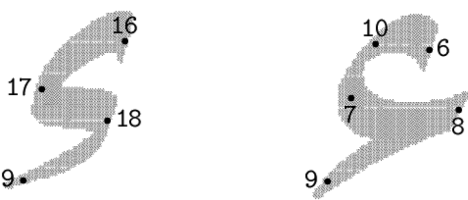

Figure 5.18: The shara of kaf and the hamza.

$$
\begin{aligned}
&\textbf{path } \mathit{shaara};\\
&z_{17} = z_{16} + (-0.7n, -0.4n);\\
&z_{18} = z_{17} + (0.55n, -0.26n);\\
&z_{19} = z_{18} + (-0.7n, -0.5n);\\
&\mathit{shaara} = z_{16}\{\text{dir } 80\} \mathrel{..} \text{tension } 1.5 \mathrel{..} \{\text{dir } -120\}z_{17}\\
&\qquad\qquad \mathrel{..} \text{tension } 1.8 \mathrel{..} z_{18}\{\text{dir } -110\} \mathrel{..} \{\text{dir } -150\}z_{19};\\
&\textbf{tstroke}(\mathit{shaara}, 75, 75, 70, 20);\\
&\textbf{path } \mathit{hamza}[\,];\\
&z_7 = z_6 + (-0.65n, -0.4n);\\
&z_8 = z_7 + (0.9n, -0.1n);\\
&z_9 = z_8 + (-1.1n, -0.6n);\\
&z_{10} = z_7 + (0.2n, 0.45n);\\
&\mathit{hamza}_1 = z_6\{\text{dir } 100\} \mathrel{..} z_{10} \mathrel{..} \{\text{dir } -75\}z_7 \mathrel{..} \text{tension } 1.3 \mathrel{..} \{\text{dir } 25\}z_8;\\
&\mathit{hamza}_2 = z_8\{\text{dir } -160\} \mathrel{..} z_9;\\
&\textbf{tstroke}(\mathit{hamza}_1, 70, 50, 60, 60);\\
&\textbf{tstroke}(\mathit{hamza}_2, 60, 50, 60, 60);
\end{aligned}
$$

Listing 5.10: METAFONT code describing the shara and the hamza.

Drawing the dot is also worth mentioning, since it is better to be drawn using

a different pen nib. In reality, calligraphers draw their dots using only one side of the pen nib, which is the straight side, hence the pen nib we have selected in Chapter 4 can not be used as is. Instead, we draw with a perfectly rectangular pen nib. Fig. 5.19 shows the pen orientation with respect to the paper and the dots it produces. Calligraphers use the one on the right to draw dots, and we use a rectangular (or razor) pen to model this effect. Note how the top left side of the dot on the left is a little curved at its ends.

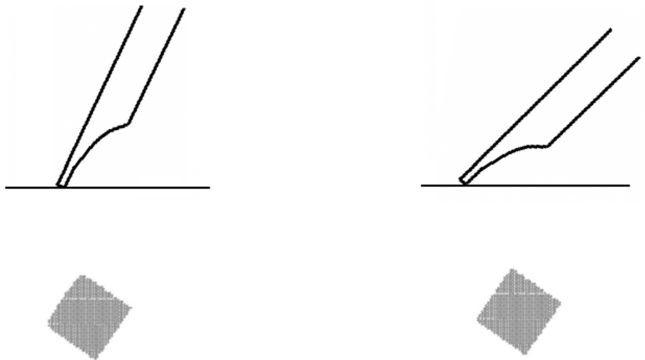

Figure 5.19: The dot as drawn by different pen orientation.

The *nuqṭā*, *kāf*'s *shāra*, and the *hamza* are all static, and can be regarded as Type-1 primitives. However, other diacritics like the short vowels: *fatḥa* and *kasra* are dynamic, and do change length and inclination angle. More about them will be told when we discuss future developments in the final chapter of this thesis.

Chapter 6

Forming Words

The preceding two chapters discussed in great detail how we model pens as closely as possible, and then how we meta-design Arabic letters. The next logical step is how such glyphs are joined together, which we will explain in this chapter. This step is the most important, and the final towards achieving the goal of the thesis, which is to produce Arabic script that is as cursive and flexible as calligraphers' writings. The parameterization of the glyphs is aimed at joining the glyphs perfectly as if they were joined with just one continuous stroke. At the end of this chapter we explain how a simple graphical user interface was developed to experiment with the developed font.

6.1 Joining Glyphs with Kashidas

As said before, the glyph used most in connecting other letters is that of the *kashīda*. In this section we will explain the mechanism we use in order to make the junction between letters as smooth as possible. In the most widely used font standards, like OpenType and TrueType, *kashīdas* are made into ready glyphs with pre-defined lengths, and are substituted when needed between letters to give the feeling of extending the letter. But since the *kashīda* is static, so as the rest of the surrounding letters, they rarely join well, and it is evident that the word

produced is made of different segments joined by merely placing them close to each other.

In our work, the *kashīda* is dynamic and can take continuous values, not just predefined or discrete values. We believe that when a *kashīda* is extended between any two letters, it does not belong to just one of them, instead it is a connection between both. This belief is the result of experimenting with different joining methods.

