# Parameterized Arabic font development for AlQalam

Ameer M. Sherif, Hossam A. H. Fahmy
Electronics and Communications Department
Faculty of Engineering, Cairo University, Egypt
ameer dot sherif (at) gmail dot com, hfahmy (at) arith dot stanford dot edu
http://arith.stanford.edu/~hfahmy

**Abstract**

We present new approaches to Arabic font development for the AlQalam system. In order to achieve an output quality close to that of Arabic calligraphers, we try to model the pen nib and the way it is used to draw curves as closely to the ideal as possible using METAFONT. Parameterized fonts are also introduced for a more flexible and dynamic combination of glyphs, to be used in forming ligatures and in drawing whole words as single entities. Quality will improve if words are created as single entities since the Arabic script is cursive. We compare our method to the basic binding of glyphs using simple box and glue mechanisms and also to currently existing font design technologies.

## 1 Introduction

The process of typesetting languages using the Arabic script is more challenging and more complex than typesetting languages using the Latin script. Previous works [3, 4, 10] indicated the special needs for high quality Arabic typesetting. This paper concentrates on just one special property of the Arabic script, namely being cursive in nature, and presents a way to model this property accurately.

Being a cursive script means that letters interact with each other, and adjacent letters affect each other in many ways. Arabic letters have many forms depending on their position in a word: initial, medial, ending and isolated. Early typesetting systems stored glyphs for each character in each of these different forms and used them when required.

Another feature of the Arabic script is the presence of a large number of ligatures in any text, unlike the Latin script which uses only a few ligatures. Arabic ligatures usually include many letters, sometimes a whole word is one ligature. The longest example the current authors have seen is a ligature of seven consecutive letters. The issue of ligatures is partially solved in contemporary systems by introducing glyphs for only a selected number of letter combinations.

Current font design technologies still treat Arabic glyphs as separate boxes. Advanced technologies like OpenType do allow interaction between different glyphs, through numerous layout features, however, there are still limitations that we discuss below.

Hence, we propose a different solution to give us more quality and typesetting flexibility. Our solution is based on accurately modeling the process of writing Arabic, using the powerful, if underutilized, language of METAFONT.

## 2 Modeling the calligraphic process

The AlQalam project was initiated with the intention of typesetting Qur'anic and other traditional Arabic texts. Our goal is to produce an output quality as close as possible to a book written by a calligrapher (the majority of Qur'ans in print today are offset images of hand-written pages). 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 has been through simplifying the Arabic script for easier modeling. Haralambous [5] discusses the typographical simplifications applied to the Arabic script in these past years. Most of these suggestions for simplification failed over the years to gain any market acceptance. Nowadays, with the existence of more powerful computers and the advances in font technologies, it makes sense to try to model Arabic writing more accurately.

Letters have to be completely interactive with neighboring ones; in fact, an Arabic writer 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 certain horizontal space remaining for one word, the calligrapher will make use of additional ligatures and compress letters together if space is short, or

break some ligatures and extend some letters if extra space is available. 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, 10]. Moreover, it is not acceptable to justify the lines in Arabic only by varying the width of the spaces between words as done in Latin.

We illustrate these ideas with examples scanned from a widespread copy of the Qur'an printed in Al-Madinah [1]. Breaking and forming ligatures is evident in words as أَصْحَبُ becoming أَصْحَبُ, and also ٱلْحَجِّ becoming ٱلْحَجَّ. Other examples show how the kashida or tatweel (elongation stroke) is used to give words extra length as in أَحَد, أَحَد, and أَحَـد. 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 أَكُونَ or even أَكُـونَ. 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 allow a higher flexibility in line justification, much more than the unacceptable method of relying only on inter-word spaces.

Completely flexible and dynamic fonts must be available to provide this facility in typesetting programs. When we surveyed the available font design technologies, we concluded that METAFONT is the most suitable. We really need to describe the letters in a very abstract way to make them more flexible, i.e. we need not only design but meta-design the Arabic letters — analogous to the Computer Modern typeface family. By METAFONT, here we mean the language itself and not necessarily the output bitmap formats.

## 3 The Computer Modern typeface family

The Computer Modern (CM) typeface family produced by Donald Knuth [8] was a main source of motivation for this work. It is one of the landmarks in producing parameterized fonts. Despite the differences between Latin and Arabic characters, we believe it is possible to apply the same concepts used in designing the CM fonts to Arabic ones.

