SYNTHESIS OF PARAMETRISABLE FONTS BY SHAPE COMPONENTS

THÉSIS N° 1905 (1998)

PRÉSENTÉE AU DÉPARTEMENT D'INFORMATIQUE

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÉS SCIENCES TECHNIQUES

PAR

Changyuan HU

Master of Computer Science and Technology, Nanjing University, Nanjing, Chine
de nationalité chinoise

acceptée sur proposition du jury:

Prof. R. D. Hersch, directeur de thése

Prof. J.P. Buser, rapporteur

Dr. J. Gonczarowski, rapporteur

Dr. P. Karow, rapporteur

Lausanne, EPFL

1998

Thanks

I wish to express my gratitude to my thesis advisor professor Roger D. Hersch, the head of the Peripheral Systems Laboratory (LSP) of the EPFL, for the unique opportunity he offered me to do research work in his laboratory, for guiding my research from the start to the end, and for providing me the best research environment.

Many thanks to all my colleagues in the LSP, who were always there to help me with small and big problems from updating computer systems to translating French. I thank in particular Dr. Patrick Emmel for helping me in understanding regulations and in setting up the typeset format of this thesis. I thank Oscar Figueiredo for translating the abstract of this thesis to French. Isaac Amidror helped me to improve the English of some parts of this thesis. I thank him. I also want to thank Fulco Houkes for developing the visual interface.

I wish to thank also professor André Gürtler, from Basel School of Design, for correcting the shape of our synthesized characters.

* * *

And finally, many thanks to my parents and all my family.

Abstract

Typographic characters implicitly incorporate structure elements such as stems, bars, round parts, arches and serifs, which are repeated throughout the characters of a font. Although this structure information is important when typographers design typefaces, it is however not explicitly described in today's outline font technology. As a consequence, coherently varying the style of an outline font has to be done by modifying contours of all characters in the font.

We propose in this thesis a new highly flexible font description method, which explicitly describes characters as structure elements, i.e. as assemblies of parametrisable shape components. Structure elements are either predefined parametrisable components such as stems or bars of parametrisable width, or can be described by assemblies of parametrisable shape components such as sweeps and half-loops. Terminal elements are either predefined parametrisable serif shape components or are described by components such as sweeps and ellipse-like round parts and by boundary correcting paths. The component based character synthesis method is illustrated by the reconstruction of the basic characters of a few traditional text typefaces.

Using this method, we have developed a prototype of our component based parametrisable font synthesis system. Fonts are characterized by the font independent structure of individual characters, by typeface category information (serif types, junction types, squareness and obliqueness of round parts), by font-dependent global parameters and by further font-dependent parameters, referring either to a group of characters or to a single character. By varying global parameters, derived fonts can be created which vary in width, weight, contrast and shape. Such derived fonts are useful for producing high-quality condensed text, for varying the character weight and for optical scaling. Varying the typeface category information as well enables exploring parts of the traditional Latin character design space.

We show the high quality of our synthesized fonts by synthesizing characters of some existing typefaces (Times, Helvetica and Bodoni). To demonstrate the application potential of this method, we have successfully accomplished typographical experiments, which are beyond the capability of traditional outline font technology, such as variation of weight, condensation, height proportion, contrast and oblique stress, and optical scaling for printing at different physical sizes.

Résumé

Les caractères typographiques comportent implicitement des éléments de structure tels que des jambages, des barres, des parties arrondies, des arches et des empattements qui sont répétés sur tous les caractères d'une même police. Bien que cette information de structure soit importante et qu'elle soit présente lorsque les typographes dessinent les polices, elle n'est pas exprimée explicitement par la technologie actuelle de représentation des caractères par contours. Par conséquent, pour modifier de façon cohérente le style d'une police, il est actuellement nécessaire de modifier les contours de tous ses caractères.

Nous proposons dans cette thèse une nouvelle méthode de description de fonte extrêmement souple qui décrit explicitement les caractères d'après leurs éléments de structure, c'est-à-dire un assemblage d'éléments de forme paramétrable. Les éléments de structure sont soit des composants paramétrables prédéfinis tels que des jambages ou des barres de largeur paramétrable, ou bien ils peuvent être décrits comme des assemblages de composants géométriques paramétrables tels que des "tracés de pinceaux" ou des demi-boucles. Les éléments terminaux sont soit des empattements paramétrables soit des composants formés par des tracés de plume, des ellipses et des courbes de correction de contour. La méthode de génération de caractères à partir de composants géométriques est illustrée par la reconstruction des caractères de base de quelques polices classiques.

En utilisant cette méthode nous avons réalisé un prototype de système de génération de polices à partir d'éléments de formes paramétrables. Les polices sont caractérisées par la structure inhérente des caractères indépendemment de leur police, par une information de catégorie de police (types de jambage, types de jonction, arrondis ronds ou carrés verticaux ou obliques), par des paramètres globaux spécifiques à la police et par d'autres paramètres spécifiques à la police relatifs à un groupe de caractères ou à un caractère individuel. En jouant sur les paramètres globaux, on peut créer des polices dérivées, variant en largeur, graisse, contraste et forme. De telles polices sont utiles pour produire du texte condensé de haute qualité, pour changer la graisse des caractères ou bien pour l'ajustement optique des caractères. La modification des paramètres relatifs à la catégorie de la police permet également d'explorer certaines variations de parties de l'alphabet latin.

Nous montrons que la qualité des polices produites est bonne lors de la reproduction des caractères de polices existantes (Times, Helvetica, Bodoni). Pour explorer le potentiel d'application de cette méthode, nous avons mené à bien un certain nombre d'expériences typographiques au-delà des capacités de la technologie traditionnelle de description de polices par contours telles que variation de graisse, condensation, modification de la proportion hauteur des minuscules / hauteur des majuscules, augmentation du contraste, accentuation de la nature oblique des caractères arrondis et ajustement optique des caractères pour l'impression à différentes tailles.

Table Of Contents

1. Introduction 5
1.1 Preface 5
1.2 Previous work in font parametrization 7
1.3 Content of this thesis 8
1.4 Terminology 10

2. Structures and components 11
2.1 Structure of characters 11
2.1.1 Character structure 11
2.1.2 Structure element connecting graph 12
2.1.2.1 Basic structure elements in structure graphs 13
2.1.2.2 Basic connections in structure graphs 13
2.1.2.3 Design of structure graphs 16
2.1.2.4 Refinement of structure graphs 16
2.2 Components 17
2.2.1 Components for straight strokes 18
2.2.2 Curves suitable for font parametrization 19
2.2.2.1 Modelling quadrant curves 19
2.2.2.2 Curvature of single curve segment 21
2.2.2.3 Curvatures of successively connected curve segments 24
2.2.3 Components for round parts 28
2.2.3.1 The loop component 28
2.2.3.2 The half-loop component 30
2.2.3.3 The sweep component 31
2.2.4 Components for terminals 34
2.2.4.1 The serif component 34
2.2.4.2 The slant-serif component 38
2.2.4.3 The dot component 39
2.2.4.4 The path component 41
2.3 Summary 41

3. Font parametrization 43
3.1 Earmark-based refinement of structure graphs 44
3.1.1 Serif styles 44
3.1.2 Analysis on common earmarks 45
3.2 Parametrisable character synthesis methods 47
3.2.1 Component position dependency 47
3.2.1.1 Relationships between components 47
3.2.1.2 Component dependency graph 50
3.2.1.3 Methods to specify dependency 51

3.2.2 Component shape tuning 52
3.2.2.1 Dimension 53
3.2.2.2 Orientation 54
3.2.2.3 Curvature 55
3.2.3 Boundary correction 56
3.2.3.1 Trim extension of component outlines 56
3.2.3.2 Smooth corner of intersected components 60
3.2.3.3 Hand tuned boundary correction 61
3.2.4 The complete parametrisable character synthesis method 62
3.3 Parameter files 65
3.3.1 Coordinate system 65
3.3.2 Parameter hierarchy 67
3.3.2.1 Global parameters and local parameters 67
3.3.2.2 Local parameter grouping 68
3.3.2.3 The parameter hierarchy 69
3.3.3 An example of parameter files 70
3.4 Technical issues regarding the font synthesizer 72
3.5 Summary 74

4. Implementation of the parametrisable font synthesizing system 75
4.1 Classes 75
4.1.1 The base class 76
4.1.2 The derived classes 78
4.1.3 Other classes 79
4.2 The implementation of the parameter hierarchy 81
4.2.1 Parameter files 81
4.2.2 Implementation 83
4.3 Output forms of synthesized characters 84
4.3.1 Component-based rasterization of synthesized characters 84
4.3.2 Outline generation of synthesized characters 86
4.4 Automatic optical spacing 88
4.4.1 The principle of automatic optical spacing 88
4.4.2 The implementation 89
4.5 Automatic hinting and grid-fitting 91
4.5.1 The theory of grid-fitting and hinting 92
4.5.2 The implementation 92
4.5.2.1 Automatic generation of traditional hints 92
4.5.2.2 Grid-fitting component-based characters without hints 93
4.6 Evaluation 95
4.7 Summary 97

5. Experiments and applications 99
5.1 A visual environment 99

5.1.1 Character visualization 99
5.1.2 Global parameter modification 101
5.2 Typographical experiments and applications 104
5.2.1 Font variation 105
5.2.1.1 Boldness 105
5.2.1.2 Condensation. 106
5.2.1.3 Stress 108
5.2.1.4 Contrast 109
5.2.1.5 Height proportion 109
5.2.1.6 Variations in multiple dimensions. 110
5.2.2 Optical scaling 111
5.2.3 Parametrization of existing fonts 113
5.3 Summary 113

6. Conclusion and future work **115**

References 117

Appendix A. The β value for a quarter of an arc 123

Appendix B. The β value for approximating an ellipse 125

Appendix C. Proof of parameter η 127

Appendix D. Description of characters by components 129

Appendix E. Enlarged resynthesized component-based characters 131

Appendix F. The variation cube of “e” 135

Appendix G. Parameter files of parametrisable Times Roman 137

CHAPTER 1

Introduction

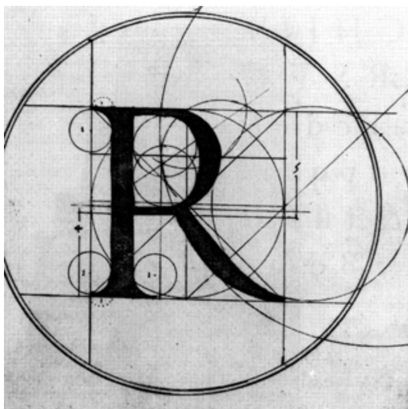
1.1 Preface

The twenty-six letters that we use today have evolved over the course of several millennia. Although printing is a far more recent phenomenon, the development of type styles — the refinement of symbols we use to express ourselves in writing — is a process that began when man first started to communicate. Legibility and beauty might be the two most important reasons which have continuously driven typographers to refine our letterforms. To make our written symbols easy to read and write, basic structure elements such as vertical stems and horizontal bars have evolved naturally in our typefaces. As man's knowledge of science accumulated, typographers started to dream: could the beauty of our characters and their structure elements be represented by mathematics, or by geometry?

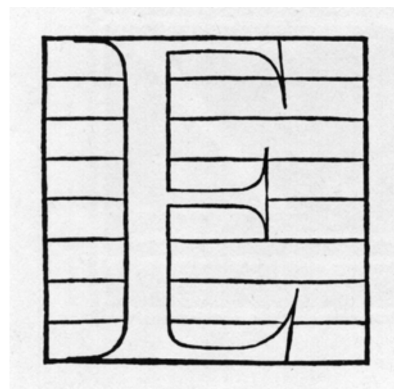
Figure 1.1: The Roman capitals of the Column of Trajan in 114 A.D.

Perhaps the closest direct ancestor of our letterform can be dated back to the ancient Roman Empire. The most important contribution of the Romans to our own alphabet was the visual refinement of capital letters. The capitals incised in the Column of Trajan in 114 A.D. have been called the perfect expression of letters (Fig. 1.1). “Indeed, they are the basis of all of our typefaces” [Labuz88, pp.31]. Roman letters were engraved with thick and thin strokes and graceful curves. Thicker strokes are usually referred to as major strokes; thinner stroke are minor strokes. Researchers have found that the tool required to produce these embellishments was a double-pointed pen. By shifting the angle of the pen, the Roman could produce lines of any width. When a stone was to be carved, a sketch of the letter was first made with this tool and then chiselled. Yet the Roman capitals are not geometrically designed; their beauty comes from the subtleties of their curves and strokes.

Fourteen hundred years later, the Renaissance humanists rediscovered the old values of the antique past. During that time, artists tended to find the mathematical rules of beauty for their art works, and scientific discoveries and methods such as geometry and the rule of proportion were applied to their artistic designs. The Roman capitals were therefore naturally described through mathematical constructions drawn with ruler and compass. For example, Fig. 1.2a, a letterform design from the middle of the sixteenth century, displays beautiful circles, curves, reference points and lines constructed by mathematical calculation. The capital letters designed by Frate Vespasiano Amphiareo in 1572 suggest that the width of a main stem should be one eighth of the letter’s height (Fig. 1.2b). These letter construction models served as references for the punchcutting of printing types, even though it was not possible to strictly apply those reference lines and curves onto the small piece of hand-cut steel types.



(a)



(b)

Figure 1.2: Character constructions in Renaissance: a) curves designed by circles; b) regularized proportion of stroke width to character height.

Only with the advent of digital typography were typographers really able to apply geometry to type design. Computers were used to calculate the geometrical shapes of characters and the generated character shapes were transferred onto negatives faithfully

by digital laser photocomposer. New character design and representation methods were developed by the cooperation of typographers and computer scientists. Nowadays, almost every typesetting system produces characters by outlines. The outline font technology represents a character by its outline description and reproduces the character's image by a computer program called the font rasterizer. Outlines are capable of describing the shape of a character either with straight lines or curved arcs, yet the outline representation lacks the ability to describe specific typographical information such as widths of strokes and the squareness of round parts. A problem came up: could we design and represent characters so as to make sure that typographical features such as stroke width, serif length and squareness of round strokes could be controlled or modified? Incorporating typographical information as parameters into the characters' representation rather than describing characters by their contours — an interesting yet difficult problem — has attracted both typographers and computer scientists.

1.2 Previous work in font parametrization

The earliest attempt known to describe typographic characters by parametrizable shape primitives was that of P. Coueignoux, who designed one of the first fully digital typesetter controllers [Coueignoux81].

The most serious published work done in the field of parametrizable fonts is the *Metafont system*, a programmable parametrizable font synthesizing system [Knuth86a]. The Metafont system relies on a few basic paradigms for generating characters and symbols. The main character parts such as horizontal, vertical, diagonal strokes and round parts are specified by describing the path of a pen with given orientations and pen widths. The pen's central path is described by a sequence of pen positions and directions. From this information, Metafont computes a smooth centerline pen trajectory. With the given pen positions, widths and orientations, and with the centerline trajectory, Metafont infers the description of the boundary of the corresponding pen stroke [Hobby89]. Serifs with font dependent serif width, height and depth information are adjusted to the computed stroke boundary. The shape boundary resulting from the assembly of serifs and stroke can either be directly filled, or traced by a small circular pen and then filled. In addition to strokes defined by pen trajectories, Metafont specifies round character parts covering one or multiple quadrants by *superarcs*, i.e. scaled arcs defined within a single quadrant whose boundaries are given by *superellipses* [Knuth86a, pp.126].

The Metafont program was used by Donald Knuth to create his Computer Modern typeface family [Knuth86b]. The Computer Modern typefaces were parametrized so as to automatically generate optically scaled fonts and to generate by simple change of parametrization sans-serif, typewriter, semi-bold, bold, condensed and slanted roman fonts. Separate character shape descriptions are used for the italic font. Since Metafont is both a complete programming language and a flexible font design tool, using it requires both advanced programming and typographic skills. Only a few individuals in the world are known to use Metafont, often for the design of non-Latin characters [Haralambous94] [Southall98]. One of the lessons learned by users of Metafont design-

ing Latin characters is that Metafont's pen paradigm does not offer sufficient freedom to generate character shapes exactly according to the designer's intention. This explains why besides Computer Modern, most Metafont designs for Latin fonts rely heavily on outline descriptions [Billawala89].

The recent Infinifont system [MacQueen93] [Bauermeister96] is a feature-based parametrizable font description and reconstruction system. Its authors describe the basic mechanism to assemble a character such as character "E" from parametrizable vertical bars, horizontal bars and serifs. However, most of their work is not published and it is therefore not known how they synthesize different typeface categories and which paradigms they use for synthesizing curved character shapes and serif variants.

Schneider describes a method for assembling parametrizable pen-based strokes into typographic characters [Schneider98]. Regarding Latin character shapes, the method suffers from the same limitations as Metafont. It seems however particularly well suited for synthesizing Chinese characters. The method of assembling characters by parametrizable strokes has already been tried for Chinese characters [Dong91] [Fan91] [Dürst93]. Industrial implementation has shown that good results can be obtained when the stroke-based Chinese characters are used for lower resolution applications. For high resolution applications, however, the quality of the reproduced characters is still not satisfying especially for connections and curved strokes.

Some researchers try to incorporate additional information (constraints) to the outline representation of characters [Shamir98] [Zalik95]. They use this information to constrain the relationship (distance or spatial order) between vertices of character outlines and, possibly, the moving directions of vertices when the outlines are modified. By carefully specifying these point-to-point constraints, some typeface variations such as variation of boldness can be carried out. But, for more complex typeface variations, it is difficult to predicate the behaviour of each contour point and thus to specify their constraints. For example, for varying the contrast and oblique stress of a font, it is not sufficient to constrain the trajectory of some character contour points to straight lines. Adding constraints to character outlines seems to be more suitable for grid-fitting character outlines during the process of rasterization. For example, some of the instructions used in TrueType [Microsoft95a] fonts specify the grid-fitting method by constraining the behaviour of character outline points.

1.3 Contents of this thesis

In this thesis, we propose a component-based parametrizable font representation method and the implementation of its prototype. Traditional Latin letter shapes which derive from Antiqua and Grotesque (sans-serif) typefaces [Tschichold65] can be decomposed into basic structure elements: vertical stems, round parts (also called curved shape parts or bowls), arches, horizontal bars, diagonal bars, serifs and terminals. Conversely, our aim is to synthesize traditional letter shapes by coherently assembling such basic structure elements (Fig. 1.3).

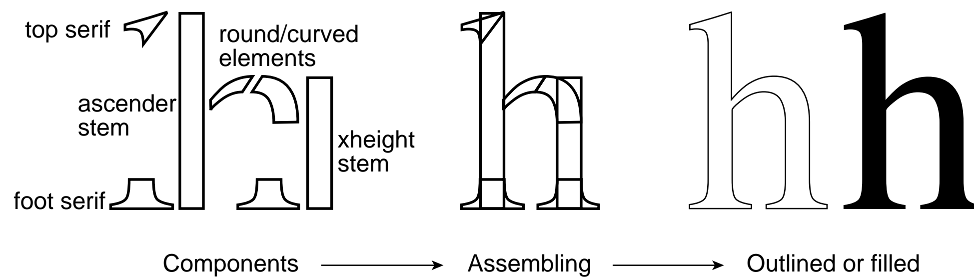


Figure 1.3: Assembling a character by basic structure elements.

The challenge resides in defining parametrizable geometric shapes (components) with which most instances of structure elements can be synthesized. We propose both components for structure elements representing straight strokes such as stems, horizontal bars and diagonal bars, as well as components for curved structure elements such as round parts and arches. Components are also designed to describe serifs and terminals. In order to support different typeface categories, we also propose a set of standard parametrizable junctions between shape components. The research work and the results of this thesis will be presented in chapters 2 to 5.

In chapter 2, we analyze and describe the character structure by a structure element connecting graph. Then, we present in the same chapter our design of *components* — the basic typographical shape synthesizing primitives. To derive our model for the parametrization of curved structure elements and strokes, our research on curve descriptions suitable for parametrizable fonts is presented.

In chapter 3, we present the model and principles of our parametrizable font synthesis system. Characters are synthesized by using a set of parametrizable character synthesis methods, one basic method for each character. To generate the character components, the system makes use of hierarchically organized parameters.

In chapter 4, a prototype implementation based on the C++ programming language is given. Ideas presented in chapter 2 and chapter 3, such as the hierarchical parameter organization, will be realized by the prototype implementation. Technical problems regarding rasterization, automatic optical spacing, grid-fitting and automatic hinting are discussed.

Typographical experiments are presented in chapter 5. These experiments demonstrate the capabilities and application potential of the proposed parametrizable component-based font synthesis system.

1.4 Terminology

In this thesis, traditional typographic terminology [Labuz88, pp.23-25] and the terminology used in digital typography will both be used. Sometimes, there is a little inconsistency between the traditional meaning of a term and its explanation according to today's computer font technology. In such a case, the meaning should adapted to the context.

- A *typeface* is a distinctive design for a set of visually related symbols. A typeface represents an abstract design idea for how letterforms are to be presented.
- A *typeface family* is a collection of typeface variations based on a single design. For example, Helvetica is a type style design. The Helvetica typeface family may contain Helvetica normal, Helvetica light, Helvetica bold, Helvetica condensed, Helvetica italic and other typefaces.
- This thesis focuses on *text typefaces* and not on *decorative typefaces*. Text typefaces are used for the continuous setting of text in books and newspapers. Legible, suitable and aesthetically pleasing letters are critical to text typefaces. Decorative typefaces (mainly created in the 19th century) are used for decoration purposes in which attractiveness, novelty and loudness are emphasized. The distinction between them can be found in Rockledge's famous Typefinder system [Rockledge91].
- A *font* is a particular example of a typeface, traditionally in a particular size, and contains symbols for each element of a particular character set. In digital typography, since outline fonts are scalable, a font is often a file containing description data for shapes of all characters which can be scaled to any particular size by computer programs. It seems that there is no practical difference between a scalable digital *font* and a *typeface*.
- A *parametrizable font* is a font with parameters controlling typographical features. Not only the size of a character is scalable, but also the style of characters can be varied. By changing certain parameters, a parametrizable *font* may generate all typefaces or scalable fonts in a *typeface family.*
- A *glyph* is an individual symbol which appears when it has been printed. In digital typography, a glyph is represented either in the form of an outline or a bitmapped image (array of dots) that is to be displayed or printed.
- A *character synthesis method* is a C++ object member function specially conceived for the typeface-independent generation of a given character (for example ASCII character "m").

CHAPTER 2

Structures and components

In out font parametrization method, a character is constructed by pieces of basic descriptive shapes, called components. *Components* are connected to or placed between each other according to a *structure graph* which describe their relationship, i.e. the character structure. The structure graph is a conceptual model used to describe the component decomposition of characters and, conversely, the assembling of characters from their components. In this chapter, we first present our methodology to describe the structure of characters. Then we will introduce the basic components we have designed and our method to parametrize components.

2.1 Structure of characters

2.1.1 Character structure

While Chinese characters are written by basic strokes, Roman characters are made of some basic structure elements, such as vertical stems, horizontal bars, diagonal bars, slanted top serifs, foot serifs, loops, bowls, arches, dots, tails and ears (Fig. 2.1). Nowadays, thousands of different text typefaces have been invented. However, one doesn't have much difficulty to recognize a letterform in any text typeface. That is due to the spatial ordering of its structure elements (stems, bars, bowls, arches, etc.). The way character elements are connected is intentionally maintained across different typefaces. This kind of intrinsic spatial ordering and connection of elements in a character is called *character structure*.

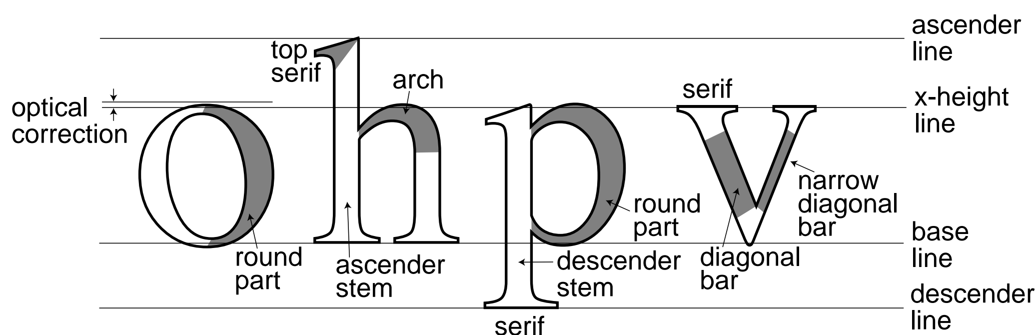


Figure 2.1: Some basic elements in Roman letters. This figure shows some important structure elements.

Previous work on character structure focused on character and feature recognition as well as character synthesis. C. H. Cox III and his colleagues [Cox82] studied the relationship between two aspects of characters: the embellishments of physical characters and the skeleton of letters. The basic elements of skeletons are (1) vertices, (2) edges which specify the spatial ordering between vertices, and (3) the relationships between vertices and edges. An extension of the symbolic representation commonly used in graph theory was employed to depict skeletons, which made it possible to do some formal graph analysis. Readers are referred to [Cox82] for a detailed description of the skeleton model. As an application, the skeleton model was used to bridge between the conceptual description of a letterform and the corresponding physical embellished character for character recognition. Cox et al. have also proposed a way to create different fonts by applying different stroke definitions to their skeleton model.

The idea of using a skeleton to represent the character structure is also used in other frameworks for different purposes, such as finding typographical features [Herz97], creating thin line or skeleton fonts [Gonczarowski98], and describing the track of a pen [Knuth86a]. However, for the sole purpose of building parametrizable fonts, we found the skeleton representation to have some disadvantages. Firstly, skeletons are suitable for describing the bones of characters, but not for describing their flesh. Secondly, according to the typographer's point of view, the meaning of a character skeleton is not very clearly defined. It is often assumed that the skeleton is the midline of a character. Should midlines of embellishments such as serifs be parts of the skeleton? Cox's work is a good abstraction which puts emphasis on the relationship and connection of "vertices". It might be suitable for letterform shape analysis. But for character shape generation, the typographical meanings of the vertices and edges are not defined and hence, not suitable for synthesizing parametrizable fonts.

The description of character structure for high-fidelity font parametrization should be capable of (1) describing the essential ordering of different parts in a character, and (2) refining the shape of the parts and their connection types.

2.1.2 Structure element connecting graph

We use the term *structure element connecting graph*, or *structure graph*, to describe character structures. The purpose of a structure graph is to explain how a character is decomposed into structure elements and, how these structure elements are assembled into a full blown character.

The symbols used in structure graphs are structure elements and connections. Structure elements are typographically functional parts, such as vertical stems, horizontal bars, full or half loops, and serifs. Structure elements can be synthesized by one or more components (see section 2.2 for a detailed description of component design). The order of structure elements in a structure graph reflects the natural spatial ordering of the corresponding parts in a character, but not their physical positions. A structure element can either be connected to one or to more structure elements or be simply isolated. If two structure elements connect, a line is drawn connecting them. The

connection of two structure elements will be implemented by means of operations or by a special connecting component (to be discussed later).

2.1.2.1 Basic structure elements in structure graphs

Three kinds of basic structure elements are introduced in the structure graph: straight strokes, round parts, and terminals. *Straight strokes* represent all visually (not geometrically) straight strokes, which include vertical stems, horizontal bars and slanted or diagonal bars. *Round parts* represent all curved or round strokes, which include loops, half loops, arches, bowls, counters, dots, etc. Stems and round parts build up the main body of characters. These body strokes are often terminated by some styled shapes, which are called *terminals*. Serifs and pears are common terminals for straight and curved strokes respectively. The graph symbols that we defined for straight strokes, round parts and terminals are shown in Fig. 2.2.

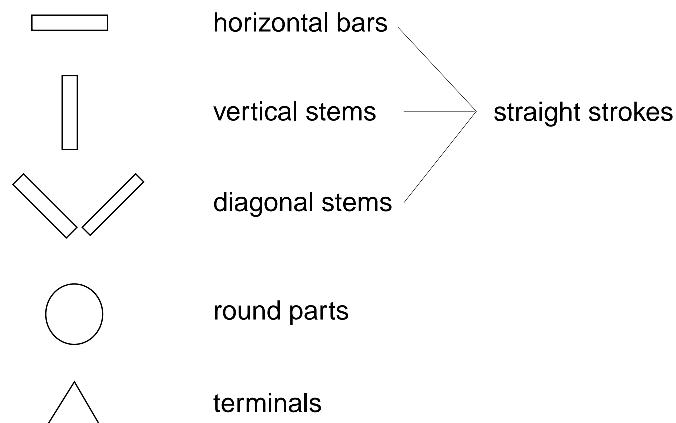


Figure 2.2: Straight strokes, round parts and terminals used in structure graphs.

Note that symbols are designed to try to reflect the common shapes of the corresponding part in a character. However, round parts and terminals are abstract symbols, which may represent very different shapes. Hence refinements are often necessary for these elements. Refinement of round parts and terminals are discussed in section 2.1.2.4.

2.1.2.2 Basic connections in structure graphs

Connections, or junctions between structure elements, are simply represented by lines. There are four different junction types used in the structure graphs: *meet*, *link*, *join* and *cross*. Names for these junction types are partly inspired by [Cox82] but oriented towards physical typefaces. However, these junctions types often cannot be one-to-one mapped to physical character shapes. They are conceptual types of connections. The

implementation of these connection types may require additional sub-types as a means of refinement. Symbols of junction types are drawn in Fig. 2.3.

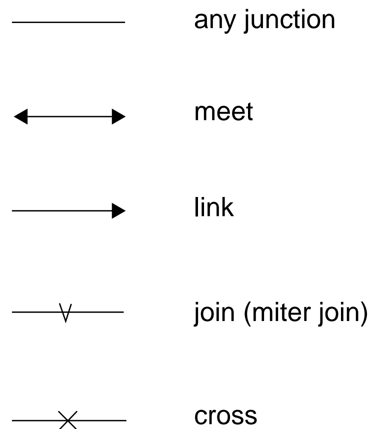


Figure 2.3: The types of connections used in structure graphs.

1) meet

Two structure elements, which may be of any type, connect each other at their ends. Most of the *meet* junctions are smooth. But not necessarily absolutely smooth. The *meet* junction is often applied when a stem connects to a serif, a stem connects to an arch or any round part, two round parts connect to each other, etc. In the structure graph, the *meet* junction is a vertex with two edges, or a vertex of degree two (Fig. 2.4).

Figure 2.4: The *meet* junction.

2) link

The end of one structure element is buried in the body of another one. Both structure elements can be either stem or round part. This kind of junction is very common when a stem is connected to a round part to produce an "arm". A *link* junction is a vertex of degree three (Fig. 2.5).

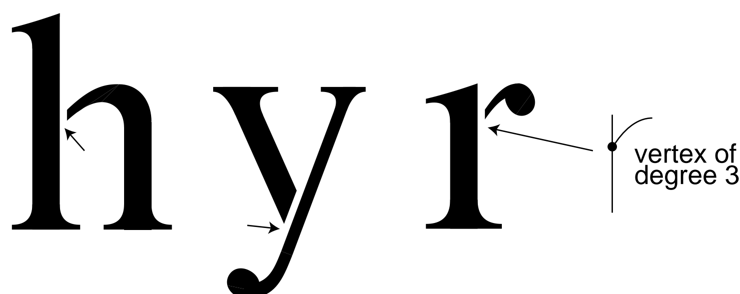


Figure 2.5: The *link* junction.

3) join

Two stems connect at their end and form a sharp angle, less than 90 degree, on the internal side and a miter and limited tip on the external side. Although in respect to graph theory, the *join* junction is also a vertex of degree two, it is quite different from the *meet* junction. The *join* junction is a sharp angle and certainly not a smooth connection. Typical examples of *join* junctions are character “v”, “w” and “z” (Fig. 2.6).

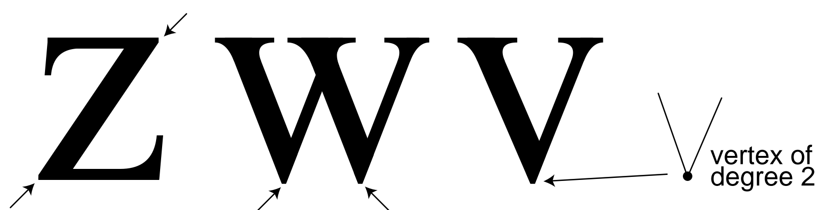


Figure 2.6: The *join* junction.

4) cross

Two stems or possibly round parts overlap each other. In the terminology of graph theory, the *cross* junction is a vertex of degree four. Typical examples are character “f”, “t” and “x” (Fig. 2.7).

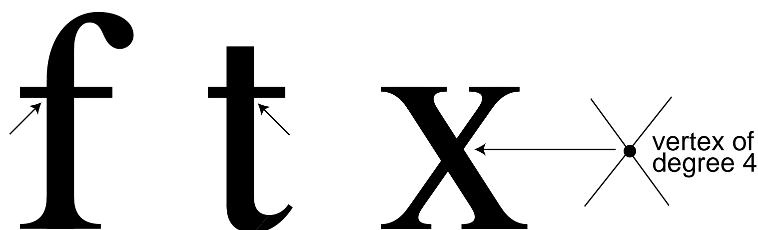


Figure 2.7: The *cross* junction.

2.1.2.3 Design of structure graphs

Structure graphs are designed for the purpose of font parametrization. In order to parametrize most of the text typeface, one structure graph should be able to describe a character for many typefaces. Practically, we found that for most of the Latin characters, one structure graph may describe different text typefaces. Only a few characters need design variations, such as the double-storey “a” and the single-storey “*a*” (Fig. 2.8), and the looped “g” or the tailed “g”.

To design the structure graph for a character, first we should carefully study the character shapes in different typeface. Then, the essential font-independent structure elements for the body of the character are abstracted, which include straight elements and round elements. The ways they connect one to another are also specified during the separation of structure elements. Decorations at the ends of body elements can be represented by terminal elements. Note that in serif typefaces terminal elements are often serifs or bulbs (pears). In sans-serif typefaces, we should allow the terminal to be null to avoid designing variations of structure graphs. Examples of basic structure graphs are shown in Fig. 2.8.

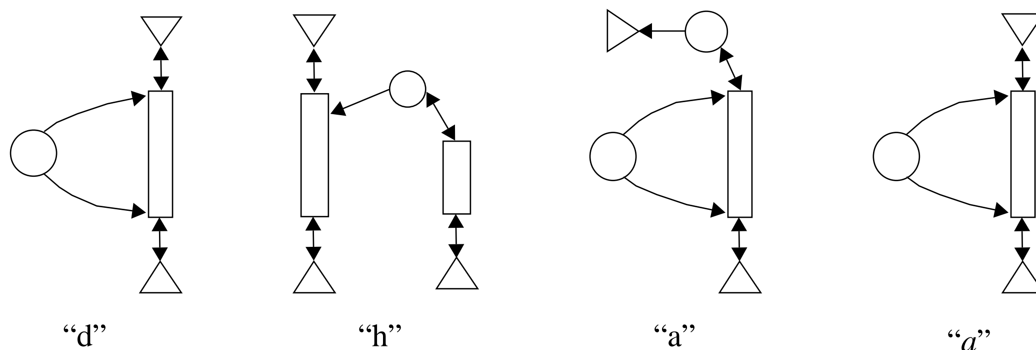


Figure 2.8: Examples of basic structure graphs. Character “a” has two structure graphs which are designed for double-storey and single-storey respectively.

2.1.2.4 Refinement of structure graphs

Both structure elements and connections in the basic structure graphs can be refined. Fig. 2.9 gives an example showing the refinement of structure elements and the corresponding components (see section 2.2 for the definition of components).

In the refined structure graphs, shortened names of components can be used to mark the refinement of a structure element. For example, *st* stands for stem, *sw* for sweep, *sf* for serif, *tsf* for top-serif, *lp* for loop and *hlp* for half-loop, etc. Some structure elements, such as stems and bars, can be synthesised by a single basic component. Others, such as round parts and terminals, may need two or more components. Connections in structure graphs will normally be implemented by connecting sweeps

and some kind of parameter dependencies. In fact, connections are more complex than the four predefined basic connection types. Therefore, junction variations may sometimes be needed. In the next chapter (Chapter 3), we will discuss in detail how to synthesize characters according to their refined structure graphs (components and parameters).

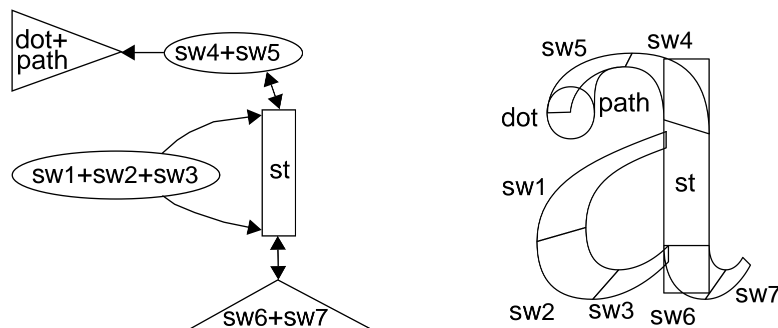


Figure 2.9: Refinement of structure elements and the corresponding components.

2.2 Components

The structure graphs discussed in the previous section enable us to separate a character into basic functional structure elements. In this section, we discuss about how to synthesize these structure elements by computer.

The challenge resides in defining parametrizable geometric shapes enabling to synthesize most instances of structure elements. We propose some basic shape synthesizing primitives for structure elements representing straight strokes, round parts and terminals. We see these shape synthesizing primitives as generators which take some *parameters* as input and create *contours* of these basic shapes as output. We call these synthesized basic shapes *components*, since they can be assembled into structure elements and, eventually, characters.

Contour: The contour of a 2-D character shape is represented by pieces of segments including straight lines and Bézier curves. In this thesis, contours are often specified by the list of their vertices.