Let us take an example to illustrate the *kashīda* joining mechanism we developed. If we consider the simple joining of the two letters *ḥā'* and *dāl*, both designs of which we discussed in detail in the previous chapter. We first experimented with adding the *kashīda* together with the initial *ḥā'* glyph, and keeping the *dāl* static. It did not work because as the *kashīda* length changes, its ending left tip to be connected to the *dāl* changes its ending direction. Hence the macro of the *dāl* glyph also had to receive the length of the *kashīda* as a parameter. This parameter which we call the `tatwil`, was first used to change the direction of the final *dāl* form to match that of the *kashīda*, as explained in 5.3. Results we still not satisfactory, since the ending direction of the *kashīda* on the *ḥā'* side was left to be decided automatically by METAFONT to produce the best Bézier curve possible.

The solution we arrived at was to pass the `tatwil` parameter to the macros producing the two glyphs, and the *kashīda* length is distributed equally between both glyphs. This enabled us to fix the ends of the glyphs to be joined at one angle, which is along the x-axis, since any *kashīda* must at one point move in this direction before going up again. Listing 6.1 shows how both macros are affected by the `tatwil` parameter. Each glyph ending point is moved further from its letter, and in order to accommodate long *kashīdas*, these points are moved slightly downwards. Long *kashīdas* need more vertical space in order to curve smoothly, sometimes pushing the letters of a word upwards.

Other than affecting the ending points, the parameter also affects the curve definition on both sides by varying the tensions, while keeping the direction of the curves at the intersection along the $-ve$ x-axis, since the stroke is going from right to left, hence the 180 degrees shown in the code. The resulting word at many different *kashīda* lengths is shown in Fig. 6.1 which can be compared with the adding of *kashīdas* using TrueType fonts freely available in Microsoft Office 2003 in Fig. 6.2 resulting in flat horizontal connections.

```
% path definition of the dal's stem
path stem;
z6 = z3 + (2.14n, −.71n) + (.5tatwil, 0) ∗ n;
z7 = z3 + (.4n, 0);
z8 = z3 + (−.36n, 2.5n);
stem = z6{dir 180} .. tension(1 + .1 ∗ tatwil)
                .. z7 .. tension 1.3 .. z8{dir 100};

% path definition of the kashida in the haa
path kashida;
z10 = endpoint;
z9 = z10 + (1.74 ∗ n, .116n) + (.5tatwil, .025 ∗ tatwil) ∗ n;
kashida = (point 1.8 of head){dir 185} .. z9
                .. tension 1.2 .. z10{dir −180};
```

Listing 6.1: METAFONT code showing how effect of the kashida on glyphs.

Fig. 6.3 shows another example of joining letters. The word *yahia* is written using four different font technologies: TrueType Simplified and Traditional Arabic[1], OpenType Naskh[2] and our own parameterized font. Notice how, the TrueType fonts connect the ligatures with a straight line, and how the OpenType font improved on this by placing curved *kashīdas*. However, these curved *kashīdas* are static and rarely join well, and if you look closely you will notice the discontinuities in the curve. In the word produced by the parameterized font,

[1]Created by Monotype and available for use within Microsoft Office 2003.
[2]Created by Tradigital.

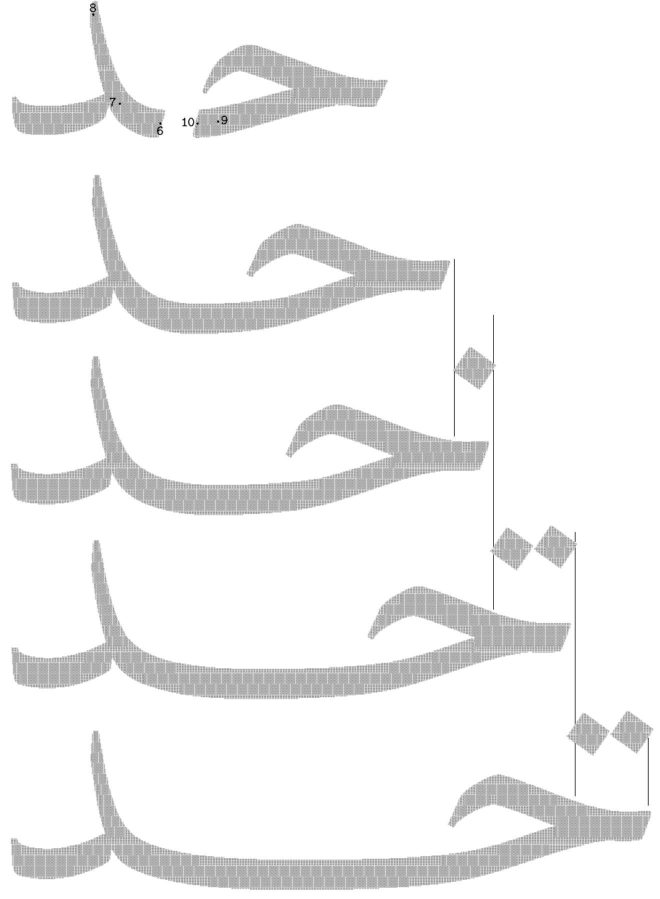

Figure 6.1: Placing a kashida between the letters haa’ and dal with different lengths: 2, 3, 5 and 7 nuqtas.