Each character or symbol of CM has a program to describe it. The font glyphs are defined by spline vectors, but unlike current outline fonts, these vectors are defined in a clear mathematical way that can be parameterized, allowing them flexibility.

**Figure 1**: Four different lowercase letter 'a' forms generated by a single description program. From left to right: roman, sans serif, typewriter, and bold.

The only drawback of this design technique is its difficulty. Knuth described his work to produce parameterized CM fonts to be "much, much more difficult than [he] ever imagined". 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" [8].

His final design of the 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 are meta-designed. Fig. 1 shows four of the lowercase letter 'a' generated by the CM family. These a's and many more are output from a single description program.

We aim to produce Arabic fonts that are as meta-designed as CM. Of course parameters would be very different, for example many parameters in the CM fonts described the serifs. In Arabic there are no serifs, but instead there will be other parameters for connecting glyphs and in forming ligatures.

## 4 Current font technologies

OpenType is currently the de facto standard font technology. 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 in connecting glyphs in cursive scripts like Arabic. Being the most common current font standard has led to the existence of many editors and tools that help design the glyph outlines. Some tools, such as Microsoft's Visual OpenType Layout Tool (VOLT), provide simple graphical interfaces for editing the GPOS and GSUB tables, among other features. In general, we find the main advantage of OpenType over METAFONT to be the ease of design and the availability of tools.

Despite the many features provided by OpenType, including those dedicated to the Arabic script, we see them as insufficient. The whole concept of letter boxes connecting together via other boxes of

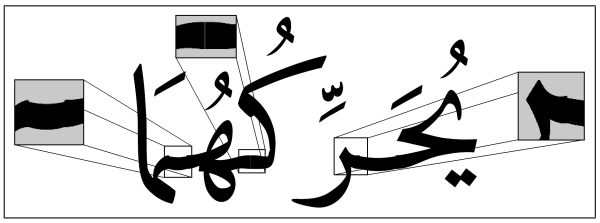

**Figure 2**: An example of OpenType font problems with junctions between glyphs. This word was obtained from sample products by Tradigital, a sister company of DecoType, with a typesetting system developed by Thomas Milo. Note that this font represents the highest quality in OpenType Arabic fonts we have seen, and if viewed at its standard original size, without enlargement, these imperfections are not visible.

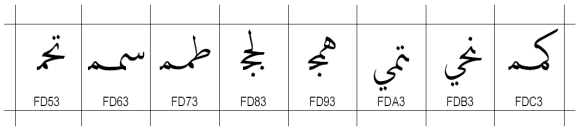


**Figure 3**: Part of the character code tables indicating code allocation to complicated ligatures, combining up to 3 letters.


**Figure 4**: 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 normally appears. 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.

elongation strokes is not suitable for highest quality Arabic typesetting, as we show in the following examples.

Outline fonts can be used to draw glyphs of characters in different forms very well when these glyphs are isolated. When connecting glyphs to one another, 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 tatweel, a ready made kashida of specific length is used to connect the glyphs together. Of course, such a kashida will not match perfectly with all the different glyphs. Fig. 2 shows examples of problems at junctions. Two of those problems are due to using kashidas. The junction after the kaf has no kashida, but it shows the non-uniform stroke width. It would be possible of course to edit the outline of these two glyphs to obtain a match, but this would certainly create a mismatch with yet other glyphs.

Another limitation is the use of already stored glyphs for different ligatures; since the number of possible ligatures is very large, only a selected portion can be made available. To model the Arabic script more accurately, each word should be considered a ligature and hence we would have an almost infinite number of ligatures, which is impossible to prepare in advance. The Unicode standard has numerous glyphs called presentation forms, each representing a unique ligature form. Unicode version 5 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 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, so following this method of code allocation to cover all ligature forms would take up every remaining free code (and more). The process of selecting a ligature should instead be left to the typesetting application.

A final feature that is more feasible to implement in METAFONT than in OpenType is the capability to program and embed information to be used in scaling glyphs for different sizes in the fonts themselves. This additional information (called 'hinting' in the OpenType terminology) may be used to enable optical scaling instead of linear scaling. The optical scaling is even more important when two strokes meet, as in the medial form of the letter 'sad' in the left-hand side of Fig. 4. At a small scale this stroke crossing produces a black blotch when it is used in a word. Knuth [7] discussed this problem, and its solution in METAFONT by decreasing the thickness of the stokes 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. 4. This solution can be parameterized such that, as the size decreases, the pen width at

**Figure 5**: Various pen nib shapes in METAFONT. From left to right: circle, inclined ellipse, square, inclined rectangle, and a polygon created by the `makepen` macro.