Component: The term component refers to a basic geometric shape used for the synthesis of a part of a character.

Parameters of component: Parameters of a component can be regarded as arguments of a function. They are real values for coordinate, dimension (width or height), angle and attributes. Parameters in a font are managed in a hierarchical manner. How to prepare parameters for components is discussed in the following chapters of this thesis.

Components represent the concretization of structure elements by geometric shapes. There are some goals we should aim at. Firstly, components should be flexible

and intelligent enough to enable style coherent modifications of characters. The synthesized shapes should response reasonably to a change of input parameters. For example, when the width at the horizontal extremum of a loop changes, the whole curve should also be adapted to maintain some curvature properties. Secondly, the less parameters, the better. Reducing the number of parameters does not only save space, but also simplifies the way to control, or tune, the shape of components.

2.2.1 Components for straight strokes

Straight strokes represent vertical stems, horizontal bars and diagonal stems. No matter what orientation they have, the two sides of the stroke are parallel. Essentially, straight strokes can be described by their position, orientation and width. The component we designed for straight strokes is named the stem component, or stem. The basic design of the stem component is shown in Fig. 2.10.

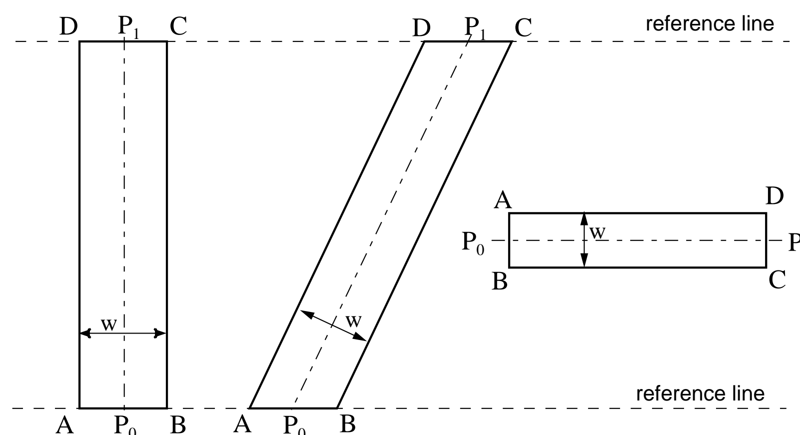


Figure 2.10: The basic design of the *stem* component.

The *stem* component needs a departure point P_0 and a arrival point P_1 to specify its position and orientation. The width of the stroke is given by parameter *w*, which is the geometrical distance between the two edges. The ends of a stem are terminated by either reference lines, such as xheight line and base line, or by lines perpendicular to its orientation. The later case is called a butt terminal. The synthesized stem component is a quadrangle labelled by its vertexes as *ABCD*.

We found that straight strokes in most of the text typefaces can be synthesized by the basic design of the stem components. Their two long edges are visually absolutely straight and parallel. However, some text typeface exist in which straight strokes have slightly concaved edges (Optima, for example). Some fonts even incorporate non-even width straight strokes. To parametrize these kinds of fonts, enhanced designs of the stem components are possible. The enhanced stem component requires additional parameters to specify the depth of the concaved edges and the different width at the two ends of the stroke (Fig. 2.11).

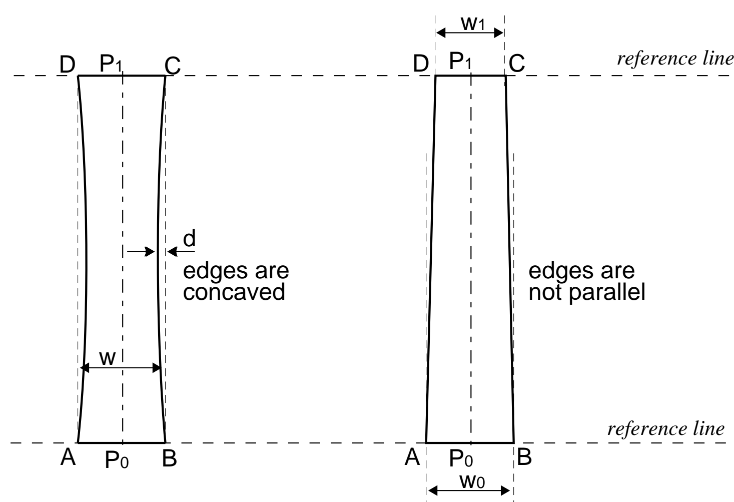


Figure 2.11: Possible enhanced design of the stem component.

2.2.2 Curves suitable for font parametrization

Before discussing the components for round parts, let us discuss our method of curve representation for parametrizable fonts. Curves are one of the most sensitive aspects of the aesthetic feeling of character shapes. In today's outline font technology, the curves which are employed to represent contours are cubic Bézier splines, quadratic B-splines and circular arcs. Industrial practice shows that cubic Bézier splines are most suitable for representing character outline parts and therefore also curved component contours. This is mainly because the slope of the curve can be easily controlled by successive pairs of on-curve and off-curve points.

2.2.2.1 Modelling quadrant curves

A cubic Bézier spline is defined by two end points, B_0 and B_3, and two control points, B_1 and B_2. We also refer to B_0 and B_3 as *on-curve* points, and to B_1 and B_2 as *off-curve* points. The polygon made by B_0, B_1, B_2 and B_3 is called the control polygon. The cubic Bézier curve can be represented in parametric form as

$$B(t) = (1 - t)^3 B_0 + 3t(1 - t)^2 B_1 + 3t^2(1 - t)B_2 + t^3 B_3 \qquad \textbf{(2.1)}$$

where, t = [0, 1]. Because the coordinate of each point contains two values (x and y), a cubic Bézier curve generally has eight free parameters.

Several pieces of Bézier curves can be connected smoothly to synthesize a long complex desired shape. It has been noticed that if the Bézier curves start and end at the locus of the local extremum of the desired shapes, it is easy to grid-fit them in order to rasterize them at a given resolution. Therefore, outline fonts are regularized to have Bézier spline end points at horizontal and vertical extrema [Stamm94b]. In our font

parametrization method, we keep this condition for the contours of components. Therefore Bézier splines for component descriptions are limited to one quadrant (Fig. 2.12b). As Fig. 2.12 shows, a Bézier spline occupying exactly one quadrant (for example the curve in Fig. 2.12b occupies the 3rd quadrant) has only six free parameters.

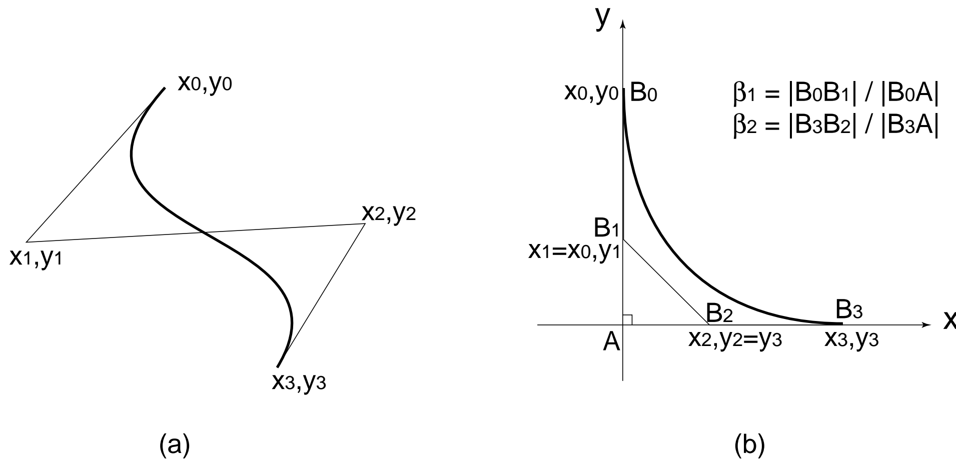


Figure 2.12: (a) A random Bézier spline has eight free parameters, while (b) a Bézier spline occupying exactly one quadrant has only six free parameters.

Suppose B_0B_1 and B_3B_2 intersect at point A (Fig. 2.12b). We define the relative positions of the control points B_1 and B_2 by parameters $\beta_1 = |B_0B_1| / |B_0A|$ and $\beta_2 = |B_3B_2| / |B_3A|$. Given departure point B_0, arrival point B_1 and the quadrant number, the parameters β_1 and β_2 control the shape of the quadrant curve. We refer in this thesis to the curves defined by the β parameters as *β-Bézier curves* or *β-Bézier splines*.

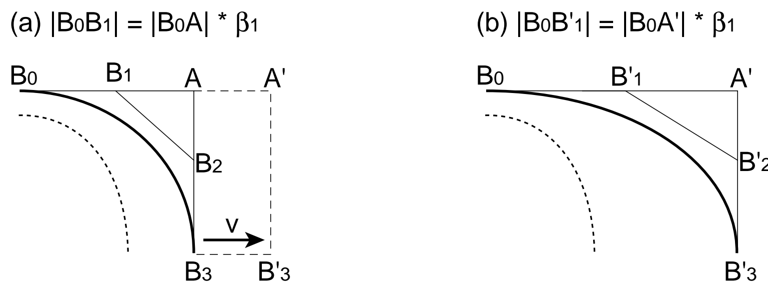


Figure 2.13: Stretch a quadrant Bézier curve: (a) original curve, in which B_3, B_2 and A will be stretched right by vector v; (b) control point B_1 defined by parameter β_1 can adjust itself reasonably.

One of the reasons we model quadrant curves with β-Bézier splines is that parameters β_1 and β_2 define control points B_1 and B_2 with some intelligence. For example, curves representing the contours of the round part of a character can be adjusted reasonably in response to a weight (stroke width) change. In Fig. 2.13, we pick

a curve in the first quadrant, which can be found in many characters such as “o”, “p” and “b”, and stretch it horizontally by vector v, so as to make the character bolder. With the β-Bézier spline method, we can adjust the control points properly.

2.2.2.2 Curvature of single curve segment

With the aim of synthesizing loops, let us analyse the flexibility offered by β-Bézier curves. Our first experiment is to see how the curvature behaves in response to the varying of parameters β_1 and β_2. Curvature radius R of a parametric Bézier spline (2.1) at the point $B(t) = [x(t), y(t)]^T$ is given by the formula

$$R(t) = \frac{((x'(t))^2 + (y'(t))^2)^{3/2}}{x'(t)y''(t) - x''(t)y'(t)} \tag{2.2}$$

And the curvature k, which is the reciprocal of curvature radius, is defined as

$$k = 1 / R. \tag{2.3}$$

For a Bézier curve defined by parametric form (2.1), the first and second derivatives of x(t) and y(t) are calculated by

$$B'(t) = -3(1 - t)^2 B_0 + 3(1 - 4t + 3t^2) B_1 + 3t(2 - 3t) B_2 + 3t^2 B_3 \tag{2.4}$$

$$B''(t) = 6(1 - t) B_0 - 6(2 - 3t) B_1 + 6(1 - 3t) B_2 + 6t B_3 \tag{2.5}$$

where $B'(t) = [x'(t), y'(t)]^T$, $B''(t) = [x''(t), y''(t)]^T$. Thus the values of curvature radius R and curvature k of a Bézier curve can be precisely calculated. By computing enough curvature values at points from t = 0, to t = 1, one obtains the curvature curve indicating the variation of curvature along the Bézier curve.

Without loss of generality, we normalize the curve shown in Fig. 2.12b by letting

$$|y_0 - y_A| = 1, |x_3 - x_A| = 1 \tag{2.6}$$

so that

$$B_0 = (0, 1), B_3 = (1, 0), A = (0, 0). \tag{2.7}$$

Then we have

$$B_1 = (0, 1 - \beta_1), B_2 = (1 - \beta_2, 0). \tag{2.8}$$

Varying the parameters β_1 and β_2 from 0 to 1, we obtain families of curves the apex of which (point with tangent parallel to baseline B_0B_1) is located within the circumscribed triangle B_0AB_3 (Fig. 2.14a). Since curvature conveys the visual information associated with a given arc, curvatures corresponding to the plotted Bézier splines are given in Fig. 2.14b.

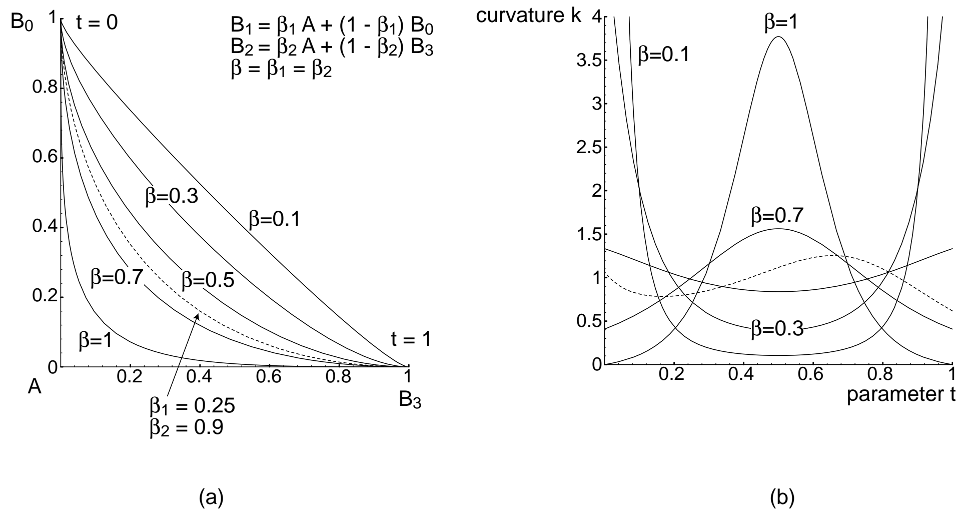


Figure 2.14: Family of cubic Bézier splines covering a quarter of arc and their respective curvatures, obtained by varying parameters β_1 and β_2.

From this experiment, we can observe some properties of the curvature of quadrant Bézier splines. Firstly, the shape of the curvature line varies more rapidly than the curve itself. This explains why curvatures are suitable for analyzing curves. Secondly, curves with β value greater than 0.7 or lower than 0.4 have widely different curvatures at their end points and in their central part. Experience shows that curves having high curvature at their end points and low curvature at their central part (for example curves with $\beta < 0.4$) are not visually pleasant. Thirdly, making β_1 very low and β_2 very high, and vice versa, does not affect the Bézier curve very much, but it does affect its curvature (see the dotted curve and curvature lines in Fig. 2.14).

Controlling the curvature of a Bézier curve is not as easy as controlling its shape. However, the curvature control at the ends of the curve is important when connecting two Bézier curves to make a sufficiently smooth long curved section, for example a curve with CG^2 continuity (to be discussed in section 2.2.2.3).

The second property shows that curves with median β values, for example between 0.4 and 0.7, have similar curvatures between their end points and their central part. The shapes of these curves are rounder and closer to a quadrant of circle or ellipse. Median β values seem to be more suitable for font parametrization. They better represent the outline of real characters from existing outline fonts. For example, if we choose one of the curves in Fig. 2.14a to approximate the quarter circle in the third quadrant, we will choose the one whose β value lies between 0.5 and 0.6. Because the curvature radius of each point on this curve is very close to the radius of the circle which radius is 1. The next experiment (Tab. 2.1) has confirmed this property. It is easy to find that the exact value of β for approximating a circular arc is around 0.55. One can require the point at

parameter t = 1/2 to have identical coordinates as the corresponding point of a quarter of a circle, by solving equation

$$B\left(\frac{1}{2}\right) = \left[1 - \frac{\sqrt{2}}{2}, 1 - \frac{\sqrt{2}}{2}\right]^T \quad \textbf{(2.9)}$$

where B(1/2) is the midpoint of the normalized Bézier curve $B_0B_1B_2B_3$ given in Fig. 2.14a. Another criterion for finding the β value which approximates a circular arc is requiring the curvature radius at B_0 or B_3 to be the radius of the arc. This requirement also generates a β value around 0.55 (see Appendix A).

Our second experiment is concerned with the third property. Will one β value be good enough for the curves of character shapes? A positive answer enables us to save one parameter and make it much easier to control the shape, or squareness, of the curves.

Table 2.1: β_1 and β_2 values for external and internal arcs of character "o" for different typeface and their approximation by a single β value.

	External contour								Internal contour							
	3rd quadrant				4th quadrant				3rd quadrant				4th quadrant			
	β_1	β_2	β	ξ	β_1	β_2	β	ξ	β_1	β_2	β	ξ	β_1	β_2	β	ξ
Bodoni	.544	.538	.541	.062	.538	.544	.541	.062	.699	.978	.842	.574	.978	.699	.842	.574
Clarendon	.639	.584	.611	.125	.584	.639	.611	.125	.656	.659	.657	.088	.659	.656	.657	.088
Courier	.609	.504	.557	.252	.504	.609	.557	.252	.580	.544	.562	.115	.544	.580	.562	.115
Helvetica	.830	.405	.633	.310	.405	.830	.633	.310	.434	.728	.590	.360	.735	.434	.593	.342
Lucida	.574	.660	.618	.161	.664	.574	.620	.149	.397	.742	.581	.475	.742	.397	.581	.475
Optima	.606	.538	.537	.190	.660	.512	.588	.282	.483	.731	.613	.227	.760	.351	.571	.613
Palatino	.573	.573	.573	.004	.584	.586	.585	.129	.430	.771	.612	.374	.700	.505	.607	.184
Times	.579	.577	.578	.108	.560	.563	.561	.143	.597	.599	.598	.043	.643	.638	.641	.051
Universe	.586	.660	.624	.123	.660	.586	.624	.123	.543	.753	.653	.072	.753	.543	.653	.072

The experiment considers character outlines of round part from several typefaces described by Bézier splines. It tries to replace them with single β parameter Bézier splines. We have analysed the Bézier splines (quadrant arcs) defining the external and internal contours of character "o" in various font families (Tab. 2.1). In most cases, β_1 and β_2 values are similar and can therefore be merged into a single β value expressing the *squareness* of an arc. Even in the case of widely different β_1 and β_2 values, an intermediate common β value can be computed by minimizing a difference function which gives the difference between the original Bézier spline and its approximation with a single $\beta_1 = \beta_2 = \beta$ value. Good visual results are obtained by minimizing the sum of the squares of the distances between points of the new Bézier spline $B^{new}(u)$ at parameter values $u = \{0.1, 0.2, 0.3, 0.4, 0.45, 0.5, 0.55, 0.6, 0.7, 0.8, 0.9\}$ and the original spline $B(t)$. In Tab. 2.1, parameter ξ gives the mean absolute distance between the points of the approximating spline (according to the parameter values given above) and the original

spline. In all cases, the mean absolute error distances ξ are very small when compared with a typical capital letter height value of 800.

For example, let us consider a Bézier curve segment from the internal contour of the lower-case letter “o” of font Palatino, which is a stressed serif font similar to Times. This curve is defined by $B_0 = (125, 257)$, $B_1 = (125, 152)$, $B_2 = (163, 13)$, $B_3 = (291, 13)$, as shown in the left part of Fig. 2.15. Its β parameters are $\beta_1 = 0.430$, $\beta_2 = 0.771$. We adjust the two control points B_1 and B_2 such that $\beta_1' = \beta_2' = \beta = 0.612$ (see Tab. 2.1). The new curve $B^{new}(u)$ is placed overlapped on the old one $B(t)$ in Fig. 2.15a. Fig. 2.15b shows the curvature radii of the two curves. Comparing the new and the old curves and their curvatures, we found that the geometric shape of the two curves are very similar, and that even the curvature of the new curve does not deviate too much from the original curvature.

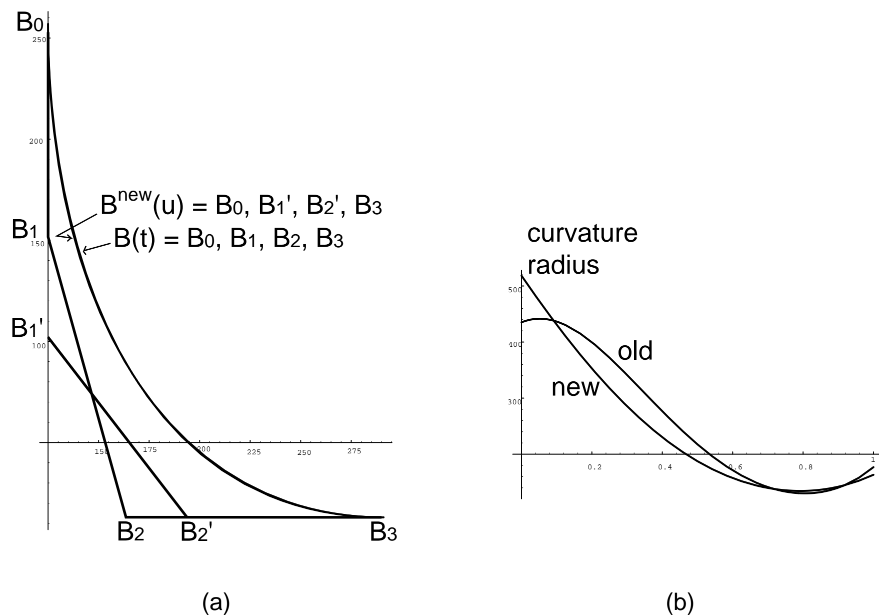


Figure 2.15: The third quadrant of the internal loop of letter “o” of typeface Palatino is replaced by the our β-Bézier curve with equal β values.

2.2.2.3 Curvatures of successively connected curve segments

If the span of a curve occupies more than one quadrant, it will be approximated by two or more successively connected Bézier splines. To connect two Bézier curves $P = (P_0P_1P_2P_3)$ and $Q = (Q_0Q_1Q_2Q_3)$ smoothly, we require the connection points of the curves to meet some continuity conditions [Farin90, pp.185-210] [Foley90, pp.480-482]. We distinguish between zero order, first order and second order geometric continuity as following (Tab. 2.2):

Table 2.2: Conditions of geometric continuity of two connected Bézier curves.

Order of GC	Conditions of Bézier curves P and Q
zero order (CG^0)	$P_3 = Q_0$.
first order (CG^1)	CG^0 and tangent directions at P_3 and Q_0 are the same.
second order (CG^2)	CG^1, normal vectors at P_3 and Q_3 have same direction, and curvatures at P_3 and Q_3 are identical.

It is easy to reach CG^1 continuity, but the condition of CG^2 continuity is not easy to meet. Connecting two cubic Bézier curves with CG^2 continuity results in a *curvature continuous yet not necessarily twice differentiable* long curve. Currently, curves used in outline fonts are only required to reach CG^1 continuity at the connection point. However, typographers design round parts of characters as a whole curve. Sudden changes of curvature along a curve are not intended. So, if we try to approximate the real curve with piecewise Bézier curves, we would like to keep the curvatures at P_3 and Q_0 as close as possible, i.e. try to meet the condition of CG^2 continuity.

From the discussion in the previous section (section 2.2.2.2), we know that Bézier curves with median β parameters have curvatures which do not vary strongly from end points to central part. If we connect two such β-Bézier splines, the curvatures at the connection points will be more likely to be close to each other. This is obviously true when we connect four quadrant β-Bézier splines to approximate a circle. Since the round part of characters are more like parts of an ellipse, we expect that the ellipse approximated by four quadrant β-Bézier splines has also a well behaving curvature.

Let us take a closer look at the curvature radius of an ellipse. Does the β value used for approximating a circle also work for an ellipse? Since an ellipse can be regarded as a scaled circle, we expect that the scale transformation applied to the Bézier curve which is the best approximation of a quadrant circle will generate a Bézier curve which is the best approximation of a quadrant ellipse. This can be proved again by comparing their curvature radii. An ellipse can be presented in a parametric form

$$P(t) = \{x(t) = a \cos t,\ \ y(t) = b \sin t, t \in [0, 2\pi]\} \quad \textbf{(2.10)}$$

where coefficient a is the half axis length in the x direction, coefficient b is the half axis length in the y direction. Using the example in Fig. 2.15, we have $a = (x_3 - x_0) = 291\text{-}125 = 166$, $b = (y_0 - y_3) = 257\text{-}13 = 244$. From (2.2) and (2.10), the curvature radius at the point P(t) = (x(t), y(t)) is

$$R(t) = (a^2 \sin^2 t + b^2 \cos^2 t)^{3/2} / (a\, b) \quad \textbf{(2.11)}$$

Fig. 2.16a shows the curvature radius for the ideal ellipse. Using β = 0.55, each quadrant of the ellipse can be approximated by a β-Bézier curve. The curvature radius of the best approximating β-Bézier spline is shown in Fig. 2.16c. We also show two curvature radii of parameter β = 0.6 and β = 0.5 (Fig. 2.16b, Fig. 2.16d). The curvature radius for β around 0.55 is the most similar to the curvature radius of an ideal ellipse. Let us notice that the parameter of an ellipse and a Bézier curve have not exactly the same

meaning. However, this comparison still holds, since Fig. 2.16(a) and 2.16(b), (c), (d) represent how curvature evolves along the ellipse and the ellipse-like contour.

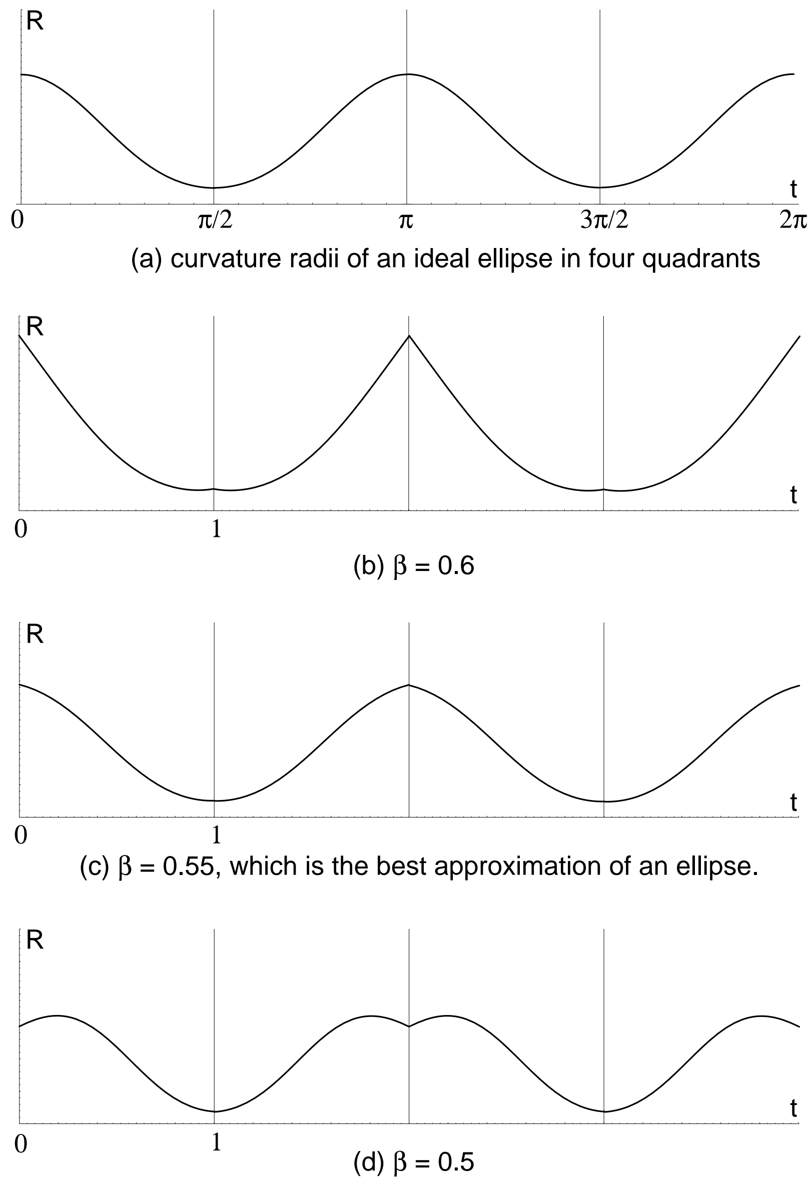


Figure 2.16: Curvature radii of the four quadrants of an ellipse: (a) an ideal ellipse; (b) the ellipse approximated by four Bézier curves with $\beta = 0.6$; (c) $\beta = 0.55$; and (d) $\beta = 0.5$. In (b), (c) and (d), the curvature radius lines of successive Bézier spline segments have been concatenated.

We can prove that the β value which creates ideal curvatures for the two end points when the β-Bézier curve is used to approximate a quadrant of a circle also creates ideal curvatures for the curve's ends when it is used to approximate a quadrant of an ellipse (see Appendix B).

If we model the round part of a character as an ellipse, we can use the Bézier curves with the ideal β value to construct its outline. However, an ideal ellipse may not have the highest visual aesthetic quality for Latin characters. Typographers tend to make the round part a bit more squared, or fatter, than the ideal ellipse. This can be found from a statistics of the lower-case letter “o” in real fonts. Letter “o” always specifies the round part of a letter in a font. In our outline font samples, both the inner contour and the outer contour of the character are designed by four Bézier splines with their connecting end points placed at local extrema. The sample fonts are all from commonly used text typeface. We do statistics on the maximum, minimum and the mean β values of the four Bézier curves (i.e. among eight β values) on the outer and inner contours respectively as well as the mean β values of both contours (Tab. 2.3). The general mean β value of all the inner and outer contours are also computed. These statistics show that different letters “o” designed by typographers have a β value around 0.6 (the general mean β value is 0.601), a little larger than the ideal β of a circle or ellipse.

Table 2.3: Statistics on the β values of some non-fancy fonts.

Font name	**β values of outer cont.**			**β values of inner cont.**			**Mean β value of both contours**
	Max.	**Min.**	**Mean**	**Max.**	**Min.**	**Mean**	
Agfa-Times	0.618	0.556	0.579	0.707	0.523	0.599	0.589
Clarendon	0.639	0.584	0.611	0.659	0.656	0.657	0.634
Courier Bold	0.62	0.538	0.578	0.623	0.557	0.589	0.586
Courier	0.638	0.504	0.563	0.613	0.544	0.57	0.567
Frutiger-Roman	0.6	0.547	0.58	0.692	0.476	0.577	0.579
Lucida	0.66	0.574	0.618	0.742	0.397	0.57	0.594
New Century Schoolbook	0.556	0.556	0.556	0.654	0.649	0.651	0.604
URW Times	0.583	0.558	0.57	0.643	0.597	0.611	0.591
Palatino	0.631	0.561	0.583	0.763	0.43	0.605	0.594
Times-Roman	0.639	0.524	0.581	0.792	0.509	0.601	0.591
Univers	0.66	0.586	0.623	0.753	0.543	0.648	0.636
URW Antiqua Normal	0.641	0.637	0.639	0.675	0.654	0.665	0.652
The general mean β value							**0.601**

To summarize, giving some restrictions to the Bézier curve used in font design is essential to the parametrization of fonts. In our font parametrization method, curves are always limited to a quadrant so as to be represented by β parameters. And frequently, the parameter β_1 and β_2 can be assigned the same value. In this way, we are able to control most of the curve segments with five free parameters instead of eight, and the curve has a better curvature and continuity at the connection point of two curves. In our method, a Bézier curve is often specified by a triangle B_0AB_3 describing the dimension of the curve, and two proportional values β_1 and β_2 controlling its curvature. Point A is the intersection of the extension of vector B_0B_1 and vector B_3B_2 (Fig. 2.12). If a Bézier curve does not exceed one quadrant, point A always exists. Curves which span less than one quadrant can be also represented by β values.

2.2.3 Components for round parts

Round parts are synthesized by components named *loop*, *half-loop* and *sweep*. The *loop* and *half-loop* components aim at synthesizing ellipse-like round parts of characters such as the round parts in characters o, b, d, p, q, g. The sweep component is used to synthesize any round stroke inscribed within one quadrant.

2.2.3.1 The loop component

The *loop* component is defined with the aim of parametrizing the shape of letter “o”, or a round part which looks like “o”. A loop has an internal and an external contour, both are ellipse-like and symmetric in respect to their center points. The contour therefore can be modelled by four quadrant β-Bézier splines. For example, character Helvetica “o” can be modelled by a single loop component, the exterior, respective interior contours being defined by 4 connected quadrant arcs having a common center (Fig. 2.17a). Character Times “o” is, however, more complex. While its exterior contour, looking like an upright ellipse, can be modelled in the same manner as the contours of character Helvetica “o”, its interior contour looks like an ellipse with an oblique orientation and would require circumscribed quadrants with 4 different centres in order to describe it by quadrant arcs (Fig. 2.17b).

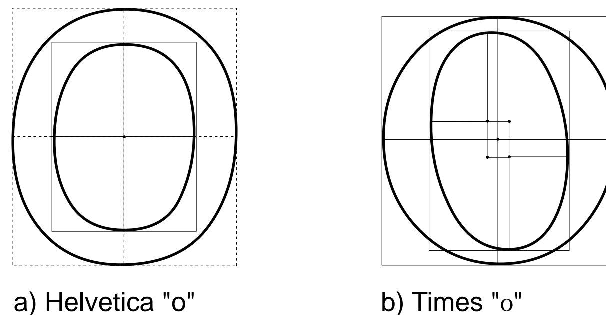


Figure 2.17: (a) Helvetica “o” is upright both on the internal and external contours, while (b) Times “o” has oblique stress on its internal contour.

If we model the oblique stress of the contour as the rotation of an ellipse, we obtain curve segments whose end points are not located at local extrema (Fig. 2.18a). This is not what we want. We prefer to have curve end points at extrema. Therefore, we model the obliqueness of ellipse-like contours with parameter η which gives the offset of extrema from the upright ellipse. Given half width *p* and half height *q* of the bounding box, *Δp* and *Δq* being the corresponding displacements at extrema, the parameter η is defined as

$$\eta = \frac{\Delta p}{p} = \frac{\Delta q}{q} \qquad (2.12)$$

This formula implies that in Fig. 2.18b, line VH connecting the horizontal and vertical tangential points is parallel to line AB connecting point $A = (o_x - p, o_y)$ and $B = (o_x, o_y + q)$ of the ellipse's bounding box. For the proof, see Appendix C.

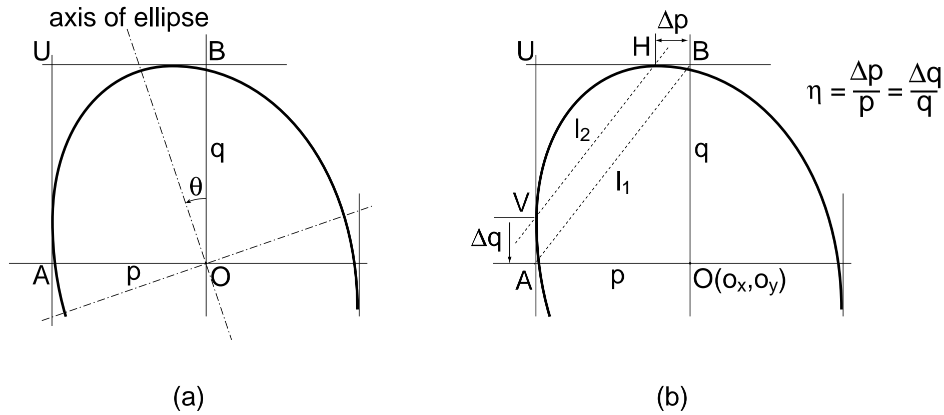


Figure 2.18: (a) Modelling the obliqueness as rotation of an upright ellipse results in end points which are not located on local extrema. (b) Modelling obliqueness by parameter η.

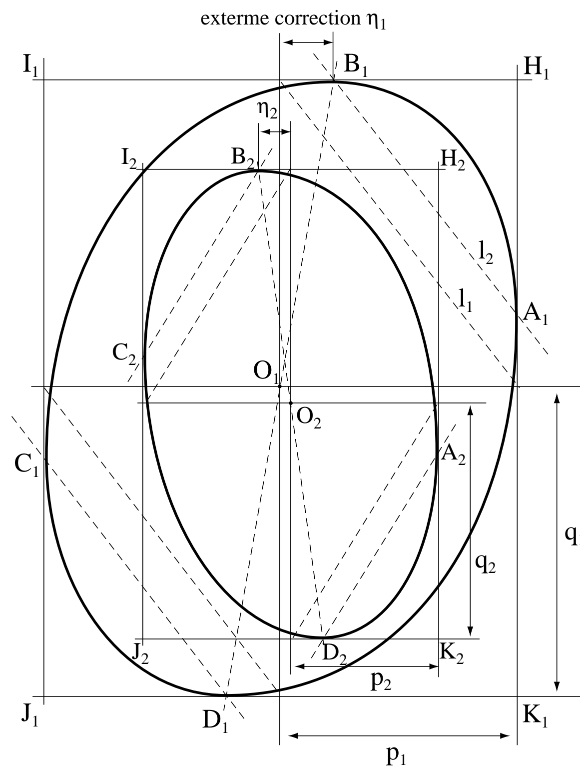


Figure 2.19: Parameters of the loop component: O_1, O_2, p_1, p_2, q_1, q_2, η_1, η_2.

Given the center of an ellipse-like contour, the half width p and half height q of the bounding box, the contour's location, dimension and bounding box can be calculated. The parameters of the external contour and the internal contour of a loop component are: the center points O_1 and O_2, the half width of bonding box p_1 and p_2, the half height of the bounding box q_1 and q_2, and the extrema correction parameters η_1 and η_2. Note that, both the internal and the external ellipse-like contours are symmetric in respect to their respective center points. Hence, the four quadrants of the loop component can be generated (Fig. 2.19). We label the external contour by four successive control triangles of β-Bézier curves ($A_1H_1B_1$, $B_1I_1C_1$, $C_1J_1D_1$, $D_1K_1A_1$) in counterclockwise orientation (positive orientation), and label the internal contour as ($A_2K_2D_2$, $D_2J_2C_2$, $C_2I_2B_2$, $B_2H_2A_2$) in clockwise orientation (negative orientation).