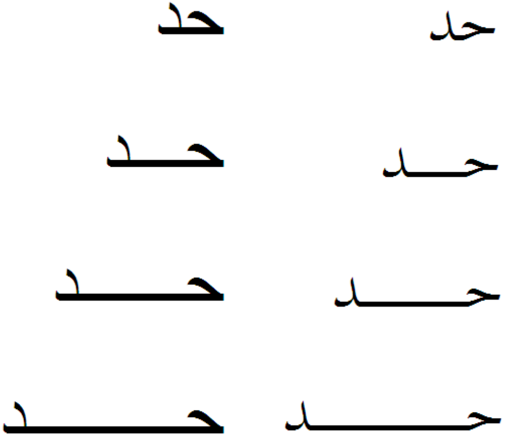

Figure 6.2: Placing kashidas using TrueType fonts.

the letters join perfectly together, and there is also the possibility of freely extending the length between the *ḥā'* and the *yā'* by any value similar to what was presented in Fig. 6.1.

6.2 Vertical Placement of Glyphs

In writing Arabic, the existence of some letter combinations may force the starting letter of a word to be shifted upwards in order to accommodate for the ending letters to lay on the base line of the writing. A very simple example of that property is the name of the Prophet Muhammad (PBUH) when written with ligatures, where the initial *mīm* is well above the baseline, as shown before in Fig. 1.2.

In the word *yahia* in Fig. 6.3, the initial *yā'* is raised above the *ḥā'*. If another *ḥā'* is placed above the already existing *ḥā'* this *yā'* will be moved further upwards. It is hence obvious that the starting letter's vertical positioning is dependent on the word as a whole. It might then be thought that it is easier to draw the words from the left going right, starting from the left at the baseline, and then move upwards while proceeding to the right. But this has two problems; one is that the horizontal positioning of the last letter depends on the position of the first letter on the right and on the length of the word. The second problem is that a left to right drawing would be against the natural direction of writing, which we were trying to preserve from the beginning.

The solution is then to walk through the word till its end and analyze each letter to know where to position the beginning letter vertically, and then start the actual writing at the right from that point going left. This process is what a calligrapher actually does before starting to write a word. To illustrate this better, see Fig 6.4. The ligature containing the letters *sīn*, *ǧīm*, and *wāw* is traced from left-to-right as shown, going through points 1-2-3, the starting points of each glyph, until the vertical position of point 1 is known. The next step is

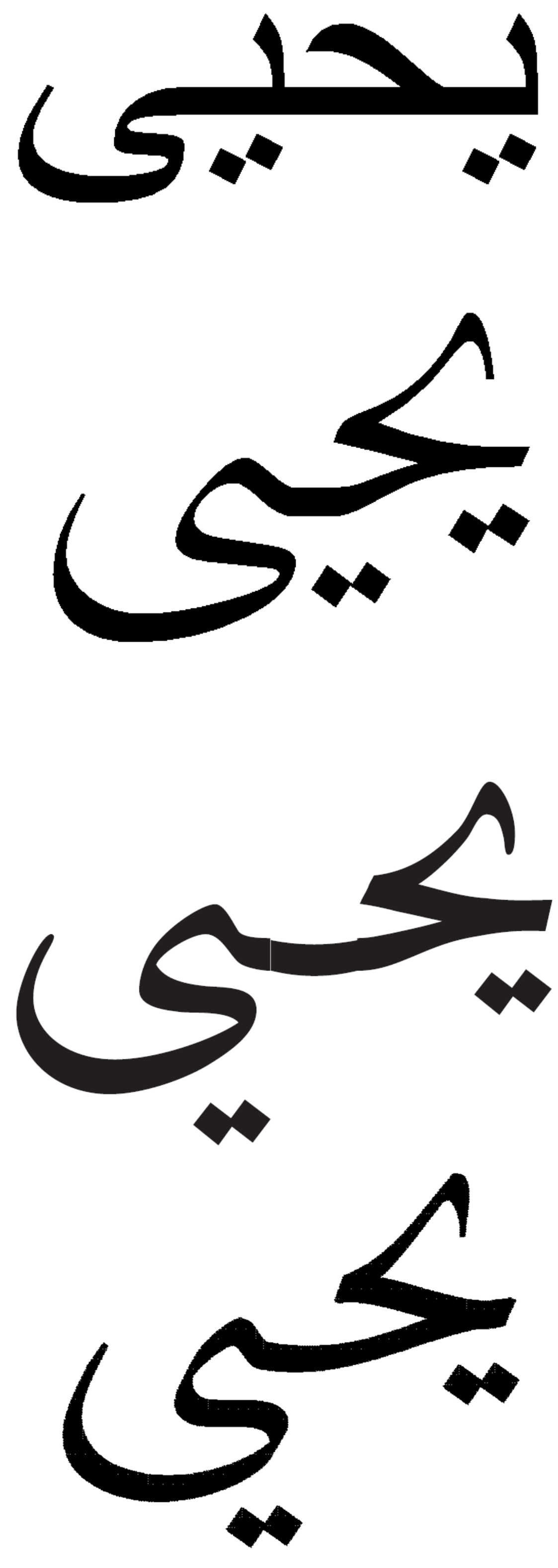

Figure 6.3: The word *yahia* as it is written using four font technologies. From top to bottom: TrueType Simplified Arabic (no ligatures), TrueType Traditional Arabic, OpenType Tradigital Naskh, and our own parameterized font.

