ELECTRONIC PUBLISHING, VOL. 6(3), 261–272 (SEPTEMBER 1993)

Analysing character shapes by string matching techniques

JACKY HERZ
Open University, Tel-Aviv, &
Hebrew University, Jerusalem,
Israel

email: jacky@ramon.openu.ac.il

ROGER D. HERSCH
Swiss Federal Institute of Technology
EPFL
CH-1015 Lausanne, Switzerland

email: hersch@di.epfl.ch

SUMMARY

Preliminary attempts at automatic analysis and synthesis of typographic shapes are described. String matching techniques are used to recover implicit relationships between character parts. A knowledge base describing local character shape parts is created and is used in order to propagate local shape modifications across different characters.

KEY WORDS Digital typography Shape analysis String matching Shape similarities Implicit design intentions

1 INTRODUCTION

Characters are carefully designed shapes incorporating both the design ideas of a skilled character designer [1] and the rules related to visual appearance.

The visual appearance of printed characters is the result of many interacting factors: the design intentions of the type designer, the artistic creation of the different character shapes as typeface masters, the font industrialization, rendering and printing processes and, last but not least, the way the characters are perceived by human beings (Figure 1).

In the last decade, the font acquisition, industrialization and rendering processes have been computerized, without however significantly modifying the way in which characters are created. The majority of type designers still hide their design intentions within manual or computer-aided type creations generally represented by contours, and filled using either black ink or a computer.

By presenting the result of the design process as large size master characters, designers *implicitly* state their design intentions through the resulting character shapes. The font industrialization process which follows is aimed at bringing the characters into a form suitable for rendering the shape of the characters on computer-driven typesetters and page printers [2]. The process may involve some uniformization and regularization of the original shapes, thus contradicting the designer's original intention.

Once the character shapes have become computer objects, they are subjected to rendering rules programmed by computer scientists in order to produce fonts of industrial quality on a variety of devices (displays, page printers, typesetters). But since the intentions of the original designer are lacking, programs responsible for producing the characters can only

CCC 0894–3982/93/030261–12

Received 15 August 1993
Revised 1 December 1993

do an approximate job. Even at character design time, the computer is merely used as a drawing system. When creating variations such as fonts of different boldness and shapes, current software can only interpolate between existing, compatible designs (Figure 2).

As was the case in traditional typography, there is a need for producing different character shapes at different point sizes (Figure 3).

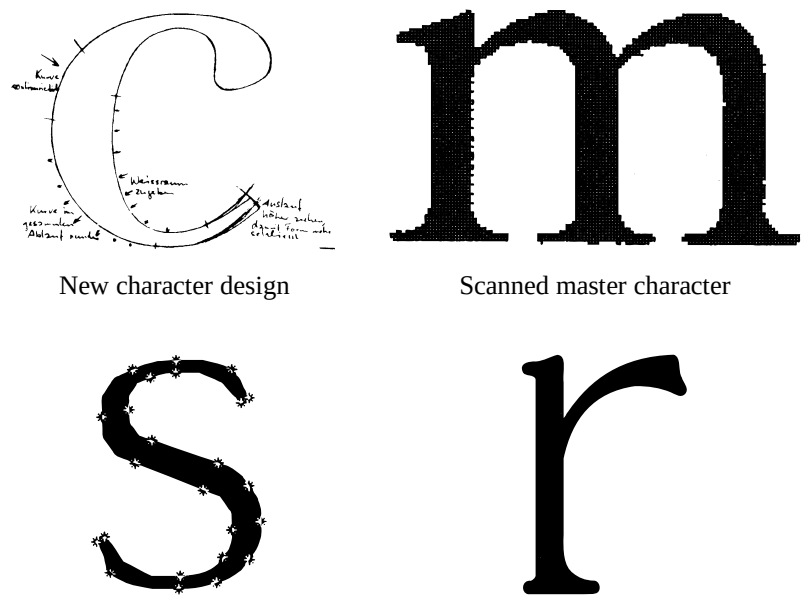

Figure 1. From the design intention to the complete character shape

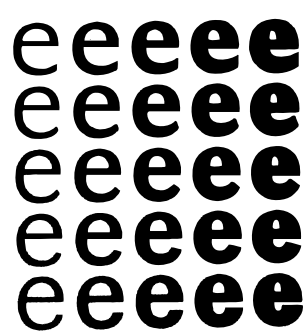

Figure 2. Interpolating between characters (from [3])