**Figure 6**: One path traced by two different pens [7]. The left-hand path was drawn using a circular pen, and the right-hand path used an elliptical pen inclined at 40 degrees from the x-axis.

**Figure 7**: Skeleton of the letter noon requires that the pen rotates to achieve different widths. In the left glyph (correct), the pen inclination from the x-axis changes from 70 to 120 degrees as it moves from right to left. In the right glyph (wrong), the same segment is drawn with a pen of fixed inclination of 75 degrees.

intersections decreases, thus giving a feel of uniform darkness at all sizes.

These limitations in new font technologies led us back to METAFONT, which has existed in its current form since 1986. What new font technologies are attempting (and sometimes failing) to achieve has mostly been feasible using METAFONT since it was created. It is only due to the complexity of the task that it was not widely tried either with METAFONT or with anything else. We think that the use of the METAFONT language to produce high-quality flexible Arabic fonts might be easier than the use of current OpenType tools. In the next section we discuss one of METAFONT's most powerful features, the notion of pens.

## 5 Modeling pens in METAFONT

The pens used in writing Arabic are of different types and were previously discussed by Benatia *et al.* [3]. In order to solve the problems of contemporary outline fonts, we propose a better pen model to achieve an output closer to the real writing of a calligrapher. We first discuss how pen nibs are defined in METAFONT and how pen strokes are modeled.

### 5.1 Pen nibs in METAFONT

METAFONT provides two predefined pen nibs, for circular and square pen nib shapes: `pencircle` and `pensquare`, respectively. Each can be scaled, with different scaling factors in each direction, allowing a multitude of elliptical and 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. 5 shows some of the nib shapes that can be used in METAFONT.

The most important pen macro in METAFONT is `makepen`. This macro enables the user to define any custom pen nib shape required, as long as it is a convex polygonal shape. The polygonal shape is defined by a number of coordinates connected by straight lines. The rightmost pen nib in Fig. 5 shows a polygonal nib produced by `makepen`. Since Arabic pens may not be purely rectangular or elliptical, `makepen` can be used for accurate modeling of the pen nib.

After defining the pen to be used in drawing, we need to define the path to trace. In METAFONT we define points in Cartesian coordinates and then describe how the path passes through these points, in what directions and angles. Bézier curves are used by METAFONT to define the equations of these paths. Again METAFONT gives unlimited flexibility when defining paths.

### 5.2 Plain METAFONT drawing macros

Given the pen to be used and the path to trace, we now have the last step, the drawing action itself. The main drawing macros defined in plain METAFONT are the `draw`, `fill`, and `penstroke` macros.

John Hobby [6] developed the algorithm defining the points traced by the pen. Fig. 6 shows a path drawn using the `draw` macro but with different pens. The limitation of the `draw` macro is its use of a fixed pen inclination for the different paths in the glyph. But in Arabic calligraphy, this is not the case. Many letters require that the calligrapher change the inclination of the pen as he draws. A clear example is the skeleton of the letter noon in its extended form; see Fig. 7. Its lower segment should be thick at the middle and thin at the tips, and this requires pen rotation while tracing.

The `fill` macro simply fills a closed contour. It does not model a pen, but we will demonstrate later how it can be used within other macros to do so.

The third drawing mechanism in plain METAFONT allows rotation of the pen while tracing the path. However, this mechanism does not use the

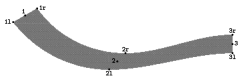

**Figure 8**: A path drawn using the `penstroke` macro. Note how the pen inclination changes as it moves across the path, resulting in a different path thickness at different parts.

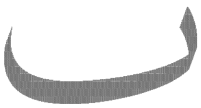

**Figure 9**: The first problem with `penstroke`: razor pen

defined pen nib, and instead approximates a stroke produced by a razor pen (a pen with zero-width). This makes the underlying algorithms used to determine the shaded contour much simpler than if a polygon pen was used. This drawing mechanism uses two macros, `penpos` and `penstroke`. `penpos` defines the width and inclination of the razor pen at each coordinate pair. `penstroke` does the actual drawing depending on the `penpos` at each point; see Fig. 8. Tracing a 2-D path with a rotating polygonal pen proves to be much more complex than the case of no rotation as with the `draw` macro, and it was not implemented in the plain METAFONT macros.