The curvature of the internal and external contours is controlled by the β parameters of the quadrant Bézier splines. To ensure the coherent appearances of loops in different characters across a font, the β parameter of loops is provided as a global font parameter (see section 3.3.3). With η controlling stress or obliqueness, and β controlling curvature, different styles of the character "o" can be simply synthesized by the loop component (Fig. 2.20).

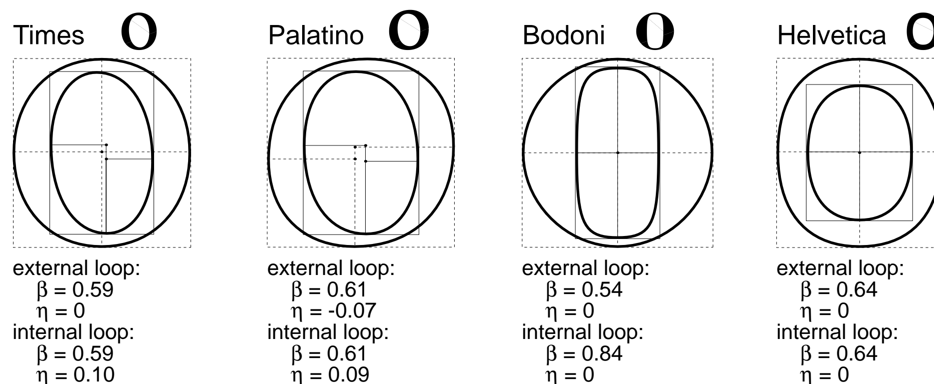


Figure 2.20: Synthesizing character "o" in different typefaces.

2.2.3.2 The half-loop component

The round part of many characters can be considered as a half (upper, lower, left or right) of a full loop (Fig. 2.21a). The *half-loop* component is designed to synthesize this kind of round structure element. Besides all the parameters of the loop component, the half-loop component needs an additional parameter specifying "which" half of the full loop it is. This parameter is a tag with four possible values specifying left, top, right and bottom orientation respectively. Since a the shape of a half-loop has no internal contour, the synthesized shape of a half-loop has only one contour which can be labelled by its vertexes as *ABCD* in counterclockwise orientation (Fig. 2.21b).

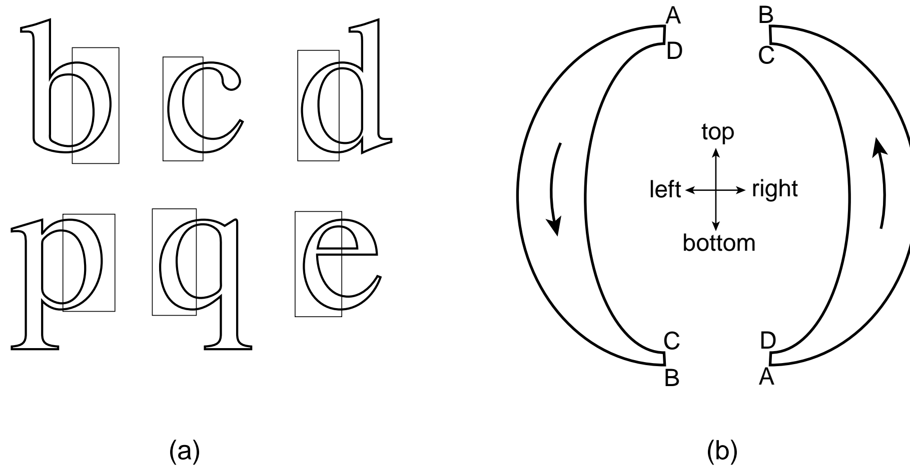


Figure 2.21: (a) Examples of the half loop component, and (b) orientation and labelling of contours of half loops.

2.2.3.3 The sweep component

Half-loops can be used for synthesizing the round parts of letters, such as “b”, “d”, “p” and “q”. When looking at one of these characters (Fig. 2.22a), one can see that the curved part connecting the loop to the bar shows a particular behaviour. Its parameters are not directly related to the parameters of the half-loop. Similarly, the arches of characters “h” (Fig. 2.22b), “n”, “m” and “u” cannot be conceived as half-loops. They are curved character parts connecting two vertical stems and therefore need their own shape description.

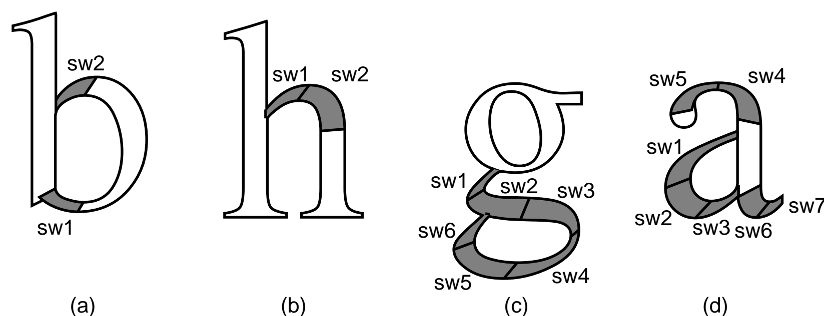


Figure 2.22: Example of the sweep components.

In order to support connecting elements made of curved parts, we introduce a shape primitive named the *sweep* component. The sweep component is used for establishing the connections between a loop and a vertical stem (Fig. 2.22a) and between

two vertical stems (Fig. 2.22b). It can also be used for creating curved parts such as the tail of character “g” (Fig. 2.22c) and the round parts of characters “a” (Fig. 2.22d), which cannot be modelled by loops and half-loops.

A *sweep* component is defined by a departure segment, an arrival segment, and a center line connecting the center point of the departure segment and the center point of the arrival segment (Fig. 2.23). The departure segment is given by end points P_d and Q_d. The arrival segment is given by end points P_a and Q_a. The center line is represented by a Bézier curve $B(t) = B_0B_1B_2B_3$. Since B_0 is the center of segment P_dQ_d, and B_3 is the center of segment P_aQ_a, only the control points B_1 and B_2 need to be provided as parameters. We can also specify the center line by a β-Bézier curve which requires one control point A and two β parameters. The *sweep* component intends to imitate the trajectory made by a flat pen moving along the center line from the departure position to the arrival position. The *sweep synthesizing algorithm* which calculates the two edges (P(t) and Q(t), in Fig. 2.23) of the pen trajectory determines the resulting shape of the sweep component.

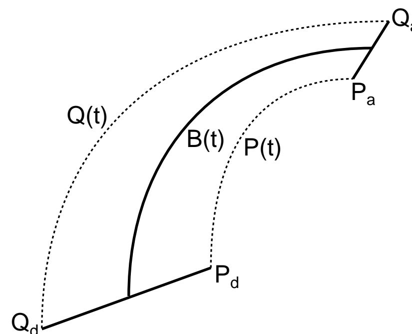


Figure 2.23: The *sweep* component is given by a departure segment, an arrival segment and a Bézier curve connecting their center points.

The first idea of computing intermediate curve points between the departure end and the arrival end might be by interpolation. One can design algorithms to interpolate both the pen orientation and the pen width at a intermediate position between the departure and the arrival ends, and thus calculate the points on both edges (P(t) and Q(t)). We have tried several interpolating methods. However, we found that interpolating methods suffer from two shortcomings: (1) they are not efficient since interpolating requires many calculations; and (2) it is difficult to control the tangent directions at both departure and arrival ends. Therefore, we developed a sweep synthesizing algorithm based on the geometric transformation of the center line. This algorithm is used in our font parametrization system.

Our sweep synthesizing algorithm consists in scaling and translating the center line from $B(t) = B_0B_1B_2B_3$ to $P(t) = P_dP_1P_2P_a$ and $Q(t) = Q_dQ_1Q_2Q_a$ respectively. Experiences show that the tangent directions at the ends are very important for the synthesis of connecting sweep shapes, especially when the sweep is supposed to connect

smoothly to another sweep or half-loop. Most of the sweeps used in our font parametrization method are within one quadrant. For this kind of sweeps, our sweep synthesizing algorithm ensures that the tangent directions at P_d and Q_d are identical to the tangent direction of the center line at B_0, and that the tangent directions of P_a and Q_a are identical to the tangent direction of the center line at B_3. In Fig. 2.24, we assume line P_dP_A is parallel to line Q_dQ_A, and line P_aP_A is parallel to line Q_aQ_A. This algorithm can be generalized by the following steps:

1. Regarding the center line as a β-Bézier which is represented by its control triangle $B_0B_AB_3$, draw a line P_dP_A from P_d so that line P_dP_A is parallel to line B_0B_A, draw a line P_aP_A from P_a so that line P_aP_A is parallel to line B_3B_A; line P_dP_A intersects line P_aP_A at point P_A.
2. Similarly, draw a line Q_dQ_A from Q_d so that line Q_dQ_A is parallel to line B_0B_A, draw a line Q_aQ_A from Q_a so that line Q_aQ_A is parallel to line B_3B_A; line Q_dQ_A intersects line Q_aQ_A at point Q_A.
3. Let $\beta_1 = |B_0B_1| / |B_0B_A|$, $\beta_2 = |B_3B_2| / |B_3B_A|$, find point P_1 on P_dP_A so that $|P_dP_1| / |P_dP_A| = \beta_1$, find point P_2 on P_aP_A so that $|P_aP_2| / |P_aP_A| = \beta_2$; then $P_dP_1P_2P_a$ is the control polygon of the Bézier curve P(t).
4. Similarly, find point Q_1 on Q_dQ_A so that $|Q_dQ_1| / |Q_dQ_A| = \beta_1$, find point Q_2 on Q_aQ_A so that $|Q_aQ_2| / |Q_aQ_A| = \beta_2$; then $Q_dQ_1Q_2Q_a$ is the control polygon of the Bézier curve Q(t).

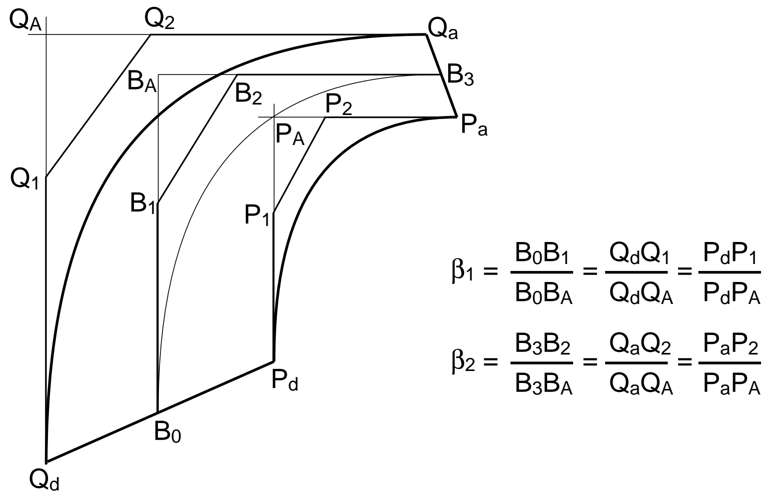


Figure 2.24: The sweep component generated by scaling and translating the centerline.

The triangle $B_0B_AB_3$ and $P_dP_AP_a$ or $Q_dQ_AQ_a$ are not required to be similar. By keeping the same β values, we ensure that the transformation from $B_0B_1B_2B_3$ to $P_dP_1P_2P_a$ and to $Q_dQ_1Q_2Q_a$ are affine transformations. Therefore, the Bézier curves $P(t) = P_dP_1P_2P_a$ and $Q(t) = Q_dQ_1Q_2Q_a$ are affine-transformed from the center line

$B(t) = B_0B_1B_2B_3$. These affine transformations ensure that most of the properties of a Bézier are kept when the curve is adapted to a new dimension. Practically, we found that this algorithm not only keeps the tangent direction at both ends, it also keeps the orientation change in the generated curve close to the orientation change of the original centerline curve. Curve P(t) and curve Q(t) are perceived as harmonious.

Note that there is an important limitation of this algorithm. The Bézier curve of the sweep's center line must not exceed one quadrant and must be without inflection point. Otherwise, the curve cannot be represented in β-Bézier format.

2.2.4 Components for terminals

Terminal structure elements represent serifs, slant serifs, dots, pears and any shapes used to terminate body strokes. Terminals convey very significant information about font styles, and hence have many variations and are very flexible. Components used to synthesize shapes of terminals are *serifs*, *slant-serifs*, *dot* and correcting *paths*.

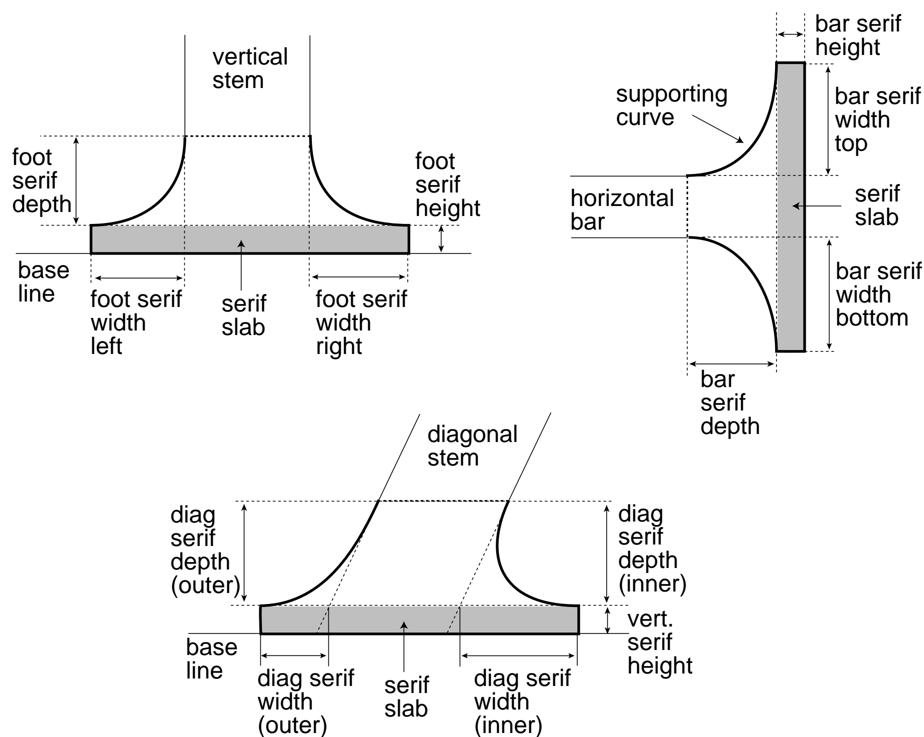


Figure 2.25: The general description of foot, bar and diagonal serifs.

2.2.4.1 The serif component

Serifs are the most important terminal elements which can be used to terminate vertical stems, horizontal bars or slanted stems. The foot and bar serifs and their respective width, height and depth parameters (Fig. 2.25) have been extensively described in the lit-

eratures [Knuth86b] [Karow94]. An essential part of foot and bar serif is the *slab* which is a line (thick or thin) terminating the respective stem or bar. The corner formed by the serif slab to its respective stem or bar is often made round by using a curve.

(1) Basic serif design

The basic shape primitive we designed for serifs is the *serif* component, also called the *foot-serif* component (Fig. 2.26). The basic parameters of the *serif* component are serif width sw_1 and sw_2, serif height sh_1 and sh_2, and serif depth sd_1 and sd_2. The parameter *dir,* which has values left, right, top and bottom indicates the direction to which the serif faces. The serif support lines l_1, l_2 and l_3 define the dimension and position of the serif (Fig. 2.26a and b) which usually are derived from the stem to which the serif connects. The support lines are not provided directly, but derived from the stem component to which the serif connects. The two round corners ABC and HGF are again β-Bézier splines even though the corner may be sharper than 90 degree (Fig. 2.26b). This is a special case where we allow β-Bézier splines to occupy more than one quadrant. The resulting shape of the serif component is represented as contour *ABCDEFGH*.

Direct parameter: Most parameters have values derived directly from global or local font parameters. We call them direct parameters. Direct parameters are for example the stem width, the curved stroke width and the height and width of serifs.

Indirect parameter: An indirect parameter depends on other components. Indirect parameters are for example required to keep a relation between two components such as a stem and its connecting serifs. The support lines l_1, l_2 and l_3 of the serif component are an example of indirect parameters. More about how to provide a component with parameters will be discussed in the next chapter.

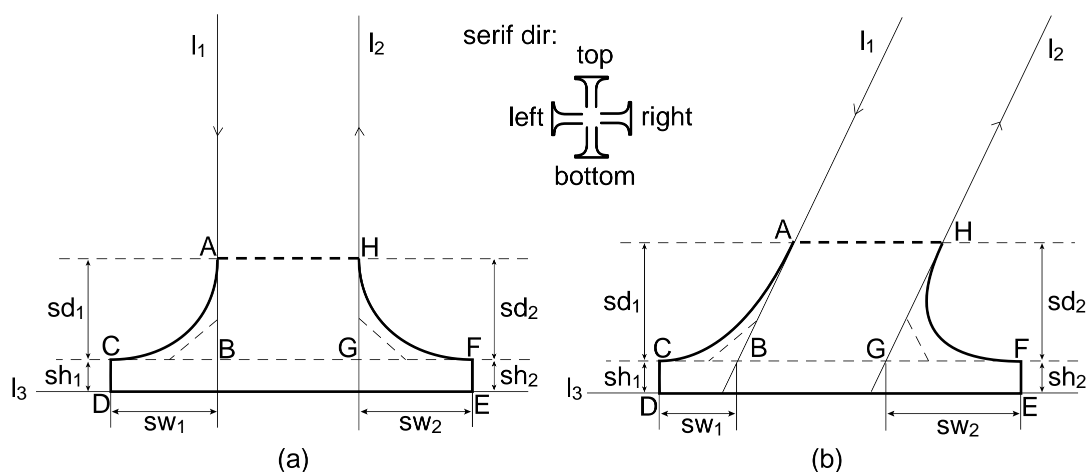


Figure 2.26: Basic parameters of the serif component.

(2) Enhanced serif design - attributes of serif types

In order to synthesize the various types of serifs described in typeface classification books [Rockledge91] [Bauermeister87], we introduce *attributes*, which are regarded as

global parameters for the serif component synthesizer. They describe variations in serif shapes. Currently, we have three serif attributes: *serif support type*, *serif end type* and *serif face type*.

The curves at the corner ABC and HGF are called *serif supports*. The type of a serif support can be *smooth*, *angled*, *straight* or *none* (Fig. 2.27). The smooth serif support connects both the stem and the serif slab smoothly, as for example the serifs of the Times typeface. The angled serif support connects the stem smoothly but may connect the serif slab with an angle. The straight serif support connects the stem to the slab by a straight line, and hence generates a triangle serif. The support type none means there is no curve or line to fill the corner ABC and HGF. This serif support type is designed for slab/square serifs or thin line serifs [Bauermeister87, pp.ix-xi].

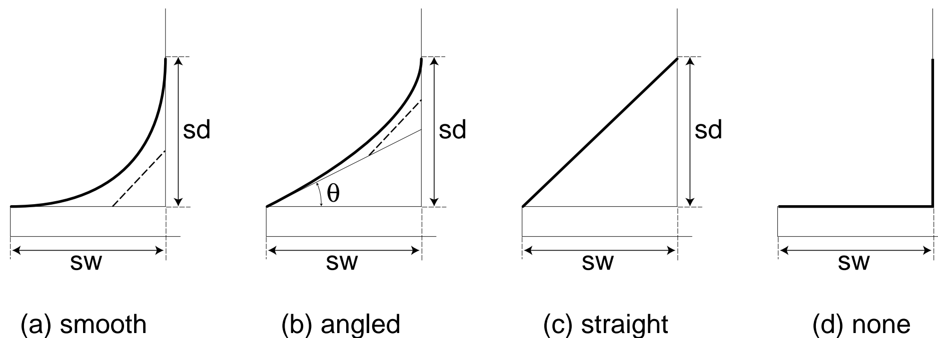


Figure 2.27: Serif support types.

The line CD and EF of the serif component are called *serif ends*. Serif ends can be *butt* or *rounded* (Fig. 2.28). The butt serif end simply connects C to D and E to F with two straight line segments, as in the serifs of Times Roman. The rounded serif end connects both C to D and E to F with two semi-ellipses constructed by two Bézier curves each. The extension of the rounded end is given as a global parameter.

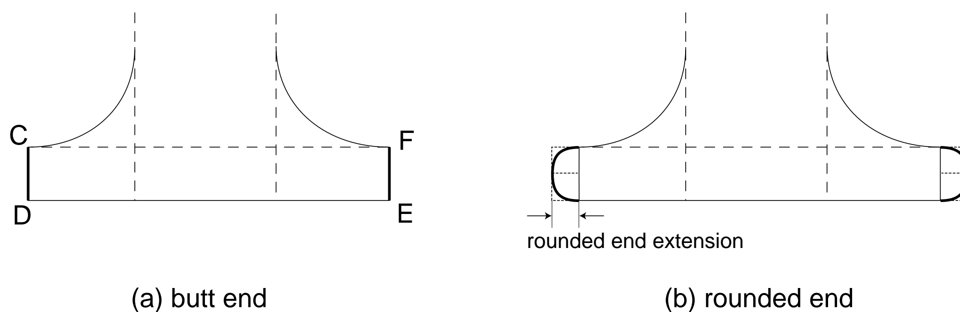


Figure 2.28: Serif end types.

The line DE of the serif component is called *serif face*. Enhanced serif design enables the serif face to be *flat*, *smoothly concave* or *directly concave* (Fig. 2.29). The flat serif face simply connect D and E with a straight line, as in Times Roman. The smoothly concave serif face and the directly concave serif face both connect D to E with a concave curve made of two Bézier spline segments. The curve of the smoothly concave serif face connects the serif slab (support line l_3) without an angle, i.e. at the junctions, serif face and slab lines have an identical orientation. The depth of the concave serif face is given as a global parameter.

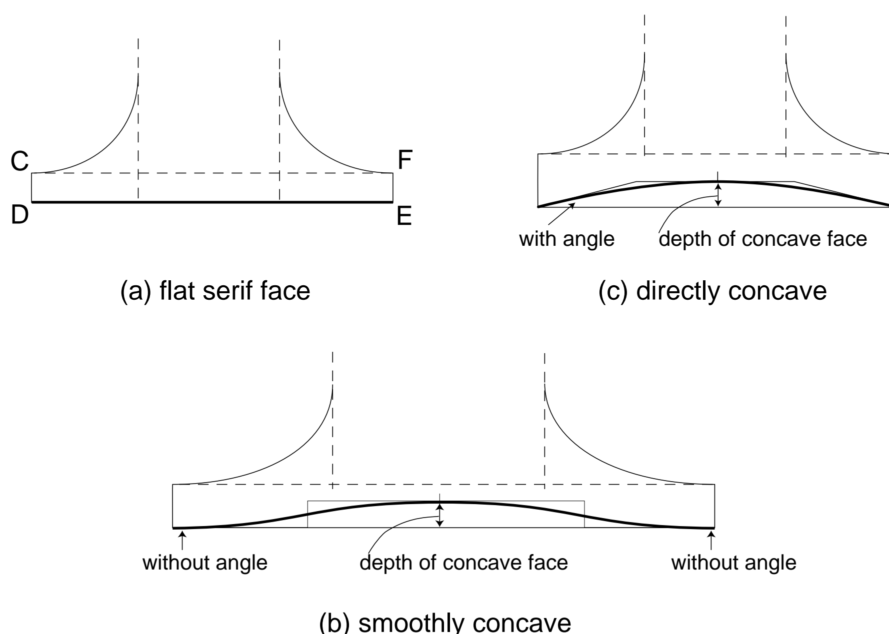


Figure 2.29: Serif face types.

With these enhanced serif component types, we are able to synthesize most of the serif styles, such as transitional serifs, slab serifs and wedge serifs. For example, we demonstrate in Tab. 2.4 how the serif classification according to Rockledge's Typefinder [Rockledge91] and the serif classification according to Bauermeister's PANOSE system [Bauermeister87] are synthesized with the enhanced design of the serif components. The classification in the Typefinder is based on the history of characters, while the PANOSE system emphasizes on the geometrical shapes of characters. This table tries to match the classification of serif types in both systems and describes these serifs with the attributes (options of serif styles) of the serif or top serif components.

We are also aware that the serif style attributes may be extended in order to synthesize a specific serif style which is not appeared in Tab. 2.4.

Table 2.4: Analysis of typographical serif types and our enhanced serif component.

Typefinder's classifications	PANOSE system's classifications	Description with attributes of the serif/topserif components
Venetian Serif Old Style Serif Transitional Serif New Transitional Serif	Cove Serif Square Cove Serif	Most Roman typefaces have these kinds of bracketed serifs: serif support = smooth/angled serif end = butt/round serif face = flat/concaved.
Modern Serif	Thin Line Serif	High contrast Roman typefaces, often no bracket: serif support = none serif end = butt serif face = flat
Slab Serif	Square Serif	No bracket, or little bracket: serif support = none or smooth with very high β values serif end = butt (big serif height) serif face = flat
Wedge Serif (Hybrid Serif)	Triangle Serif Exaggerated Serif	Triangle serif: serif support = straight serif end = butt (very small serif height) serif face = flat Unable to do exaggerated serifs.
Sans Serif	Sans Serif (stroke with various end)	null serif components (stroke end controlled by the parameter of stem component)

2.2.4.2 The slant-serif component

The *slant-serif*, also called *top-serif*, component is designed mainly for top serifs. We tend to called it slant-serif because it is not only for the top serif, but can also be used to synthesize the shape of beaks, tips and etc., for example in character "z" and "T" (Fig. 2.30).

The slant serif can be modelled by the intersection of the stem and a small diagonal slab (Fig. 2.31), similar to the serif component. Support lines l_1, l_2, l_3 are support lines derived from the stem it connects to. Parameter θ gives the slant angle between the slab and the support line l_3. The serif height *sh*, serif width *sw* and serif depth *sd* parameters are defined similarly to corresponding parameters of the serif component. Since a *slant-serif* often has a tip overshoot over the support line l_3, generally for optical correction, the additional parameter *overshoot* specifies the amount of overshoot. In the basic slant serif design, the corner DEF and the face ABC are each connected with a Bézier spline with curvature controlled by β values. The parameter *dir* describes the orientation of the slant

serif, which can be top-right (*tr*), top-left (*tl*), bottom-right (*br*), bottom-left (*bl*), right-top (*rt*), right-bottom (*rb*), left-top (*lt*) and left-bottom (*lb*).

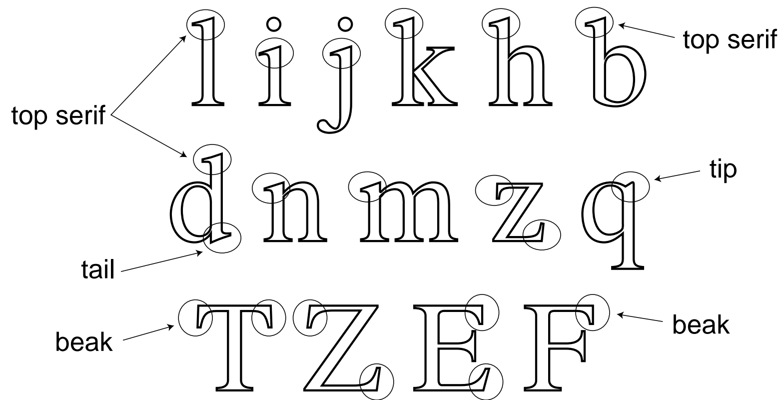


Figure 2.30: The slant serif component can be used for many purposes.

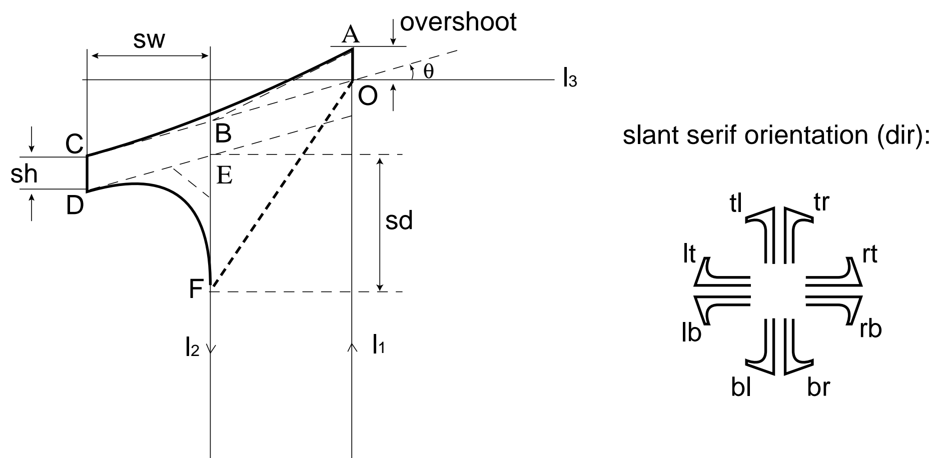


Figure 2.31: Basic parameters of the slant serif component.

Enhancements similar to the ones applied to the serif component can be also applied to the slant serif component in order to synthesize variations of the *serif support*, *serif end* and *serif faces*. Readers are referred to the previous section (section 2.2.4.1).

2.2.4.3 The dot component

The *dot* component synthesizes ellipse-like character parts, such as the dot on top of the character "i" and "j", and the terminal (pear/bulb) of round strokes in character "a", "f", "j" and "r". It can be modelled as an ellipse with a given rotation angle. Therefore, parameters are the ellipse center O, two axes of the ellipse a and b, and the rotation angle θ. The method to represent the rotated ellipse is different from that of the loop

component. Here, the rotated ellipse is generated by an upright ellipse, constructed by four quadrant Bézier splines and rotated by the rotation angle θ. Hence the end points of the Bézier may not be located at the local extrema (Fig. 2.32). The resulting contour can be labelled by its vertexes as *ABCD*.

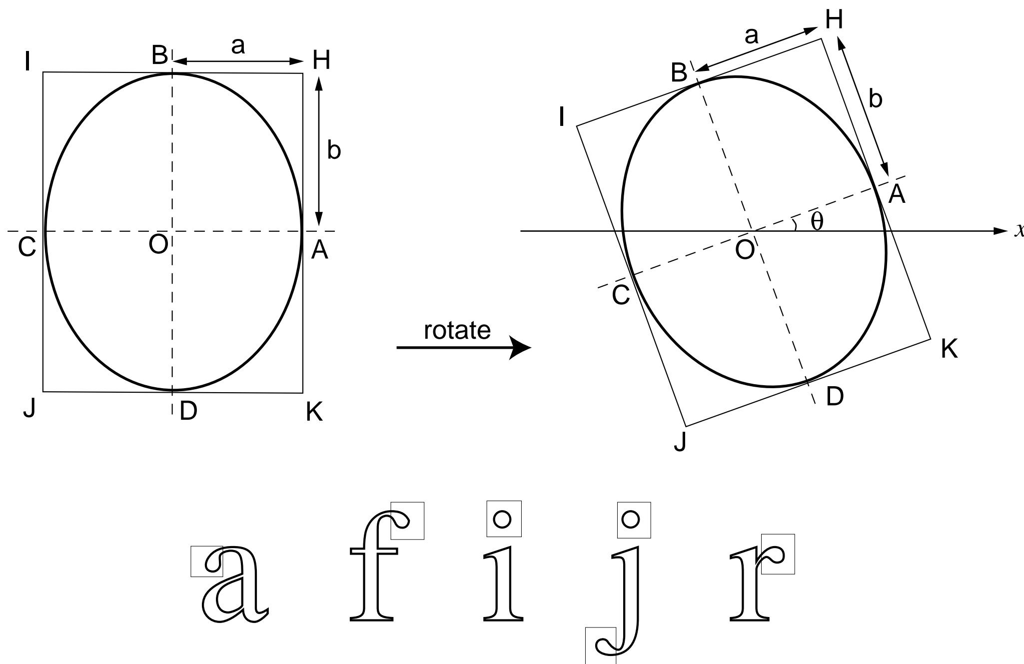


Figure 2.32: The dot components and its examples.

There are some reasons to allow end points of Bézier spline segments not to be located on local extrema. First, the dot components are generally small parts of characters. The advantage of placing end points at local extrema is not obvious here. Second, an isolated dot in character, such as “i” and “j”, is often upright; however the terminating dot/pear/bulbs may have an important rotation angle, which cannot be modelled by parameter η as in the loop component. And last, rotating the axes of an upright ellipse makes it easy to connect the dot to the end of a round stroke, enforcing the same orientation for the round stroke and for one of the ellipse’s axes.

As an extension, the dot component can also be a rectangle for the purpose of synthesizing the square dot in some sans-serif font, such as “i” and “j” in the Helvetica typeface. No additional parameter is needed except a flag giving the type (ellipse or rectangle) of the dot. If the type is rectangle, the dot component will synthesize a rectangle by four straight line segments instead of quadrant Bézier splines. By the way, a rectangle dot can also be synthesized by a stem component.

2.2.4.4 The path component

One cannot expect to parametrize all kinds of terminals with a limited set of predefined shapes since the freedom of the typeface designer cannot be limited. With the components discussed above, we are able to synthesize nearly all body and terminal parts of regular text typefaces. However, there do exist some terminals which cannot be directly synthesized by the predefined components. For example in the Times Roman typeface, some sort of smoothing curves are needed for the connection of a round stroke to a dot terminal in characters “j” and “f”. The cross junction of character “t” also needs to be corrected by a special triangle-like shape (Fig. 2.33).

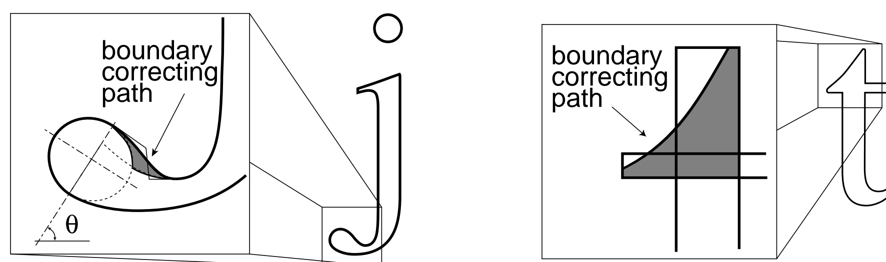


Figure 2.33: Example of the boundary correcting path.

The *path* (or *boundary correcting path*) component is designed mainly for correcting the shapes built by those predefined components. It has no predefined parameters. The shape of a correcting path is constructed by lines and Bézier curves given explicitly. Correcting paths are parametrized since their coordinates are only allowed to have relative values, not absolute numbers. These coordinate values are given as proportions of global font parameters such as *x-height* and *caps-height*, and other components which have already been parametrized. To be consistent with other components, correcting path shapes should also be closed.

Theoretically, the flexibility of the path component enables us to construct any arbitrary shape. One may notice that it introduces additional complexity into the font parametrization. Fortunately, we use the correcting path only for a few shape corrections. Depending on implementation, we may generate the correcting paths automatically by designing specific operations, such as the *smooth* operator (section 3.2.3.2) which adds a smoothing shape to a corner formed by the intersection of two components.

2.3 Summary

In this chapter, we studied the structure of typographical characters and designed components to synthesize typographical structure elements.

The structure graph we presented is based on the invariant typographical structure features of characters. This structure graph differs from the skeleton models by the fact

that it combines the connection type between structure elements and the refinable existing structure elements into one graph. In the next chapter, we will see that the structure graph is useful for explaining and specifying the font parametrization.

The basic set of components we designed enables us to synthesize most structure elements of text typefaces. The idea of using intelligent shape primitives to generate typographical character parts creates the fundament of our font parametrization method. Through systematic experiments, we introduced the β and η parameters for describing single curves and loops. Since these parameters reflect existing typographic and aesthetic properties, the components' round parts behave closely to what typographers may expect in response to variations of weight, obliqueness and contrast.

The stem component is used for the synthesis of all straight parts (vertical, hotizontal or diagonal) of characters. Round parts are synthesized by the loop/half-loop component and the sweep component. The sweep component can also be used for the synthesis of curved connections between character parts. The serif component has different style options (attributes) enabling the synthesis of different serif types. The dot component is used to synthesize the isolated dot as well as to synthesize terminals of curved strokes.

CHAPTER 3

Font parametrization

In the previous chapter, we have discussed the structure element connecting graph and the basic character components. In this chapter, we will present how parametrizable characters and fonts are synthesized.

First of all, the framework of our font parametrization system consists of two main parts: the *parametrizable character synthesizer* and the *parameter files*, as shown in Fig. 3.1. The parametrizable character synthesizer consists of font-independent *parametrizable character synthesis methods* (one for each character) and a *library of components and operations* that is used by the synthesis methods. The font-independent parametrizable character synthesis methods define for each character the component shapes and their connection types. The refined structure graph of each character gives a visual representation of component shapes and connection types. Parameters (which can be regarded as function arguments) are used to control shapes, positions and orientations of components. Hence they are font specific. The *parameter files* contain concrete parameter values for each parametrizable font, including those for global font styles and those for individual character features. Parameter files have been organized into hierarchical layers. The parametrizable character synthesis methods contain *parameter entries* to be filled by parameters from the parameter files. The parametrizable character synthesizer has a *component and operation library* containing procedures for creating component shapes, and geometric algorithms for realizing assembly operations in order to assemble components into a full character.

In order to completely specify font-independent parametrizable character synthesis methods, we first need to refine the structure element connecting graph for all the typefaces which we would like to synthesize. Typefaces whose structure does not differ, but whose height proportion, weight and contrast differ can be synthesized by just modifying parameters. However, typefaces whose structure differ require different parametrizable character synthesis methods. Let us first present the refinement of the structure element connecting graph before we present the parametrizable character synthesis methods. The parameter file organization is another important aspect which will be also discussed in detail in this chapter.