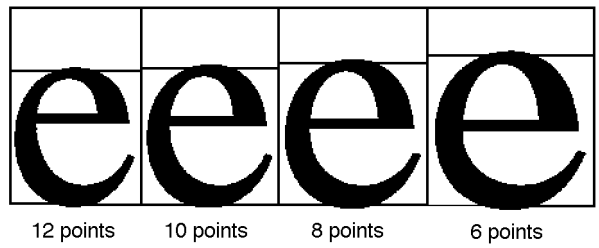

Figure 3. Character shapes which have been optimized for different point sizes

But since current software is not able to retain explicit information about the designer's intentions, character shapes for variable point sizes can only be approximated by interpolation between different character shapes.

Future font design software may, to a certain extent, capture the designer's explicit intentions and therefore provide the basis for automatically generating families of related fonts.

The goal of the present contribution is to explore to what extent implicit similarities [4] in a given font design can be made explicit using string matching techniques. Recovered information about similarities of character shape parts in different characters can be used when modifying representative shape parts, to propagate such modifications to all their instances.

Automatic recovery of similarities may be useful to computer-aided font design in situations where the font designer would like to start from one of his previous designs, modifying certain parts of it to generate a new design [5].

2 REASONING WITH SHAPES

The expression 'Reasoning with shapes' sounds quite bizarre.

Reasoning is the process whereby new facts are deduced from a given set of rules and facts, usually by means of a logical mechanism.

Then what is the meaning of our expression? The main difference between a well-designed group of typographic shapes (e.g. a font) and an arbitrary group of shapes is the cohesion the font expresses in its appearance.

Clearly, there is a logic behind a font design. It is this logic we would like to explore by reasoning.

Unfortunately, unless the font has been defined using an appropriate computer language such as METAFONT [6], this logic remains implicit. This is the reason why grid-fitting systems require explicit knowledge that has to be inserted in addition to the geometric character outline descriptions. This additional knowledge may be expressed by declarative statements such as *hints* as in [7], or can take the form of a general topological description valid for a large group of fonts [8].

Our character shape reasoning system tries to automatically detect some aspects of the logic behind the font design, in order to minimize the explicit human intervention required in font design and rendering systems.

We will show a simplified example of what can be done with typographic shapes using reasoning. We first present a short overview of basic steps in our reasoning system, then

we demonstrate a simple implementation, and we conclude by mentioning some limits of the described approach.

2.1 From contours to strings

Our reasoning system is based upon a syntactic representation of forms [9]. To achieve the translation between contours and strings, we examine all the contours of a given font in order to list repeated attributes. A contour is nothing but a sequence of segments where the last point (chosen arbitrarily) is also the starting point. It can be modelled very well by a cyclic string of syntactic tokens. Each token represents a segment of the contour. Whenever two segments happen to be similar by some pre-established criteria, they are represented by the same token.

Example: In Figure 4, concave segments are denoted by the token `a`, convex segments are denoted by the token `c`, and approximately flat segments are denoted by `b`. Starting at the bottom-left point and turning clockwise we obtain the string `cababccbaccbbcb*`.

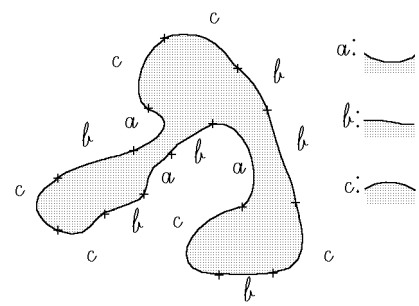

Figure 4. Contour represented by strings (from [10])

Note: We add `*` at the end of a string whenever the string is cyclic.

