ELECTRONIC PUBLISHING, VOL. 6(3), 133–143 (SEPTEMBER 1993)

Coordinate-independent font description using Kanji as an example

MARTIN J. DÜRST

Multimedien-Laboratorium,
Institut für Informatik der Universität Zürich,
Winterthurerstrasse 190,
CH–8057 Zürich, Switzerland

email: mduerst@ifi.unizh.ch

SUMMARY

Abstract, font-independent character descriptions are important for a systematic approach to automated and semi-automated font design. This is particularly so for large character sets such as Kanji. The paper defines a completely coordinate-independent notation for Kanji, which contains all the necessary information to produce legible character sketches.

KEY WORDS Abstract character description Coordinate- independent font Large fonts Kanji Prolog

1 INTRODUCTION

In his comment on Knuth's METAFONT [1], Hofstadter [2] envisions that a future-generation font design system should consist of a) an abstract description of characters in terms of roles such as 'crossbar'; b) 'an ability to generalize from a few letterforms . . . to an entire typeface', and c) 'an integration of perception with generation'.

Intermixing of structure and parameter values leads to tools for the implementation of fonts and metafonts based on the concepts of procedural, functional, and structured programming [1,3,4]. However, to exploit the benefits of object-oriented programming techniques [5], such as inheritance and flexible code reuse, requires the strict isolation of abstract structural character description (as a kind of superclass) from which concrete fonts can be derived.

The research in structured character description has a long history; already Coueignoux [6] developed a grammar for the family of Roman typefaces. His rules were so detailed that they implicitly described all characters of the alphabet while excluding any others. The abstract structure of the Roman alphabet is also studied and used in recent work [7–9]. Using more details and coarse coordinate grids, Hersch and Bétrisey [10,11] fit contours to characters of different fonts for automatic hinting.

1.1 Requirements for abstract character descriptions

The requirements for an abstract, font-independent character description can be summarized as follows:

CCC 0894–3982/93/030133–11
Received August 16 1993
Revised December 3 1993

1. *Compactness*: A character description should not contain more information than really necessary. This excludes font-specific information and coordinates.
2. *Completeness*: There must be enough information so that the character can be identified. This can be specified more exactly as follows:

   (a) *Productivity*: It must be possible to produce legible graphical sketches from the abstract representation without additional information. Such sketches serve as a kind of proof of the completeness of the representation. Legibility here only means that the sketches can be identified, and does not imply any aesthetic criteria.
   (b) *Separability*: Characters considered different in a given character set must differ in their abstract representations. The criteria for separating characters are not completely uniform (see subsection 5.2 for an example). Combined with the requirement for compactness, this implies that abstract character descriptions have to provide a flexible degree of detailedness.
   (c) *Derivability*: Enough information should be present to allow font implementations to use rules such as 'on a free bottom end of a stem, there is a double serif'.

1.2 Coordinate-independence

A specific property of an abstract character description is its coordinate-independence. Coordinate-independent or coordinate-free approaches are popular in mathematics because they allow abstraction from the invariants of orientation, translation, and scaling. Still in this case exact geometry is retained.

In font design, coordinate-independence serves abstraction in a similar way. However, what is ignored is not the coordinate system, but the exact, font-dependent coordinate *values*. In general, a character turned 180 degrees is not the same character any more. Thus coordinate-independence for font design has to retain concepts such as *top*, *bottom*, *left*, and *right*, but is otherwise more related to topology than to geometry. For the use of the term topology in this context, see also [11, p. E-1].

1.3 Large character sets

Developing abstract, structured character descriptions is much more rewarding for large character sets than for small ones. There, the importance of structure relative to exact geometry is higher than for small character sets. Theories, techniques, and tools developed for large character sets can later be used for smaller character sets.

For large character sets, the reuse frequency of the basic graphic elements is much higher than for small character sets[1]. Indeed, the reuse frequency is higher than in most other CAD application areas, so that font design for large character sets can also serve as a case study for structural aspects in CAD in general [13].