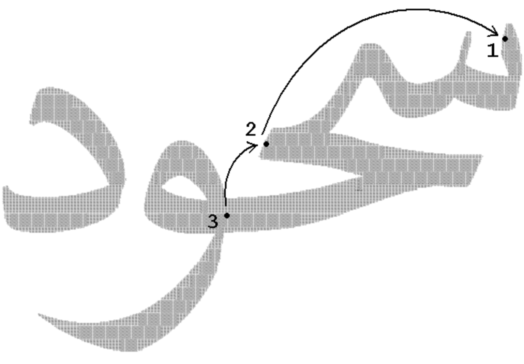

Figure 6.4: Tracing a word from left-to-right to know starting vertical position.

to start writing the word starting from point 1. Isolated letters like *dāl*, are not taken into consideration, because they don't affect the preceding letters vertically.

6.3 Word Lengths

In reality word lengths are not selected by the calligraphers per word, but instead, it is a decision based on the justification requirements of a whole line. When a word length is decided according to the line it exists in, this length is passed to a main macro that calls glyph macros in order to form the word. This macro decides the length of *kashīdas* to be added depending on the minimum length of each letter. For example, if the word under consideration is the same as that in Fig. 6.1, and the required total length of the word is 10 *nuqṭās*. In order to calculate the extension or the `tatwil` parameter between the letters, it subtracts all the minimum lengths of the individual letters. In our example, the head of the *ḥā'* is 4 *nuqṭās* wide, and the base of the *dāl* is 3 *nuqṭās*, hence the word can have a minimum length of only 7 *nuqṭās*. In order to stretch it to 10, the added *kashīda* is 3 *nuqṭās* wide.

6.4 A Final Example

This section describes a more illustrative example shown in Fig. 6.5. This example, showing four instances of the word *sujud*, demonstrates the many properties and benefits of our parameterized font. First, it shows flexibility in stretching and compressing words for line justification purposes. This flexibility is due to two capabilities of the font: dynamic length *kashīdas* and glyph substitution. For a very small line spacing, the *sīn* is written on top of the *ḥā'*, and the *kashīda* after the *ḥā'* is almost zero.

When more space is available, the *kashīda* after the *ḥā'* is stretched and the *senn* connecting the *sīn* and the *ḥā'* is also made slightly longer. Further elongation, is made possible by breaking the ligature between *sīn* and *ḥā'*. And finally, the maximum length is obtained by elongating the *kashīda* between the two letters. Theoretically speaking, we can get more stretching of this *kashīda* and even adding another one after the *ḥā'*, but calligraphic rules are the limits in this case.

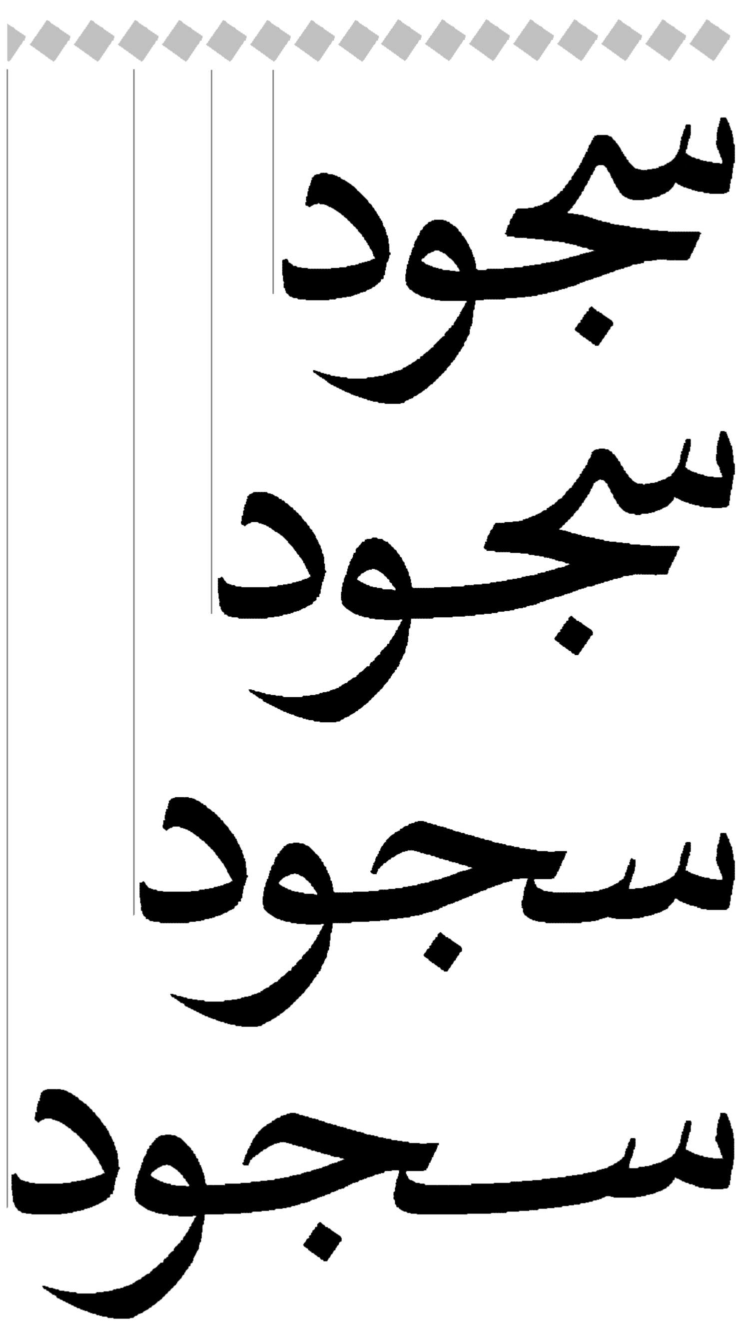

Figure 6.5: The word *sujud* written with different lengths ranging from 10.5 to 16.5 nuqtas.