It is clear that translating segments into tokens makes sense only if we are sure that the geometric description is coherent, i.e. the basic assumption is that similar segments (chosen by some criteria) of different contours share the same attributes. This assumption is crucial to the whole reasoning process.

The string representation of a contour is a common technique in pattern recognition [9]. As such it is very promising: it enables the use of the huge quantity of algorithms available for string treatment [10]. Once we have translated our contours into a group of strings, we can proceed by looking for repeated elements embedded in the description of the contours. But first of all let us define what we are looking for.

- **Definition 1:**
  A *repeated element* `E` in a group of strings `S` is a token sequence that appears in all members of this group.
  Example: (tokens are represented by `a`, `b`, . . .)
  `S` = { `aabccd*` , `dcdaaddcd*` , `ccdaa*` }
  The repeated elements of `S` are { `a` , `c` , `d` , `aa` , `cd` , `da` , `cda` , `daa` , `cdaa` }; note that repeated elements are usually not cyclic (except when `S` consists of a unique cyclic string).

- **Definition 2:**
  A repeated element E of a group of strings S is called a *maximal element* if at least one of its occurrences in one of the strings is not part of another repeated element of S.
  Example: The maximal elements of S are: d (the third one in dcdaaddcd), c (the first one in ccdaa - the second one is not maximal: it is part of cd and cdaa.)
  In most cases we will be looking only for maximal elements. Therefore, we will simply refer to them as *elements*.
- **Definition 3:**
  A *trivial element* is an element which consists of a unique token.
  In the previous example, c and d were trivial elements of S, while ccdaa was a nontrivial element.

When converting a contour into a string, it is important to keep track of the contour segments the tokens actually represent. Each such segment may be represented by its two end points, its control points (for arc segments) and the contour's unique identification. Even if at reasoning time we may ignore this information, it remains crucial for the manipulation of shape parts.

2.2 Extracting elements of a font

Let S = { A,B,C, ...} be a group of n strings, each string representing a contour.

For every couple of strings, we extract its nontrivial elements. Any of the many string pattern comparison algorithms may serve this purpose [11]. The worst-case complexity for a naive algorithm is proportional to the squared length of the compared strings. Thus if the maximal length of a string is L, then the complexity of extracting all elements for each contour couple is $O(L^2 \times n^2)$.

The result of this operation may be summarized in the following table:

Contours	*Element*
A, B, D, ...	E1
A, C, D, ...	E2
...	

This table contains for each element the list of all the contours it belongs to.

At this stage we create a dependency graph for the elements we have found so far (Figure 5). A dependency graph is a graph where in every vertex V_i we keep the element E_i

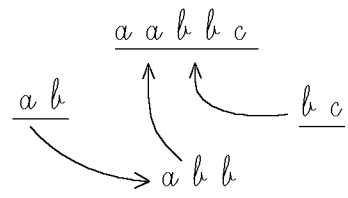

Figure 5. A dependency graph

and where, for each element V_p that is a substring of another element V_q, there is a unique path from V_p to V_q.

We build this graph in two stages:

1. For each string in V_p that is a substring of another string in V_q, we add an arc from V_p to V_q.
2. We delete all redundant arcs. An arc from V_i to V_k is considered redundant if in the graph there is already another path from V_i to V_k.
   For example, in Figure 5, the arc from `ab` to `aabbc` has been removed since these elements were already connected in the same direction via `abb`.

The dependency graph is the tool we use when we want to propagate a substitution made upon an element `E` to all other elements that contain `E`. In this way, we maintain the coherence of the font outlines whenever any of its elements are redesigned (manually or even automatically, as could be the case for grid-fitting purposes). Let us proceed with a concrete character shape modification case.