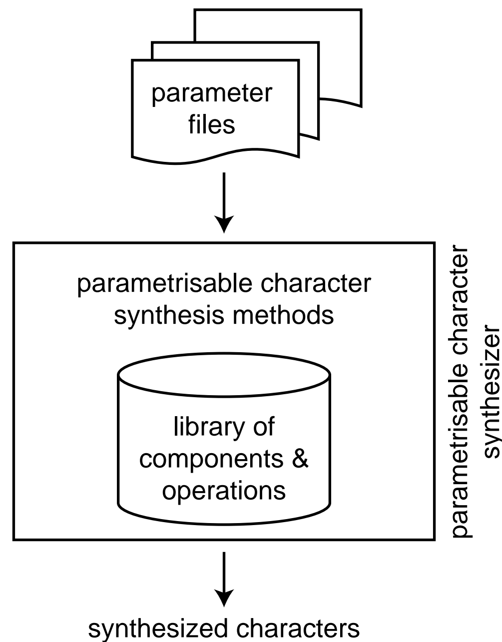


Figure 3.1: Framework of our parametrizable font synthesis system.

3.1 Earmark-based refinement of structure graphs

A structure element connecting graph is a typeface-independent representation of the spatial relationship between structure elements of a character. To synthesize the structure elements, we have designed several shape primitives called components. Each structure element can be synthesized by one or more components depending on its concrete shape.

However, there are so many kinds of physical character embellishments among all the typefaces that one refined structure graph for each character can hardly cover all typefaces. We would like to resynthesize most of the commonly used text typefaces. The question is how many variations of the refined structure graph will be needed to represent a letter in all target typefaces. The answer resides in both the flexibility of the components we are designing and the number of styles that we would like to support. We use components not only to synthesize the shapes of concrete character parts, but also to modify the styles of the junctions between structure elements. To create variations of refined structure element connection graph, we will analyse traditional typeface styles. Based on this analysis, we deduce the required variations.

3.1.1 Serif styles

The most important feature of a typeface is the serif style. It is the first rule that typographers apply to classify or identify typefaces [Rockledge91], [Bauermeister87]. The enhanced serif and slant-serif terminal components introduced in section 2.2.4 are

supposed to cover all the serif styles. According to Tab. 2.4, serifs of most text typefaces can be synthesized by our predefined serif and top-serif components. The serif style therefore can be selected by just one font parameter. If a terminal in a structure graph is supposed to be a serif or a slant-serif, it can be represented by just one serif component or one top serif component respectively. The serif or slant-serif component synthesis method (library functions) will look up the global serif style parameters to generate the serif or slant-serif components.

3.1.2 Analysis on common earmarks

Since serif terminals are easy to refine, structure graph refinements are mostly due to other typeface features, especially those which identify typefaces. Features which obviously identify or classify typefaces are called *earmarks*. Again, we refer to the classical typographic work - C. Perfect and G. Rookledge's book "International Type Finder" [Rockledge91], which collects more than 300 traditional text typefaces. They have divided earmarks of these typefaces into two kinds: *common earmarks* which are typical common features that can be used to identify a typeface, and *special earmarks* which are distinctive ones giving special ornamentation to letters in specific fonts. Earmarks can be any typeface features including weight, contrast, connection style, terminal style and height proportion. Among them, many earmarks are decorations of stroke ends (or terminals) and styles of connections in a character.

Not all of the features which earmarks represent need to be present in the refined structure graph. Only earmarks which represent features of terminals and connections between structure elements need to be considered.

Let us consider refinements of terminals and junctions. Examples are taken from the set of lower-case roman characters. Terminal and junction types for upper-case letters are similar but simpler. Sometimes it is not easy to tell whether an earmark is related to a terminal or a junction, for example in character "t" and "w". In these cases, we assume it is related to a terminal.

1) Variations of terminals

There are two sorts of terminals: the serifs which terminate straight strokes and the terminals which terminate curved strokes. For serif typefaces, terminals of curved strokes, such as bowls, arches and tails, are often bulbs (or pears) and bars. For sans-serif typefaces, there is often no special decoration at the end of curved strokes. Curved strokes of sans-serif typeface are terminated abruptly, leaving the end squared or pointed (Fig. 3.2a). We shall also enable variations to provide terminals of special letters such as the lower terminal of "b", the upper terminal of "t" and the upper-center terminal of "w" (Fig. 3.2b).

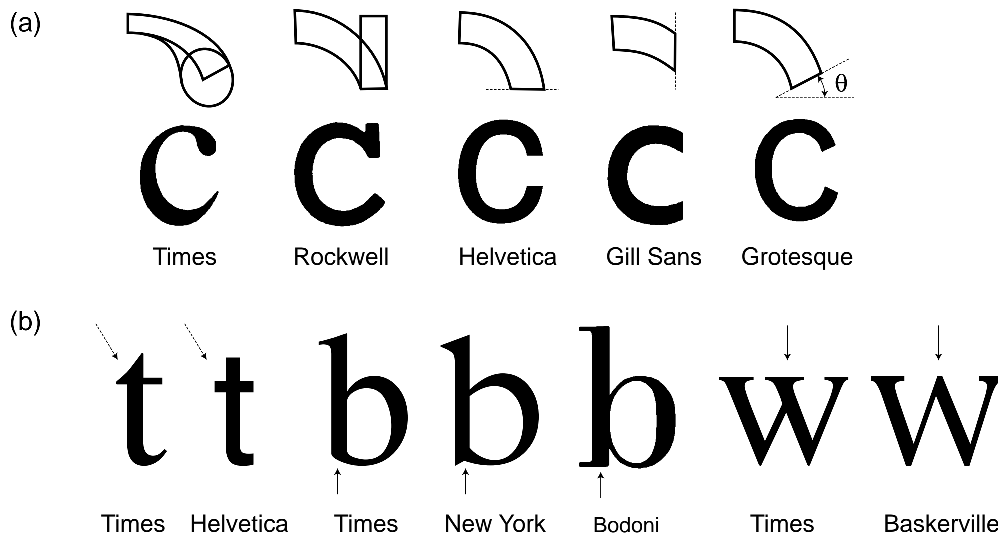


Figure 3.2: Terminals of (a) curved strokes and (b) special letters.

2) Variations of junctions

Junctions, or connection types have been classified to be *meet*, *link*, *join* and *cross*. There is no variation for the *meet* connection. The width of the miter tip of the *join* connection, such as “v” and “z”, can be controlled by the distance and orientation of the slanted stems (to be discussed later), therefore no variation is needed. The *cross* junction has no variation either. Only the *link* junction which connects an arch or bowl to a stem needs variations: *smooth* or *angled* (Fig. 3.3a). The connection of the two diagonal bars to the vertical stem in letter “k” is an interesting case which needs variations to distinguish the order of connections (Fig. 3.3b).

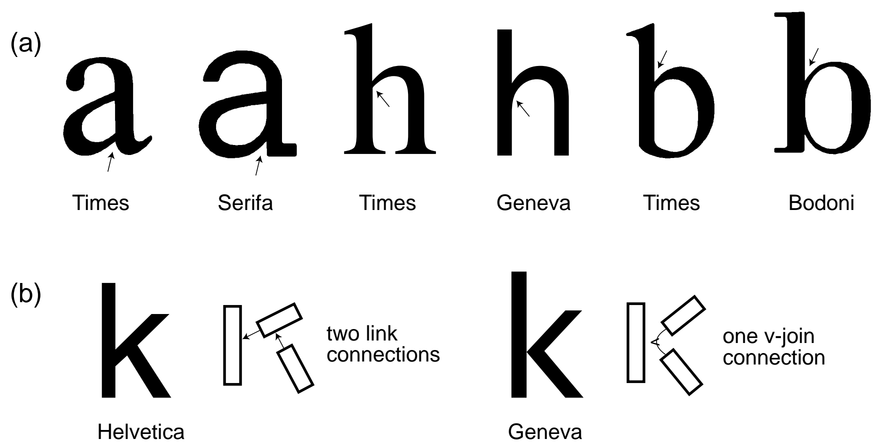


Figure 3.3: Junction variations: (a) angled or smooth *link*, (b) letter “k” with either two *link* connections or one *v-join* connections.

3) Variation of global character structure

Besides the local variations of terminals and junctions, some characters have different global structures, such as the double storey “a” vs. the single storey “*a*” (Fig. 2.8) and the “g” with a belly vs. the “g” with a tail. If the shape of a specific character can not be obtained by appropriate combinations of components, a new global structure can be introduced.

3.2 parametrizable character synthesis methods

The refined structure element connecting graph is an abstract character part specification. This specification is used to write the code of the *parametrizable character synthesis method* of a given character.

The parametrizable character synthesis method requires for each component its corresponding parameters. The parameters of a component specify the position of the component and its precise shape. Certain relationships between components must be maintained. The shape of a component is controlled by the dimension, orientation, angle and curvature parameters. The shape of a component (for example the extension of a stem) may also need to be trimmed in order to fit well with other components. The synthesized character shapes, or glyphs, will be represented in the form of trimmed component outlines, which in turn can be intersected and merged into glyph outlines or scan-converted into glyph bitmaps.

3.2.1 Component position dependency

The first step is the placement of each component at a proper position. The vertical position is controlled by the traditional reference lines. In respect to the horizontal position, the origin and the left side bearing point are not specified in the parametrizable character synthesis method. We can generally place the first synthesized component at a random horizontal position. Positions of the other components are relative to this first component.

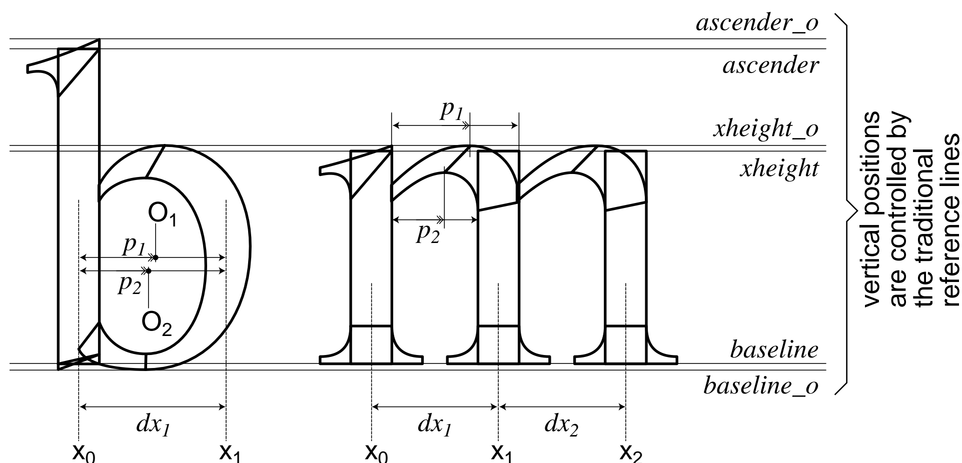
3.2.1.1 Relationships between components

There are two kinds of component-to-component relationships: the distance between two components, such as two vertical stems, and the connection of two components, such as a serif and its connecting stem. Some relationships must be maintained to ensure that, providing any proper parameters, the font synthesizer can generate reasonable character glyphs.

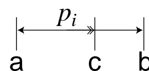
Relationships between character parts were used in the traditional outline font technology to specify hinting methods, such as “the line connecting p_i and p_j is vertical”, “the distance between the horizontal projections of p_i and p_j is the standard stem width”

and "p_i lies between p_j and p_k". Readers are referred to the hinting systems in Microsoft TrueType [Microsoft95b], Agfa Intellifont [Agfa91] and Adobe Type1 [Adobe90]. In our component based font parametrization system, relationships between character parts are used to determine the positions of components. Relationships between contour points are not necessary, since they will automatically be maintained if the relationships between components are kept.

The distance relationships may constrain both horizontal and vertical relative positions of components. Most of the vertical positions depend on the character height reference lines, such as, for small letters, the base line, the x-height line, the ascender line and the descender line. So, when two components are put together, we shall study how to maintain their horizontal distance relationship. We found that almost every character has two or three main components whose horizontal position relationship (*main component horizontal distance*) must be kept, primarily because their horizontal distance reflects the *character width*. For example, consider the vertical stem and the half-loop of character "b", and the left, middle and right vertical stem of character "m" (Fig. 3.4). Other horizontal positions can be specified as a function of the horizontal distances between the main components. If necessary, some dimension parameters are defined by *proportion parameters*. Examples are the center position of the half-loop in character "b" and "d", and the arch's top extreme points in character "m", "n" and "h".



Legend:



Interpolation c = a + p_i * (b - a),
where p_i is a proportion parameter.

Figure 3.4: The relationships describing the components horizontal positions: main component horizontal distances dx_1 and dx_2. Vertical positions are determined by the character's horizontal reference lines.

Character width: The traditional typographical definition is the horizontal distance from the left side bearing point (lsb) to the right side bearing point (rsb). The character origin, the left side bearing point and the right side bearing point are not suitable for

parametrization. Therefore the traditional character width is not defined in our font parametrization system.

Main component horizontal distance: This term generally refers to the horizontal distance between centerlines of main components. In this thesis, it is often given as parameters named dx, dx_1, dx_2. When all parameters of a character are provided, the character's bounding box width and its main component horizontal distance can be derived.

Proportion parameter: A proportion parameter is a parameter specifying proportional percentage values. It is often a local parameter used to specify one parameter related to another. It is quite useful for interpolating between positions in order to modify standard dimension parameters. We write proportion parameter as p_i or ppp_i. Parameter names are always in *italic* style.

Each main component horizontal distance is represented in proportion to one of the standard character width parameters. Traditionally, typographers describe the width feature of a typeface by referencing the width of lower-case letter "o". We accordingly define the bounding box width of letter "o" as the standard "*round letter width*", which is supposed to be referenced by all the letters. To ensure the coherence of character width features, we define three kinds of main component width: the *curve-to-curve width*, the *stem-to-curve width* and the *stem-to-stem width* (Fig. 3.5). The curve-to-curve width can be applied to all round letters such as "o", "e" and "c". The stem-to-curve width can be applied to characters which have a main vertical stem and a main vertical curve, such as "b", "d", "p" and "q". And the stem-to-stem width can be applied to characters with two or three main vertical stems, such as "h", "m", "n" and "u". Other characters should specify their component distance as a proportion of the round letter width. The curve-to-curve width can be directly derived from the standard round letter width by subtracting from it the standard vertical curved stroke width.

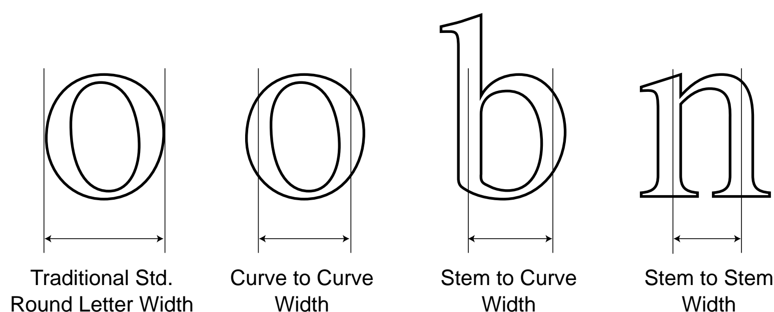


Figure 3.5: Standard width parameters for the reference of main component distances.

Another important component-to-component relationship are the connection types (see examples in Fig. 3.6). When a serif is connected to a stem, its position and dimension shall all depend on the stem so as to ensure that the serif never flies apart. When a sweep is used to connect a half-loop to a vertical stem, its span will be determined by both the half-loop and the vertical stem so that it really touches both of them. Such connection relationships represent the dependency of one component on another. For example, a serif depends on a stem.

Varying the design of junctions and terminals may require different component shapes, but generally their respective dependency remains. In Fig. 3.6, for example, no matter if there is a spur or not, one end (departure end) of connecting sweep c_4 will always depends on the position of the connected stem c_0. The same applies to the other connecting sweep c_5.

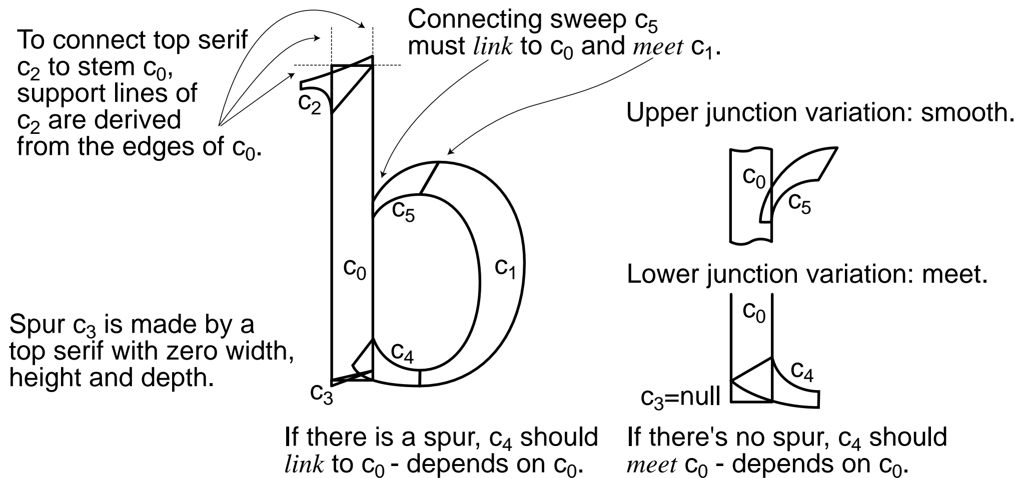


Figure 3.6: The relationship describing the connection dependency: serifs depend on the stem and connecting sweeps depend on their two neighbouring components. This figure also shows that local variations of the junction don't affect the component's connection dependency.

The purpose of studying the relationships between component is to clarify the dependency of positions between components and to ensure that no redundant or disturbing parameter is introduced to control the position of components. For example, the departure points P_d, Q_d and the arrival points of the connecting sweep c_5 in the example of character "b" (Fig. 3.6) should be determined by the half-loop c_1 and the stem c_0. Otherwise, if we would introduce direct parameters to control the sweep's positions, these parameters would have to be hand-tuned each time the distance between the stem and the half-loop is changed.

3.2.1.2 Component dependency graph

We can represent the distance and connection relationship by an acyclic directed graph, the *component dependency graph*. A component dependency graph has generally one or, sometimes, two root nodes which stand for the horizontal position of the main components of a character. The first level child nodes are main components which depend on the distance at the root node(s). The second level child nodes are terminals (serif, dots etc.) and connecting sweeps. More levels are possible but not common. The dependency relationships are represented by arrow-headed lines. An example can be found in Fig. 3.7.

Generally, varying the design of junctions and terminals does not affect their component dependency graph. The dependency relationship between components often remains the same. But the implementation of the dependency may be different according to the different styles of junctions or terminals.

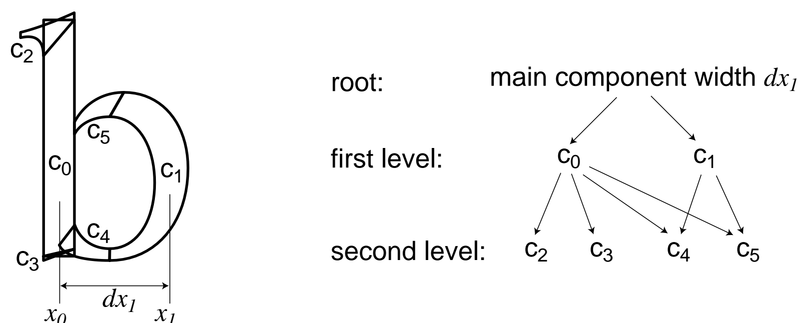


Figure 3.7: The relationship between components can be represented by a component dependency graph.

The component dependency graph also represents the steps for assembling components into a character. The distances represented by the root nodes should be provided as direct parameters at the beginning. Components located at the first level of the graph are synthesized first. Then, components on the second level can be synthesized and linked to the first level components.

3.2.1.3 Methods to specify dependency

We generally place one of the main components which will be synthesized first at a random horizontal position and name it the x_0 position (see Fig. 3.4 and Fig. 3.7 for examples). We introduce direct parameters named dx_1, dx_2, ... for the distance between the first synthesized main component and other main component(s) of the first level in the component dependency graph. Thus, their positions are $x_0 + dx_1$, $x_0 + dx_2$, ... respectively.

The connection relationship is a little more complex to specify. The position parameters of components at the second level of the component dependency graph depend on the first level components. The position parameters of the second level components are indirect parameters derived from the first level components. Support contour points of the synthesized components can be used to derive indirect parameters. For example, if c_0 is a stem, $c_0.A$ refers to the support point A of the synthesized stem. Here are some notations:

Notation 3.1: $Pt\ (x, y)$ is a point with coordinate (x, y).

Notation 3.2: $Ln\ (p_1, p_2)$ is a line connecting point p_1 and p_2.

Notation 3.3: If p is a point, $p.x$ is the x element of p, $p.y$ is the y element of p.

Notation 3.4: If *c* is a component, *c.p* refers to a support point of *c*.

Notation 3.5: Given two points or numbers *a* and *b*, and given interpolating factor *f* varying between 0 and 1, the interpolation can be written as *iplt (a, b, f)*, which equals "*a + f * (b - a)*" (see also Fig. 3.4.).

Notation 3.6: The form *{a_1, a_2, a_3, ...}* stands for the argument list of a component.

With these notations, we specify, for example, the dependency relationships "top serif c_2 depends on stem c_0" and "sweep c_5 depends on stem c_0 and half loop c_1" with the mathematical expressions given in Fig. 3.8. For the connecting sweep c_5, the example gives the argument list for an "*angled*" connection. A "*smooth*" junction (see also Fig. 3.6) requires slightly different expressions for sweep c_5's arrival points P_a and Q_a so as to make the sweep end appear horizontal.

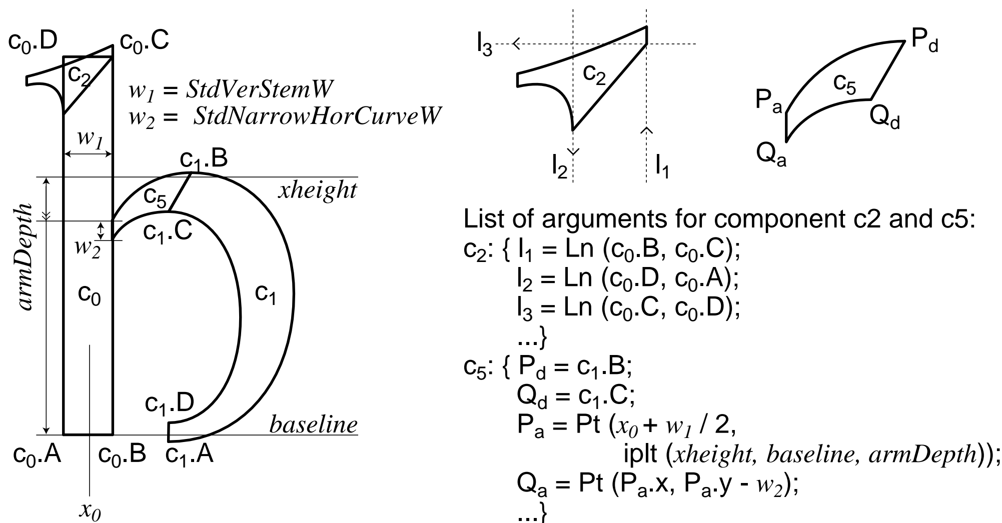


Figure 3.8: An example of mathematical expressions specifying dependency. The expressions exist in the argument list of component c_2 and c_5. This figure doesn't show c_3 and c_4.

3.2.2 Component shape tuning

In the previous section (section 3.2.1), we have shown how to place a component at a proper position and how to maintain the relationships between components. In this section, we present means of controlling the precise shape of a component.

The shape of a component is synthesized with its parameters, or arguments of the component synthesis functions (we also use the term "component" to refer to the component synthesis function). The shape of a component can be tuned by controlling the parameters which control the component's dimension, orientation and curvature.

3.2.2.1 Dimension

The first aspect of a component shape is its dimension: the width of a stem, the width of a loop or half-loop, the height and width of a serif, the radius of a round dot, the width of the ends of a sweep, etc. Most of the dimensional characteristics of character parts can be standardized for all characters in a font. For example, the width of a vertical stem remains the same for all lower-case characters or, respectively, for all capital characters in typefaces such as Times or Helvetica. Arguments of main components are specified as global font parameters (section 3.3).

If the standard dimension parameter needs to be enlarged or reduced for a specific part in a given character, a factor can be introduced as a local parameter to modify (scale) the standard dimension parameter. The scaling factor keeps a relationship between a special component dimension and the standard one, because most special dimensions do have a relationship with standard parameters. This relationship can be preserved when, for example, the weight of the font is varied.

In detail, let us summarize the dimensional arguments of the previously defined components (see section 2.2 for the definition of component parameters).

- *Stem*: Stem width is controlled by parameter w. Stem length is limited by the departure point P_0 and the arrival point P_1 of its center line.
- *Loop* and *half-loop*: Their overall size is controlled by p_1, q_1, p_2, and q_2, which represent the half width and half height of the external and internal loop bounding boxes. If the internal loop center and external loop center are common, the difference between p_1 and p_2 is the stroke width of the vertical part of the loop, and accordingly the difference between q_1 and q_2 is the stroke width of the horizontal part of the loop. The dimensional parameter p_1 and q_1 are often controlled by the character height reference lines since the size of the loop or half-loop always occupies the space between the baseline to the x-height (for lower-case letters) or the caps-height (for capital letters) lines. Therefore, parameters p_2 and q_2 can be derived from p_1 and q_1 by subtracting from them the standard width of vertical and horizontal curved strokes respectively.
- *Sweep*: The width of a sweep is controlled by the width of both the departure line segment and the arrival line segment given by the departure points P_d and Q_d and the arrival points P_a and Q_a respectively. If a sweep connects to some other component, its departure and/or arrival points will depend on other points. The open end of a sweep, such as the tail of character “e” and “t”, are controlled by introducing a local parameter controlling the tail’s width.
- *Serif* and *top-serif*: Their dimension width, height and depth are controlled by sw_1, sw_2, sh_1, sh_2, sd_1 and sd_2 respectively (see Fig. 2.26). These parameters are required to be uniform in a font, and hence are assigned directly corresponding global font parameters. The length of the serif face is also affected by the support lines l_1 and l_2, which represent the two sides of the connecting stem.

- *Dot*: Width and height are controlled by parameter a and b (Fig. 2.32). Isolated dots have a standard size given by global font parameters. However, the dot as a terminal element may have a varied size, which requires local dimension modifying parameters (Fig. 2.33).
- *Path*: Dimension of arbitrary paths should also be controlled as much as possible by standardized global font parameters. We try to find dimensional relationship between them so that the points which construct a path are controlled by local relative parameters and the number of local parameters is as small as possible.

3.2.2.2 Orientation

Orientation of components includes the slant angle of loops, half-loops, dots and diagonal stems, the slope angle of the open end of sweeps, and the orientation of serifs, top-serifs and half-loops. Many of the angles are controlled by some method other than by giving the angle in degrees. We summarize the details below.

- *Stem*: The orientation of a diagonal stem is determined by the stem's departure point P_1 and arrival point P_2. The positions of these two points are often controlled by the character width, such as "x", "v", and "z", not by angle (Fig. 3.9d).
- *Loop* and *half-loop*: The slant angle, or stress, of the external and internal loop are controlled by parameter η_1 and η_2 respectively, which are the displacement of the extreme points (Fig. 3.9a). Half-loop quadrants are given as directions by parameter *dir*, whose value may be left, right, top or bottom (for example, the orientation of the half-loop in Fig. 3.9b is "left").
- *Sweep*: The open end of a sweep has a slope, for example the end of the tail of character "e" (Fig. 3.9b), "a" and "t". The slope orientation is an important typeface feature which needs to be controlled if the weight, stress or contrast varies. We often introduce a local parameter specifying the degree of the slope angle. The end of a sweep is given by two points P_a and Q_a which can be derived from the angle and the tail's width. Trigonometric functions sin(x) and cos(x) are needed.
- *Serif* and *top-serif*: Orientations are specified by parameter *dir*, which has values left, right, top or bottom. The topserif has a slanted serif slab (Fig. 3.9c). The angle of the serif slab is specified by the component's parameter θ. The angle of a topserif can be given as a global font parameter since it is often uniform throughout a whole font. Top-serifs can be used to synthesize beaks in character "z" (Fig. 3.9d), "F", etc. In these cases, the slab angle defines the beak's angle.
- *dot*: If the dot is slanted, the angle θ can be explicitly specified (for example the terminal dot in Fig. 3.9c).

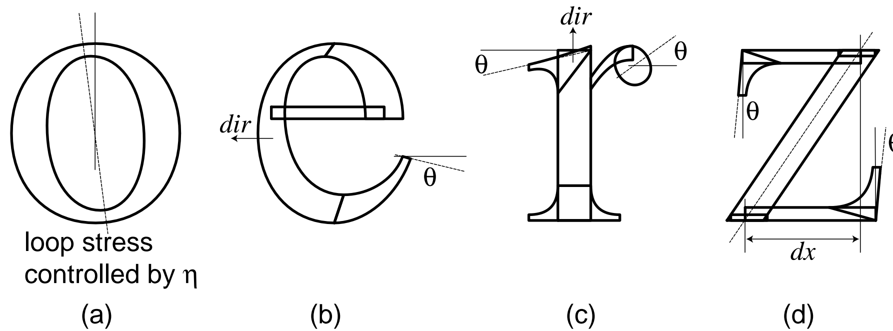


Figure 3.9: Various kinds of orientation of components.

3.2.2.3 Curvature

The curvature features of a typeface are reflected by the two sides of a sweep, the internal and external contour of a loop or half-loop, the outline of a dot, and the support (bracket) of a serif or topserif. We summarize the curvature control elements as follows.

- *Sweep*: The midline of a sweep is a β-Bézier curve controlled by control point A and the center points at both ends of the sweep. Since both the departure end and the arrival end of a sweep are given by points P_d, Q_d, P_a and Q_a, the curvature of the sweep is mostly determined by the position of control point A. Generally, point A has two degrees of freedom. But when a sweep is used to connect two part of a character, we need to constrain point A to a single degree of freedom. For example, the curvature of the upper connecting sweep (c_5 in Fig. 3.8) is controlled by a local proportion parameter which specifies the horizontal relative displacement of the β-Bézier control point A. Control point A is constrained to move horizontally (Fig. 3.10). Typographically, control point A controls the junction angle between the loop or arch to the vertical stem in characters such as “b” and “n”.

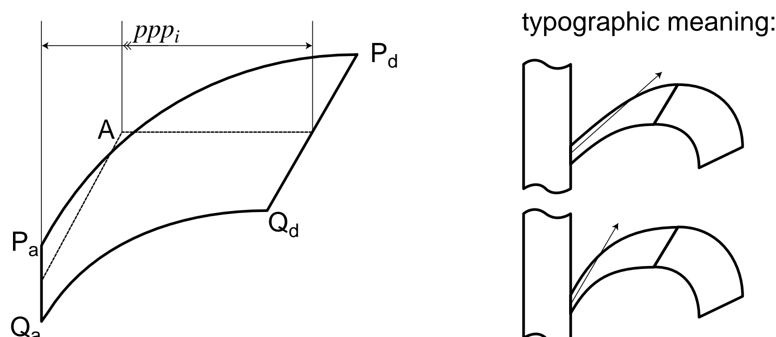


Figure 3.10: Controlling the curvature of a connecting sweep. The local parameter ppp_i controls the horizontal relative position of the control point A, hence, typographically, it controls the junction angle.

- *Loop* and *half-loop*: The squareness of the external and internal contours of a loop are controlled by β_1 and β_2 respectively. Some high contrast typeface, such as Bodoni, require the internal contour to be more square, which requires a larger β_2 parameter. Examples can be found in Fig. 2.20 in chapter 2.
- *Dot*: Squareness is controlled by a font level global parameter β_{dot}.
- *Serif* and *top-serif*: The squareness of the serif support, or bracket, is controlled by a font level global parameter $\beta_{serifsupport}$.

3.2.3 Boundary correction

Some characters cannot be synthesized by simply assembling shape primitives (components). Components for some junctions or terminals need to be trimmed or patched in order to modify their extension or smooth their corners out. These modifications are realized by *boundary correcting paths*. Some of the correction cases are quite common and will be specified by predefined *boundary correcting operations*.

Boundary correcting path: The boundary correcting path is also a sort of shape primitive which will be used to add a patch to or substitute a part of the synthesized character shapes. For example, a boundary correcting path can fix the corner formed by the intersection of two stem or sweep components and therefore smooth out the corner.

Boundary correcting operation: The operation which adds a boundary correcting path to an existing component(s). Boundary correcting operations are defined as functions which takes one or two components and correcting parameters as input and return the corrected contour.

The two most important boundary correcting operations are *cut* and *smooth*.

3.2.3.1 Trim extension of component outlines

Sometimes a component needs to be trimmed to limit its extension. This trimming work is much like what a pair of scissors does - it cuts one piece of paper into two pieces along a line, and keeps only one piece. We define this function as a boundary correcting operation named *cut*, which divides a contour into two parts along a given *cutting line* and keeps one of the resulting contours.

Since the contour of a component has a direction, the cutting line should also have a direction. The *cut* operation uses the direction of the cutting line to select the resulting contour which will be kept. The resulting contour contains part of the original contour and a cutting edge (Fig. 3.11). Orientation of the original contour and of the cutting edge should remain coherent so that the resulting contour is closed and has the same orientation as the original one. The straight line is the most commonly used cutting line in our font parametrization system. In concrete cases, a cutting line is derived from an existing component.

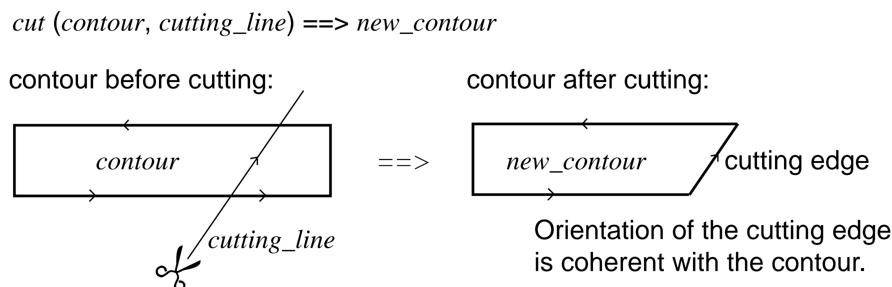


Figure 3.11: Definition of the *cut* operation.

The cut operation is simple and useful to synthesize character shapes, especially in realizing junctions and special optical corrections. Here are some examples:

1) a stem and a sweep "meet" each other

The connection symbol in the structure graph for *meet* is ◄——►. When a sweep "meets" a stem smoothly, the stem should be trimmed at the meeting position to be sure it does not extend out of the sweep (Fig. 3.12).

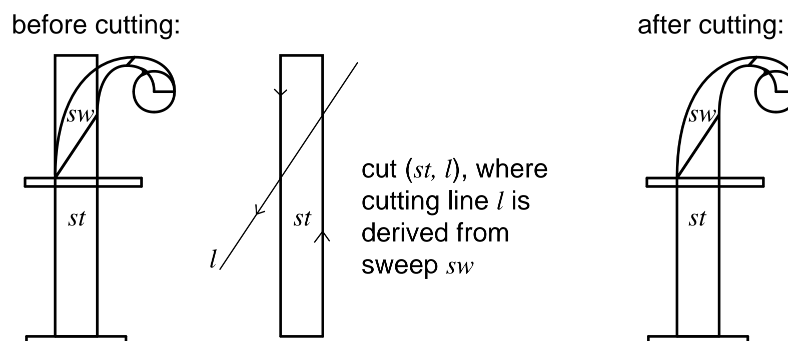


Figure 3.12: Example of the use of the *cut* operation for realizing the *meet* connection.

2) two stems "join" into a tip

The connection symbol in the structure graph for *join* is —v— , which is derived from the typical join case in character "v". From this connection, the ends of both stems need to be trimmed so that none of them will extend out of the boundary of the tip (see the example in Fig. 3.13).