Also, the broader statistical base can facilitate the investigation of the relation between geometry and perception. The work of Uchio et al. [14], who normalized character size based on area and complexity, can be seen as an early attempt in this direction. In any

[1] For actual frequencies, see e.g. [12]

case, generalization from say 1,000 characters to 20,000 characters should be considerably easier than generalization from say 10 to 26 characters.

The largest character set in the world is formed by the common East Asian ideographic characters, called Hanzi in China, Kanji in Japan, Hanja in Korea, and written 漢字 in all countries. In this paper, the Japanese term Kanji is used. Although knowledge of Kanji is advantageous for reading the paper, it is in no way necessary.

Whereas there is an abundance of fonts for the Roman script, there still is a shortage of high-quality Kanji fonts for computers. This problem is aggravated by the extension of the character set. Whereas current implementations in Japan contain about 6,000 Kanji, the new 16-bit universal character encoding standard Unicode/ISO10646 [15] increases their number to 21,000. The design of a typeface for such a huge number of characters is virtually impossible without the aid of automation.

1.4 Overview of the paper

Section 2 gives an overview of the *box-bar model*, our way to view the structure of Kanji. The coordinate-independent notation for the bar layer is introduced in Section 3. Section 4 presents the algorithm that produces character sketches from the coordinate-independent notation. Extensions to the notation and the algorithm are discussed in Section 5.

2 THE GLOBAL STRUCTURE OF KANJI

For any script, there is a broad informal consensus about how to describe characters. This is particularly so for Kanji, as it is necessary in daily life to describe unusual Kanji, for example, to explain the writing of a name over the phone, or to look up a character in a dictionary. In our formalization, the description of Kanji is split into two layers and is called the box-bar model [16]. Similar structures can be found in most other work on Kanji, but in many cases these layers are not clearly separated, and their constituents receive vague or misleading names such as ‘basic element’.

2.1 The box layer

The greatest number of Kanji are combinations of two parts that are themselves Kanji. These two parts are in most cases placed one besides or atop the other, and adjusted to fit into a square area. As this composition principle is similar to Lego bricks or TEX boxes [17], we call this the *box layer*. Some, but not all, of these parts are so-called radicals (部首). The traditional 214 radicals are distinguished characters that serve as division heads in dictionaries. As only one part of a Kanji has to be a radical, 214 or even fewer radicals are enough for indexing. A structured description, however, has to specify all components of a character, and therefore uses many more than 214 boxes.

A notation for the box layer, based on the principle of direct character representation and written using the logic programming language Prolog [18], has been described in [16] and is briefly reviewed here. Prolog programs have both a declarative and a procedural interpretation. The same simple notation is used for programs (rules) and data (facts). The set of all facts is also called the database. Any identifiers, even Kanji[2], can be used as symbols.

[2] This depends on the implementation. We use Quintus™ Prolog release 3.1.

Direct character representation means that characters represent themselves. This is in contrast to using numbers as in [19] or names as in [20]. In [4], direct character representation is used for most characters, but for parts such as radicals, Japanese terms are used that are not understood in China or Korea. Of all these representations, direct character representation requires least memorization, is language-independent, and most resembles the finally desired graphical representation. On the other hand, the character set has sometimes to be extended by hand to account for forms that are used as boxes, but not as Kanji.

We take the character 驚 to show how we represent Kanji on the box level. The following four facts of the database describe this character:

```
ch(驚,comp(敬,=,馬)).
ch(敬,comp(苟,‖,攵)).
ch(苟,comp(艹,=,句)).
ch(句,comp(勹,┐,口)).
```

The relation `ch(Character,Description)` associates a single character with its description. This description can recursively reference other character descriptions. The functor `comp(Box1,Operator,Box2)` describes the composition with two boxes and an operator. The operators ‖ and = stand for horizontal (side-by-side) and vertical composition. Other operators, mainly self-describing, are ⌐, ⌙, ¬, ⌟, ⊂, ∪, ∩, and ⊃. Besides the `comp`-functor, there are other functors that describe duplication and triplication of parts, as for example in 森, and other, rarer compositions.