6.5 Design of a Graphical User Interface

To test the ideas presented in this thesis, a Graphical User Interface (GUI) was designed. This GUI is a simple interface between a user and the METAFONT programs beneath. Fig. 6.6 shows the block diagram describing the operation of the GUI and Fig. 6.7 shows the different window components of the GUI.

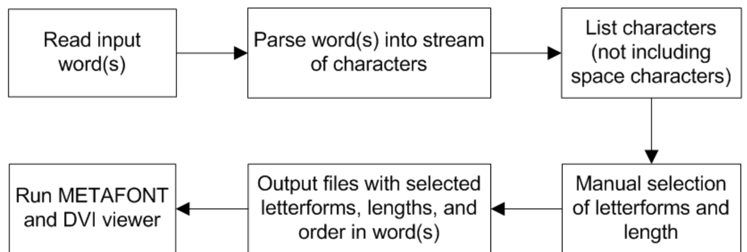

Figure 6.6: Block diagram describing the operations of the graphical user interface.

First, the user types in a word (or a sequence of words) in the input word textbox, then presses the parsing button. This parses the input word(s) into a string of letter, removing the space characters. Each character can be selected from a list and its shape determined from the letterform list at the bottom of the screen. Also a length extension can be input in the length textbox which has the default value of '0'. Extra letter length is only set to integer values.

When the output button is pressed, the letterforms selected and their extra length are written in files, then METAFONT is called to execute the glyph programs. Finally, a DVI previewer is called to open the resulting output as seen in Fig. 6.8. The GUI code is written using the Visual C++ language.

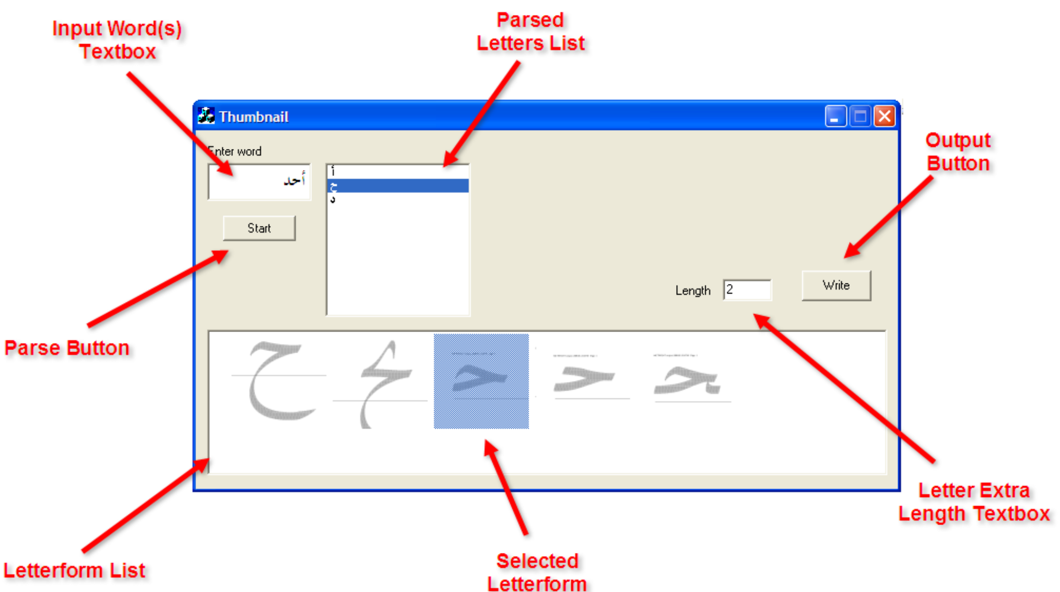

Figure 6.7: Screenshot of the designed GUI.

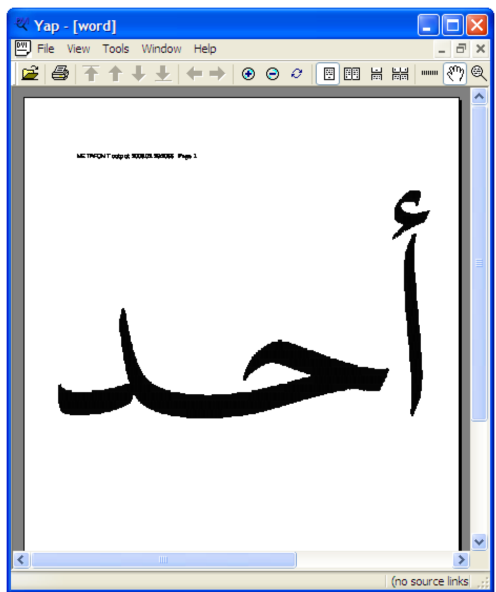

Figure 6.8: Screenshot of the DVI previewer displaying the output word.