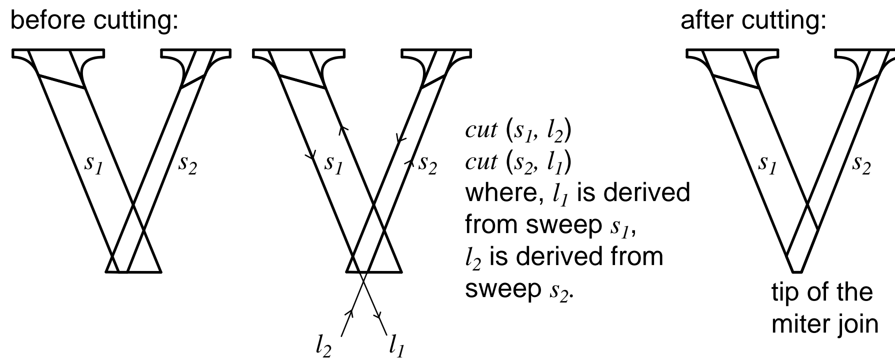
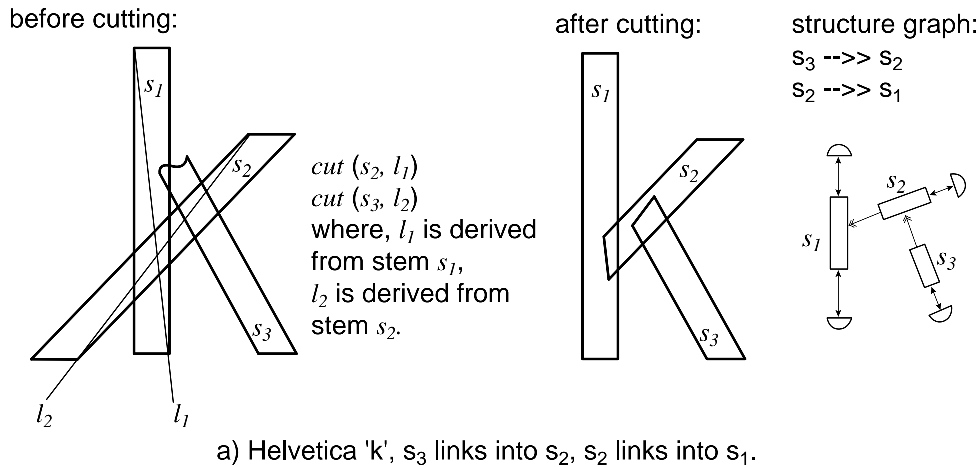
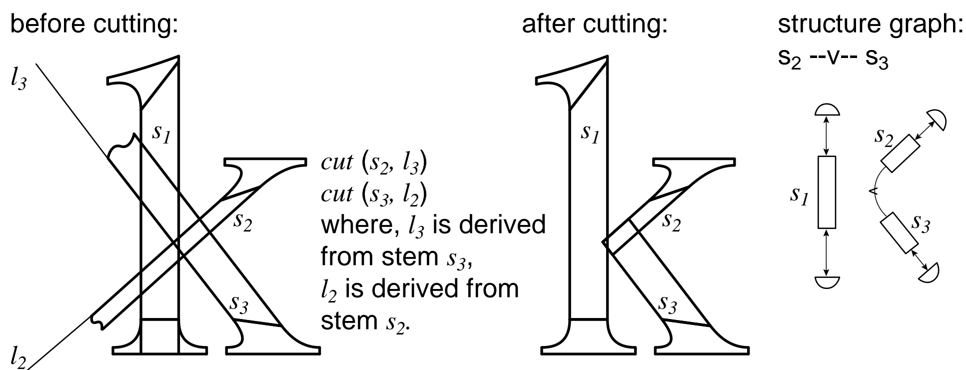


Figure 3.13: Example of the use of the *cut* operation for realizing the *join* connection.



a) Helvetica 'k', s_3 links into s_2, s_2 links into s_1.

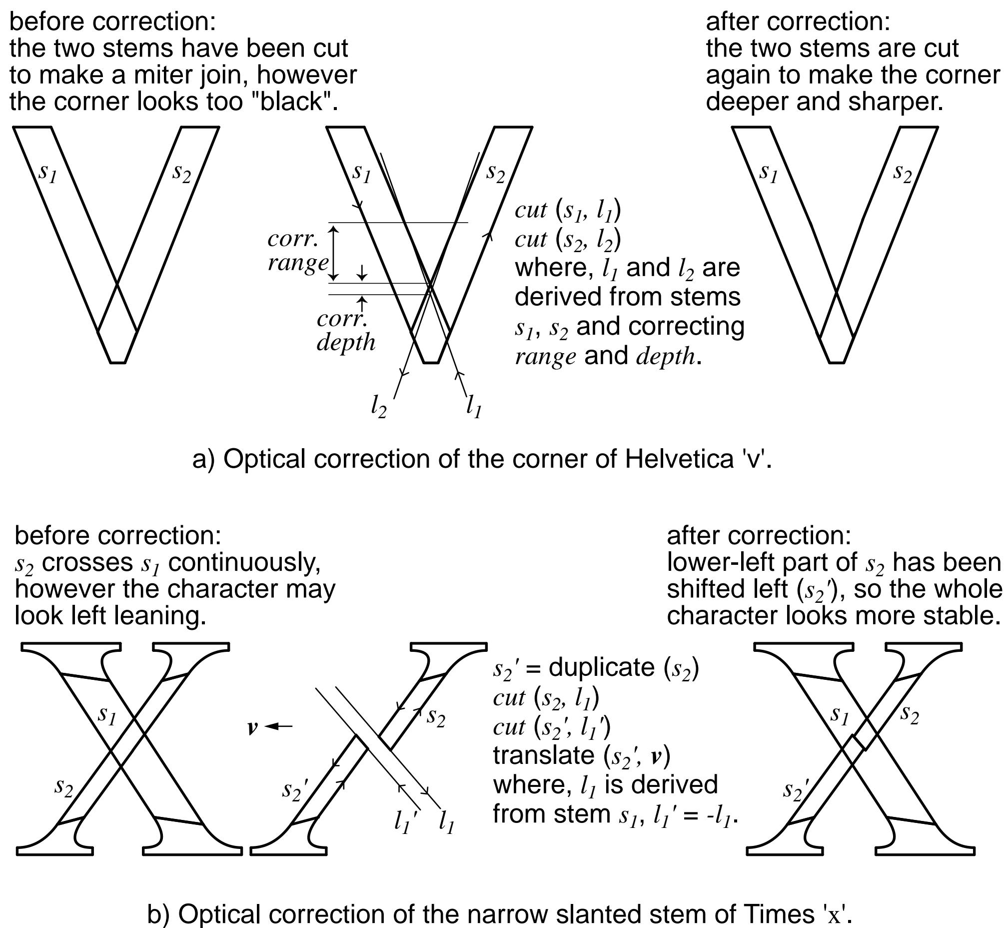


b) Times 'k': s_2 and s_3 from a tip. Whether the tip overlaps s_1 or not depends on their distance, which is a local parameter of this character.

Figure 3.14: Use of the *cut* operation for synthesizing character “k”.

3) one component "links" into another

The connection symbol in the structure graph for *link* is $\longrightarrow\!\!\gg$. In a *link* connection, the end of one component is buried in the body of the other one. To be sure the end of the linking component never exceeds the body of the linked one, the *cut* operation can be applied. The examples in Fig. 3.14 demonstrate the different realizations of the two kinds of junctions for Helvetica character "k" (Fig. 3.14a) and Times character "k" (Fig. 3.14b), and their corresponding structure graphs (an interesting example of design variation).



a) Optical correction of the corner of Helvetica 'v'.

b) Optical correction of the narrow slanted stem of Times 'x'.

Figure 3.15: Examples of the use of the *cut* operation for realizing special optical character shape corrections.

4) special optical corrections

Optical correction of character shapes is needed so as to ensure that they look as expected. For example the height of a round letter should be a little higher than that of a square letter so as to look the same. Generally, optical correction is made by using the character height reference lines. There are still some special shapes which need to be

optically corrected and are not subject to the reference lines. For example the sharp corner of Helvetica “v” (Fig. 3.15a) should be corrected to look really sharp, and the thin diagonal stroke in the cross junction of Times “x” (Fig. 3.15b) should be corrected so that the character looks stable [Jamra93]. These special optical corrections can be realized with the help of the *cut* operation.

3.2.3.2 Smooth corner of intersected components

The bulb/pear-like terminal may need to smooth the corner formed by the sweep and the dot. In some typefaces, corners formed by two straight component also need to be smoothed. We define the *smooth boundary correction operation* to fulfill this task. The *smooth* operation generates a smoothing shape from the two given components and some smoothing parameters (Fig. 3.16). A smoothing shape is a path component which fills the corner smoothly. The size and shape of the smoothing shape are controlled by smoothing parameters. This operation first finds the corner by possibly intersecting the two contours which form the corner. Then the *smooth* operation constructs a smoothing shape from the parameters specifying the range of the corner to be smoothed and the tangent length of the smoothing curve.

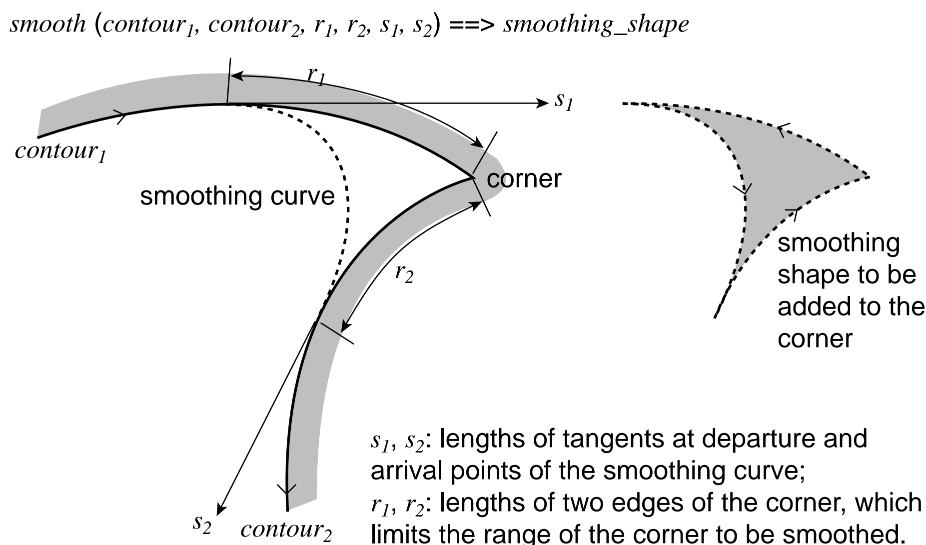


Figure 3.16: Definition of the *smooth* operation.

In Fig. 3.16, smoothing parameter r_1 and r_2 limit the range of the corner to be smoothed by restricting the length of the two edges of the corner. Parameters s_1 and s_2 control the curvature of the smoothing curve by the length of tangents at both departure and arrival point. Tangent directions are computed by the *smooth* operation in order to ensure smooth connection between the smoothing curve and the two edges of the corner. The final constructed smoothing shape should cover the smoothed area of the corner without leak. We can make sure the smoothing shape touches or slightly overlaps the

smoothed corner. Practically, r_1, r_2, s_1 and s_2 can be given as local parameters relating to the character's width and height.

The *smooth* operation works quite well in smoothing out the corner of terminals. Examples in Fig. 3.17 show the smoothing path component added to the corners of two components in case of a sweep to dot (character “j”) and a sweep to stem (character “s”) smoothing correction.

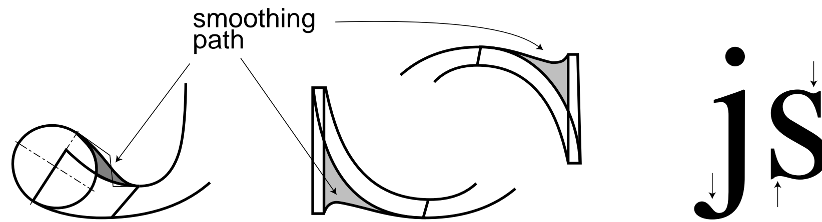


Figure 3.17: Examples of smoothing paths which are added by the *smooth* operation.

3.2.3.3 Hand tuned boundary correction

Sometimes, the predefined operations, *cut* and *smooth*, are not able to generate the correction path for some special terminals. In this case, we can construct a correcting path and add it at the proper position. This hand tuned path component can be used together with the predefined ones.

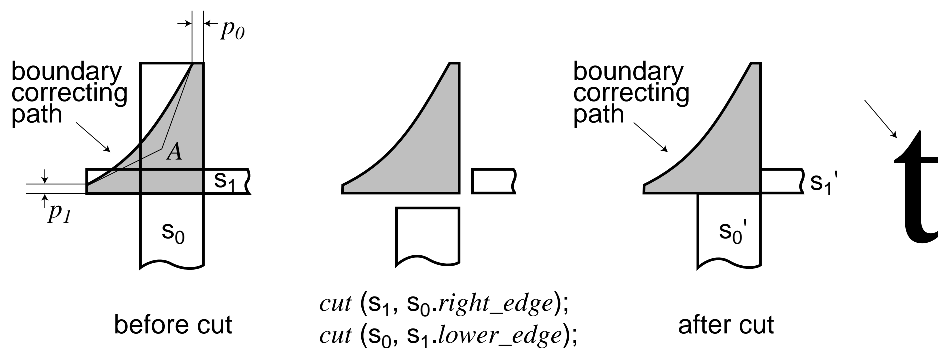


Figure 3.18: The special terminal of Times “t” requires a hand-made correcting path as well as two *cut* operations.

For example, the upper terminal of the Times character “t” is a coved triangle-like shape. As demonstrated in Fig. 3.18, we can compose a path component to describe this terminal with a few local parameters specifying the dimension relationship to the existing stems s_0 and s_1. The local parameter p_0 specifies the width of the vertical tip in proportion to the width of stem s_0, and p_1 specifies the width of the horizontal tip in proportion to the width of stem s_1. The control point of the β-Bézier curve (point A in the

figure) is also given in relation to the position of the two stems. To add this new path component, we apply the *cut* operation to both of the stems s_0 and s_1 to make sure neither of them will exceed the boundary of the correcting path.

3.2.4 The complete parametrizable character synthesis method

As described above, we are able to specify the position relationship between components, to specify how to tune component shapes by parameters, and to specify how to trim or correct the boundary of components. Now, we can describe the complete *parametrizable character synthesis method*.

A complete parametrizable character synthesis method consists of two parts. A *parameter declaration* part and a *character synthesis method specification* part. The parameter declaration part contains place holders for all parameters used for this character, including *global parameters* and *local parameters* (section 3.3.2). The character synthesis method specifies the type and parameter (argument) list of each component, as well as the boundary correcting operations. Relationship between parameters and components are maintained through mathematical expressions. A pseudo C/C++ parameter passing format is used to explain the character synthesis method.

Components are often named subsequently, for example, like c_i or s_i, i = 0, 1, 2.... Parameters of each component, which are defined in section 2.2, are written in the parameter list form as below:

- *stem* $\{P_0, P_1, w, terminator_0, terminator_1\}$, which also computes and stores support points *A*, *B*, *C* and *D* (see Fig. 2.10);
- *loop* $\{O_1, O_2, p_1, p_2, q_1, q_2, \eta_1, \eta_2\}$;
- *halfloop* $\{O_1, O_2, p_1, p_2, q_1, q_2, \eta_1, \eta_2, dir\}$, which also computes and stores support points *A*, *B*, *C* and *D* (see Fig. 2.21);
- *sweep* $\{P_d, Q_d, P_a, Q_a, B_1, B_2\}$ or *sweep* $\{P_d, Q_d, P_a, Q_a, A\}$;
- *serif* $\{sw_1, sw_2, sh_1, sh_2, sd_1, sd_2, dir, l_1, l_2, l_3\}$, which also computes and stores support points for serif connecting positions *A* and *H* (see Fig. 2.26);
- *topserif* $\{sw, sh, sd, \theta, overshoot, dir, l_1, l_2, l_3\}$, which also computes and stores support points for serif connecting positions *O* and *F* (Fig. 2.31);
- *dot* $\{O, a, b, \theta\}$;
- *path* $\{segment_i, i = 0, 1, ...\}$.

As defined in notation 3.4 in this chapter, once a parameter *p* of component *s* is evaluated, it can be referred to as *s.p* later. The result of component stem and half-loop contain support points which can also be referred in the same way.

We explain the details of the parametrizable character synthesis method by providing as example the character “b”. To simplify, we only provide the method for synthesizing a character with one given junction and terminal type. As indicated in Fig. 3.19, character “b” has a top-serif, a null spur, its upper junction type is “angled link” and its lower junction type is “meet”. Local parameters and some global parameters are indicated. The refined structure graph gives the component type and the connection between components.

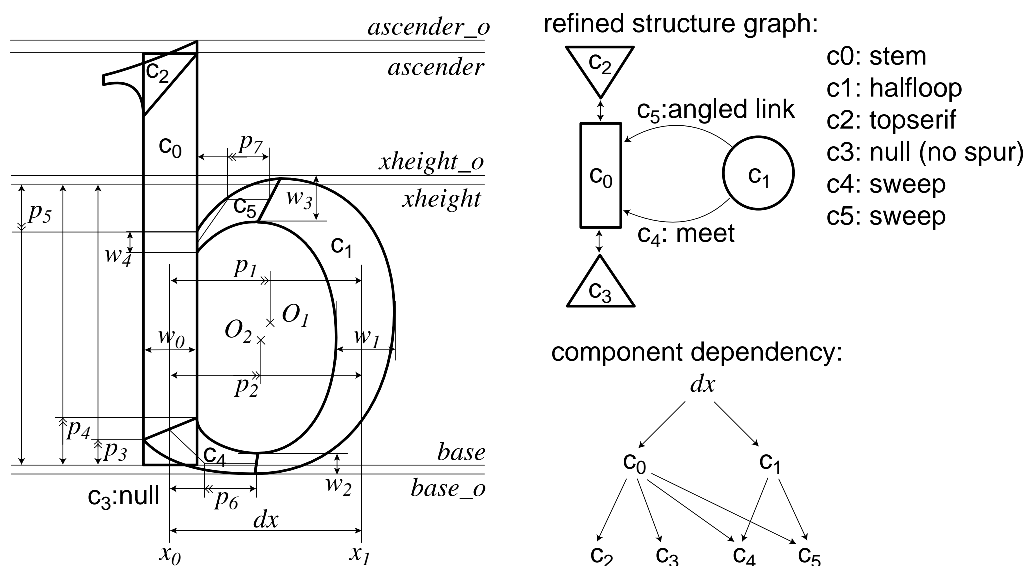


Figure 3.19: Parameters of character “b”, its refined structure and component dependency.

Referring to Fig. 3.19, we can write the complete *parametrizable character synthesis method* as follows. Font parameters, both global and local, are printed in *italic*.

Parameter declaration (place holders for parameters):

- dx : the distance between the left stem and the right half-loop;
- w_0 : the standard vertical stem width;
- w_1 : the standard vertical curved stroke width;
- w_2 : the standard narrow horizontal curved stroke width;
- w_3 : the standard horizontal curved stroke width;
- w_4 : the width of the upper junction;
- p_1 : relative h-position of the half-loop external center;
- p_2 : relative h-position of the half-loop internal center;
- p_3 : relative height of the lower junction (c_4), external;
- p_4 : relative height of the lower junction (c_4), internal;
- p_5 : relative depth of the upper junction (c_5);
- p_6 : relative control point position of connecting sweep c_4 (see also Fig. 3.10 for the design of sweep control point);
- p_7 : relative control point position of connecting sweep c_5;

eta_1 : external loop extrema correction, standard;
eta_2 : internal loop extrema correction, standard.

Character synthesis method specification:

// auxiliary names: x_0, x_1
x_0 = random value for the position of the left stem;
$x_1 = x_0 + dx$;
// components: c_0, c_1, c_2, c_3, c_4, c_5
c_0 = stem {
P_0 = Pt (x_0, *base*); // departure point
P_1 = Pt (x_0, *ascender*); // arrival point
w = w_0; // stem width
$terminator_0$ = the base line; // stem terminator at P_0
$terminator_1$ = the ascender line; // stem terminator at P_1
};
c_1 = halfloop {
O_1 = Pt ($x_0 + p_1 * dx$, (*xheight_o* + *base_o*) / 2); // center, external
$p_1 = x_1 + w_1 / 2 - c_1.O_1.x$; // half width
q_1 = (*xheight_o* - *base_o*) / 2; // half height
O_2 = Pt ($x_0 + p_2 * dx$,
((*xheight_o* - w_3) + (*base_o* + w_2)) / 2); // center, internal
$p_2 = x_1 - w_1 / 2 - c_1.O_2.x$; // half width
q_2 = ((*xheight_o* - w_3) - (*base_o* + w_2)) / 2; // half height
$\eta_1 = eta_1$; // external loop extrema correction
$\eta_2 = eta_2$; // internal loop extrema correction
};
c_2 = topserif {
sw = standard serif width; // topserif width, global
sh = standard serif height; // topserif height, global
sd = standard serif depth; // topserif depth, global
θ = standard topserif angle; // topserif angle, global
overshoot = standard topserif overshoot; // overshoot, global
dir = top; // orientation, constant
l_1 = Ln {c_0.B, c_0.C}; // support line, derived from c_0
l_2 = Ln {c_0.D, c_0.A}; // support line, derived from c_0
l_3 = Ln {c_0.C, c_0.D}; // support line, derived from c_0
};
c_3 = null; // no spur, a spur is made by a topserif
c_4 = sweep {
P_d = Pt ($x_0 - w_0 / 2$, iplt (*base*, *xheight*, p_3)); // departure right
Q_d = Pt ($x_0 + w_0 / 2$, iplt (*base*, *xheight*, p_4)); // departure left
$P_a = c_1$.A; // arrival right
$Q_a = c_1$.D; // arrival left
A = Pt (iplt ((c_1.A.x + c_1.D.x) / 2, x_0, p_6),
(c_1.A.y + c_1.D.y) / 2); // control point
};

```
c5 = sweep {
        Pd = c1.B;                                              // departure right
        Qd = c1.C;                                              // departure left
        Pa = Pt {x0 + w0 / 2, iplt (xheight, base, p5));        // arrival right
        Qa = Pt (x0 + w0 / 2, c5.Pa.y - w4);                    // arrival left
        A = Pt (iplt ((c1.B.x + c1.C.x) / 2, x0 + w0 / 2, p7),
                (c1.B.y + c1.C.y) / 2);                         // control point
};
// boundary correcting operations:
// the stem component c0 needs to be trimmed to connect serif c2 and sweep c4
c0 = cut (c0, Ln (c2.A, c2.Z));                      // connect topserif c2
c0 = cut (c0, Ln (c4.Pd, c4.Qd));                    // stem c0 and sweep c4 meet
// end of the parametrizable character synthesis method for character "b"
```

3.3 Parameter files

The parametrizable character synthesis method defines how to generate the shape of a character. To synthesize an instance of the character, one needs to provide every parameter in the parametrizable character synthesis method with its proper parameter instances, or to feed parameter files to the parametrizable character synthesizer. In this section, we will discuss parameters used in our font parametrization system and our approach to organize parameter files.

3.3.1 Coordinate system

Before describing parameter files, we should answer some questions about the coordinate system for character design.

The first question is about the unit or granularity of the character design space. Generally, there are two kinds of parameters: character-size-dependent and character-size-independent. Character-size-dependent parameters specify the dimension of character parts. The change (scaling) of a character will need the value of these parameters to be changed (scaled) proportionally. Character-size-independent parameters do not relate to the size of a character and hence remain unchanged in response to the character scaling. Such kinds of parameters are, for example, those which describe angles, proportions, serif styles, junction and terminal styles etc.

The value of a character-size-dependent parameter depends on the size of the character. Since physical size (in inch, point or centimetre) depends on physical device resolution, digital font manufacturers tend to define outline characters in a specific font design coordinate system which incorporates logical coordinate units (“FUnit” in TrueType [Microsoft95b] and “character space unit” in Adobe Type1 [Adobe90]). The font design space, which is a rectangle enclosing all characters, can be granulated into a certain amount of logical coordinate units. And the logical coordinate unit is the smallest

unit for the coordinate of character outlines. Originally, this granulation is caused by the digitizing character ourlines from a design master [Karow94, pp.97-98]. In some outline fonts, however, it is intended to restrict the precision of the coordinate in order to speed up the rasterization process. For example it is common to have 1000 font units in a PostScript Type1 font and 2048 font units in a TrueType font.

In our font parametrization system, we will not restrict our flexibility to integer coordinates. Instead, parameters can be real numbers whenever needed. Therefore, the number of font units in a font design space does not limit the precision of the synthesized character outlines. But, to compare the fonts we generate with Adobe Type 1 fonts, we tend to assume the character space to be 1000 by 1000, but with no granularity limitation (Fig. 3.20).

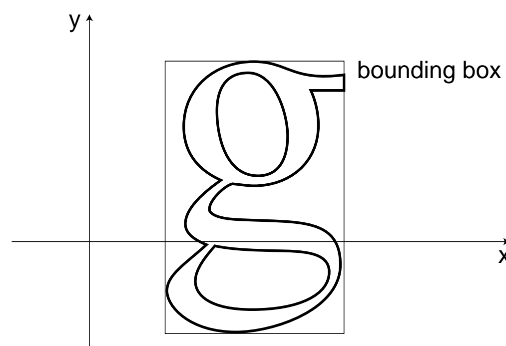


Figure 3.20: The coordinate system for our font parametrization system is similar to the one used in the traditional outline font technology except that it has no granularity limitation and that the x-origin is not defined. The bounding box is calculated on-the-fly and will be used to compute optical spacing between two characters.

The second question is about the values of character-size-dependent parameters. Theoretically, these parameters depend on the size of the character, and therefore could be represented as a proportion or percentage of the character size, which usually is represented by the height or width of some typical letters, such as the height of capital letters or the width of lower-case letter “o”. For example, in the Times fonts, the standard vertical stem width is about 20% of the width of lower-case letter “o”. However, we have observed that the proportion between size-related parameters varies from typeface to typeface and that it is not important for parametrized fonts. Therefore, most of the character-size-dependent parameters in our font parameter system have direct values specifying the dimension of character parts.

The third question is about character origin and width measurements. Since the coordinate origin is not defined in our parametrizable character synthesis system, it is often assigned an arbitrary value when the character is synthesized. The traditional width measurements, such as character width, left side bearing and right side bearing, are not defined either. Thus, when two characters are placed against each other, the space

between them is not given by any kind of traditional metric tables. We will therefore use an automatic optical spacing method to calculate character spacing on-the-fly.

3.3.2 Parameter hierarchy

Parameters are organized hierarchically and, for convenience, stored separately in three parameter files: *global parameters* in the global parameter file, *group parameters* in the group parameter file, and *local parameters* in the local parameter file.

3.3.2.1 Global parameters and local parameters

Basically, there are two hierarchical layers of parameters: the *global parameters* and the *local parameters*. Global parameters specify general characteristics of all characters in a font, such as the main stroke widths, character height alignment and serif styles. On the other hand, local parameters specify individual characteristics of each characters, such as the modification (enlargement or reduction) of the main stroke widths, the proportion parameters for relative positions, and special terminal or junction styles.

It is not easy to determine whether a parameter is global or local. Our rule is to restrict global parameters so that global parameters are really “global”. Here are some rules for global parameters:

- Global parameters have clearly defined typographical meanings, which means they are understandable by a typographer without special knowledge of our component based font synthesis system. Accordingly, global parameters have names from traditional typographic terminology.
- The purpose of global parameters is to unify font features across individual characters, and to enable coherent typeface feature modification in a font. Hence special individual character features are not global parameters.
- Global parameters are common typographical features of all text typefaces, they therefore often enable typeface identification.

Generally, if a parameter is not global, it is local. Local parameters have special meanings which are interpreted by the parameter place holders in the parametrizable character synthesis method for each individual character. A local parameter can only be seen within the parametrizable character synthesis method which declares it and defines its meaning. Local parameters do not need to have meaningful names.

We should point out that, since a parametrizable character synthesis method depends on the character’s structure graph, the definition of a local parameter in a parametrizable character synthesis method will not change between different typefaces as long as they correspond to the same refined structure graph. Thus, local parameters have relatively fixed meanings between “similar” typefaces or, quite often, throughout a family of typefaces.

3.3.2.2 Local parameter grouping

Sometimes a typographic feature is similar within a group of characters, but not all characters in a font. The local parameters controlling such kind of features can be grouped in order to achieve coherent feature modification. The parameter used to represent a group of coherent local parameters is called a *group parameter.*

Parameters which can be grouped and the characters affected by a parameter group are not fixed for different typefaces. Therefore, a mechanism allowing dynamic grouping of parameters has been introduced. This mechanism enables the parameter files to have the following two functions:

1. *name definition* - to define a name for a group parameter and to associate a value to the defined name;
2. *name reference* - to refer to the value of a predefined name and to offer support for simple algebra operations (add, subtract, multiply, divide etc.).

For example, we can define a group parameter to control the junction depth of character h, m, n and u, because they have similar arches. We can also define a group for the relative position of the extrema of the arch and the relative position of the curvature control point for this group of characters (Fig. 3.21a). Similarly, characters b, d, p and q have some coherent features which can be grouped, such as stress (slant) of the loop, loop-stem connecting position and connecting sweep curvature (Fig. 3.21b). These groups are common in many serif and sans-serif typefaces. By grouping these kinds of feature controlling parameters, we are able to generate coherent feature modifications of a group of characters (Fig. 3.22).

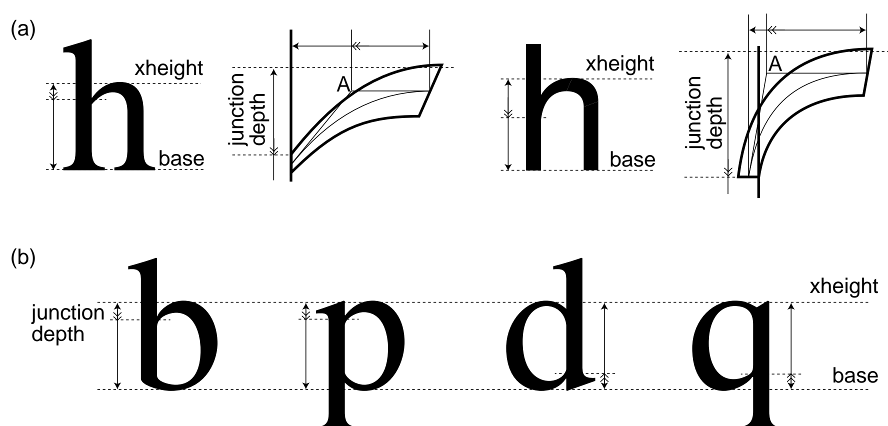


Figure 3.21: An example of features which can be controlled by group parameters: (a) the arches of characters h, m, n, and u; (b) the round parts of characters b, d, p and q.

Coherent feature modification is quite useful for typographers because their jobs are not only to design completely new typefaces but also to modify, improve or vary

existing fonts under special requirements. A new typeface may be derived from an existing typeface by coherent modifications of typographic features. For example, the last line of Fig. 3.22 imitates a typeface called “Corona”, which is derived from Times Roman by smoothing “link” junctions and thickening serif slabs.

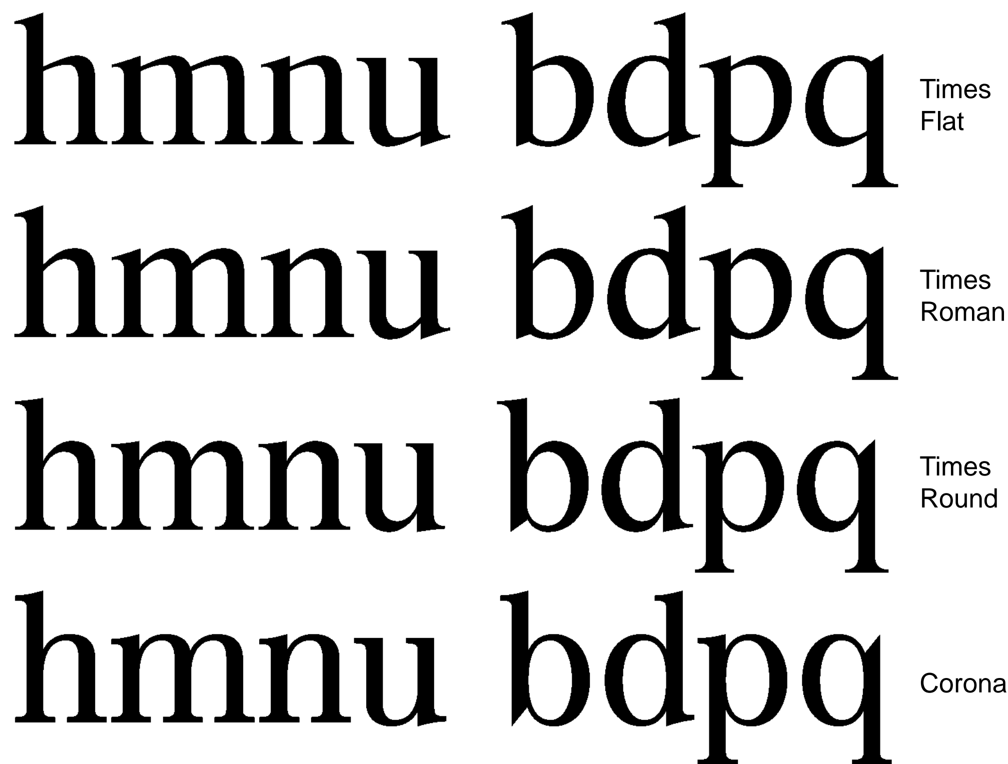


Figure 3.22: Coherent feature modification across character groups. Note that the depth and curvature of arches in characters h, m, n and u are coherent and so are the round parts of characters b, d, p and q.

The purpose of group parameters is mainly to achieve coherent local feature modification. The grouping of local parameters does not reduce the use of local parameters. The number of local parameters declared in the parametrizable character synthesis method remains the same but by grouping, they can be modified coherently. The grouping of local parameters from different characters can be considered as establishing a link between them.

3.3.2.3 The parameter hierarchy

The three kinds of parameters, global, local and group, are organized hierarchically. The top-most layer in the hierarchy are global parameters and the bottom-most layer are local parameters. The group parameters lie between global parameter and local parameters and are optional (Fig. 3.23).

global parameter	required, valid throughout a whole font, common between all text typefaces
group parameter	optional, valid for a group of characters
local parameter	required, valid for individual character, defined and used in a character's parametrizable character synthesis method

Figure 3.23: Parameter hierarchy.

The parameter hierarchy offers much flexibility for preparing parameter files. Different fonts with trivial feature modification can sometimes be achieved by modifying local parameters or by grouping some local parameters if they are intended to be coherent. Modifying a global parameter will result in the coherent modification of typographic features across a font. However, modifying a local parameter only affects the character which defines it. The group parameter provides a name definition and reference mechanism. By defining a name and referring it from several local parameters, the font synthesizer knows that these local parameters are linked (grouped) together.

3.3.3 An example of parameter files

For our font synthesis experiments, we selected about 100 important global parameters, which are present in most text typefaces. Some of the global parameters describe earmark styles, β-curve parameters and character height alignment positions. Others describe dimensions of character parts, slant serif angles and standard optical corrections. Different parameters for capital letters, lower-case letters and digits (including symbols) are present. The set of selected global parameters is important because it represents important typeface features. But not each of these global parameters are actually used in every typeface. For example, the sans-serif typeface Helvetica does not use any of the parameters related to serif features.

In the following table (Tab. 3.1), the global parameters were named according to their typographic meanings. The first letter indicates the type of the parameter: "i" stands for an index which often describes type style, and "p" for a parameter which intends to be a real number (floating or fixed-point integer). The parameters for Times, Bodoni and Helvetica are based on the same RastWare[1] or Type 1 outline fonts. So, the corresponding character design space is 1000 by 1000. Parameters which may have different values for capital letters, lower-case letters and symbols are distinguished by the second letter in their parameter names: "m" for lower-case (minuscular) letter, "c" for capital letters and "d" for digits and symbols.

1. Many of the RastWare fonts are converted directly from corresponding Type 1 or URW fonts.

Table 3.1: Examples of important global parameters in the global parameter file.

Name	Times	Helvetica	Bodoni	Description
iSerifStyle	bracket	sans	slab	general style description
iSerifSupportType	smooth	n/a	none	serif support style
iSerifEndType	butt	n/a	butt	serif end style
iSerifFaceType	flat	n/a	flat	serif face style
iTopSerifSupportType	smooth	n/a	none	topserif support style
iTopSerifEndType	butt	n/a	butt	topserif end style
iTopSerifFaceType	flat	n/a	flat	topserif face style
pBaseLine	0	0	0	base line position
pXheight	445	521	396	x-height line position
pCaps	659	726	668	capitals height position
pNumbers	659	706	668	digits and symbols h. pos.
pAscender	679	726	683	ascender line position
pDescender	-220	-220	-234	descender line position
pStdBetaXS	0.54	0.64	0.54	extremely flat curve
pStdBetaS	0.59	0.64	0.59	loop internal and external
pStdBetaM	0.67	0.64	0.62	serif support, sweep
pStdBetaL	0.85	0.64	0.75	very bent curve
pStdBetaXL	0.95	0.64	0.92	extremely bent curve
pStdLoopEta	0.13	0	0	oblique stress of a loop
pmVerStemW	86	84	100	vertical stem width
pmHorStemW	34	68	25	horizontal stem width
pmNarrowVerStemW	20	42	18	narrow vert. stem width
pmNarrowHorStemW	31	68	20	narrow hori. stem width
pmVerCurveW	93	87	120	vertical curve width
pmHorCurveW	33	77	20	horizontal curve width
pmDiagStemW	79	79	120	diagonal stem width
pmNarrowDiagStemW	37	79	25	narrow diag. stem width
pmRoundLetterW	440	474	420	standard round letter width
pmStemStemW	266	332	270	stem-to-stem width: h,m,n,u
pmStemCurveW	312	382	312	stem-to-curve width: b,d,p,q
pmVerSerifW	76	n/a	68	vertical serif width
pmVerSerifD	76	n/a	n/a	vertical serif depth
pmDiagOuterSerifW	49	n/a	46	diag. outer serif width
pmDiagOuterSerifD	49	n/a	n/a	diag. outer serif depth
pmDiagInnerSerifW	64	n/a	56	diag. inner serif width
pmDiagInnerSerifD	64	n/a	n/a	diag. inner serif depth
pmVerSerifH	15	n/a	20	vertical serif height
pmHorSerifW	105	n/a	145	horizontal serif width
pmHorSerifD	90	n/a	n/a	horizontal serif depth

Name	Times	Helvetica	Bodoni	Description
pmHorSerifH	20	n/a	20	horizontal serif height
pmTopSerifAngle	13	n/a	0	degree of slant slab angle
pmDotR	52	52	57	dot radius
pmOptCor	12	11	10	optical correction
pcXXXXXX	...	...	...	similar to pmXXXXXX, specific for capital letters
pdXXXXXX	...	...	...	similar to pmXXXXXX, for digits and symbols

For the local parameters, the average number of local parameters per character is around 15. Some local parameters can be grouped. In our experiments, we made a group for parameters controlling the arches of letters “h”, “m”, “n” and “u”, and a group for parameters controlling the round part of letters “b”, “d”, “p” and “q”.