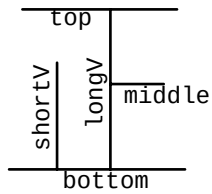
For many applications, the box layer is sufficient without any additions. An example is the deduction of the radical of a character according to the rules of Nelson [21]. This was used to show how alternative representations of a character can be generated [16]. Whereas a description should be concise and therefore should not contain alternative representations, such representations are necessary to support different views or variants of a character that can be expressed simultaneously or alternatively in different fonts.

It is obvious that the box layer is coordinate-independent. Also, the requirements of Section 1.1 are fulfilled if we assume that finally for every box there is an adequate description. An application area for the box layer in Roman characters is diacritics, especially multiple ones. For other writing systems, such as Khmer [22], a representation similar to the box layer can also be very useful.

2.2 The bar layer

On a lower level, Kanji cannot be divided into conceptually separated, individually scaled boxes. It becomes necessary to describe the individual lines and their intersections to define characters abstractly. We call these lines *bars*.

Bars are not identical to strokes (画), their traditional equivalent. Strokes are the elements of a Kanji that are written without lifting the brush in standard style. Individual strokes are easily visible in some fonts, but can disappear completely in other fonts, especially in sanserif styles. A stroke is composed of more than one bar if it has sharp corners. As an example, the character 凸 (convex) consists of eight easily identifiable bars, but is written with five strokes only. The four bars on the top and right side are drawn with one stroke. As the rules to draw a character are fairly regular, strokes can be extracted again from a bar-based representation, and be used for font design as well as other applications like computer-aided instruction and dictionary lookup.




```
ch(正,bars([
      bar(longV, |,[top,middle,bottom]),
      bar(middle,一,[longV,empty]),
      bar(shortV,|,[empty,bottom]),
      bar(top,   一,[empty,longV,empty]),
      bar(bottom,一,[empty,shortV,
                     longV,empty])])).
```

Figure 1. Definition of character 正

Uehara *et al.* [12] use the term *substroke*, and Tanaka *et al.* [4] the term *element* for a concept similar to bars, whereas Hobby & Goan [20] just invariably speak of strokes. Hofstadter [23, p. 295] and Zeng *et al.* [19] do not divide strokes.

3 A COORDINATE-INDEPENDENT NOTATION FOR THE BAR LAYER

In the bar layer, it is much more tempting to rely on coordinates than in the box layer. Even Tanaka *et al.* [4], the only group we know that until now did not use coordinates in the box layer, make use of coordinates in the bar layer.

For characters whose strokes are connected, it is mainly the orientation of the bars and their intersections that allow the reader to distinguish one character from another. In this section, we describe our notation for such intersections. Section 4 will explain how this notation can be converted to a visible form. The relative length of the bars is in general not of importance. Therefore its treatment is delayed to subsection 5.2. To explain the notation, the character 正 (right, correct) is used. Its description is shown in Figure 1.

The relation `ch(Character,Description)` is the same as in subsection 2.1. This time, the `Description` is the functor `bars(Barlist)`, which indicates that 正 is described in terms of a list of bars. Lists in Prolog are enclosed in `[` and `]`.

Each bar is described by a functor with the name `bar` and three terms, the name of the bar, the type of the bar, and an intersection list. The bar names are arbitrary symbols that are used to identify the bar locally inside a single description. The bar type is given in direct character representation and is used mainly to deduce bar inclination. The third argument is a list of the other bars that are intersected. The special symbol `empty` indicates an end of a bar without intersection, i.e. a free end.

The sequence of the intersections is fixed: top to bottom in general, and left to right for horizontal strokes. This is identical to the traditional drawing direction, which simplifies writing character specifications. As a whole, the representation is easy to write, and can also be output by a program that allows character abstractions to be drawn interactively and graphically.

4 CONVERSION TO CHARACTER SKETCHES

This section presents an algorithm to convert the descriptions introduced in Section 3 into character sketches according to the requirement of productivity defined in subsection 1.1.

4.1 Constraint programming

The notation introduced in Section 3 can be interpreted as a system of constraints. There is a large amount of literature on constraint programming and constraint solving, especially also with respect to geometric problems. As an entry point to this field, the interested reader may start with [24].

However, several problems occur when applying an existing constraint solving system. First, our constraints are highly underdetermined, as there is no exact geometrical information. Second, some constraints are implicit and difficult to formulate in a standard system, such as the condition that a Kanji has to fit into a square of a given size. On the other hand, a general constraint-solving system is much more powerful than necessary.