2.3 Character shape modification example

Assume we have a font consisting of the Times-Roman outlines of the letters E, H, L. The partition of the outline into segments, and the choice of the control points, are given in advance. As we said before, we assume that this geometric description is not arbitrary but that it reflects the structure of the font.

In this example we use an expression such as X_k to denote the *length* of the segment starting at point k (and ending at the next point) in the contour X. The operator $\rightarrow$ indicates a token assignment. Thus $X_k \rightarrow$ `a` signifies that the contour segment X_k is represented by the token `a`.

Our string translation criterion is based upon the length of the segments: *only equal-length segments are translated into the same token*. This is by no means the unique possible criterion. We choose it for the sake of simplicity.

Formally: let X, Y be a couple of distinct contours in a font.

$$\text{if } (X_i = Y_j) \text{ and } (X_i \rightarrow \texttt{a}) \text{ then } Y_j \rightarrow \texttt{a}$$
$$\text{and if } (X_i \mathrel{!}= Y_j) \text{ and } (X_i \rightarrow \texttt{a}) \text{ and } (Y_j \rightarrow \texttt{b}) \text{ then } \texttt{a} \mathrel{!}= \texttt{b}$$

Following these rules we obtain the corresponding strings for E, H and L, starting from the bottom leftmost point, as shown in Figure 6. Looking at this example, it may seem that equal-length segments have distinct tokens. In reality these segments are different, but this is hardly noticeable because of the figure's size.

E= `acdcagrastueklawalkemnopvg`$*$
H= `acdcabacefecabacdcabacefecab`$*$
L= `acdcabacdxopvg`$*$

Note that even at a glance we can see that the strings have well-structured internal patterns. For instance they each contain at least one palindrome (a string read equally in both directions) such as `klawalk` in E, `cabac` in H, `acdca` in L etc. This phenomenon reflects the highly structured nature of the design itself.

Returning to our method, let us extract elements. We first write the couple to be compared and then list the extracted elements.

E, H: `acdca`
E, L: `opvgacdca`
L, H: `acdcabac`

Since we have one unique element in each list, this is also the element table. For this example, the resulting dependency graph is very simple (Figure 7):

Any modification of any part of the segment sequence represented by the string `acdca` (or any of its sub-strings) will automatically be propagated throughout the font. For instance if the letter E is modified as illustrated in Figure 8, then the letters H and L will follow the same nonlinear transformation.