Although `penstroke` solves the problem of pen rotation, the use of a razor pen leads to three other problems while drawing Arabic letters. Most notably, the zero width of the pen razor causes some unwanted effects as shown in Fig. 9 when we try to draw the letter baa'. Close observation of the resulting glyph shows two defects directly. The first is that the left tip of the letter is 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 base of the letter. This intersection is thinner than usual due to the same reason related to the pen.

Yet, this is not the only issue with `penstroke`, and not even the most prominent. There are two other 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 two paths, right and left, connects them at the endpoints with straight lines, then fills the resulting contour. In fact the macro expands to:

**Figure 10**: The second problem with `penstroke`: figure-8 shape.

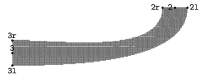

**Figure 11**: The third problem with `penstroke`: bad pen approximation.

```
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.

This implementation causes the two problems. The first is when we try to draw a shape like that in Fig. 10. 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 crossing 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, means that at some points the distance between the two 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. 11. 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. 10 also shows the same problem as a significant chunk of the stroke is missing at the middle of the path.

Although the `draw` and `penstroke` macros 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 neither macro is sufficient, and we need the best of both worlds: a polygonal pen that traces a path while rotating. The next section discusses some proposed solutions for accurate modeling of pen strokes. These solutions make use of the previously mentioned plain METAFONT macros in various ways.

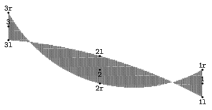

**Figure 12**: The error resulting from left and right paths crossing is resolved, even in the case of multiple crossings.

### 5.3 Enhancements to plain METAFONT drawing macros

In this section we describe four proposed methods to provide better modeling of the pen nib tracing a path while rotating, in order of their ascending quality. With the last being the most accurate drawing method. The first method was proposed by Knuth for his CM fonts, while we developed the other three in the course of our work. 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, are solved.

METAFONT does not fill a non-simple contour that crosses itself. In order to solve this, we propose finding the crossing 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 loop until there are no more crossing points. Fig. 12 shows an example of a stroke drawn by a pen that rotates 180 degrees from point 1 to 2 and then 180 degrees more from 2 to 3, hence rotating 360 degrees in total. Such a stroke would result in an error if the plain `penstroke` is used.

#### 5.3.1 The `filldraw stroke` macro

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, `filldraw` fills the contour and then traces its outline with a small circular nib pen, thus adding thickness to very thin segments of the "virtual pen stroke". The reason we say it is 'virtual' is because Knuth's definition of a glyph like 'e' keeps the pen rotating in such a way that the left and right paths of `penstroke` do not cross, and this is certainly different from what a person would do while drawing the 'e', hence it is not a real pen stroke.

The `stroke` macro is defined in the CM base file, and it merely defines the closed contour created by `penstroke` without filling it. Fig. 13 shows the letters e and baa' with `penstroke` and then with `filldraw stroke`. Note the thickness effect and how Knuth used this method to give letter tips round edges. The letter baa' is shown with its contour and after filling with `penstroke` on the left and with `filldraw stroke` on the right.

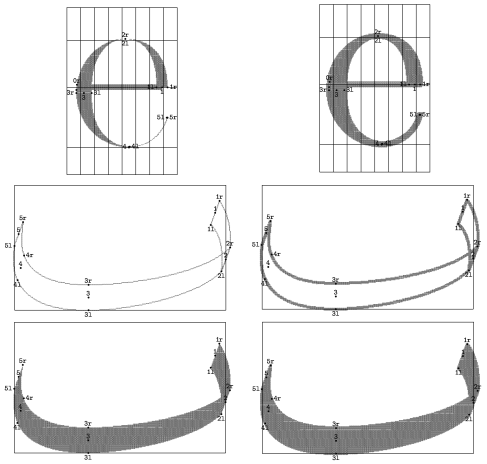

**Figure 13**: First solution: `filldraw`. The letter 'e' is shown on the top right as it is used in the CM fonts, and on the left how it would look like if drawn using `penstroke` instead of `filldraw stroke`. The top pair of baa' letters shows the skeleton of the letter, i.e. the closed contour. The left contour is drawn with a very fine pen nib, while on the right with a thicker nib. The pair of baa' letters at the bottom show the contours after filling.

#### 5.3.2 The `astroke` macro

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 example, for a rectangular pen nib, a macro is defined which essentially breaks into four `penstroke`s, each to model the area covered by one side of the rectangle. The `penpos` macro also had to be modified in order to evaluate the four corner points of the pen nib [l, r, n, m] instead of only two (left and right). Fig. 14