Therefore, a special constraint-solving algorithm tailored to the problem was developed. The main idea of the algorithm is to divide constraint solving into two subproblems, one along the *x*-direction (horizontal) and the other along the *y*-direction (vertical).

4.2 Extracting intersection points

The first step of the algorithm combines the information on different bars and assembles a list of intersection points. First, a list of its intersection points is built separately for each bar. Intersection points here and in the following include `empty` endpoints. Each point carries its bar's information, a list of crossing bars, if any, and its ordinal position on the bar. Ordinal positions are written in the form `2/3`, read 'second of three'. The `top`-bar of 正, for example, results in the three points

```
pt(top,一,[],     1/3).
pt(top,一,[longV],2/3).
pt(top,一,[],     3/3).
```

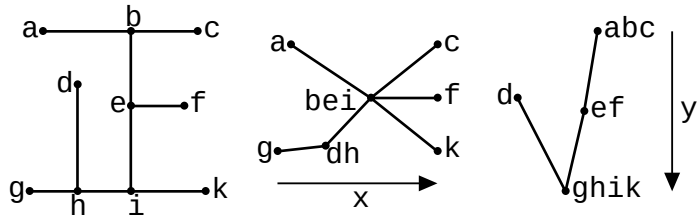
Next, corresponding points on intersecting bars are merged by matching a point on bar `A` intersecting bar `B` with a point on bar `B` intersecting bar `A`. The first point on the `longV`-bar, described as `pt(longV,|,[top],1/3)`, is merged with the second point on the `top`-bar, and the resulting point is described as `mpt([(top,一,2/3),(longV,|,2/3)])`. Altogether, 正 has ten such points, denoted `a` to `k` in Figure 2.

4.3 Building partial orders

This and the next subsection describe operations that are carried out separately for both the *x* and the *y* directions. First, for each direction, a partial order is constructed. This partial order expresses the constraints on the ordering of the intersection points that can be deduced from the bar notation. The two partial orders for the character 正 are shown as graphs in Figure 2.

The partial order is built as follows: points that are connected by a bar running orthogonal to the direction in question are unified in equivalence classes. So the points `b`, `e`, and `i`, which all lie on the `longV`-bar that runs orthogonal to the *x*-direction, form a single equivalence class in Figure 2b.

The sequence of the intersection points on the remaining bars serves to define the relations between the equivalence classes. So, for example, the group `dh` is displayed left of the group `bei` and connected to it in Figure 2b, because `h` and `i` appear on the `bottom` in this sequence.



(a) (b) (c)

Figure 2. Character 正; (a) intersection points; (b) partial order in x-direction; and (c) partial order in y-direction

The two partial orders completely express the conditions on the *x* and *y* coordinates of the points of the character. Points in the same equivalence class have the same respective coordinates, and a line in the graph between two equivalence classes expresses an ordering constraint between the respective coordinates of the classes.

4.4 Building total orders

This step is again carried out separately for each direction. On the partial order derived in subsection 4.3, for a second time, equivalence classes are built, and then total orders are generated. These total orders are possible total orders for the coordinate values of the sketches that we want to generate. The equivalence classes are due to the fact that the same coordinate value can be assigned to different elements of the partial order of subsection 4.3. In the algorithm, the building of equivalence classes and the final total ordering are intertwined and therefore will be described together.

In most cases, there are several possible total orders for each partial order. For Figure 2c, for example, the following five total orders exist: `abc=d<ef<ghik`, `abc<d=ef<ghik`, `d<abc<ef<ghik`, `abc<d<ef<ghik`, and `abc<ef<d<ghik`. For Figure 2b, there are 65 possible total orders.

Using the generate-and-test mechanism of Prolog, the different total orders are subsequently generated for each direction. Priority is given to orders with a low number of equivalence classes, because there is no need to choose different coordinate values if there is no additional information. Doing otherwise would mean that additional information is introduced that may be font-dependent.