Chapter 7

Results and Future Work

This chapter contains a discussion on a subjective test which was performed to evaluate the output words created by the suggested system, followed by many ideas for future work on the same subject.

7.1 A Subjective Test

A subjective test was carried out by surveying a sample of 29 people. The object of the test was to know if reader will be comfortable with the way out parameterized font looks as compared to other Naskh fonts. In this section, we explain the testing methodology used, how the test was designed, and finally, test results and comments.

7.1.1 Testing Methodology

The test methodology used is that of the Mean Opinion Score (MOS). This is a subjective test which is very often used to evaluate the perceived quality of media (like audio or video) after compression or transmission in the field of electrical communications and digital signal processing. The MOS is expressed as a single number in the range of 1 to 5, where 1 is lowest quality and 5 is highest quality.

7.1.2 Design of the Test

In our case, the MOS is a numerical indication of the perceived quality of a written word. A survey was designed asking a reader to evaluate words in terms of their written quality and he is asked to rate them in terms of how comfortable he is with them. A rating of 1 is given to low quality and uncomfortable words, and a rating of 5 is given to a word of high written quality and which is comfortable to the reader.
The test was composed of 16 words, each written in four different Naskh fonts:

- Simplified Arabic
- Traditional Arabic
- DecoType Naskh
- The parameterized METAFONT generated font presented in this thesis

The 16 words were chosen to test for three main features:

- Connections between letters
- Extension/Tatwil of letters
- Kerning

For a more reliable and unbiased test, the order of fonts used was varied in consecutive rows and all words are set to approximately the same sizes although same point sizes of different fonts were not exactly equal. Finally, the MOS is calculated as the arithmetic mean of all the individual scores. The designed test was composed of three pages and is shown in the next three pages.

a

a

a

7.1.3 Test Results

The survey was conducted on a sample of 29 people (14 males and 15 females) with ages ranging from 10 to 70 years. Table 7.1 shows the MOS scores for each font and the total average and Table 7.2 shows the average scores for individual words in the test.

Table 7.1: MOS results for each font.

Line No.	Simplified Arabic	Traditional Arabic	DecoType Naskh	AlQalam Parameterized
1	3.5	2.0	1.7	3.6
2	3.0	3.3	2.7	3.9
3	2.8	2.5	2.4	3.9
4	2.5	3.5	2.2	3.8
5	1.7	2.1	3.4	4.3
6	1.6	2.3	3.6	3.8
7	2.4	2.4	3.9	3.7
8	1.9	1.9	3.6	3.6
9	1.3	2.1	3.9	3.8
10	2.6	3.5	3.1	4.2
11	2.2	1.6	3.6	3.8
12	1.6	2.0	3.4	4.2
13	2.0	2.5	3.3	4.0
14	2.0	1.7	4.0	4.1
15	2.7	3.2	3.1	3.8
16	2.5	2.0	3.8	4.0
MOS	**2.3**	**2.4**	**3.2**	**3.9**

Table 7.2: MOS results for individual words in the test.

	Column A Scores	Column B Scores	Column C Scores	Column D Scores
1	هال 1.7	هال 2.0	هال 3.6	هال 3.5
2	هدهد 3.9	هدهد 2.7	هدهد 3	هدهد 3.3
3	حال 2.8	حال 2.4	حال 2.5	حال 3.9
4	هذا 2.2	هذا 3.8	هذا 2.5	هذا 3.5
5	أوزان 1.7	أوزان 2.1	أوزان 4.3	أوزان 3.4
6	سواك 1.6	سواك 3.6	سواك 2.3	سواك 3.8
7	سجود 3.9	سجود 3.7	سجود 2.4	سجود 2.4
8	سوف 1.9	سوف 1.9	سوف 3.6	سوف 3.6
9	ادوارد 1.3	ادوارد 3.9	ادوارد 2.1	ادوارد 3.8
10	ساجد 3.1	ساجد 2.6	ساجد 4.2	ساجد 3.5
11	وداع 1.6	وداع 3.8	وداع 3.6	وداع 2.2
12	واحـــد 2.0	واحـــد 1.6	واحـد 4.2	واحـد 3.4
13	رأى 3.3	رأى 2.5	رأى 2.0	رأى 4.0
14	أوجاع 2.0	أوجاع 4.1	أوجاع 1.7	أوجاع 4.0
15	يحيي 3.1	يحيى 3.2	يحيى 2.7	يحيي 3.8
16	حاول 4.0	حاول 2.0	حاول 3.8	حاول 2.5

7.1.4 Comments and Conclusion on Results

The results clearly show the superiority of the parameterized font with respect to the other popularly used fonts. One feature clearly making the difference between the different fonts is kerning. This is very evident in words 5, 9, 11, 13, and 14 as can be seen in Table 7.2. Another feature which is adding kashidas for extension also cause a large difference in scores as can be seen in word 12. We believe that more work and collaboration with calligrapher can harness even better results.

7.2 Future Work

This thesis proposed a new font technology that will enable computers to produce Arabic texts of similar quality to the works of calligraphers. This proposed parameterized font will also enable better typesetting, by enabling words to be more flexible. The work covered in this thesis is just the beginning and a small step towards the realization of such a system that produces output comparable to writings of humans writing Arabic, and still a lot of work is needed.