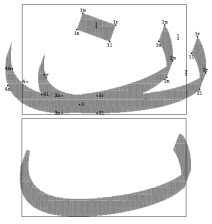

**Figure 14**: Second solution: `astroke`. `penpos` defines four points for each coordinate pair, named l, r, n, and m (see dot at top). `astroke` macro then produces four `penstrokes`; the top figure shows two of these sides (l–r) and (n–m).

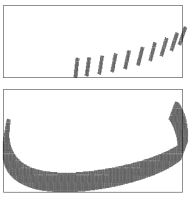

**Figure 15**: Third solution of stroke modeling: `qstroke`. It forms the pen stroke by drawing many instances of the pen nib along the path. The pen nib used can be any shape, not just rectangular as in the `astroke` macro.

shows in the top figure two of the four penstrokes resulting from the pen nib shown. The pen nib is enlarged for better understanding. The bottom figure shows all four `penstroke`s but with reasonable pen nib dimensions. The resulting baa' skeleton is much better than the one produced using only one `penstroke`, and slightly better than the one using `filldraw stroke`. Note that this macro can only model strokes for rectangular pen nibs.

A pen used to write Arabic is rarely moved in the direction of the smaller side. Hence the need to model the smaller sides of the pen is limited to tips of the glyph. This means that it is also possible to produce the same output with only two `penstroke`s (l–r) and (n–m) together with two nib dots at the start and end.

#### 5.3.3 The `qstroke` macro

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 the segment's starting and ending inclination angles. At a given finite resolution, a finite number of pen dots gives the effect of a continuous pen stroke. As the distance of the path increases more pen footprints are needed. Also, since the path times in METAFONT are not linearly distributed, more instances are needed. Finally, when the pen rotation is large in a specific segment more instances are needed as well. Fig. 15 shows an example letter baa' drawn with the macro.

#### 5.3.4 The `envelope` macro

For high resolutions, the `qstroke` macro needs to draw many dots to yield a smooth stroke. A refinement of this idea is to compute the exact envelope of the razor pen and then fill it. This `envelope` macro

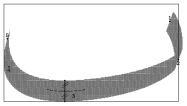

**Figure 16**: Fourth solution of stroke modeling: `envelope`.

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 more accurate model of a razor pen than `penstroke`; see Fig. 16. Applying four of this new `envelope` stroke as in the `astroke` solution produces the most accurate glyph.

Of course, 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, `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 section how parameterized glyphs are designed.

## 6 Arabic font meta-design

With the satisfactory pen nib models of the last section, we now discuss their usage to mimic the way calligraphers draw the different letters. Our main approach is to make the writing as dynamic as possible, while obeying the traditional rules of calligraphy [2, 9]. This enables us to simulate the cursive nature of the Arabic script. In order to do that we started to design a font that is parameterized in many ways. This parameterization comes in two forms: parameterization of coordinates and of curves.

Parameterization of coordinates means that our 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, or sometimes both together. This complete 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

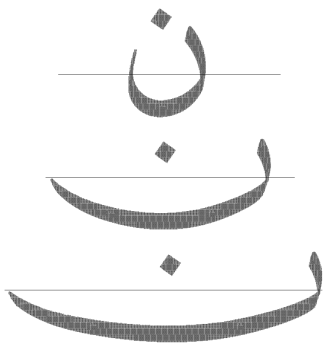

**Figure 17**: The letter noon shown with kasa widths of 3, 9, and 13.

to be drawn, which is more difficult. Outline vector fonts like TrueType can be created by merely scanning a handwritten sample and then digitizing it by converting it to vectors. The more meta-designed the glyph, the more difficult and more time it takes to describe it to the computer, but on the other hand, the better the final results become.