Once a total order for a given direction is found, the integers 0,1,2 ... are assigned as coordinate values, so that intersection points are equally spaced in this direction. This again can be justified as being the simplest solution without additional assumptions. These values are later scaled appropriately for drawing so that the character shape fills a square.

With increasing number of elements of a partial order, the number of corresponding total orders can grow exponentially. This limits the complexity of the characters that can be sketched. We are currently working on improvements based on algorithmic and geometric considerations.

4.5 Avoiding undesired intersections

The algorithm, as explained above, leads to readable results in many cases. However, one fact has been ignored up to here. The notation introduced in Section 3 not only defines all

the intersection points of a character, but also contains the implicit assumption that there are no other intersection points. Therefore, the generated solutions are tested, and those with undesired intersections rejected. Prolog then automatically generates additional solutions with its backtracking mechanism. As an example, the total order `abc=d<ef<ghik` for Figure 2 means that the point `d` lies on the `top`-bar, resulting in 正, which is clearly unacceptable. The second total order with only three groups, `abc<d=ef<ghik`, then produces 正, having no undesired intersections. Therefore, this solution is accepted and drawn after appropriate scaling.

5 EXTENSIONS

5.1 Additional bar types and multiple intersections

Until now, only the bar types 丨 and 一 have been used. However, the algorithm has no problems dealing with slanted bars. The two basic varieties are 丿 and 乀. For 丿, there is a variant 丿, which behaves like 丿 in the last segment, but like 丨 in the first few segments. Segments are parts of bars that lie between two consecutive intersection points. This is the place where the ordinal position information extracted in subsection 4.2 is used. The bar type 丿 is used in characters such as 大 and 夫, whereas 丿 is used in characters such as 文. For simplicity, straight lines are used to draw the segments of slanted bars.

A variant for 丨, namely 亅, allows characters such as 干 and 于 to be distinguished. Although it is rare that this feature[3] is the only distinction between two characters, discriminating on the abstract description level is advisable because the feature is expressed consistently in all but a very few fonts.

For intersection points involving more than two bars, as at the centre of 大, the notation in Section 3 is extended. The horizontal bar of 大 for example is then described as `bar(horizontal, 一, [empty,[leftSlanted,rightSlanted],empty])`. Transitivity of the intersection relation is exploited so that a list of simultaneously intersected bars has to be given only for one of the bars.

5.2 Hidden bars and constraints

Usually, there is no need to specify the relative length of unconnected bars, as this may affect readability but will not influence identification. There are however some examples where this is different, such as in the pair 土 (earth) and 士 (knight). These are resolved by introducing bar types with special properties.

The bar type `xGreater`, for example, extends in the *x*-direction like a horizontal bar, but is ignored with regard to the *y*-direction. This results in additional restrictions for the partial order in the *x*-direction, without affecting the partial order in the *y*-direction. In addition, an `xGreater`-bar is *hidden*, i.e. it does not produce any drawn segments. Connecting two intersection points by an `xGreater`-bar thus serves to express the constraint that of their *x* coordinates, one is greater than the other.

5.3 Derived definitions

The character 三 (three) is an example for another application of hidden bars. It can be derived from 王 (king) by hiding the centre bar. Arbitrary bars are hidden by qual-

[3] Called *hane* in Japanese.

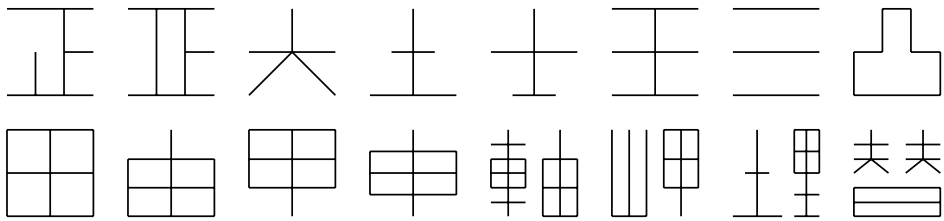


Figure 3. Sketches of (first row) 正, 正 *before intersection test,* 大, 土, 士, 王, 三, 凸, *and (second row)* 田, 由, 甲, 申, 軸 *(axis),* 岬 *(cape),* 埋 *(bury), and* 替 *(replace)*

ifying the bar type with the functor `hidden`, as in `bar(center, hidden(|), [top,middle,bottom])`. Instead of writing the whole character definition twice, once for 王 and another time for 三 with the additional `hidden` functor, it is much more convenient to derive the definition for 三 from the definition for 王, using the notation `ch(三,hide(王,center))`. This expresses that 三 can be derived from 王 by hiding the bar with the name `center`.