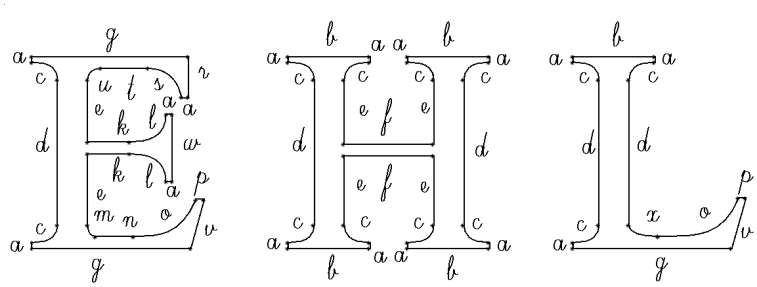

Figure 6. The contours of the font represented as cyclic strings

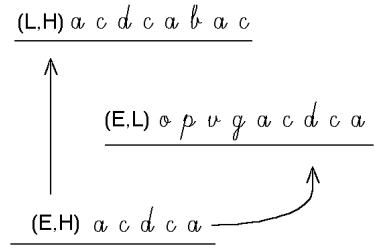

Figure 7. In brackets: the contours containing the element

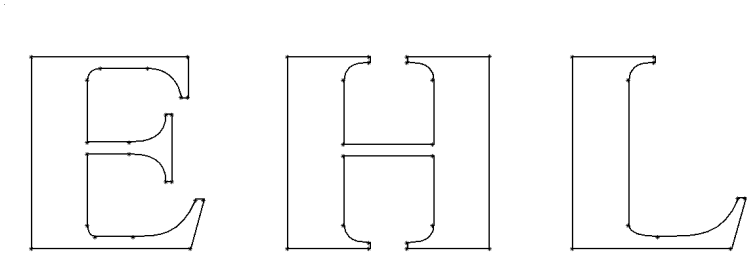

Figure 8. A modification of E *is propagated to* H *and* L

Note that a modification of a segment sequence represented by a pattern like vga leads to an alteration of E and L but not of H (Figure 9).

In Figure 10 we see how the same method is used to transform a sequence of Hebrew characters with serifs to a similar sanserif sequence.

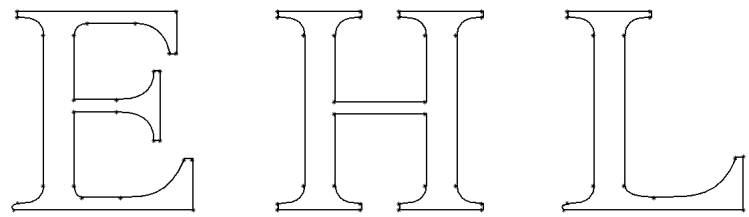

Figure 9. A modification in E *is propagated to* L *but not to* H

3 BIDIRECTIONAL ANALYSIS

With the exception of special kinds of strings such as cyclic ones or palindromes, strings always have a direction. When using strings to identify elements, this property should be considered. Using strings together with a distance unification criterion should provide easy detection of isometrics like rotations and translations. In order to obtain reflections as well, just invert the order of one of the strings that is being compared. Thus, any string comparison should be made twice: first in the normal order and secondly after one of the strings has been inverted (Figure 11).

4 TWO-DIMENSIONAL ELEMENTS

Every representation has its inherent limits. Representing elements as strings enables the recognition of one-dimensional elements. These elements correspond to the basic structure of strings, which are just directed chains of symbols. At a first glance this fact restricts the usefulness of this representation, because we are dealing with typographic shapes which are two-dimensional objects. Fortunately, there are some exceptions.

We have already encountered such a case: cyclic strings represent elements that have closed contours, thus representing the two-dimensional area enclosed within its frontiers. The discovery of two-dimensional elements within a given set of shapes adds much more power to a mechanism for coherent font processing: one-dimensional elements are used to propagate modifications carried out upon the element itself, thus affecting only a contour segment. Any modification within the frontiers of the two-dimensional element may be propagated to all its occurrences within the font, even when the frontier itself remains untouched (Figure 12). In two-dimensional elements the contour segments, as well as the entire internal area, may be propagated over the whole font.

The former case (Figure 12) is a trivial one. Obviously, two distinct contours that have exactly the same form should be treated equally. However, combining string comparison with some topological knowledge (see Definition 4) which may be deduced from the outline enables us to recover less trivial two-dimensional elements.