Capital letters and symbols are synthesized by components according to similar principles as lower-case characters. Fig. 3.24 shows the components and resulting characters for a few representative upper-case Times Roman characters. Generally, stem and bar widths of capital letters are slightly larger than those of lower-case letters. For a complete description of the component-based design of each capital and lower-case letters, readers are referred to Appendix D.

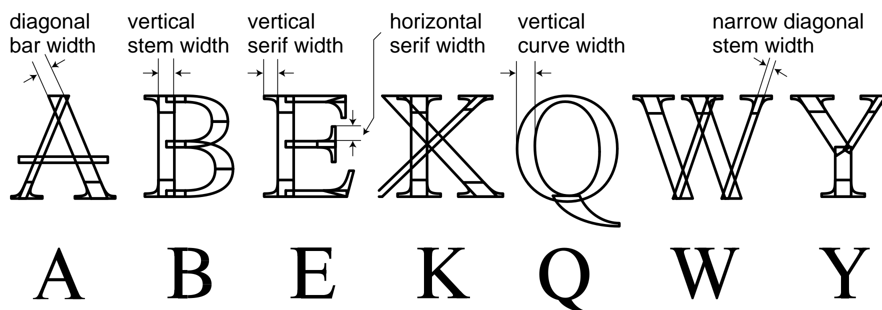


Figure 3.24: Components and parameters for capital letters.

3.4 Technical issues regarding the font synthesizer

The parametrizable font synthesizer contains a component library which includes functions to synthesize component instances, functions to do necessary geometric operations (cut/clip and intersection) and functions to do rasterization (scan conversion, grid-fitting and optical spacing). The concrete structure of the font synthesizer is implementation oriented. But no matter how we implement a font synthesizer, it must take the following technical issues into consideration:

1. Grid-fitting for rasterization purpose: The actual physical size of a character is the result of scaling of parameters in respect to the device resolution. As in traditional outline font technology, intelligent grid-fitting (or hinting) should be applied when the size of characters is small, because rounding effects may be detrimental to character quality. Traditional outline technology usually uses considerable additional information called hints to improve fitting of outline characters to the target grid. Our parametrizable character synthesis method already includes the necessary information for grid-fitting, therefore the overhead (time and storage) for grid-fitting with our font parametrization system should not be noticeable.
2. Flexible scan conversion: Scan conversion is the process of converting a geometric outline into pixels for raster devices. It can be based on component outlines provided that the filling rule[1] enables shape overlapping. Alternately, one can merge component outlines to a traditional outline font.
3. Automatic metric table calculation: The metric table contains important information for optical spacing of character pairs such as left-side bearing, right-side bearing, character width and character pair kerning. In a font parametrization system, none of these typesetting information should be stored permanently. Instead, they will automatically be calculated on-the-fly.
4. Specific geometric operation algorithms: Since general purpose geometric operation algorithms, for example intersection of two arbitrary shapes, are very complex and hence not efficient, we intend to design specific algorithms for shape merging and trimming. Fortunately, the shape of a synthesized component is more or less known and therefore simplified algorithms can be applied, reducing their complexity and accelerating their speed.
5. Font organization: This is an almost stand-alone topic beyond the technology of font parametrization. But, some issues still need to be considered by the font synthesizer, such as composing hybrid characters and providing some font information necessary for font management including font name, font creation and modification date, copyright, etc.

In chapter 4, we will present some important algorithms and solutions along with a detailed description of our experimental font parametrization system.

1. Readers are referred to the PostScript language for an explanation of filling rules. The “non-zero winding” rule ensures that pixels in the overlapped area are activated.

3.5 Summary

This chapter presented concepts and methods of our component-based font parametrization system. The framework of the system consists of two main parts: parametrizable character synthesis methods and parameter files.

The parametrizable character synthesis methods enable synthesizing characters by parametrizable components. Characters are built by using predefined shape primitives - components, and by specifying the relationships (dependency) between components. We introduced variations of terminals and junctions in order to cover a large character design space.

The parameter files provide instances of parameters for the parametrizable character synthesis methods. Parameters have been organized hierarchically: global parameters control uniform features over the whole font and local parameters control only features of individual characters. To achieve coherent feature modification of similar typographical parts across several characters, we introduced the concept of local parameter grouping.

To explain in detail the parametrizable character synthesis system, a full example for one character and its parameters have been given in this chapter. Statistics about the number of required parameters are provided in section 4.6.

CHAPTER 4

Implementation of the parametrizable font synthesizing system

In the previous chapter, we have presented the algorithms and techniques used in our parametrizable character synthesis system and the specification of the hierarchy of global, group and local parameters. In this chapter, focusing on the font synthesizer (see the system framework in Fig. 3.1), we describe our experimental implementation of the parametrizable font synthesizing system. The font synthesizer contains a set of functions which deal with component synthesizing (building), parameter file interpretation, specific geometric operations such as shape trimming and merging, intelligent scaling (hinting), and automatic optical spacing.

We started our study on basic font parametrization problems, such as modelling of curves and shape primitives for character parts. We used Mathematica, a comprehensive tool suitable for building mathematics and computation models. After the first experimenting and prototyping phase, we decided to integrate the results and to design a prototype system. This prototype system allows to carry out more typographical experiments and to evaluate the results.

The font synthesizer and the individual parametrizable character synthesis method are written in C++. The parameter files, however, are written as readable text. This text is parsed and interpreted by the synthesizer. The readable and editable parameter files facilitate the creation of derived fonts by appropriately modifying the parameters.

The problems of hinting and automatic optical spacing [Hersch95] are themselves two stand-alone research topics. They are outside the scope of the present research. We have however tried to adapt results from previous work into our font parametrization system. We also have slightly improved the technique used for automatic spacing.

4.1 Classes

There are some similarities in the data and in the behaviour of each character's parametrizable character synthesis method. This suggests designing a base object which contains all the common data elements and functions used by each parametrizable character synthesis method. Each parametrizable character synthesis method corresponds to an individual derived object of this base class. This model also helps implementing the parameter hierarchy (Fig. 4.1).

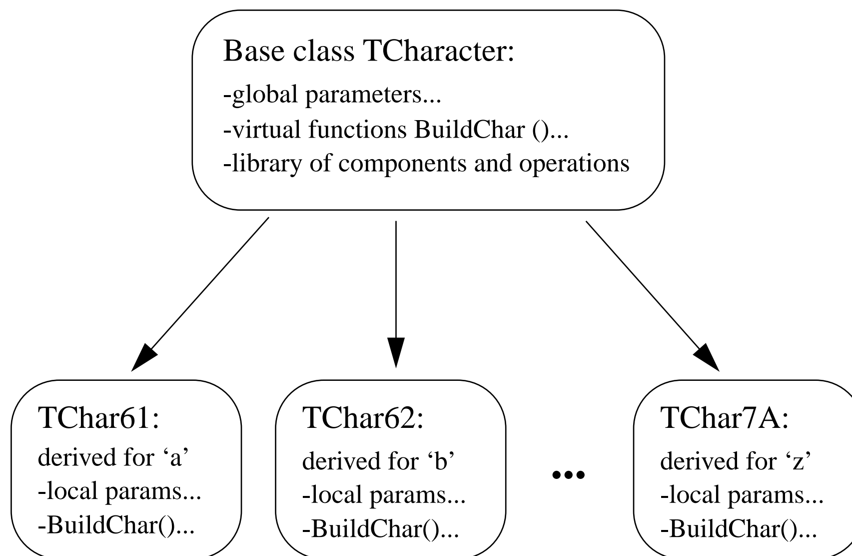


Figure 4.1: The base and derived class model reflect the similarity of parametrizable character synthesis methods and the hierarchy of parameters.

4.1.1 The base class

The *TCharacter* class is the base class for each character's parametrizable character synthesis method. Data members of this class consist of all global parameters. Component synthesizing functions and specific purpose geometric operations are member functions of this class. Some member functions such as the *BuildChar()* are virtual. They intend to be overridden in derived classes.

We first define the data type for parameters, which is a real number or a flag.

```
typedef float TParameter;
typedef short TFlag;
```

The first part of the base class contains all global parameters. Names of parameters begin with a letter indicating whether the parameter is a flag (for example junction or terminal types) or a real number. A parameter may have three values respectively for capital characters, lower-case characters and digits (including symbols).

```
class TCharacter {
   // Data members...
public:
   /* The first letter of a global name has             */
   /* a special meaning:                                */
   /*   i - an integer, often used for earmark types    */
   /*   p - parameter, whose type is defined above      */
   /* serif styles */
   static TFlag iSerifStyle;
   static TFlag iSerifSupportType;
   static TFlag iSerifEndType;
   static TFlag iSerifFaceType;
   ...
```

```
/* global parameter names: ***************************/
/*   a name is like: p(c/m/d)Xxxx...xxx(W/H/D/A) ******/
/*   where, c - capital, m - minuscule, d - digit *****/
/****************************************************/
/* reference lines ----------------------------------*/
static TParameter pBaseLine;
static TParameter pXheight;
...
/* beta ---------------------------------------------*/
static TParameter pBestBeta;
...
static TParameter pLoopExternalBeta;
static TParameter pLoopInternalBeta;
...
/* stem ---------------------------------------------*/
static TParameter pcVerStemW, pmVerStemW, pdVerStemW;
static TParameter pcHorStemW, pmHorStemW, pdHorStemW;
...
/* character width ----------------------------------*/
static TParameter pcRoundLetterW, pmRoundLetterW, pdRo...;
static TParameter pcStemStemW, pmStemStemW, pdStemStemW;
static TParameter pcStemCurveW, pmStemCurveW, pdStemCurveW;
/*serif ---------------------------------------------*/
static TParameter pcVerSerifW, pmVerSerifW, pdVerSerifW;
static TParameter pcVerSerifD, pmVerSerifD, pdVerSerifD;
static TParameter pcVerSerifH, pmVerSerifH, pdVerSerifH;
...
/* others -------------------------------------------*/
static TParameter pcOptCor, pmOptCor, pdOptCor;
...
/* spacing ------------------------------------------*/
static TParameter pCapitalSpacing;
static TParameter pSmallSpacing;
...
/* end of global parameters **************************/
```

The second part of the base class mainly contains functions for synthesizing components and functions for inputting parameters and handling synthesized results. Components are stored in a list written as *s[]*. The data member *theGlyph* holds the result of *BuildChar ()*, and *theMergedGlyph* holds the result after merging all components in *theGlyph*. Both of these results (contours) are stored in global memory.

```
protected:
   TComponent s[kMaxNbOfComponents];
   TGlyph theGlyph;
   TGlyph theMergedGlyph;
```

The constructor of the class is supposed to initialize the parameters. One can also update the parameters or write parameters to a file by calling the corresponding function to load or write a parameter file.

```
// Member functions
public:
   TCharacter ();
   virtual ~TCharacter ();
public:
   static void LoadGlobalParameters (FILE *fp);
   static void LoadGroupParameters (FILE *fp);
   virtual void LoadLocalParameters (FILE *fp);
   static void WriteGlobalParameters (FILE *fp);
   static void WriteGroupParameters (FILE *fp);
```

```
    virtual void WriteLocalParameters (FILE *fp);
```

The parametrizable character synthesis method is represented by a function *BuildChar ()*, which will be overridden for each individual characters. The function *UnBuildChar ()* frees memory when the object is destroyed.

```
    virtual void BuildChar ();
    virtual void UnBuildChar ();
```

A function for synthesizing a component is named as *makeXxxx ()*, which creates an *TComponent* object (see section 4.1.3).

```
protected:
    TComponent nullComponent ();
    TComponent makeStem (TRealPoint p0, TRealPoint p1,
           float w, int end0, int end1);
    TComponent makeSerif (float sw1, float sh1, float sd1,
           float sw2, float sh2, float sd2,
           TLine l1, TLine l2, TLine l3, int dir);
    TComponent makeTopSerif (float sw, float sh, float sd,
           float theta, float overshoot,
           TLine l1, TLine l2, TLine l3, int dir);
    TComponent makeSweep (TRealPoint Pd, TRealPoint Qd,
           TRealPoint Pa, TRealPoint Qa,
           TRealPoint B1, TRealPoint B2);
    TComponent makeLoop (TRealPoint center1,
           float a1, float b1, float eta1, RealPoint center2,
           float a2, float b2, float eta2);
    TComponent makeHalfLoop (TRealPoint center1,
           float a1, float b1, float eta1, TealPoint center2,
           float a2, float b2, float eta2, int dir);
    TComponent makeDot (TRealPoint center,
           float a, float b, float theta);
```

Operations specific for components are also member functions of the base class.

```
    TContour cut (TContour theContour, TLine theLine);
    TContour unite (TContour cont1, TContour cont2);
    TContour smooth (TContour cont1, TContour cont2,
           float r1, float r2, float tang1, float tang2);
```

There are some functions for handling outputs, such as preparing drawing lists for displaying the components and character glyphs. Among them, the function *MergeGlyphShapes ()* merges components in *theGlyph* into a non-overlapping character outline and stores it in *theMergeGlyphShape*.

```
    void MergeGlyphShapes ();
    ...
}; // end of the base class TCharacter
```

4.1.2 The derived classes

Each parametrizable character synthesis method is a derived class of the base class. They are named according to their ASCII code with a prefix "*TChar*". For example the class for character "a" is named as "*TChar61*", the class for character "z" is named as "*TChar7A*", etc. Generally the *TCharXX* class has two parts. The first part declares all the parameters used by this character specification.

```
class TCharXX : public TCharacter {
// Data members
private:
   TParameter ppp1;
   TParameter ppp2;
   ...
```

The second part contains the overriding function *BuildChar ()* which defines the method to build this character. The constructor *TCharXX ()* acts as the parameter entry which loads parameters to the parameter place holders.

```
// Member functions
public:
   TChar61 ();
   ~TChar61 ();
protected:
   void BuildChar ();
   ...
}; // end of TCharXX
```

The *BuildChar ()* function is the parametrizable character synthesis method. In this function, components are synthesized by calling the corresponding component synthesizing functions (*makeStem*, *makeSerif*, etc.). Components are trimmed by calling the proper function and assembled in the same order as the component dependency graph (section 3.2.1.2). The result is stored in *theGlyph*. It comprises all trimmed component outlines.

4.1.3 Other classes

This prototype system also defines some other useful classes, such as *TGlyph*, *TComponent*, *TPool* and *TFont*.

The *TGlyph* class defines and manages a dynamic data structure for storing character glyphs. It starts from the definition of a point, the class *TRealPoint*. A segment, class *TSegment*, consists of a departure point, an arrival point and, if it is a curve, some control points. A contour, class *TContour*, is a segment ring. And finally, a glyph is a set of contours.

The *TComponent* class defines the data structure for components. It contains a “union” of different component types. Both the input parameters and the synthesized results including the auxiliary points are stored in this class.

```
typedef struct stem_struct {      // -- stem and bar
   // args
   TRealPoint p0;
   TRealPoint p1;
   float w;
   int end0;
   int end1;
   // outputs
   TRealPoint A, B, C, D;         // support points
} TStem;
typedef struct serif_struct {     // -- foot serif
   ...
```

```
} TSerif;
typedef struct topserif_struct { // -- top-serif
  ...
} TTopSerif;
typedef struct sweep_struct {    // -- sweep
  ...
} TSweep;
typedef struct loop_struct {     // -- full loop
  ...
} TLoop;
typedef struct halfloop_struct { // -- half-loop
  ...
} THalfLoop;
typedef struct dot_struct {      // -- dot
  ...
} TDot;
typedef struct path_struct {     // -- others
  ...
} TPath;

typedef struct component_struct {
  int ComponentType;
  union {
    TStem Stem;
    TSerif Serif;
    TTopSerif TopSerif;
    TSweep Sweep;
    TLoop Loop;
    THalfLoop HalfLoop;
    TDot Dot;
    TPath Path;
  };        // an anonymous union, so you can access
            // its member shortly like s0.Sweep
  TContour theContour;
  TContour theAuxContour;
  ...
} TComponent;
```

The glyph is a dynamic data structure and memory allocating and freeing are frequent. Since the basic unit of memory allocated for a glyph is the memory of a segment, we defined the class *TPool* to take up the memory management for segments. The class *TPool* manages a pool of memory units for segments and allocates or frees a segment in its own segment pool. This solution is more efficient than calling the OS memory management.

The *TFont* class manages all characters of a font. It contains a character encoding table (element *theTable*) which indexes each character's entry by its ASCII code. The entry of a character contains the name of the character and the *TCharXX* object of the character. This enables looking up a character by its name. The class also contains documentary information for font management. The member function *GetAChar ()* is the first application interface through which an application obtains the object for this character.

```
typedef struct char_entry {
  char name [kMaxCharNameLength];
  TCharacter *theChar;
} TCharEntry;

class TFont {
```

```
// data members:
public:
   char sFontname [kMaxStringLength];
   char sFontfamily [kMaxStringLength];
   char sCreator [kMaxStringLength];
   char sVersion [kMaxStringLength];
   char sDescription [kMaxStringLength];
   char sCopyright [kMaxStringLength];
   char sEncodingStandard [kMaxStringLength];
   ...
   TCharEntry theTable [kMaxNumberOfCharacters];
   // member functions
public:
   TFlexFont ();
   ~TFlexFont ();
   TCharacter *GetAChar (int charcode);
   ...
}; // end of TFont
```

4.2 The implementation of the parameter hierarchy

4.2.1 Parameter files

Parameters are stored in files in a readable text format. Each concrete font needs three kinds of parameters and, accordingly, three parameter files: the global parameter file, the group parameter file and the local parameter file. In order to simplify their manipulation, such as preparation, comparison and modification, and to maintain portability, parameter files are text files instead of binary ones. Parameter files have a predefined understandable format and the parameter loading functions of the *TCharacter* class interpret this format. C or C++ style comments are allowed in the parameter file. The basic format is a parameter assignment which looks like

```
parameter_name = parameter_value;
```

Global parameters have standard names which can be recognized by the file parser. Local parameters have no standard name, but the set of local parameters for each character is fixed. We could give each local parameter a fixed name and make it recognizable by the parameter file parser. However, parameter groups may vary from typeface to typeface, meaning that the grouping of local parameters ought to be dynamic.Therefore, the basic parameter assignment format has been extended to enable parameter group definition in the parameter files. This is realized by a macro definition and a reference mechanism.

A macro name is a string starting with the letter “$”. A macro definition looks similar to a parameter assignment except the left side of the sign “=” is a macro name. When the file parser scans in a macro definition, it installs a “name-value” pair in a macro definition table which is part of the parser. Then, this macro name can be used as a parameter value in a later parameter assignment. For example, the following two lines first associate a macro name called “$APercentage” with a value 0.9, then assign this macro name as a parameter value to the parameter named “proportion1”.

```
$APercentage = 0.9;            // define a macro
proportion1 = $APercentage;    // refer to the macro
```

Sometimes, a simple expression as the right side value of the parameter assignment will provide much flexibility for parameter grouping. For example, if a parameter “proportion2” is supposed to be the difference between 1.0 and the macro “$APercentage”, we may want to assign expression “1.0 - $APercentage” to this parameter. In order to use simple expressions, we introduce PostScript-like post-fixed expressions using a limited set of operators: *add* for add, *sub* for subtract, *mul* for multiply, *div* for divide, *neg* for negative, and *iplt* for interpolate (Tab. 4.1). To provide for more flexible parameter assignations, we also introduce the *if/ifelse* branch operators and some boolean operators (Tab. 4.2).

Table 4.1: Algebra operators for writing parameter files.

Usage of Operator	Meaning in C/C++
a b *add*	a + b
a b *sub*	a - b
a b *mul*	a * b
a b *div*	a / b
a *neg*	- a
a b f *iplt*	a * (1 - f) + b * f

Table 4.2: Branch and boolean operators.

<table>
<tr><th>Usage of Operator</th><th>Meaning in C/C++</th></tr>
<tr><td>a b eq</td><td>a == b</td></tr>
<tr><td>a b ne</td><td>a != b</td></tr>
<tr><td>a b ge</td><td>a >= b</td></tr>
<tr><td>a b gt</td><td>a > b</td></tr>
<tr><td>a b le</td><td>a <= b</td></tr>
<tr><td>a b lt</td><td>a < b</td></tr>
<tr><td>a b and</td><td>a && b</td></tr>
<tr><td>a b or</td><td>a || b</td></tr>
<tr><td>a not</td><td>! a</td></tr>
<tr><td>(0|1)[a] a if</td><td>if (1) a</td></tr>
<tr><td>(0|1) a b ifelse</td><td>if (1) a else b</td></tr>
</table>

a. Boolean values “true” is represented as “1”, “false” is represented as “0”. They are often generated and pushed into the stack by the boolean operations.

The reason to use post-fixed expressions is to simplify the implementation of the grammar parser, despite the fact that the expressions may not look natural. Using the subtraction operator, we can write the above example as follows.

```
proportion2 = 1.0 $APercentage sub; // 1 - &APercentage
```

Now, let us present the method to link several local parameters into a group parameter. For example, we want to define a group parameter to achieve the coherent modification of the arch depth of characters “h”, “m”, “n” and “u” (see also Fig. 3.21). We can define a macro name “$ArchDepth” and then refer to that macro name in those characters.

Since the macro name must be defined before it is referred to, the parameter files should be loaded in a fixed order: first the global parameters file, then the group parameter definition file and finally the local parameter file. Modifying the group parameter file requires reloading both the group parameter file where the macros are defined and the local parameter file where the macros are referred to.

The flexible format of parameter files also provides a way to specify parameter varying rules in order to make typeface variation. Experiments on the synthesis of derived font by modifying parameters are presented in section 5.2.1. In Appendix G, we give the complete global, group and local parameter files for Times Roman with a macro defined for applying boldness variation.

4.2.2 Implementation

The macro definition and reference mechanism is realized by a table called the *MacroTable*, which is functionally similar to the PostScript dictionary. The post-fixed expressions are implemented by a stack called the *OperandStack*, which is a simplified version of the operand stack in a PostScript interpreter.

The macro definition and reference mechanism is realized by a table which holds name-value pairs. To define a macro, the macro name and values are stored (installed) into the table. To reference (or use) a macro, the macro name is used as a key to search the table until it finds the most recently installed name-value pair which has the same macro name. The value is returned to replace the macro’s name.

To process a post-fix expression, the parameter file parser pushes each operand it meets into the *OperandStack* until it comes across an operator name. These are defined in Tab. 4.1 and Tab. 4.2. Then the parser pops the operands required by the operator, from the *OperandStack*. The binary operators *add*, *sub*, *mul* and *div* require two operands, the unary operator *neg* requires one operand, and the interpolation operator *iplt* requires three operands. The boolean operand for the branch operators *if* and *ifelse* is treated according to this rule: value 0 means *false*, value 1 means *true*. After executing the function of the operator, the result is an operand ready for the next operation, pushed on top of the *OperandStack*. The end-of-line character (any comments will be skipped) indicates the end of the expression and the parameter file parser knows to pop out the latest operand as the result value of the expression. Then the *OperandStack* should be empty again as it was before parsing the expression.

4.3 Output forms of synthesized characters

Typographic characters are first synthesized as a set of trimmed components. To make use of them in an application, the synthesized characters can either be converted to displayable bitmaps, or to outlines in order to synthesize traditional outline fonts.

In both cases, the character is scaled just by scaling the global parameters, because all local parameters are expressed as proportions of global parameters.

4.3.1 Component-based rasterization of synthesized characters

When synthesized component-based characters are converted to character bitmaps, for example for printing and display purpose, they are rasterized. The rasterization process converts geometrically represented character shapes into bitmaps. The high quality “flag-fill” rasterization algorithm [Hersch88] can correctly convert non-overlapped contours into bitmaps. However, the component-based character may comprise overlapping component contours, which require the flag-fill algorithm to be extended to support overlapped contours.

The basic flag-fill algorithm can be described by the following two steps:

Step 1: Scan-convert all segments of a contour into pixels and mark them as flags in an empty device bitmap:
M[i][j], i = 0 .. n-1, j = 0 .. m-1,
which is an array with *n* rows and *m* columns of pixels.

Step 2:
```
FOR i = 0 TO n-1 DO        // for each ecan line
    v := 0;                // values of pixels: 0 for white, 1 for black.
    FOR j = 0 TO m-1 DO
        // 1. toggle pixel values
        IF M[i][j] has been marked as a flag THEN
            IF v == 0 THEN v := 1 ELSE v := 0 ENDIF;
        ENDIF;
        // 2. set pixel values
        M[i][j] := v;
    ENDFOR
ENDFOR
```

If we represent the flagged pixels on a scan line as p_1, p_2, ..., p_k, black pixels are located between successive pairs of flags: p_1p_2, p_3p_4, ..., $p_{k-1}p_k$. This filling result can be best described as the “even-odd” filling rule in PostScript terminology (Fig. 4.2a, see also [Adobe85, pp.70-71]). If two contours overlap, the overlapped area will be regarded as the outside of the filling area. To fill the component based character shapes, an overlapped area need to be treated as the inside of a filled area. This requires using the filling rule known as the “non-zero winding” rule (Fig. 4.2b).

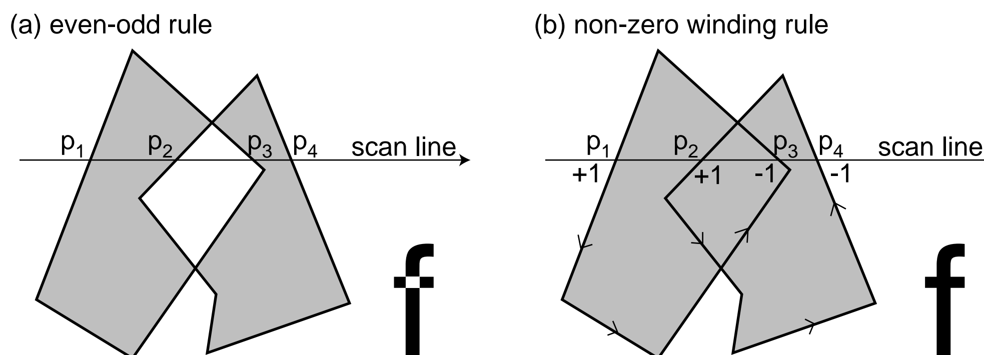


Figure 4.2: Two filling rules and their effects on the overlapped character contours.

To apply the "non-zero winding" filling rule, the basic flag-fill algorithm is modified. The non-zero winding filling rule requires oriented input contours. When a scan line intersects a contour, the algorithm can detect if the scan line enters or leaves a contour part. In this thesis, the positive contour orientation is counterclockwise. At an intersecting point, if the intersecting contour segment traverses the scan line from left to right, the scan line enters the contour and vice-versa (see also Fig. 4.2b). To remember the type of intersection between the scan line and the contour, intersection attributes, which indicate *in* or *out*, should be associated to each flag.

The basic flag-fill algorithm needs also to be improved in the case that two flags overlap in the bitmap. We store flags on each scan line by their *x* coordinates as well as their intersection attributes (*in* or *out*) in a "sorted flag array", instead of bit-setting in a bitmap. In this data structure, two overlapped flags will appear twice in the sorted flag array. For character rasterization, a flag can be represented by a two-byte integer with the highest bit indicating the intersection attribute of this flag, and the remaining bits representing the *x* coordinate. The maximum length of the flag array depends on the maximum number of possible intersections a scan line has with the contour of a character. This number is not large, normally less than ten for Latin characters. The modified flag-fill algorithm for the "non-zero winding" filling rule is described below.

Step 1: Scan-convert all segments of a contour into pixels; store the *x* coordinate of each pixel associated with its intersection attribute (*in* or *out*) in a 2-D flag array:

S[i][j], i = 0 .. n-1, j = 0 .. m-1,

where *i* is the index of a scan line, *n* is the maximum number of scan lines, *j* is the index of a flag in a scan line, and *m* is the maximum number of intersections in one scan line. Sort flags of each scan line (S[i]) in an incremental order. The number of flags (an even number) in each scan line is also counted and stored in an auxiliary array:

NbOfFlags[i], i = .. n -1.

Step 2: Find "black" spans on each scan line and paint them to the raster device. A span is represented by two *x* coordinates, one for the *first* pixel, the other for the *last* pixel.

FOR i = 0 **TO** n-1 **DO** // for each scan line s[i]
  counter := 0; // initialize to count the winding number
  FOR j = 0 **TO** NbOfFlags[i]-1 **DO** // for each flag in the scan line
    IF S[i][j].attribute == *in* **THEN**

```
                        counter := counter+1;
                        IF counter == 1 THEN
                                first := S[i][j].x;
                        ENDIF;
                ELSE // the attribute is out
                        counter := counter-1;
                        IF counter == 0 THEN
                                last := S[i][j].x;
                                paint_span (first, last, i);      // in scan line i
                        ENDIF;
                ENDIF;
        ENDFOR;
ENDFOR.
```

These filling functions are enclosed in a class named *TFlagFill*, which uses a table data structure to store flags and attributes of each scan line.

4.3.2 Outline generation of synthesized characters

When a synthesized character is to be used as an outline, for example to draw the outline of the character or to convert the synthesized characters from a component-based shape description into a traditional outline-based shape description, our system can merge all individual component contours of the character into a character outline. The merged character outline may contain several contours but none of the contours is overlapped.

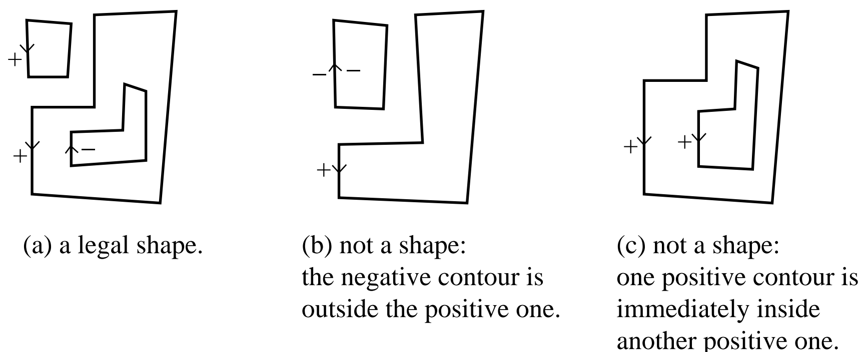
General purpose union or intersection algorithms for geometric shapes can be applied to merge components, for example [Vatti92]. We have developed our own specific shape merging package, which has been used in our font synthesizer. We give only a short overview of its working principles, since the detailed description of its functionality is outside the scope of the present thesis.

Assuming contours have an orientation, our algorithm first detects all intersections between two contours. Then a tracing process is applied to find the boundary of the final merged shape. The principle of the tracing algorithm is simple. Suppose the direction of a positive contour is counterclockwise, the tracing algorithm for finding the union of two contours will always choose the right-most branch candidate to follow at each intersection between the two contours.

Since a character may incorporate one or several external and one or several internal contours, we call the counterclockwise external contour a positive contour, and the clockwise internal contour a negative contour. Then a shape is defined as follows (also illustrated in Fig. 4.3).

Definition 4.1: A shape is a set of disconnected positive and negative contours, which satisfy the following conditions:
1) A negative contour represents a hole and must reside inside a positive contour.

2) A shape may contain one or more positive contours if none of the positive contours is immediately inside another positive contour.



(a) a legal shape.

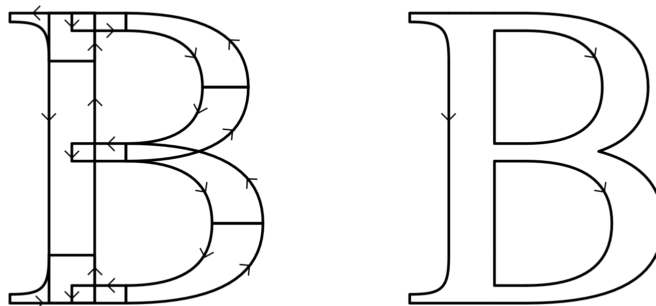
(b) not a shape: the negative contour is outside the positive one.

(c) not a shape: one positive contour is immediately inside another positive one.

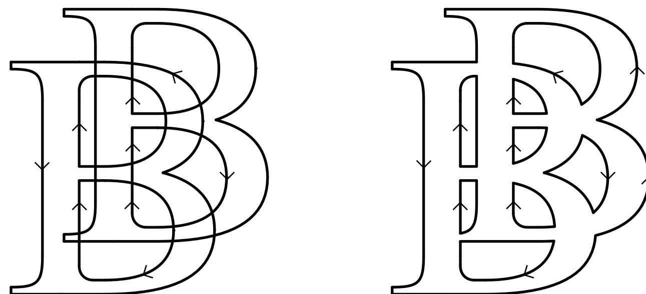
Figure 4.3: Definition of shapes: good shape vs. bad shapes.

Shapes satisfying these conditions can be correctly merged with our shape merging algorithm, and the result is also a shape respecting these conditions. The shape merging process can be repeated to merge more than two shapes. In the case of merging components of a character, each component is merged successively. This shape merging algorithm not only merges character components (Fig. 4.4a), but can also be used to create combined characters or glyphs (Fig. 4.4b) and to create outlines of complex graphic shapes (Fig. 4.4c).

(a) Merging characters components (left) into outlines (right).



(b) Combining two characters (left) into a new glyph (right).



(c) Creating the outline of a complex graphic shape.

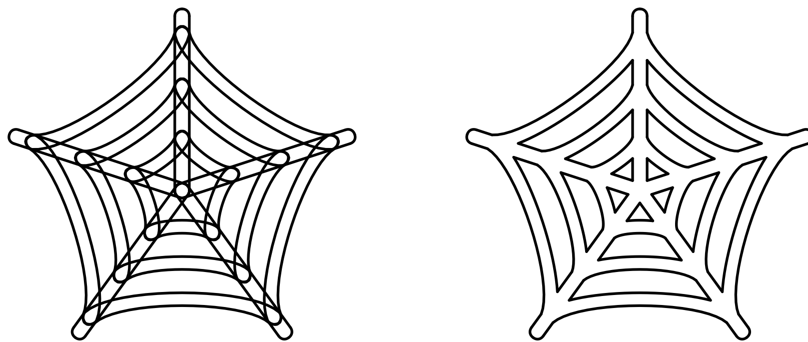


Figure 4.4: Merging components (graphical shapes).

4.4 Automatic optical spacing

To layout text, spacing information such as character pair kerning is needed. However this kind of fixed information is not suitable for parametrizable font generation. The ideal space between a character pair depends on the space perception of the human vision system. Traditionally, typographers tune spaces between many pairs of characters by hand and save them in a kerning table [Karow94, pp.173-192]. This is costly, because n characters may have n^2 possible pairs. For parametrizable fonts, we need an automatic method for building kerning tables. In case of change of weight, contrast or stress, we have to generate spacing values on-the-fly.

4.4.1 The principle of automatic optical spacing

Previous work [Hersch95] has shown that automatic spacing of character pairs is possible and good results have been obtained for both bilevel and graylevel character rasterizations. The algorithm for automatically computing the best possible spacing value between two given characters is based on a model of how the human visual system perceives a space between two character shapes. This model enables us to compute the perceived space between two characters from the geometry of their outline shapes.

The model is based on the assumption that, when we read a text line, our vision system recognizes successive characters by activating a spatial-frequency channel in a band one octave wide, ranging from one cycle to two cycles per character. The perceived intercharacter space is therefore filtered and the shape details of the two character parts are smoothed out. Because the filtering process tends to close open cavities, such as the one in character “c”, the following criteria are used to model the perceptual space between two characters:

- The inner shape of a character does not have much influence on the perceived intercharacter space.

- Only the parts of cavities that can be seen from the exterior of the character influence the visual intercharacter space.
- The cavity space and exterior contour parts that are close to the next character's contour have the largest effect on the perceived intercharacter space.

The *perceptually equivalent space (PES)* can be computed based on the above criteria and then compared with the *ideal optical space (IOS)*. The *kerning adjustment space (KAS)*, which can be either positive or negative, is used to compensate the difference between *PES* and *IOS*. The *kerning adjustment width (KAW)* is the result of dividing the *KAS* by the *character's height (CH)*.

$$KAS = IOS - PES \quad (4.1)$$

$$KAW = KAS / CH \quad (4.2)$$

In our font parametrization system, characters have no predefined width since predefined character width is not suitable for parametrizable fonts. Therefore, we compute the *kerning adjustment space* by placing two character's bounding boxes against each other. Suppose the left side of "this" character is placed as x_{this}, then the "next" character position x_{next} can be computed by adding the *bounding box width* of this character BBW_{this} and the kerning adjustment width *KAW* to x_{this}.

$$x_{next} = x_{this} + BBW_{this} + KAW \quad (4.3)$$

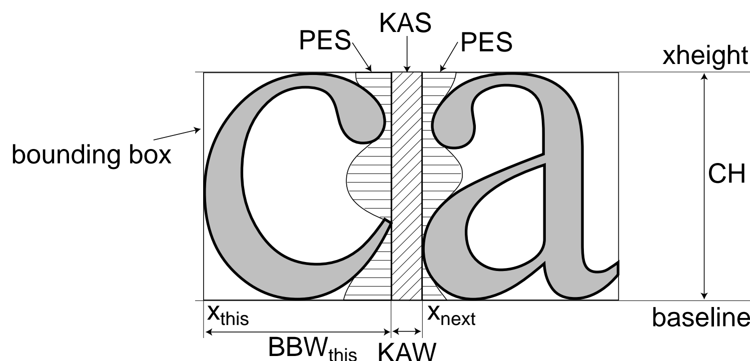


Figure 4.5: Computing optical spacing between two successive characters.

The challenge resides in computing the *perceptually equivalent space* between characters. The work presented in [Hersch95] is based on a series of geometric shape transformations for character outlines. In this thesis, we introduce another method which is based on the character images instead of the geometrical shapes.