The proposed idea of producing such an output using computers opens a vast opportunity for further research in the topic. We classify this possible future work into three categories, research within the font technology itself using METAFONT, within the typesetting system, TEX, and within the output format, whether PDF or other formats. We will mention open topics of research in each of those categories.

7.2.1 Font Technology

Even with the work covered in this thesis on fonts, there is still much to be done:

- Finalizing meta-design of all possible letter forms. Some letters like *sīn* and *bā'* have a very large variety of forms. See Fig. 7.1

Figure 7.1: Different forms of the letters baa' and seen [23]. Note the many forms of the initial seen.

Figure 7.2: A verse from the Qur'an as written by Zayed [23] showing variable diacritic lengths and their placement.

- Developing algorithms and methods for dots and diacritics placement. Keeping in mind that their placement may, in some cases, force the calligrapher to move letters or words to free space for them. Also it should be kept in mind that this placement should not impede the legibility of the text, especially since dots and diacritics are intended to improve legibility and understanding in the first place. Some diacritics, especially short vowels like *fatḥa* and *kasra* change their lengths and inclination, and hence are dynamic. Fig. 7.2 shows some variable length forms of the *fatḥa*. Also note how diacritic placement changes vertically.

- Decrease the computational complexity of the current pen modeling techniques.

- Due to the difficulty of designing using METAFONT, it would be worthwhile to research the possibility of developing a graphical user interface for METAFONT.

- When a calligrapher writes on paper, the ink used to draw usually spreads

beyond the outline of the pen nib. This effect has to be modeled for more accurate writing. A related topic is that calligraphers move the pen with different speeds when drawing different segments of a glyph. This means that the ink spread will not be constant in all the glyph, but will be more in strokes with slower pen moving speed.

- Research the possibility of generating output from METAFONT other than the resolution limited bit-mapped glyphs for high quality printing or screen viewing. Another solution to research is to embed the METAFONT code inside PDF then let the PDF viewer run METAFONT to produce the adequate font size for the current resolution.

- The generation of other writing styles than *Naskh*, with minimal changes to the already meta-designed font.

7.2.2 Typesetting System

The work done in this thesis together with the future work in the font technology, aims at the end to provide the typesetting system with more flexibility. Below are some of the points that need work in this area:

- The selection of the most suitable glyph to be place in a word is a very complicated task. This is because each letter may have more than 5 forms in one location (see Fig. 7.2). This decision is based on many factors; most importantly, justification, and dots and diacritics placement conflicting with ligatures. The form of letter may be affected by not only its closest neighbors, but in some cases a letter's form may be changed depending on the 5th or 6th following letter. Fig. 7.3 shows how consecutive occurrences of the letter *bā'* together with *sīn* might result in the letters changing their forms. The selection may be done using state machines or through trees.

Figure 7.3: Example on how a letter's shape changes depending on following letters.

- Automatic page layout is fully applied in TEX for English, it is time Arabic layout rules are made available for typesetting Arabic, for example handling footnotes, titles, and references.

- Line-breaking algorithms are a very rich topic. The flexibility in the Arabic script, adds to the complexity of this task. Rules have to be added whether a ligature is to be broken or where a kashida is to be added, and which ligatures are more important to keep.

- TEX based systems have always had the drawback of lacking easy user interfaces. The target system is to be used by all people not just for research and technical writing, and hence a simple user interface has to be developed.

7.2.3 Output Format

The following points are research points targeting different output formats:

- Currently, most output from TEX systems is in the form of PDF. Work has to be done to facilitate the selection and copying of Arabic text in PDF for

pasting elsewhere. This will most probably deal with encoding.

- Embedding the font glyphs created in this thesis in PDF will result in large output files. Efficient font embedding mechanisms have to be developed.

- Screen versus paper output is also a hot research point even for viewing Latin scripts, similar work is needed for Arabic. See section on Optical Considerations on page 25.

- Another output format that needs research is producing output for the Web.

Finally, there is a strong need for non-engineering research on the readability and legibility of the different kinds of Arabic fonts comparable to many studies conducted on Latin letters. It is important to measure if calligraphers' writings are easier and feaster to read than regular computer typefaces used in long texts or not.

References

[1] *The Holy Qur'an*. Madinah, KSA: King Fahd Complex for Printing the Holy Qur'an, 1986.

[2] A. M. Sherif and H. A. H. Fahmy, "Parameterized Arabic font development for AlQalam," in *EuroBachoTEX 2007: Proceedings of the* 17th *Annual Meeting of the European TEX Users, Bachotek, Poland*, Apr. 2007.

[3] M. J. E. Benatia, M. Elyaakoubi, and A. Lazrek, "Arabic text justification," in *The Annual Meeting of the International TEX Users Group, Marrakesh, Morocco*, Nov. 2006.

[4] H. A. H. Fahmy, "AlQalam for typesetting traditional Arabic texts," in *The Annual Meeting of the International TEX Users Group, Marrakesh, Morocco*, Nov. 2006.

[5] T. Milo, "Arabic script and typography: A brief historical overview," in *Language Culture Type: International Type Design in the Age of Unicode* (J. D. Berry, ed.), pp. 112–127, Graphis, Nov. 2002.