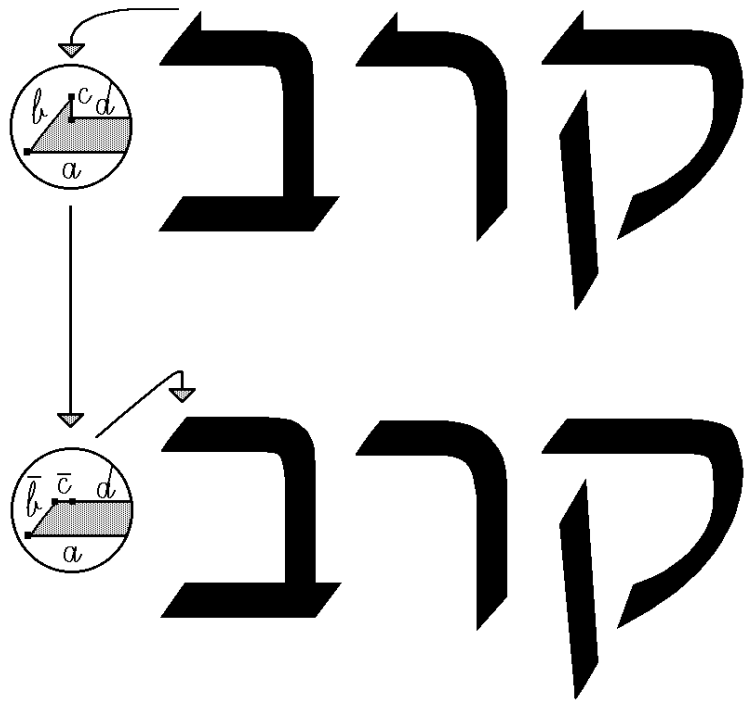

*Figure 10. An element (here, a serif) of several Hebrew characters (*Bet, Resh *and* Kouf*) is discovered, modified, and used to create a new coherent version of the same characters*

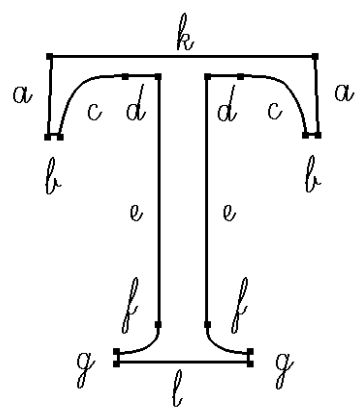

Figure 11. Reflected elements represented by `abcdefg` *and* `gfedcba` *are identified using a comparison between a string and its inversion*

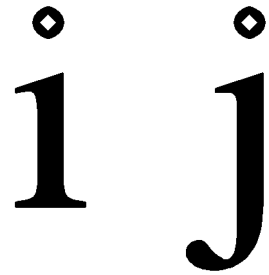

Figure 12. The modification of the internal area of the dot in i *leads to the same modification in the letter* j

- **Definition 4**
  Let $\mathtt{E} = \mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n$ be an element in a group of typographic shapes.
  E is a *two-element* of this group if and only if the (possibly null) straight line segment traced from $\mathtt{a}_1$ to $\mathtt{a}_n$ never crosses any contour.

Definition 4 concerns any kind of element, but in most cases we will be interested in the recognition of maximal elements which meet the requirements of Definition 2. Note that while every two-dimensional element is also a one-dimensional element, the inverse is not necessarily true.

In consequence a maximal two-dimensional element is not necessarily a maximal one-dimensional element. For example, in Figure 6 the string cdc is a white two-dimensional element, but cdc does not represent a maximal one-dimensional element because it is part of the one-dimensional element (only) represented by acdca.

There are only 3 cases which meet the requirements of Definition 4: The line $\overline{\mathtt{a}_1,\mathtt{a}_n}$ may be entirely within the painted area of the character (1), or entirely within the white area (2), or $\overline{\mathtt{a}_1,\mathtt{a}_n}$ is a null segment (3).

In the first case the two-dimensional element will be defined as a black element, while in the second case we have the occurrence of a white element (Figure 13).

When $\overline{\mathtt{a}_1,\mathtt{a}_n}$ is a null segment, the colour is defined by the adjacent interior colour. Usually, input data is given with contour orientations determining the interior colour [7]. If this is not the case, the colour of an area within a shape can be found by using parity fill techniques [12].

Any shape modification that is included entirely within the frontiers of a two-dimensional element can be propagated immediately to all its occurrences within a given font. In

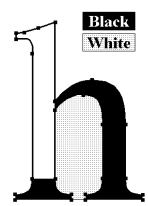

Figure 13. An example of 'black' and 'white' elements in the outline of the letter h

Figure 14 the modification of a white element in the letter h has been propagated to the letters n and m.