4.4.2 The implementation

To compute the perceptually equivalent space, we need to extract the *left wall* and the *right wall* from a character's flag-filled bitmap. A *left wall* is a set of pixels which are the

leftmost flags of each scan lines. A *right wall* is a set of pixels which are the rightmost flags of each scan lines. In case there is no flag on a scan line, the right-most pixel and the left-most pixel of the bounding box are used accordingly. The example in Fig. 4.6 illustrates the extracted right wall of character “c” and the extracted left wall of character “a” in order to compute the optical space between the character pair “ca”.

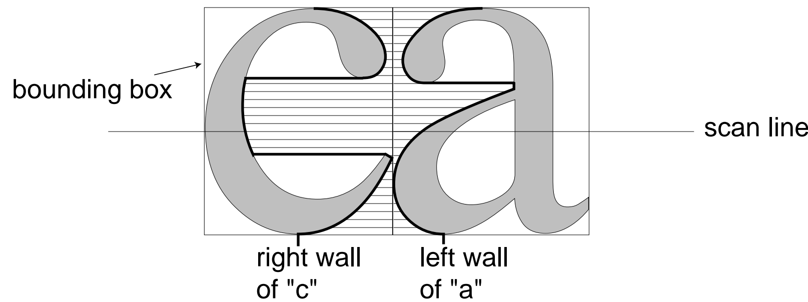


Figure 4.6: Walls extracted from characters.

The walls are then filtered to imitate the edge of the perceptually equivalent space. We apply a simple linear filtering process which fills out the concavities with $\pm 45^{\circ}$ straight lines. The space between the filtered right wall of *this* character (character “c” in the example) and the filtered left wall of the *next* character (character “a” in the example) is the area which the human visual system is most likely to recognize as the white space separating the two characters. The filtered walls of the example in Fig. 4.6 are shown in Fig. 4.7.

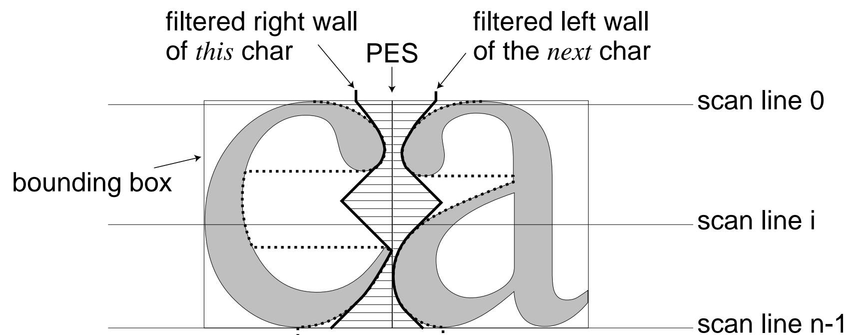


Figure 4.7: Filtered walls and perceptually equivalent space (PES).

Suppose the filtered right wall of the *this* character is stored in an array $FilteredRightWall_{this}$ [i], i = 0 .. n-1; and the filtered left wall of the *next* character is stored in an array $FilteredLeftWall_{next}$ [i], i = 0 .. n-1. The *perceptually equivalent space (PES)* can be computed as the sum of the length of all scan lines between the two walls.

$$\text{PES} = \sum_{i=0}^{n-1} (\text{FilteredLeftWall}_{next}[i] - \text{FilteredRightWall}_{this}[i]) \quad \textbf{(4.4)}$$

Let us analyze the time and space requirement of this image processing based implementation. Once the character has been rasterized by our flag-fill algorithm, extracting left and right walls consists in returning the minimum and maximum points of each scan line. The running time is *O(n)* in respect to the number of scan lines *n*. The linear filtering algorithm and the final computation of equation (4.4) can be both finished in *O(n)* running time. Therefore, the total running time is *O(n)*, assuming that the characters have been rasterized. Additional memory to store the extracted walls requires $n + n$ elements, far less than the memory requirement for the flag tables and the character bitmaps used at character rasterization time (flag-fill). Fig. 4.8 gives an example of automatically spaced characters computed by this image processing based algorithm.

hamburgefonts

Figure 4.8: Layout of synthesized characters by automatic optical spacing.

To make the result more pleasant, several tuning parameters are provided. For example, the space between a character pair “nn” is regarded as the standard “stem-stem spacing”, and the space between character pair “oo” is regarded as the standard “curve-curve spacing”. The parameter *VisualDepthLimit* is used to limit the maximum depth of a concavity so that certain character pairs such as “vo” can be properly spaced. The parameter *MinSpace* gives the minimum space between character walls in order to avoid overlapping of character parts, for example the space between the serifs of character pair “vw”.

4.5 Automatic hinting and grid-fitting

Even though automatic hinting and grid-fitting is a problem mainly related to character rasterization, it is still interesting to see how component-based parametrizable fonts may be grid-fitted to generate high-quality rasterized characters at small size. We found that our font parametrization technology facilitates the process of grid-fitting and largely removes the need for hinting, since information about components is generally sufficient for grid-fitting.

4.5.1 The theory of grid-fitting and hinting

When an outline character is rasterized into a discrete bitmap, the quality of the character image can be damaged due to rounding effects. To improve the quality of character rasterization, a process called grid-fitting is applied to a character outline before it is scan-converted. Grid-fitting is based on the piecewised deformation and grid adaptation of outline parts [Hersch93]. Since traditional character outlines contain only straight line and curve segments, additional information called hints, constraints or instructions is required by traditional outline font rasterizers. The hints or instructions normally specify how a character outline is to be modified in order to preserve features like symmetry, thickness and uniform appearance on the rasterized text page. Typical hinting information are the TrueType instructions [Microsoft95b], the Adobe Type1 hstem/vstem based hints [Adobe90], the URW intelligent font scaling technology [Karow94, pp.105-149] and EPFL RastWare constraint specification [Betrisey89].

Hinting information can be added either by experienced digital typographers or by automatic hinting tools. The quality of automatically generated hints depends on the tool's intelligence to locate typographical parts from geometric outline description of characters. The automatic hinting system described in [Hersch91] uses a model matching method to find vertices which are to be hinted. This method requires predefined font-independent character hinting models for each character or symbol.

With the parametrizable fonts, however, automatic generation of traditional hinting information can be realized without additional information such as predefined models. Grid-fitting can also be executed on-the-fly without pregenerated hinting information.

4.5.2 The implementation

4.5.2.1 Automatic generation of traditional hints

To use traditional outline font rasterizers, the synthesized parametrizable fonts can be converted into traditional outline fonts. In this case, besides the outline representation, hint information needs to be added to the converted outline font. With the component-based character description, many important hints can be generated automatically.

Typographic features which are intended to have uniform appearance at small bitmap size, for example main vertical stem width, are controlled by standard feature measurement hints, such as the *Control Value Table (CVT)* in Microsoft TrueType and the *stem width snapping array* in Adobe Type1. Due to rounding, slight differences of the real stem width may result in a one pixel difference of the rasterized stem width. This difference can be easily noticed when the stem width is only a few pixels. Therefore, at small bitmap size, single standard stem width values are used for lowercase, uppercase and digits to avoid uneven rasterization of similar stems. Obviously, this problem is solved with our component-based characters, where the global font parameters control vertical and horizontal stem widths throughout the whole font.

Hint information for individual characters specifies distances between contour points and grid-fitting requirements. With component-based characters, these contour points and their relationships are part of the component designs. As in Adobe Type 1, grid-fitting requirements may be directly derived from component information.

Some hints concern very specific character features. They are usually added by experiments [Stamm98]. However, since printing resolutions are increasing, they become less and less important. Therefore, automatic hinting methods should generate only regular and important hints.

4.5.2.2 Grid-fitting component-based characters without hints

When the synthesized characters are rasterized by our component based character rasterizer (section 4.3.1), grid-fitting can be executed without any pregenerated hinting information. The component representation of typographical parts and the dependency between components already comprise the necessary information to ensure good quality grid-fitting.

Our experimental grid-fitting of parametrizable characters uses basic grid-fitting rules which have been proved to be the most important ones. These grid-fitting rules for bilevel characters include:

- Consistent character features. It is a typographer's experience that all identical character parts should have an identical look. Character features in our parametrizable font, such as stem width and serif height, are global parameters. This means that character features are automatically kept consistent throughout the whole font. Grid-fitting requires for phase control the displacement of the contour of a stem. If the displacement is done based on a whole component instead of pieces of segments, the width of the stem is automatically kept consistent.
- Phase control of reference lines. To generate horizontally symmetric round letters such as "o", we normally center the area between the base line and the x-height line on the grid. In our component-based parametrizable font synthesis system, reference line positions are global parameters. Therefore, this phase control of reference lines can be done by controlling the global parameters of the scaled reference line positions.
- Centering of vertical and horizontal stems. Experiences show that vertical stems and horizontal stems have the most impact on the human vision system especially when a character is rasterized at a small bitmap size. Centering a stem at the center of a grid not only ensures that stems with the same width have the same rasterized stem width, but also helps to maintain the symmetry of the stem's serifs after rasterization. Main vertical stems in our *component dependency graph* (section 3.2.1.2) are always at the second level which is directly controlled by the *main component width* of the character. To grid-fit a stem, the *main component width* is slightly modified to ensure that successive stems are centred on the grid.

- Phase control of vertical and horizontal round parts. There are two kinds of round parts in a character: the connecting sweep and the loop or half-loop. The position and dimension of a connecting sweep usually depend on the connected components. To adapt itself to the displacement of connected character parts during the grid-fitting process, the contour or vertices of a connecting round part has to be specified as “elastic” according to traditional hinting systems [Betrisey89]. However, our component synthesizer of sweep components has enough intelligence to adjust a connecting sweep in response to the displacement of the connected components. Regarding the loop and half-loop components, phase control can be applied to have nicely rasterized extrema. Keeping the phase of the extremum of a round part in a certain range, for example between 1/16 to 9/16 [Hersch95], will prevent a “single pixel” or a flat “long run” at the extrema. To control the phase of a round part, the position of the extrema need to be known. Fortunately, our component design for loop and half-loop specifies the extrema explicitly.
- Dropout control. For a bilevel character bitmap, pixel dropout can be a very bad defect caused by rasterization at very small size. However, no effective grid-fitting rule can prevent pixel dropout of a slanted or curved part of a character by simply modifying its contour. Traditional outline font technology controls dropout by the filling algorithm. For our component-based fonts, we control pixel dropout at filling time for each component. Therefore, the pixel dropout of the whole character is controlled. Besides, since we know the types of each component, we can selectively control whether a component is allowed to have pixel dropout or not. For example, we may specify that a serif slab is allowed to have pixel dropout at a certain small bitmap size.

Figure 4.9: On-the-fly grid-fitting of our synthesized characters for grayscale display. Character sizes measured by capital letter height are 8 pixels, 10 pixels and 12 pixels respectively. There are 16 graylevels from black to white, printed without gamma correction. As stated in [Hersch95], grayscale characters look best when viewed on a LCD display.

Grid-fitting rules for generating good quality grayscale characters are basically the same as the rules for bilevel characters except that dropout control is not needed. Some modification are however necessary to enhance the contrast and the overall perceptual weight [Hersch95]. Instead of centering vertical stems, we align the left edge of a vertical stem to the border of a pixel so that the left side of the stem appears as “black” as possible. Fig. 4.9 describes the results of our on-the-fly grid-fitting method applied on the synthesized characters for grayscale display. The grid-fitting methods used in these preliminary experiments include centering the band area between the x-height and the base lines and left aligning vertical stems to pixel boundary.

4.6 Evaluation

Component based font descriptions require global parameters, group parameters and local parameters. Tab. 4.3 shows how many parameters are needed for our experimental versions of parametrizable lower-case Times, Bodoni and Helvetica characters. Bodoni requires less parameters, since all its serifs are slab serifs (zero serif depth). Sans-serif font Helvetica further reduces the number of required parameters. The number of local parameters varies from character to character and from typeface to typeface, thus a mean value is used for statistics.

Table 4.3: Nb of parameters and size of font comprising lower-case characters “a” to “z”

	Times	Bodoni	Helvetica
nb of global parameters	110	98	63
nb of group parameters	31	31	31
mean number of local parameters per character	15.3	16.0	14.0

A 26 lower-case character component-based parametrizable font of the complexity of Times requires approximately 540 parameters, i.e. 1080 bytes. Comparatively, TrueType requires approximately 1500 bytes for the global parameters and a mean of 154 bytes per character for storing the outlines of lower-case characters[1]. The grid-fitting instructions needed by TrueType for character generation at medium and low resolution require an additional mean amount of 385 bytes per character. In contrast, component based character descriptions already include all information about their structure (stems, bars, arches, serifs). From our experiment of on-the-fly grid-fitting, we can conjecture that the additional grid-fitting information [Adobe90][Hersch91][Karow94] required to rasterize characters at medium and low resolution is negligibly small. In addition, component-based fonts enable generating derived fonts (condensed, semi-bold, etc.) by

1. Composite characters such as accentuated characters are described by pointing to the basic character shape and to the accent description. Therefore, only basic characters are considered when giving the mean storage cost of single character descriptions.

changing the values of a few global parameters (see chapter 5 for possible variations). A single component-based font may therefore replace several traditional outline fonts. We therefore expect component-based fonts to require an order of magnitude less storage space than traditional outline fonts.

With our current implementation of the font synthesizer, we either synthesize characters made of partly overlapping components, excluding components which extend out of the character by using the *cut* operation (section 3.2.3.1) or we can, after generating the components, convert the component-based character descriptions into traditional outline-based characters by using our specifically designed shape merging algorithm to assemble components into character outlines. Tab. 4.4 shows the character generation speed for producing component-based character descriptions and for producing traditional outline-based descriptions.

Table 4.4: Lower-case character outline generation speed on a Power Mac 7200/90

	Times	Bodoni	Helvetica
synthesize component-based character outlines	418 char/s	400 char/s	1350 char/s
synthesize components and merge them into outline characters	70 char/s	71 char/s	160 char/s

Our experimental component-based font synthesizer consists of the synthesizer kernel (ComponentEngine, ShapeMerging, Kerning, and FlagFill) and of the parametrizable character synthesis method (CharFiles). It comprises about 15000 lines of C++ programs, among which 30% are comments. The mean size of one parametrizable character synthesis method is about 220 lines of C++ without comments. The Metrowerks CodeWarrior IDE 1.7.4 compiler generates about 239KB of PowerPC 602 native code and 41KB of static data for both the kernel and the 26 characters' parametrizable character synthesis methods. The linked ANSI libraries of this compiler are comparatively large, i.e. about 600KB including static data.

Table 4.5: Analysis of the programs of our component-based font synthesizer.

	CodeWarrior IDE .cp and .h	PowerPC 602 native code	Static data
ComponentEngine	8800 lines .cp and 1800 lines .h	78K	21K
ShapeMerging		51K	0.9K
Kerning		4K	0.4K
FlagFill		5K	0.4K
CharFiles	5700 lines .cp and .h	101K	18K
Linked Metrowerks' ANSI C/C++ libraries		495K	110K

4.7 Summary

This chapter presented our implementation of the component-based parametrizable font synthesizing system. Since our component-based font synthesis system is suitable for object-oriented programming, the programs are written in the C++ programming language. The comprehensive base class makes adding a new design of a parametrizable character synthesis method as easy as creating a new derived class.

A simple grammar for writing the parameter files has been developed in order to carry out typographic experiments. To rasterize synthesized characters, a scan-conversion algorithm adapted for component-based character representation has been presented. The component-based representation can also be converted into traditional outline-based representation using a specifically designed shape merging algorithm.

Automatic optical spacing and automatic hinting are two challenging topics for the traditional outline font technology, yet have not been solved perfectly. We found that, with our component-base font parametrization method, they are both worth a new study. The preliminary results are encouraging. The component-based description of characters is well suited for doing grid-fitting on-the-fly, using our specific component-based character rasterizer.

CHAPTER 5

Experiments and applications

Besides the high storage compression rate (section 4.6), our component-based font parametrization method is also suitable for applying typographical typeface variations, which are often beyond the capabilities of the traditional outline font technology. Being able to carry out typeface variations has been an important goal (yet only partially reached) of several previous research projects in digital typography [Knuth86a][Shamir98][Zalik95] [Schneider98]. In this chapter, we first introduce a visual font design and modification environment based on the component-based parametrizable font synthesizer. Then, we will present a few typographical experiments.

5.1 A visual environment

To test the prototype of our component-based parametrizable font synthesizer and to demonstrate the flexibility and functionality of our font parametrization method, a visual environment for parameter editing and modification has been developed in the Peripheral Systems Laboratory of the EPFL. This environment, named *FlexFont Editor*, integrates functions such as character visualization, parameter file saving and loading, global parameter visual modification, and specimen printing using a *multiple document interface (MDI)*[1] Windows application.

The term "visual" concerns two aspects. First, any parameter modification can be seen immediately in the character display window. Second, the modifications of global parameters are done with slide bars and iconed buttons.

5.1.1 Character visualization

The overall appearance of the environment is shown in Fig. 5.1. Some frequently used tools are displayed in tool bars. There are four different tool bars: the character set bar, the file bar, the parameter bar and the preview bar.

The *character set tool* bar displays the character set in the opened parametrizable font. By clicking on a character in this tool bar, the selected character is displayed in an

1. A terminology of Microsoft Windows describing the application's architecture. A multiple document interface (MDI) differs from a single document interface (SDI) in that the multiple document interface enables a user to open multiple documents each with its own windows, while the single document interface allows a user to work with just one document at a time.

zoomed character observing window. Characters can be displayed as either filled character images or outlined components.

Figure 5.1: Multiple character displaying windows

The current version of the FlexFont Editor does not support local parameter modification. Therefore, we modify local parameter for individual characters by editing their local parameters in the local parameter file with a text editor. After saving the modified parameter file, we reload the parameter file to update the characters in the character observing window. The file tool bar provides a short-cut for parameter updating.

The *preview tool bar* has a character displaying area which displays automatically spaced character samples. The text is entered through a text editing box. Clicking on the "waterfall" button opens a window for displaying and printing the sample text at different sizes.

The "large character" printing tool in the file tool bar enables printing a component-based character at full-page size according to the size that typographers generally use to design characters. Clicking on the printer icon brings out a character printing window which previews the printing result for the currently selected character (Fig. 5.2). Reference lines can be added as options. The character can be printed either as a filled image or as outline components.

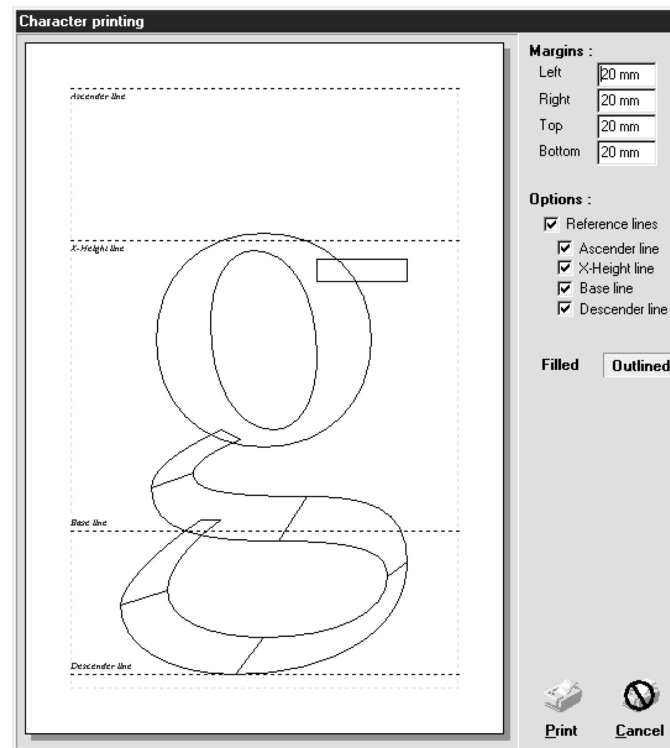


Figure 5.2: The "large character" printing window.

5.1.2 Global parameter modification

The *parameter tool bar* contains tools for applying global parameter modification and typeface style variation. Tools for global parameter modification include: font information editor, reference lines editor, serif style selector, and global parameter editor. Tools for typeface style variation include: boldness variation tool, contrast variation tool and character width variation tool.

The *font information editor* is a dialogue window which holds editing fields for each item of descriptive information in the opened parameter file. Descriptive information, sometimes called "font header", is not part of parameters, but helps the user to manage a parametrizable font.

Character height and the height proportion of different parts of characters are controlled by *reference lines*, such as the base line, the x-height line, the caps line, the ascender line and the descender line. Changing the position of reference lines modifies the character height throughout the whole font. This process is visualized by the *reference line editor window* (Fig. 5.3). In this tool, one can click a reference line and drag it to move its position higher or lower. The effect can be seen immediately on the preview characters, which are selected in the preview text editor field.

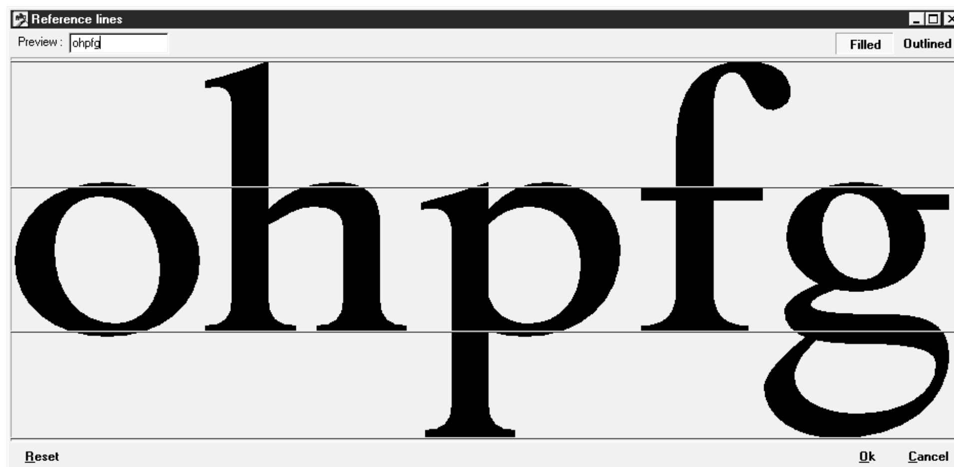


Figure 5.3: The reference line editor window.

The selection of *serif styles*, which are defined as global parameters for the serif component synthesizer (section 2.2.4.1), are visualized through a set of icons illustrating the meaning of each serif style in a dialogue box (Fig. 5.4). By selecting an icon, the corresponding serif style is applied to the preview characters and can be immediately visualized.

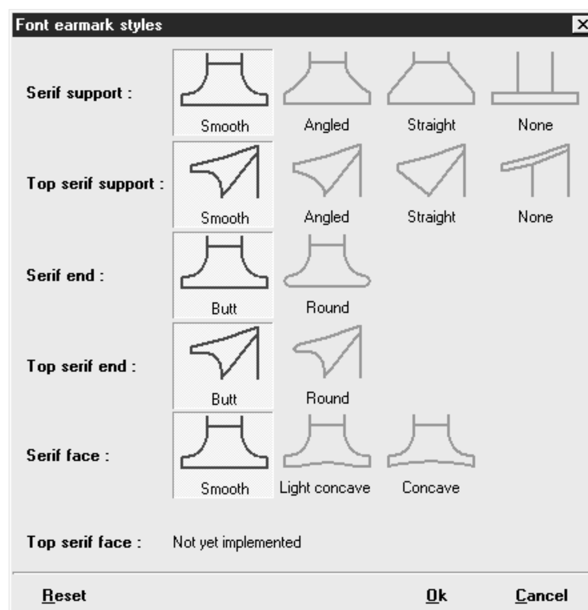


Figure 5.4: The serif style selection dialogue box.

Modification of other global parameters, such as stem widths, bar widths and serif dimensions can be done within the *global parameter editing window* (Fig. 5.5). The modification of each parameter is immediately visualized in the window's preview field.

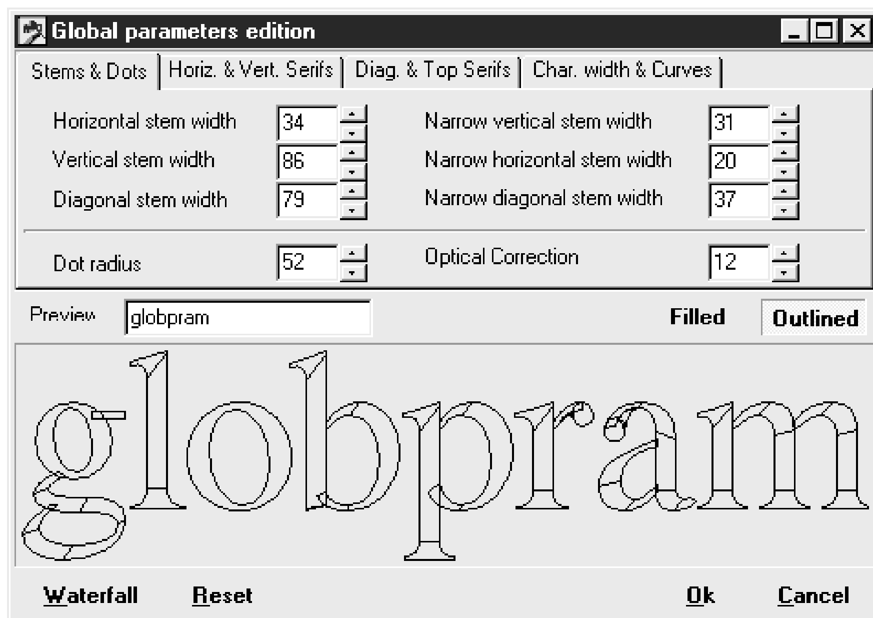


Figure 5.5: The general global parameter editing window.

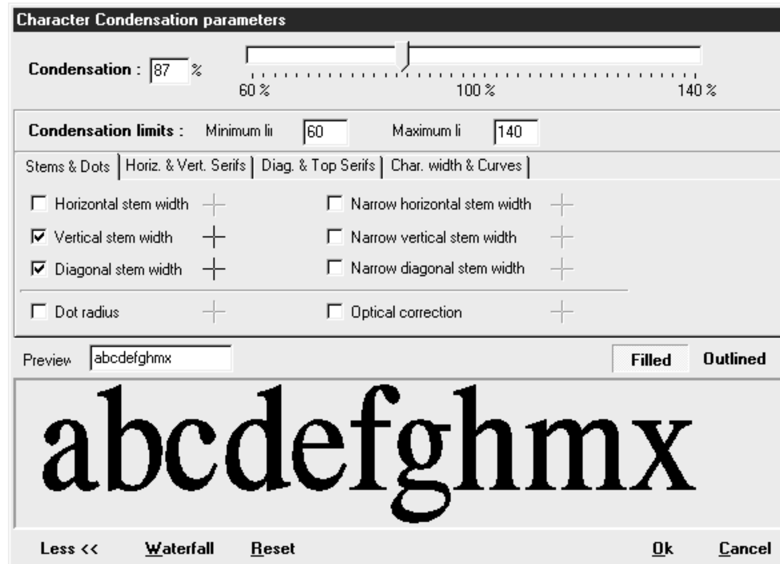


Figure 5.6: The parameter editing window for font condensation.

In order to achieve typographically pleasant typeface style changes, certain relationships between individual global parameter modifications should be maintained. For example, to increase the weight (or boldness) of a typeface by 20%, we increase the width of all vertical stems and diagonal bars by 20%, but only increase the width of horizontal stems by 15% instead of 20%. We have a special parameter editing tool to create parameter variation rules for typeface variation of character weight (boldness),

width (condensation), stress (obliqueness) and contrast. Fig. 5.6 shows the parameter editing window for doing font condensation. An interface is available for varying the font's weight, stress and contrast.

The amount of typeface style variation is visualized by a slide bar. The global parameter which is to be modified in response to this variation can be selected with check boxes. However, the amount of modification of each parameter may not be in proportion to the amount of the style variation, and may not be the same for different parameters. Therefore, parameter modifications are specified by a curve which is editable in the parameter rule editor window (Fig. 5.7). Each parameter has its own curve and therefore is edited in its own parameter rule editor window.

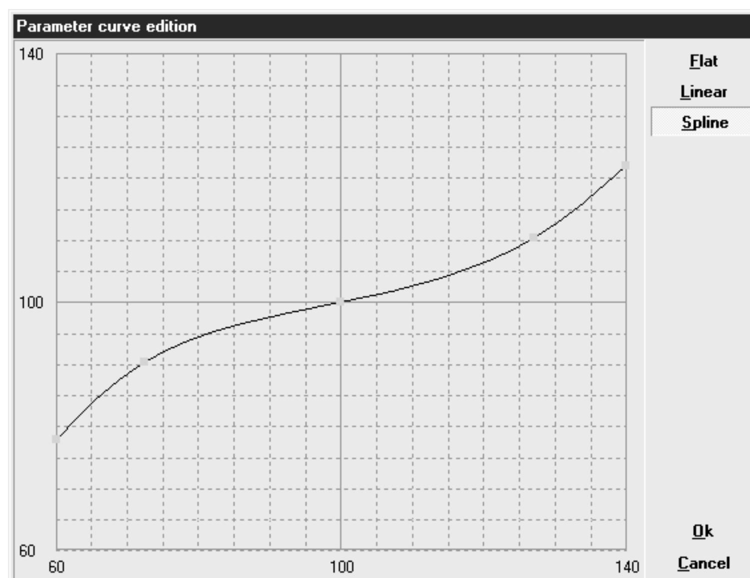


Figure 5.7: The parameter rule editing window.

5.2 Typographical experiments and applications

parametrizable fonts are developed because traditional outline font technology is not suitable for deriving character or font variations, such as variations of character weight (boldness) and non-linear condensation of characters. These variations require information not only in respect to character outlines but also to character structures. Our component based font parametrization method explicitly describes a character's structure and the shape of each of its parts. It is therefore suitable for deriving typographical font variations. It is also suitable for special typographic applications, such as optical scaling for high-quality printing.

5.2.1 Font variation

A set of fonts corresponding to one typeface but varying in styles such as boldness (weight) and character width is called a *font family*. Fonts in one family are distinguished by styles and by "high level" typographical features, such as boldness, contrast, stress, condensation and proportion. One may vary any of these features by modifying corresponding parameters. The parametrizable font will create a new font in the family. Starting with Times Roman, we have carried out several experiments trying to vary boldness, condensation, stress, contrast and other features. Similar experiments could have been carried out starting from other typefaces.

5.2.1.1 Boldness

Character boldness, or weight variation is obtained by changing the width of vertical stems, vertical curves, slanted bars, horizontal bars, horizontal curves, etc. Horizontal bar width and vertical stem width will be modified to different extents in response to the increasing character boldness, because bolder characters tend to have higher contrast between vertical and horizontal strokes. For example, if we want to make Times characters 20% bolder (or 120% of the normal weight), we will increase the characters' vertical stem width by 20% (in this sense, we measure the boldness by vertical stem width), and increase the horizontal bar width by an amount less than 20%, say, 10%. When the boldness is increased by more than 30%, we also decrease a little the serif width parameter, since bolder characters have less space for serifs. This rule is illustrated in Fig. 5.8.

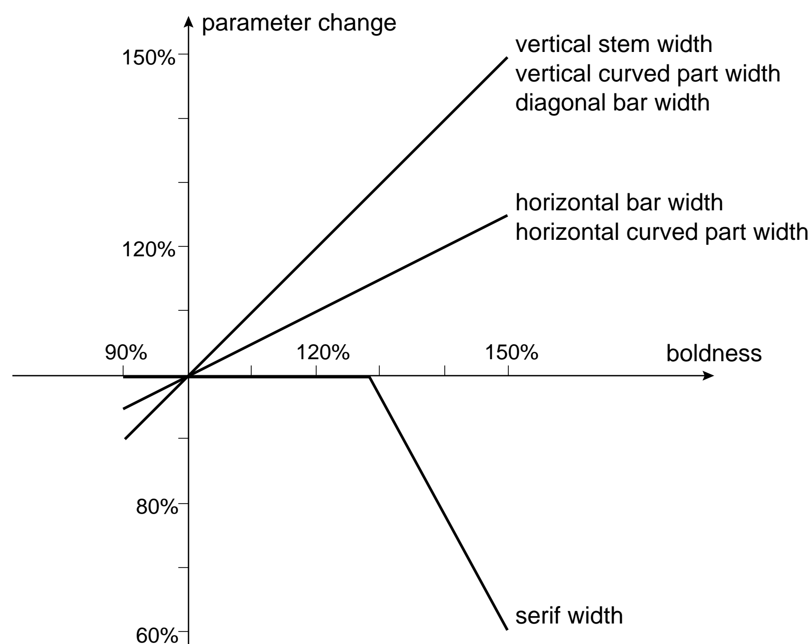


Figure 5.8: Parameter modification rules for varying the characters' boldness.

Using this rule, we derive from Times Roman new fonts with boldness changed to 90%, 110%, 120%, 130% and 150% (Fig. 5.9). The 150% boldness font imitates the Times Bold font. One can compare the character quality by looking at the enlarged characters in Fig. 5.10.

90% abcdefghijklmnopqrstuvwxyz
Times Roman abcdefghijklmnopqrstuvwxyz
110% abcdefghijklmnopqrstuvwxyz
120% abcdefghijklmnopqrstuvwxyz
130% abcdefghijklmnopqrstuvwxyz
150% abcdefghijklmnopqrstuvwxyz

Figure 5.9: Fonts derived from Times Roman (displayed in the second line) by varying the boldness factor.

Times Roman hamburgefonts
Times Bold boldness = 50% hamburgefonts
Times Bold components hamburgefonts

Figure 5.10: Times Roman and the derived Times Bold. The last line shows the synthesized Times Bold as trimmed components. See also Appendix D for the component descriptions of Times Roman.

5.2.1.2 Condensation

High-quality horizontally condensed fonts are needed where space is scarce, for example in telephone books. Furthermore, condensed fonts may offer increased flexibility for the

presentation of information at display resolution, for example in web browsers. High-quality horizontally condensed fonts are generated by reducing the character width without reducing in the same proportion the thickness of the strokes (stems, bars and curved parts). Since individual character width is controlled by a function of the round letter width, reducing the size of the global parameter *round letter width* ensures the width reduction of most of the characters of the font. Serif width of serifs for vertical stems and diagonal bars should also be reduced according to the amount of global character condensation (see Fig. 5.11 for our experimental rules).

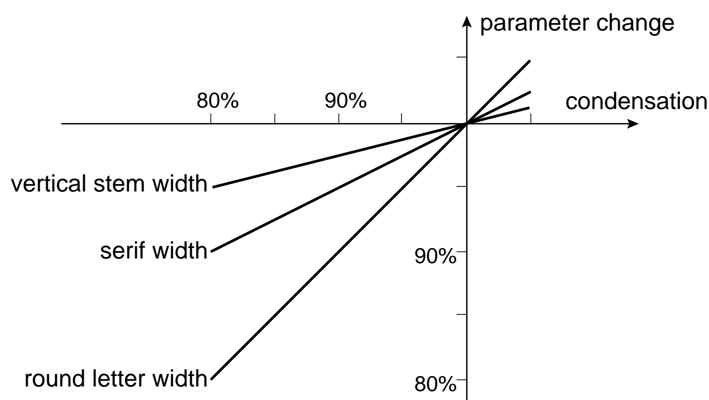


Figure 5.11: Rules of parameter modification for horizontal condensation.

Fig. 5.12 shows text condensed to 95%, 90%, 85% and 80%. Condensation up to 90% is barely perceptible and enables the generation of high-quality characters. Condensation to 80% already considerably distorts the original character shapes, hence it will be considered as a new typeface. Readers can compare the character quality by the enlarged characters in Fig. 5.13.

Times Roman abcdefghijklmnopqrstuvwxyz
95% abcdefghijklmnopqrstuvwxyz
90% abcdefghijklmnopqrstuvwxyz
85% abcdefghijklmnopqrstuvwxyz
80% abcdefghijklmnopqrstuvwxyz

Figure 5.12: Fonts derived from the normal Times typeface (displayed in the first line) by varying the character width.

Times Roman hamburgefonts

90% width hamburgefonts

80% width hamburgefonts

Figure 5.13: Condensation up to 90% is barely perceptible; condensation to 80% considerably distorts the original character shapes.

high stress bcdeghmnopqu

Times Roman bcdeghmnopqu

low stress bcdeghmnopqu

Figure 5.14: Fonts derived from Times Roman (displayed in the middle line) by varying the amount of oblique stress.

5.2.1.3 Stress

Characters with oblique stress derive from pen-based manuscript writing and were created soon after the invention of moveable metal type (for example typeface Jenson, created by a Frenchman Nicolar Jenson in 1470 in Venice, Italy). Design of typefaces evolved from oblique stress to vertical stress characters. Vertical stress characters became fashionable with the designs of the Bodoni and Didot typeface at the end of the 18th century. Increasing stress obliqueness requires increasing the obliqueness parameter η of the internal loops or half loops. Increasing stress obliqueness may also require some slight modification of the horizontal curved stroke width, and arch junction orientation (for letter h, n, m and u). In figure 5.14 we try to vary the obliqueness of Times Roman.

The derived font with reduced oblique stress looks nicer than the one with increased oblique stress.

5.2.1.4 Contrast

By *contrast* we mean the difference between horizontal strokes and vertical strokes and the transition from thick strokes to thin strokes. Old style typefaces have reasonable contrast while "modern" typefaces, such as Bodoni and Didot have very high contrast. The thin strokes and serifs can be as thin as hairlines, and the transition from thin strokes to thick strokes is abrupt. Increasing the contrast requires reducing the widths of horizontal strokes including horizontal bars and curved parts. Curvature of inner loops also need to be increased when the contrast is made higher.

Figure 5.15: Fonts derived from Times Roman (displayed in the middle line) by varying horizontal to vertical contrast.