[6] Y. Haralambous, "Simplification of the Arabic script: Three different approaches and their implementations," in *EP '98/RIDT '98: Proceedings of the 7th International Conference on Electronic Publishing, Held Jointly with the 4th International Conference on Raster Imaging and Digital Typography*, vol. 1375, (London, UK), pp. 138–156, Springer-Verlag, 1998.

[7] R. Rubinstein, *Digital Typography: An Introduction to Type and Composition for Computer System Design.* Reading, MA, USA: Addison-Wesley, 1988.

[8] D. E. Knuth, *The TEXbook*, vol. A. Reading, MA, USA: Addison-Wesley, 1986.

[9] Y. Haralambous and J. Plaice, "Multilingual typesetting with Ω, a case study: Arabic," in *Proceedings of the International Symposium on Multilingual Information Processing, Tsukuba*, pp. 63–80, Mar. 1997.

[10] Y. Haralambous and G. Bella, "Omega becomes a sign processor," in *EuroTEX 2005: Proceedings of the 15th Annual Meeting of the European TEX Users, Pont-à-Mousson, France*, pp. 8–19, Mar. 2005.

[11] K. Lagally, "ArabTEX: A system for typesetting Arabic," in *Multilingual computing: Arabic and Roman Script: 3rd International conference*, (Durham, UK), p. 9.4.1, Dec. 1992.

[12] K. Lagally, *ArabTEX: Typesetting Arabic and Hebrew.* User manual version 4.00, Stuttgart University, Mar. 2004.

[13] A. Lazrek, "RyDArab — Typesetting Arabic mathematical expressions," *TUGboat*, vol. 25, no. 2, pp. 141–149, 2004.

[14] H. T. Thành, *Micro-typographic extensions to the TEX typesetting system.* Dissertation, Faculty of Informatics, Masaryk University Brno, Oct. 2000.

[15] D. E. Knuth, *Digital Typography.* CSLI Publications, Stanford, CA, USA, 1999.

[16] K. Larson, "The technology of text," *IEEE Spectrum*, vol. 44, no. 5 (INT), pp. 20–25, 2007.

[17] D. E. Knuth, *The METAFONTbook*, vol. C of *Computers and Typesetting.* Reading, MA, USA: Addison-Wesley, 1986.

[18] The Unicode Consortium, *The Unicode Standard, Version 5.0.* Reading, MA, USA: Addison-Wesley, 2006.

[19] D. E. Knuth, *Computer Modern Typefaces*, vol. E of *Computers and Typesetting.* Reading, MA, USA: Addison-Wesley, 1986.

[20] J. D. Hobby, *Digitized Brush Trajectories.* Ph.D. dissertation, Department of Computer Science, Stanford University, Stanford, CA, USA, June 1986. Also published as report STAN-CS-1070 (1985).

[21] F. S. Afify, *ta'leem al-khatt al-'arabi [Teaching Arabic calligraphy].* Tanta, Egypt: Dar Ussama, 1998.

[22] M. E. Mahmoud, *al-khatt al-'arabi, derasa tafseelyya mowassa'a [Arabic calligraphy, a broad detailed study].* Cairo, Egypt: Maktabat al-Qur'an, 1995.

[23] A. S. Zayed, *ahdath al-toroq leta'leem al-khotot al-'arabiya [New methods for learning Arabic calligraphy].* Cairo, Egypt: Maktabat ibn-Sina, 1990.

[24] P. J. M. Coueignoux, *Generation of Roman Printed Fonts.* Ph.D. dissertation, MIT, June 1975.

[25] Y. Haralambous, "Typesetting the Holy Qur'an with TEX," in *Proceedings of the 2nd International Conference on Multilingual Computing (Latin and Arabic script), Durham*, p. 2.1.1, 1992.

[26] T. Milo, "ALI-BABA and the 4.0 Unicode characters," *TUGboat*, vol. 24, no. 3, pp. 502–511, 2003.

[27] T. Milo, "Authentic Arabic: A case study. Right-to-left font structure, font design, and typography," *Manuscripta Orientalia*, vol. 8, pp. 49–61, Mar. 2002.

[28] P. A. MacKay, "The internationalization of TEX with special reference to Arabic," *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pp. 481–484, Nov. 1990. IEEE catalog number 90CH2930-6.

[29] D. E. Knuth and P. A. MacKay, "Mixing right-to-left texts with left-to-right texts," *TUGboat*, vol. 8, no. 1, pp. 14–25, 1987.

[30] K. Lagally, "ArabTEX-Typesetting Arabic with vowels and ligatures," in Zlatuška [31], pp. 153–172.

[31] J. Zlatuška, ed., *EuroTEX 92: Proceedings of the 7th European TEX Conference*, (Brno, Czechoslovakia), Masarykova Universita, Sept. 1992.

[32] D. Berry, "Stretching letter and slanted-baseline formatting for Arabic, Hebrew, and Persian with ditroff/ffortid and dynamic POSTSCRIPT fonts," *Software—Practice and Experience*, vol. 29, no. 15, pp. 1417–1457, 1999.

[33] M. F. Plass and D. E. Knuth, "Breaking paragraphs into lines," in *Digital Typography* (D. E. Knuth, ed.), pp. 67–155, CSLI Publications, Stanford, California, 1981.

[34] Y. Haralambous, "The traditional Arabic typecase, Unicode, TEX and METAFONT," *TUGboat*, vol. 18, no. 1, pp. 17–29, 1997.