In many cases (such as in Figures 6, 10, 12 or 13) two-dimensional elements may also be viewed as structural elements, in other words also as basic typographic structures such as serifs, arms, or stems which are replicated throughout an entire font.

Figure 14. The modification of a white element in the letter h *has been propagated to the letters* n *and* m

4.1 Some final remarks

Up to now, the propagation of element alterations has always occurred from bottom to top. Any modification made to an element E was reproduced in all the elements that E belonged to. Is this propagation possible from top to bottom (for instance from the string abc to the string ab)? Unfortunately, in most cases, we cannot provide an automatic way to do it. The reason is that, while the boundary points of ab are included in abc, the inverse is not necessarily true. For example, if the sequence of the 3 segments represented by abc is transformed into a unique segment, we usually lose the information about the exact transformation of each component: a, b, c, ab and bc.

In geometric terms, endpoints of recognized elements are not allowed to move, and this limits supported shape manipulations.

Nevertheless, as the interdependence is known, in interactive systems the user can be notified about the possible consequences of modifications made to a contour which contains recognized elements.

As seen from the example above, the translation procedure between contours and strings depends heavily upon useful criteria.

Appropriate criteria can be attributes of individual contour segments represented as distinct symbols, e.g. length, orientation and/or curvature profile.

Such criteria may be suggested by type specialists as well as by programmers.

Once they are established, the reasoning system tries to ensure coherent contour manipulation.

5 CONCLUSIONS

Most known shape manipulation techniques originate from the fields of image processing and pattern recognition. In these fields, one tries to extract contour descriptions and features from pixmap images in order to classify and recognize objects. In the field of digital typography, one may start either with character descriptions incorporating explicit design knowledge [6] or with character outline descriptions having implicit design features.

This contribution shows how string analysing and matching techniques can be used to recover information about similar character parts. However, only modifications that are local to considered shape parts are successfully applied and propagated. Further research is necessary in order to express how shape parts may be joined together and how modifications of shape part extremities may affect neighbouring shape parts.

REFERENCES

1. R. Southall, 'Character description techniques in type manufacture', in *Raster Imaging and Digital Typography II*, eds. R. A. Morris and J. André, pp. 16–27. Cambridge University Press, (1991).
2. R. D. Hersch, 'Font rasterization, the state of the art', in *Visual and Technical Aspects of Type*, ed. R. D. Hersch, 78–109, Cambridge University Press, (1993).
3. G. Noordzij, 'The shape of the stroke', in *Raster Imaging and Digital Typography II*, eds. R. A. Morris and J. André, pp. 34–42. Cambridge University Press, (1991).
4. D. Adams, 'abcdefg, a better constraint driven environment for font generation', in *Raster Imaging and Digital Typography*, eds. J. André and R. D. Hersch, pp. 54–70. Cambridge University Press, (1989).
5. Hans Ed. Meier, 'On the design of Barbedor and Syndor', in *Visual and Technical Aspects of Type*, ed. R. D. Hersch, 148–164, Cambridge University Press, (1993).
6. D. E. Knuth, *The* METAFONT*book*, Addison-Wesley, Reading, MA, 1986.
7. Adobe Systems Inc., *Adobe Type 1 Font Format*, Addison-Wesley, 1990.
8. R. D. Hersch and C. Bétrisey, 'Model-based matching and hinting of fonts', *Proceedings SIGGRAPH'91, ACM Computer Graphics*, **25**, 71–80, (1991).
9. R. C. Gonzales and M. G. Thomson, *Syntactic Pattern Recognition*, Addison-Wesley, Reading, MA, 1978.
10. L. Miclet, *Méthodes structurelles pour la reconnaissance des formes*, Eyrolles, Paris, 1984.
11. J. W. Hunt and Th. G. Szymans, 'A fast algorithm for computing longest common subsequences', *Communications of the ACM*, **20**(5), 350–353, (1977).
12. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics — Principles and Practice*, Addison-Wesley, New York, 1990. pp. 964–967.