There are many more possibilities for derived definitions. One of them is bar extension, another is bar identification. The following definitions should be self-explanatory:

```
ch(由,extend(田,center,start)).
ch(甲,extend(田,center,end)).
ch(申,extend(由,center,end)).
ch(斉,identify(立,bottom,可,top)).
```

In derived definitions, bar names are used for reference from outside the local scope of a character definition. When two definitions are merged to a single one (e.g. 斉), the program that performs the derivation also has to qualify the local names to avoid name clashes.

5.4 Results

Figure 3 shows sketches for characters of varying complexity. In the case of the last four characters, the bar layer has been integrated with the box layer. This is not overly difficult, especially in the case of the operators ‖ and =.

Overall, the notation introduced in Section 3 is clearly very compact. That this notation satisfies the requirements of productivity and separability has been shown in this section. Derivability has not been shown explicitly, but poses more technical than fundamental problems.

To give an additional impression of the working of the algorithm and the resulting sketches to readers who are not familiar with Kanji, we have used our system to define the fifteen capital letters of the Roman alphabet containing only straight lines. The result is shown in Figure 4. Please note that Figures 3 and 4 do not claim any aesthetic quality; this is not required on the level of abstract, font-independent character descriptions.

Figure 4. The capital letters of the Roman alphabet that contain only straight lines

6 CONCLUSIONS AND FUTURE WORK

The use of an abstract, font-independent character description was advocated especially for large character sets such as Kanji. A corresponding notation was introduced, and an algorithm to produce legible sketches from this notation was developed.

For the box layer, our database is mostly complete [16], but not yet in the polished state necessary for public release. We are presently increasing the number of defined characters in the bar layer, thereby further refining and testing our notation. Of course, any formal notation for character structure has its limits, in a similar way that parameterizations for metafonts have their limits [2]. However, the generation of new characters is proceeding along much more predictable lines and much more slowly.

Using abstract character descriptions as a data source for existing tools and systems working on lower levels is possible. However, to fully exploit the advantages of abstract descriptions, a closer integration of tools on different levels is needed. This will open new ways of working with fonts, such as the structured reimplementation of existing high-quality fonts. Comparing the character sketches or any intermediate results to the original font, differences can be accumulated along the structure of the character set and can indicate those components of the structure that are responsible for the largest differences. Such an approach guided by abstract character descriptions promises to lead to a much faster convergence to the final design.

ACKNOWLEDGEMENTS

My thanks go to Peter Stucki for providing a fruitful working environment, to Tamami Loosli for the idea of hidden bars, to Tomoko Klopfenstein-Arii for advice on Kanji history, to Jacques André for his kind collaboration when testing inline Kanji, to the reviewers for their valuable comments, and to my colleagues and friends for interesting discussions and proof-reading.

REFERENCES

1. Donald E. Knuth, ‘The concept of a meta-font’, *Visible Language*, **16**(1), 3–27, (1982).
2. Douglas R. Hofstadter, ‘Metafont, metamathematics, and metaphysics’, *Visible Language*, **16**(4), 309–338, (1982).
3. Yunmei Dong and Kaide Li, ‘A parametric graphics approach to Chinese font design’, in *Raster Imaging and Digital Typography II*, eds. Robert A. Morris and Jacques André, 156–165, Cambridge University Press, Cambridge, (1991).