The Arabic alphabet, although consisting of 28 different letters, depends on only 17 different basic 'skeletons'. Dots added above or below these skeletons differentiate one letter from another. For example the letters haa', jeem, and khaa' all have the same shape as haa', but jeem has a dot below, and khaa' has a dot above. The separation of the dots from the skeletons as well as breaking some complex skeletons to smaller parts enables us to reduce the amount of design required by considering only some primitive shapes that are repeated and used in many letters. The construction of individual letters by assembling smaller parts is the traditional method of teaching Arabic calligraphy as well.

For example, the body of the letter noon, called the kasa, is used as a part of the isolated or ending forms of many Arabic letters: seen, sheen, saad, daad, lam, and yaa'. An important property of the kasa is that it can be extended to much larger widths. Its nominal width is 3 nuqtas (Arabic dots), and when in its extended form, it can range from 9–13 nuqtas. Fig. 17 shows several instances of the longer form. Note that its width can take any real value between 9 and 13, not just integer values.

Another example of a meta-designed primitive that is used in justification is the kashida. Kashidas can be used in almost all connective letters. Here we illustrate the kashida in use with the letter haa'. Fig. 18 shows the letter haa' in its initial form with two different kashida lengths, which differ by almost 3 nuqtas. A parameter is input to the program describing the letter in order to decide on the position of point 10 in relation to 9. For a longer kashida point 10 is moved to the left and also to the bottom. We will see in the next section how kashidas are set to join letters together smoothly.

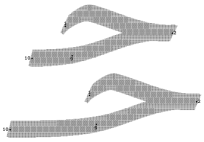

**Figure 18**: The initial form of the letter haa' with two different kashida lengths.

## 7 The formation of words

After modeling the pens and meta-designing individual letters, the next logical step is to join these glyphs together to form complete words. The parameterization of the glyphs allows perfect junction points as if these glyphs were drawn with just one continuous stroke.

In the most widely used font technologies, like OpenType and TrueType, kashidas 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 kashida is static, as is 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 one another.

In our work, the kashida is dynamic and can take continuous values, not just predefined or discrete values. Our experiments with different ways of joining various letter combinations lead us to think that when a kashida is extended between any two letters, it is neither a separate entity nor does it belong to only one of the two letters. Instead it is a connection belonging to both.

Consider for example the simple joining of the two letters, haa' and dal (Fig. 19). Each letter is designed in a separate macro and when used to form the word, the elongation parameter of the kashida in between is passed to both macros, and the kashida is distributed on both glyphs. The two glyphs then meet at the point where the pen stroke moves exactly horizontally (parallel to the x-axis). This junction point is not necessarily at the middle of the distance between both letters. The ending point of each glyph is moved further from its letter, and in order to accommodate long kashidas, these points are moved slightly downwards. Long kashidas need more vertical space in order to curve smoothly, sometimes pushing the letters of a word upwards.

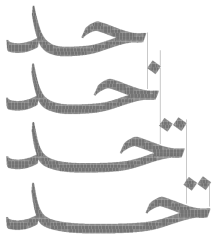

**Figure 19**: Placing a kashida between the letters haa' and dal with different lengths 2, 3, 5 and 7 dot lengths.

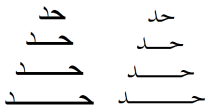

**Figure 20**: Placing kashidas using TrueType fonts available in Microsoft Office 2003.

Other than affecting the ending points, the parameter also affects the curve definition on both sides by controlling the tensions of the paths. The resulting word at many different kashida lengths is shown in Fig. 19, which can be compared with the adding of kashidas using the TrueType fonts available in Microsoft Office 2003 as shown in Fig. 20.

Further examples of joining words can be seen in Fig. 21. This figure shows the word 'yahia' written using different fonts. Notice how the TrueType fonts connect the ligatures with a straight line, and how the OpenType font (bottom left) corrects this by placing curved kashidas. But unfortunately the curved kashidas of this OpenType font are static and do not join well. In the word produced by our parameterized font (bottom right), the letters join perfectly together, and there is also the possibility of freely extending the length between the haa' and the yaa' by any value as done in Fig. 19.

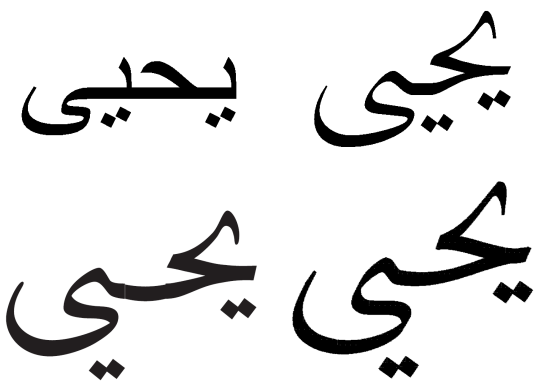

**Figure 21**: The word 'Yehia' as it is written using four fonts. From top left to bottom right: TrueType simplified (no ligatures), TrueType traditional, OpenType Tradigital Naskh, and our own parameterized font.

## 8 Future work

This paper presented new font design ideas that will enable computers to produce Arabic texts of similar quality to the works of calligraphers. The proposed parameterized font will also enable better typesetting, by providing better flexibility to the words. The work covered here 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 much remains to be done.

The proposed idea of producing such an output using computers opens a very vast opportunity for further research in the topic. We classify this possible future work into two categories, research within the font technology itself using METAFONT, and within the typesetting system, TEX.

First, regarding the font design:

- Finalize meta-design of all possible letter forms. Some letters like seen and baa' have a very large variety of forms.
- Develop algorithms and method for placement of dots and diacritics, 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 its use is to improve legibility and understanding. Some diacritics, especially short vowels like fat'ha and kasra change their lengths and inclination, and hence are dynamic.
- Decrease the computational complexity of the current pen modeling techniques.
- Research the possibility of generating output from METAFONT other than the resolution limited bitmapped glyphs for high quality printing or screen viewing. Otherwise, to embed the METAFONT sources within new file formats such as pdf and extend the current pdf viewers to read these sources and use them to produce the correct resolution for the screen or the printer on the fly.
- Design of other writing styles besides Naskh, like Thuluth and Riq'ah, with minimal changes to the already meta-designed font.

Second, the work done in this paper together with the points mentioned above aims at the goal

of providing the typesetting system with more flexibility. The typesetting engine needs some work as well:

- The selection of the most suitable glyph to be placed in a word is a very complicated task. Each letter may have many alternative forms in its specific location in the word, and these alternatives have different widths and heights. Hence, the selection of a certain form is based on many factors; most importantly, justification, and placement of dots and diacritics conflicting with ligatures. The form of the letter may be affected not only by its closest neighbors, but in some cases a letter's form may be changed depending on the fifth or sixth following letter.
- 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 to decide whether an alternative form should be used, a ligature is to be used or broken (including which ligatures are more important than others), or where a kashida is to be added.

**References**


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

[2] Fawzy Salem Afify. *ta'aleem al-khatt al-'arabi*. Dar Ussama, Tanta, Egypt, 1998.

[3] Mohamed Jamal Eddine Benatia, Mohamed Elyaakoubi, and Azzeddine Lazrek. Arabic text justification. *TUGboat* 27(2):137–146, November 2006. Proceedings of the 27th Annual Conference of the TEX Users Group, Marrakesh, Morocco. http://tug.org/TUGboat/Articles/tb27-2/tb87benatia.pdf.

[4] Hossam A. H. Fahmy. AlQalam for typesetting traditional Arabic texts. *TUGboat* 27(2):159–166, November 2006. Proceedings of the 27th Annual Conference of the TEX Users Group, Marrakesh, Morocco. http://tug.org/TUGboat/Articles/tb27-2/tb87fahmy.pdf

[5] Yannis 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*, volume Lecture Notes 1375, pages 138–156, London, UK, 1998. Springer-Verlag. http://omega.enstb.org/yannis/pdf/arabic-simpli98.pdf.

[6] John Douglas 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). http://wwwlib.umi.com/dissertations/fullcit/8602484.

[7] Donald E. Knuth. *The METAFONTbook*, volume C of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[8] Donald E. Knuth. *Computer Modern Typefaces*, volume E of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.

[9] Mahdy Elsayyed Mahmoud. *al-khatt al-'arabi, derasa tafseelyya mowassa'a*. Maktabat al-Qur'an, Cairo, Egypt, 1995.

[10] Thomas Milo. Arabic script and typography: A brief historical overview. In John D. Berry, editor, *Language Culture Type: International Type Design in the Age of Unicode*, pages 112–127. Graphis, November 2002. http://www.decotype.com/publications/Language_Culture_Type.pdf.