4. Tetsurou Tanaka, Yuuichirou Ishii, Kenji Nagahashi, Mikio Takeuchi, Hideya Iwasaki, and Eiiti Wada, 'Kanji skeleton font creation support system', in *Proceedings of the 32nd Programming Symposium*, ed. Nobuo Yoneda, pp. 1–8, 1991. (in Japanese).
5. T. Korson and J. D. McGregor, 'Understanding object-oriented: A unifiying paradigm', *Communications of the ACM*, **33**(9), 40–60, (September 1990).
6. Ph. J. M. Coueignoux, *Generation of Roman Printed Fonts*, Ph.D. dissertation, Massachusetts Institute of Technology, 1973.
7. Debra A. Adams, 'abcdefg: a better constraint driven environment for font generation', in *Raster Imaging and Digital Typography*, eds. Jacques André and Roger D. Hersch, 54–70, Cambridge University Press, Cambridge, (1989).
8. Marc Nanard, Jocelyne Nanard, Marc Gandara, and Nathalie Porte, 'A declarative approach for font design by incremental learning', in *Raster Imaging and Digital Typography*, eds. Jacques André and Roger D. Hersch, 71–82, Cambridge University Press, Cambridge, (1989).
9. Marc Nanard and Jocelyne Nanard, 'An anti-aliasing method for low resolution fonts based on font structure', in *Raster Imaging and Digital Typography*, eds. Jacques André and Roger D. Hersch, 111–122, Cambridge University Press, Cambridge, (1989).
10. Roger D. Hersch and Claude Betrisey, 'Model-based matching and hinting of fonts', *Computer Graphics*, **25**(4), 71–80, (1991).
11. Claude Bétrisey, *Génération automatique de contraintes pour caractères typographiques à l'aide d'un modèle topologique*, Ph.D. dissertation, Ecole Polytechnique Fédérale de Lausanne, 1993.
12. Tetsuzou Uehara, Motohide Kokunishi, Kenji Shimoji, and Hideko Kagimasa, 'Character shape representation and generation by skeleton vector method', *IECE Japan*, **J74-D-II**(8), 1020–1031, (August 1991). (In Japanese.)
13. Martin J. Dürst, 'Computers and culture: The case of the Japanese writing system', in *Computer Science, Communications and Society: A Technical and Cultural Challenge*, ed. Adelheid Bürgi-Schmelz et al., 264–270. Swiss Informaticians Association and Swiss Sociological Association, (1993).
14. Fumitaka Uchio, Toshiya Higuchi, Tadahiro Kitahashi, Hidehiko Sanada, and Yoshikazu Tezuka, 'A method for normalizing the appearance size of brush-written Chinese characters', in *Raster Imaging and Digital Typography*, eds. Jacques André and Roger D. Hersch, 144–153, Cambridge University Press, Cambridge, (1989).
15. The Unicode Consortium, *The Unicode Standard – Worldwide Character Encoding*, vol. 2, Addison Wesley, Reading, MA, 1992. Version 1.0.
16. Martin J. Dürst, 'Structured character description for font design: A preliminary approach based on Prolog', in *Computer Graphics and Applications*, eds. Sung Young Shin and Tosiyasu L. Kunii, 369–380, World Scientific, Singapore, (1993). Proceedings of Pacific Graphics '93.
17. Donald E. Knuth, *The TEXbook*, Addison-Wesley, Reading, MA, 1984.
18. Leon Sterling and Ehud Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA, 1986.
19. Jianchao Zeng, Takeshi Inoue, Hidehiko Sanada, and Yoshikazu Tezuka, 'A data structure suitable for representing the calligraphic rules for Chinese character evaluation', in *Proc. 9th International Conference on Pattern Recognition*, pp. 181–183. IEEE, (1988).
20. John D. Hobby and Gu Guoan, 'A Chinese meta-font', *TUGboat (TEX users group newsletter)*, **5**(2), 119–136, (1984).
21. Andrew Nathaniel Nelson, *The Modern Reader's Japanese-English Character Dictionary*, Charles E. Tuttle, Tokyo, 1962.
22. Yannis Haralambous, 'The Khmer script tamed by the lion (of TEX)', *TUGboat (TEX users group newsletter)*, **14**(3), 260–270, (1993). (1993 TEX Users Group Annual Meeting).
23. Douglas R. Hofstadter, *Metamagical Themas: Questioning for the Essence of Mind and Pattern*, Basic Books, New York, 1985.
24. Maarten J.G.M. van Emmerik, 'Interactive design of 3d models with geometric constraints', *The Visual Computer*, **7**(5/6), 309–325, (September 1991).