5.2.1.5 Height proportion

Character height proportion mainly concerns the proportion of the lower-case letter *x-height* to the upper-case letter *cap-height*. The proportion reflects the size relationship of upper-case and lower-case letters. The bodies of lower-case letters rest between the x-height line and the base-line. The part that reaches above the *x-height line* is called ascender and the part that reaches below the *base line* is called descender. The *ascender line* and the *descender line* control the height relationship of the ascender, the descender and the character body of lower-case letters. The proportion of these height lines vary considerably and is a factor having an impact on text readability as well as on the overall darkness of a printed page.

Two variations of the Times Roman typeface are created by changing the vertical position of the x-height line. The ascender line and the descender line are adjusted in proportion (Fig. 5.16).

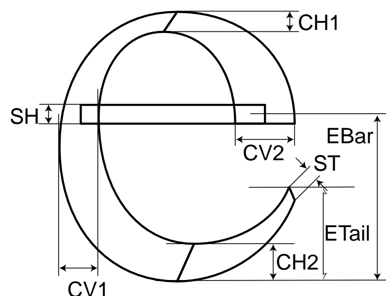
Figure 5.16: Fonts derived from Times Roman (displayed in the middle line) by varying the height proportion.

5.2.1.6 Variations in multiple dimensions

New fonts can also be derived by varying more than one dimension of typeface styles. Inspired by the designs of Gerrit Nordzij, a dutch type designer [Nordzji91], we tried to vary the few parameters determining the shape of character “e” so as to generate derived designs along three different dimensions.

Definition of interpolation:

iplt (a, b, percentage) =
a * (1 - percentage) + b * percentage.



1. CH1 = iplt [iplt [normalCH1, maxCH1, boldness], minCH1, contrast]
2. CH2 = iplt [iplt [iplt [normalCH2, maxCH2, boldness], minCH2, contrast], iplt [normalCH2, maxCH2, boldness], obliqueStress * 0.6]
3. CV1 = iplt [minCV1, maxCV1, boldness]
4. CV2 = iplt [minCV2, maxCV2, boldness]
5. SH = iplt [iplt [normalSH, maxSH, boldness], minSH, contrast]
6. ST = iplt [iplt [minCV2, maxST, boldness], minST, contrast]
7. EBar = iplt [maxEBar, minEBar, boldness]
8. ETail = iplt [iplt [minETail, maxETail, boldness], minETail, contrast]
9. ηext = minη = 0; ηint = iplt [minη, maxη, obliqueStress]
10. βext = normalβ;
    βint = iplt [iplt [normalβ, maxβ, (contrast + boldness) / 2], normalβ, obliqueStress]

explanations:
1: Boldness increases CH1 value, contrast decreases CH1 value.
2: Boldness increases CH2 value, contrast decreases CH2 value, stress obliqueness increases partly CH2 value.
3, 4: Boldness increases CV1 and CV2 values.
5, 6: Boldness increases contrast and decreases SH and ST values.
7: Boldness decreases EBar values.
8: Boldness increases contrast and decreases ETail value.
9: Stress obliqueness increases obliqueness η of interior loops.
10: Contrast and boldness increase obliqueness, reduces squareness β.

Figure 5.17: Parameters determining the shape of character “e”.

The first dimension is the *weight* of the character (boldness). All horizontal and vertical width parameters are equally (which is a bit different from the rules in Fig. 5.8) influenced by the boldness parameter. The second dimension is *contrast*. Increasing the

contrast requires reducing the horizontal bar and curve width parameters. The third dimension is *stress obliqueness*. Increasing stress obliqueness requires increasing the obliqueness parameter η of the internal half-loop and slightly increasing the horizontal curve width parameter determining the stroke width at the bottom of the character. The parameters of the character “e” and the interpolating algorithm for determining the intermediate values are given in Fig. 5.17.

In each dimension, weight, contrast and oblique stress, we consider 5 evenly spaced values to generate 125 different styles for character “e”. These characters are placed in a cube representing the three typeface style dimensions (see Appendix F).

5.2.2 Optical scaling

Previous generation phototypesetters produced high-quality characters by reproducing them photographically from carefully made film masters. Only one master was usually used to create all of the sizes of a particular style [Rubinstein88, pp.40-41]. Similarly digital font technology used in digital laser photocomposing systems uses often only one or two master fonts for producing all sizes. However, in traditional metal type, the shape of letters in the same typeface varies systematically as the size changes [Rubinstein88, pp.40]. Smaller letters must be bolder and fatter to retain their legibility [André94], [Haralambous93], [Johnson87].

When a font is *optically scaled*, the shape of characters varies as in traditional metal type design. The master design for a digital font is usually optimized for the purpose of printing running text, whose size is around 12 or 10 point. In our optical scaling experiment, we assume that a parametrizable font is best printed or displayed at 12 point. To print or display characters of the font at a size smaller than 12 point, parameters of the font should be corrected so as to make the synthesized characters relatively fatter and larger. Not only the characters’ stroke width is enlarged, but also their x-height and their character width are made slightly larger than corresponding values obtained by plain scaling. Some previous researchers also pointed out that when scaled to a larger size, character stroke width should not be scaled to the same extend, i.e. the character stroke width should be thinner than the plain scaled result.

It has been shown that a parabolic correcting function might be suitable for determining each optically scaled character parameter [Haralambous93] [Johnson87]. In our optical scaling model, optical scaling correction values given as fractions of the capital letter height are used for generating characters at sizes between 12 point and 5 point. Maximal correction values (*maxCoor*) for the various parameters (character width, stem width, bar width, curved element width) are experimentally determined for the 5 point character size. Between 5 point and 12 point, these correction values (*corrValue*) are interpolated by the parabola giving the maximal correction at 5 point and a zero correction value at 12 point (equation 5.1). The optical scaling correcting function is shown in Fig. 5.18.

$$\text{corrValue(ptSize)} = \frac{\text{maxCorr}}{(5\text{pt} - 12\text{pt})^2}(\text{ptSize} - 12\text{pt})^2; \qquad 5\text{pt} \leq \text{ptSize} \leq 12\text{pt} \tag{5.1}$$

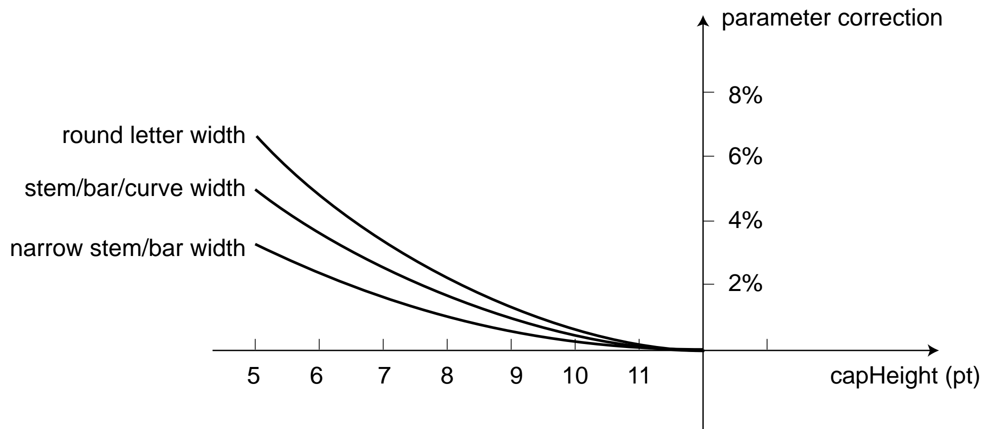


Figure 5.18: The parabolic correcting functions for optical scaling between 5pt and 12pt.

To determine the coefficients of *maxCorr* for different parameters, the printing resolution and the properties of the printing media (papers and films) need to be taken into account. The specimen in Fig. 5.19a has been generated by photo-composition on a 3000dpi laser photocomposer. The maximal correction value for the round letter width and for the stem to stem and stem to curved element part spacing is 1/15 of the capital letter height. The stem, bar and curved element width have a maximal correction of 1/20 capital letter height. The narrow diagonal bar, the narrow horizontal bar and the narrow curved element width have a maximal correction value of 1/30 capital letter height (Fig. 5.18). Fig. 5.19a shows a comparison of characters printed with and without optical scaling. Clearly, optical scaling improves legibility. Fig. 5.19b shows the corresponding optically scaled character shapes magnified to the same size.

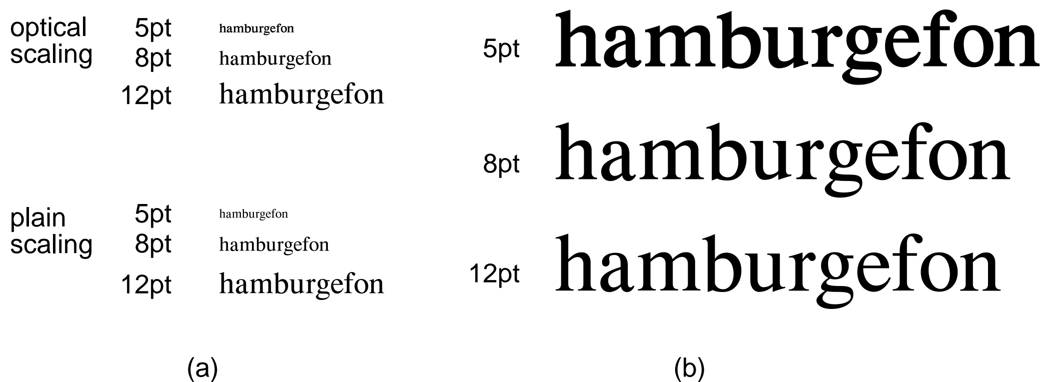


Figure 5.19: Comparison between optical scaling and plain scaling.

5.2.3 Parametrization of existing fonts

By making traditional outline fonts parametrizable, we are able to apply to them typeface variations and optical scaling. Because many fonts can be derived from one parametrizable font by varying weight, width and other typographic features, parametrization of existing outline fonts may also save large amounts of storage space.

Appendix E gives enlarged character examples of our parametrizable fonts for the Times Roman, Helvetica and Bodoni typefaces. The master fonts for each typeface were taken from several outline fonts created and digitized by different companies (mainly from URW and Adobe Type 1) and also from published typeface category books such as [Berthold88], [Bauermeister87] and [Rockledge91].

To parametrize an existing font, the first step is to select serif styles, junction types and terminal refined types (see section 3.1.1 and section 3.1.2). Serif styles are global parameters and are specified in the global parameter files. In Times Roman, serifs are bracketed with smooth serif supports, flat serif faces and butt serif ends. The Bodoni typeface has slab serifs which have thin-line serif slabs (flat faces, butt ends) but no serif support. The Helvetica typeface is sans-serif (a French word meaning “without serif”). In Bodoni and Helvetica typefaces, most of the arches and bowls connect to a vertical stem smoothly, and hence should be refined and specified as a smooth junction in the global, group or local parameter files.

The second step is to measure parameters which give dimension or orientation. Parameters of the produced experimental parametrizable fonts have been carefully measured, extracted and tuned by hand working on masters printed on papers. To accelerate the speed of parameter extraction, we have also proposed an automatic parameter extracting method based on a “window matching method” [Herz98]. The window matching method is used to help locating specific typographic features on outline characters in order to measure them.

5.3 Summary

parametrizable fonts can be used in many application areas. In this chapter we have presented several typographical experiments which were focused on coherent typeface variations such as variations of boldness, condensation, oblique stress, contrast and x-height proportion. All of these variations are beyond the capabilities of traditional outline font technology, since these variations require besides the outline character shape description additional information about the character structure.

Some of the experiments may have an important application potential. The boldness variation can be used to control the overall darkness of a printed page, which is useful in applications such as fine printing. Character condensation is useful for putting more letters in a limited space, for example for more flexible paragraph formatting. The

optical scaling experiment was able to produce designs similar to the traditional metal type where character shapes are adjusted at different sizes.

CHAPTER 6

Conclusion and future work

This dissertation presents the study and the development of a component-based parametrizable character synthesis method. Following the introduction in chapter 1, we describe in detail in chapter 2 our design of components, i.e. parametrizable shape primitives used for synthesizing structure elements of typographic characters. In chapter 3, we discuss the parametrizable character synthesis system which describes characters by predefined parametrizable components and shape trimming operations. In chapter 4, our implementation of a C++ prototype for this font parametrization system is presented. In chapter 5, we carry out typographical experiments demonstrating the capabilities of our component based parametrizable font synthesizing system.

Let us summarize the main contributions and achievements of this thesis.

Our attempt to reproduce existing text typefaces by parametrizable shape primitives has been successful. Curved parts of characters are difficult to synthesize; they are of primary importance to obtain high-quality characters. Based on a systematic study of the curves suitable for parametrization of round parts of characters, we have designed the loop, half-loop and sweep components. These components were adequate for synthesizing the shapes of round parts and of curved connections in typographic characters. Terminals of strokes differ across different text typefaces. We describe terminals of straight strokes (such as vertical stems, horizontal and diagonal bars) by the serif component which enables us to synthesize most serif types. Terminals of curved strokes are synthesized by the combination of the dot component, the sweep component and automatically generated smoothing shapes.

Using our parametrizable font synthesizing system, we have successfully created high-quality derived fonts. Relationship between character structure elements were maintained. We ensure that no components fall apart when we vary font properties such as boldness, condensation, contrast and oblique stress. The hierarchical parameter file organization and the PostScript-like parameter expressions provide the means to define parameter modifying rules for each parameter in order to achieve coherent font style variations. We are also able to make coherent typographical feature modifications across a whole font or across a set of characters by modifying respectively the global parameters or the group parameters.

Typographical experiments demonstrate the application potential of parametrizable component-based fonts. Font style variations and optical scaling can be applied to improve text printing quality. With our font parametrization system, one can create personalized font by coherently modifying some parameters. Component-based

parametrized fonts require much less storage space than outline fonts. With a few parameter changes, derived fonts can be instantly created. A restricted set of parameters is sufficient to generate all characters needed within a document. Component-based parametrized fonts can therefore be delivered together with documents over electronic networks (Internet).

The research started with this thesis can be continued in the future, at least in the following directions:

The use of local parameters is cumbersome at this stage. In a future research project, local parameters may be regularized and simplified. Visual tools may be developed to help editing local parameters.

The idea of describing characters by predefined parametrizable shape components is quite suitable for stroke-based characters, for example Chinese characters. The attempt to describing Chinese characters by “basic strokes” has been tried by some typographers and computer scientists. However, the quality of curved strokes and the junctions between strokes remain problematic. The author believes that some results of this dissertation, such as the β-Bézier curve controlling method, the sweep component and the smoothing operation, can be extended to help solving problems for Chinese character parametrization.

References

[Adams89] D. Adams, abcdefg - A Better Constraint Driven Environment for Font Generation, *Raster Imaging and Digital Typography (RIDT'89)*, Eds. J. André and R. D. Hersch, Cambridge University Press, 1989, 54-70.

[Adobe85] Adobe Systems Inc., *PostScript Language Reference Manual*, Addison-Wesley, 1985.

[Adobe90] Adobe Systems Inc., *The Type 1 Format Specification*, Addison-Wesley, 1990.

[Adobe92] J. Seybold, Adobe's MultiMasters Technology: Breakthrough in Type Aesthetics, *Seybold Report on Desktop Publishing*, Vol. 7, No. 5, 1991, 3-7, see also "Designing Multiple Master Typefaces", Adobe Systems Inc., http://www.adobe.com.

[Agfa91] Agfa Corp., *Intellifont Scalable Typeface Format*, 1991.

[André94] Jacques André, Irene Vatton, Dynamic Optical Scaling and Variable-Sized Characters, *Electronic Publishing - Origination, Dissemination and Design*, Vol. 7, No. 4, 1994, 231–250.

[Bauermeister87] Benjamin Bauermeister, *A Manual of Comparative Typography (the PANOSE system)*, Van Nostrand Reinhold, NY, 1987.

[Bauermeister96] US Patent 5'586'241, Method and System for Creating, Specifying and Generating Parametric Fonts, issued Dec. 17, 1996, inventors: B. B. Bauermeister, C. D. MacQueen, M. S. DeLaurentis, P. M. Higinbotham, D. E. Lipkie, D. J. Munsil, R. G. Beausoleil.

[Berthold88] H.Berthold AG, *Berthold Types*, H. Berthold AG, Berlin, Germany, 1988.

[Betrisey89] Claude Betrisey and Roger D. Hersch, Flexible Application of Outline Grid Constraints, *Raster Imaging and Digital Typography (RIDT'89)*, Eds. J. André and R. D. Hersch, Cambridge University Press, 1989, 242-250.

[Billawala89] N. Billawala, Panadora-an Experience with Metafont, *Raster Imaging and Digital Typography (RIDT'89),* Eds. J. André and R. D. Hersch, Cambridge University Press, 1989, 34-53.

[Coueignoux75] Ph. Coueignoux, Generation of Roman Printed Fonts, Ph.D Thesis (adviser: Prof. Schreiber), MIT, June 1975.

[Coueignoux81] Ph. Coueignoux, Character Generation by Computer, *Computer Graphics and Image Processing*, Vol. 16, 1981, 240–269.

[Cox82] Charles H. Cox III, Philippe Coueignoux, Barry Blesser and Murray Eden, Skeletons: A Link Between Theoretical and Physical Letter Descriptions, *Pattern Recognition*, Vol. 15, No. 1, Pergamon Press Ltd., Great Britain, 1982, 11-22.

[Dong91] Yunmei Dong & Kaide Li, A Parametric Graphics Approach to Chinese Font Design, *Raster Imaging and Digital Typography II (RIDT'91)*, Eds. R. Morris and J. André, Cambridge University Press, 1991, 156-165.

[Dürst93] Martin J. Dürst, Coordinate-Indenpendent Font Description Using Kanji as an Example, *Electronic Publishing - Origination, Dissemination and Design (RIDT'94)*, Vol. 6, No. 3, 1993, 133-143.

[Fan91] Jianping Fan, Towards Intelligent Chinese Character Design, *Raster Imaging and Digital Typography II (RIDT'91)*, Eds. R. Morris and J. André, Cambridge University Press, 1991, 166-176.

[Farin90] Gerald Farin, *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide (2nd edition)*, Academic Press, 1990.

[Foley90] James D. Foley et al, *Computer Graphics: Principles and Practice, Second Edition*, Addison-Wesley, 1990.

[Gaskell76] P. Gaskell, A Nomenclature for the Letterforms of Roman Type, *Visible Language*, Vol. 10, No. 1, 1976, 41–51.

[Gonczarowski93] Jakob Gonczarowski, Curve Technique for Auto-Tracing, *Visual and Technical Aspects of Type*, Ed. R. D. Hersch, Cambridge Unicersity Press, 1993, 126-147.

[Gonczarowski98] Jakob Gonczarowski, Producing the Skeleton of a Character, *Electronic Publishing, Artistic Imaging, and Digital Typography (EP'98/ RIDT'98)*, Eds. R. D. Hersch et al., LNCS 1375, Springer-Verlag, 1998, 66-76.

[Haralambous93] Yannis Haralambous, Parametrization of PostScript Fonts Through METAFONT - An Alternative to Adobe Multiple Master Fonts, *Electronic Publishing - Origination, Dissemination and Design (RIDT'94)*, Vol. 6, No. 3, 1993, 145–157.

[Haralambous94] Yannis Haralambous, Typesetting Khmer, *Electronic Publishing - Origination, Dissemination and Design*, Vol. 7, No. 4, J. Wiley, 1994, 197–215.

[Hersch88] Roger D. Hersch, Vertical Scan-Conversion for Filling Purpose, *Proceedings CGI'88*, Geneva, Ed. D. Thalmann, Springer Verlag, 1988.

[Hersch91] Roger D. Hersch, Claude Bétrisey, Model-based Matching and Hinting of Fonts, *Proceedings Siggraph'91, ACM Computer graphics*, Vol. 25, No. 4, 1991, 71–80.

[Hersch93] Roger D. Hersch, Font Rasterization: the State of the Art, *Visual and Technical Aspects of Type*, Ed. R. D. Hersch, Cambridge University Press, 1993, 78-109.

[Hersch95] Roger D. Hersch, Claude Bétrisey, Justin Bur, and Andre Gurtler, Perceptually Tuned Generation of Grayscale Fonts, *IEEE Computer Graphics and Applications*, Vol. 15, No. 6, November 1995, 78-89.

[Herz94a] Jacky Herz and Roger D. Hersch, Towards a Universal Auto-Hinting System for Typographic Shapes, *Electronic Publishing - Origination, Dissemination and Design*, Vol. 7, No. 4, J. Wiley, 1994, 251-260.

[Herz94b] Jacky Herz and Roger D. Hersch, Analyzing Character Shapes by String Matching Techniques, *Electronic Publishing - Origination, Dissemination and Design (RIDT'94)*, Vol. 6, No. 3, J.Wiley, 1994, 261-272.

[Herz97] Jacky Herz, Coherent Processing of Typographic Shapes, Ph.D thesis N° 1676, École Polytechnique Fédérale de Lausanne, 1997.

[Herz98] Jacky Hertz, Changyuan Hu, Jakob Gonczarowski and Roger D. Hersch, A Window-Based Method for Automatic Typographic Parameter Extraction, *Electronic Publishing, Artistic Imaging, and Digital Typography (EP'98/RIDT'98)*, Eds. R. D. Hersch et al, LNCS 1375, Springer-Verlag 1998, 44-54.

[Hobby89] J. D. Hobby, Rasterizing Curves of Constant Width, *Journal of the ACM*, Vol. 36, No. 2, April 1989, 209–229.

[Hofstadter85] D. R. Hofstadter, Metafont-Metamathematics and Metaphysics, Metamagical Themas: Questing for the Essence of Mind and Pattern, Bantam Books, NY, 1985.

[Hu91] Changyuan Hu and Fuyan Zhang, Automatic Hinting of Chinese Outline Fonts Based on Stroke Separating Method, *Proc. the 1st Pacific*

Conference on Computer Graphics and Applications (Pacific Graphics'93), Seoul, Korea, World Scientific, 1993, 359-368.

[Jamra93] Mark Jamra, Some Elements of Proportion and Optical Image Support in a Typeface, *Visual and Technical Aspects of Type*, Ed. R. D. Hersch, Cambridge, 1993, 47-55.

[Johnson87] Bridget Lynn Johnson, A Model for Automatic Optical Scaling of Type Designs for Conventional and Digital Technology (MSc. thesis), School of Printing, Rochester Institute of Technology, 1987.

[Karow89] Peter Karow, Automatic Hinting for Intelligent Font Scaling, *Raster Imaging and Digital Typography (RIDT'89)*, Eds. J. André and R. D. Hersch, Cambridge University Press, 1989, 232-241.

[Karow92] Peter Karow, *Schriftstatistik*, URW Verlag, Hamburg, 1992.

[Karow94] Peter Karow, *Font Technology: Description and Tools*, Springer Verlag, 1994.

[Knuth86a] Donald E. Knuth, *The METAFONT book*, Addison Wesley, 1986.

[Knuth86b] Donald E. Knuth, *Computer Modern Typefaces* (Volume E of Computers and Typesetting), Addison-Wesley, 1986.

[Labuz88] Ronald Labuz, *Typography and Typesetting*, Van Nostrand Reinhold, 1988.

[Lancaster86] P. Lancaster, K. Salkauskas, *Curve and Surface Fitting*, Academic Press, 1986.

[MacQueen93] C. D. McQueen, R. G. Beausoleil, Infinifont: A Parametric Font Generation System, *Electronic Publishing - Origination, Dissemination and Design (RIDT'94)*, Vol 6, No. 3, 1993, 117-132.

[Morris98] R. A. Morris, R. D. Hersch, A. Coimbra, Legilility of Condensed Perceptually-Tuned Grayscale Fonts, *Electronic Publishing, Artistic Imaging and Digital Typography (EP'98/RIDT'98),* Eds. R. D. Hersch, J. André, H. Brown, LNCS 1375, Springer-Verlag, 1998, 281-193.

[Microsoft95a] Microsoft Corp., *TrueType Open Font Specification, Version 1.0*, July 1995.

[Microsoft95b] Microsoft Corp., *TrueType 1.0 Font Files, Technical Specification*, Revision 1.66, November 1995.

[Nordzji91] G. Nordzij, The Shape of the Stroke, *Raster Imaging and Digital Typography II*, Eds. R. Morris and J. André, Cambridge University Press, 1991, 34-42.

[Rockledge91] G. Rockledge and C. Perfect, *Rockledge's International Type finder: The Essential Handbook of Typeface Recognition and Selection*, Moyer Bell Ltd, distributed by Rizzoli International Publications, 1991.

[Rubinstein88] Richard Rubinstein, *Digital Typography*, Addison-Wesley, 1988.

[Shamir96] Ariel Shamir and Ari Rappoport, Extraction of Typographic Elements from Outline Representations of Fonts, *Proc. EUROGRAPHICS'96*, (Eds. J. Rossignac and F. Sillion), *Computer Graphics Forum*, Vol. 15, No. 3, 1996, 259-268.

[Shamir98] Ariel Shamir and Ari Rappoport, Feature-Based Design of Fonts Using Constraints, *Electronic Publishing, Artistic Imaging and Digital Typography (EP'98/RIDT'98)*, (Eds. R. D. Hersch, J. André, H. Brown), St. Malo, France, LNCS 1875, Springer-Verlag, 1998, 93-108.

[Schneider98] U. Schneider, An Object-Oriented Model for the Hierarchical Composition of Letterforms in Computer-Adided Typeface Design, *Electronic Publishing, Artistic Imaging and Digital Typography (EP'98/ RIDT'98)*, Eds. R. D. Hersch, J. André, H. Brown, St. Malo, France, LNCS 1875, Springer-Verlag, 1998, 108-125.

[Southall91] Richard Southall, Character Description Techniques in Type Manufacture, *Raster Imaging and Digital Typography II (RIDT'91)*, Eds. R. Morris and J. André, Cambridge University Press, 1991, 16–27.

[Southall98] Richard Southall, Metafont in Rockies: the Colorado Typemaking Project, *Electronic Publishing, Artistic Imaging and Digital Typography (EP'98/RIDT'98)*, Eds. R. D. Hersch, J. André, H. Brown, St. Malo, France, LNCS 1875, Springer-Verlag, 1998, 167-180.

[Stamm94a] Beat Stamm, Object-Orientation and Extensibility in a Font-Scaler, *Electronic Publishing - Origination, Dissemination and Design (RIDT'94)*, Vol. 6, No. 3, 1993, 159-170.

[Stamm94b] Beat Stamm, Dynamic Regularisation of Intelligent Outline Fonts, *Electronic Publishing - Origination, Dissemination and Design (RIDT'94)*, Vol. 6, No. 3, 1993, 219-230.

[Stamm98] Beat Stamm, Visual TrueType: A Graphical Method for Authoring Font Intelligence, *Electronic Publishing, Artistic Imaging and Digital*

Typography (Proc. EP-RIDT'98), Eds. R. D. Hersch, J. André, H. Brown, St. Malo, France, LNCS 1875, Springer-Verlag, 1998, 77-92.

[Tschichold65] J. Tschichold, *Meisterbuch der Schrift*, Otto Maier Verlag, Ravensburg, Germany, 1965.

[Vatti92] Bala R. Vatti, A Generic Solution to Polygon Clipping, *Communications of the ACM*, Vol. 35, No. 7, 1992.

[Zalik95] Borut Zalik, Font Design with Incompletely Constrained Font Features, *Proc. 3rd Pacific Conference on Computer Graphics and Application (Pacific Graphics'95)*, Eds. S. Y. Shin, T. L. Kunii, ISBN 981-02-2337-4, World Scientific, 1995, 512-526.

APPENDIX A The β value for a quarter of an arc

Referring to Fig. 2.14, if we require the point at parameter t = 1/2 to have identical coordinates as the corresponding center point of a quarter of a circle, we get equation 2.9 which can be rewritten as (see also equation 2.1)

$$\left(1-\frac{1}{2}\right)^3\begin{bmatrix}0\\1\end{bmatrix}+\frac{3}{2}\left(1-\frac{1}{2}\right)^2\begin{bmatrix}0\\1-\beta\end{bmatrix}+3\left(\frac{1}{2}\right)^2\left(1-\frac{1}{2}\right)\begin{bmatrix}1-\beta\\0\end{bmatrix}+\left(\frac{1}{2}\right)^3\begin{bmatrix}1\\0\end{bmatrix}=\begin{bmatrix}1-\frac{\sqrt{2}}{2}\\1-\frac{\sqrt{2}}{2}\end{bmatrix}$$

Solving this equation, we have,

$$\beta = \frac{4(\sqrt{2}-1)}{3} \approx 0.552285$$

If we require curvature radii at B(0) and B(1) to be the radius of the corresponding circle with radius 1, we obtain the equation

$$R(0) = R(1) = 1$$

From equations 2.2, 2.4 and 2.5, we can obtain the curvature at both ends.

$$R(0) = R(1) = \frac{3\beta^2}{2(1-\beta)}$$

Therefore, the equation we need to solve is

$$\frac{3\beta^2}{2(1-\beta)} = 1$$

Solving this equation with respect to the condition $\beta > 0$ yields

$$\beta = \frac{-1+\sqrt{7}}{3} \approx 0.548584$$

Though these two results are slightly different, they both prove that the β value around 0.55 is the best for approximating a quarter of an arc with one β-Bézier curve.

The difference between these two β-Bézier curves is very small and is not noticeable by the human eye. This difference can be seen by plotting out their curvature radii (Fig. A.1). This plot also shows that theoretically an arc cannot be precisely approximated by a piece of Bézier curve. However, the shape of either of the two β-Bézier curves is very similar to the quadrant of arc which it approximates (Fig. A.2).

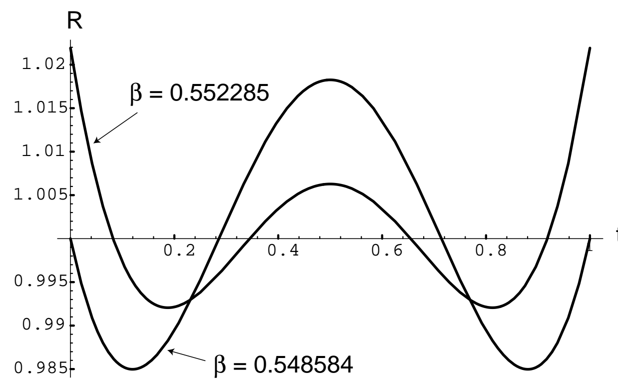


Figure A.1: Curvature radii of two Bézier curves which approximate a quadrant of a circle.

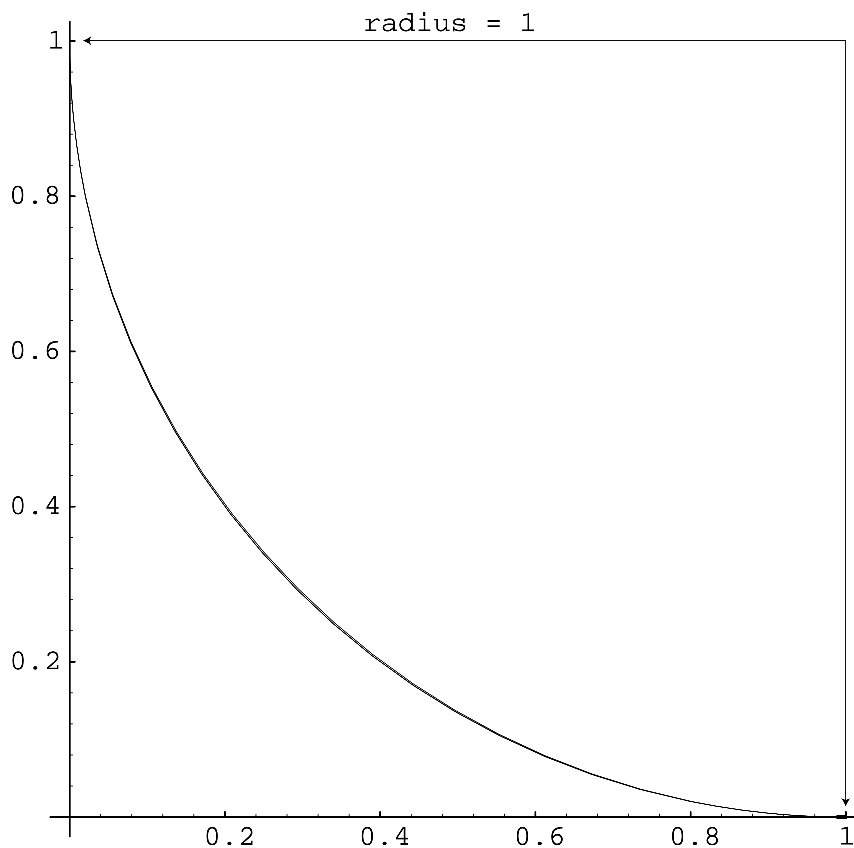
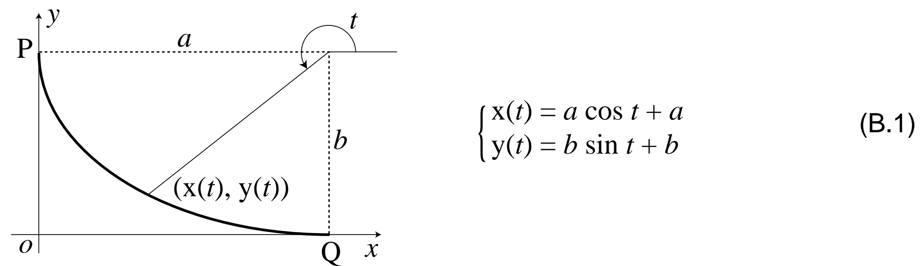


Figure A.2: The two Bézier curves and the quadrant circular arc they approximate are placed at a same place within a coordinate system. Curves are drawn with very thin lines, however the difference between the three curves are not easily noticeable by the human eye.

APPENDIX B The β value for approximating an ellipse

We show that the β value which creates ideal curvatures at the two ends of a β-Bézier curve which approximates a quarter of a circular arc also creates ideal curvatures when it approximates a quarter of an ellipse.

Without loss of generality, let us study the third quadrant of an ellipse whose long axis is *a*, short axis is *b* and center located at point $[a, b]^T$. This ellipse can be represented in parametric form in Fig. B.1.



$$\begin{cases} x(t) = a\cos t + a \\ y(t) = b\sin t + b \end{cases} \quad \text{(B.1)}$$

Figure B.1: Parametric representation of a quadrant of an ellipse.

Let P and Q represent the two end points of the curve, then $P = [x(\pi), y(\pi)]^T$, $Q = [x(3\pi/2), y(3\pi/2)]^T$. From equation 2.2 and equation B.1, we can derive the formula for the curvature radius of an ellipse

$$R(t) = \frac{(a^2(\sin t)^2 + b^2(\cos t)^2)^{\frac{3}{2}}}{ab}$$

Therefore,.

$$R_P = R(\pi) = b^2/a, \quad R_Q = R(3\pi/2) = a^2/b. \quad \text{(B.2)}$$

We try to approximate the quarter of an ellipse with a piece of β-Bézier curve. Let $B_0 = P = [0, b]^T$, $B_1 = [0, b(1 - \beta)]^T$, $B_2 = [a(1 - \beta), 0]^T$, and $B_3 = Q = [a, 0]^T$. In order to apply the equation 2.2, we calculate the first and second derivatives of B_0 and B_3 in a Bézier curve defined by equation 2.1 respectively as following.

$$\begin{aligned} B'(0) &= -3B0 + 3B1 = [0, -3b\beta]^T, \\ B''(0) &= 6B_0 - 12B_1 + 6B_2 = [6a - 6a\beta, -6b + 12b\beta]^T, \\ B'(1) &= -3B_2 + 3B_3 = [3a\beta, 0]^T, \\ B''(1) &= 6B_1 - 12B_2 + 6B_3 = [-6a + 12a\beta, 6b - 6b\beta]^T. \end{aligned} \quad \text{(B.3)}$$

With equations B.3 and 2.2, one obtain the following results.

$$R(0) = \frac{3\beta^2}{2(1-\beta)} \cdot \frac{b^2}{a}$$
$$R(1) = \frac{3\beta^2}{2(1-\beta)} \cdot \frac{a^2}{b} \qquad \text{(B.4)}$$

Comparing equation B.4 and equation B.2, if we require the curvatures at the two end points of the β-Bézier curve to be identical to the corresponding curvatures of an ellipse, we have

$$\frac{3\beta^2}{2(1-\beta)} = 1$$

This is exactly the condition for a β value which creates ideal curvatures at the two ends of a β-Bézier curve which approximates a quarter of a circular arc (see also Appendix A). Solving this equation with respect to the condition $\beta > 0$ yields

$$\beta = \frac{-1+\sqrt{7}}{3} \approx 0.548584$$

APPENDIX C Proof of parameter η

Given: Ellipse with center O and two tangents h and v, with respective tangential points H and V. Consider parallelogram UAOB constructed by the tangents h and v and by their parallels through O.

HB = Δp; UB=p; VA = Δq; UA= q

To be shown: $\frac{HB}{UB} = \frac{VA}{UA} = \frac{\Delta p}{p} = \frac{\Delta q}{q} = \eta$

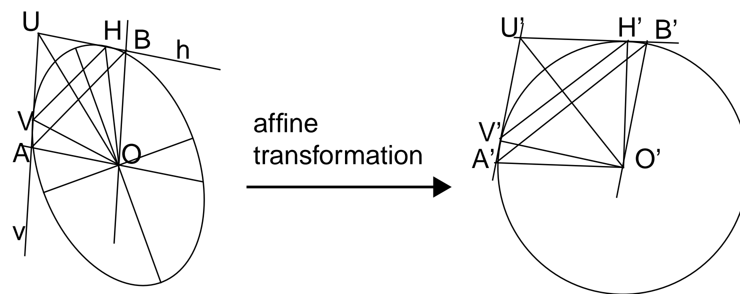


Figure C.1: Affined transformation.

Proof:

If AB is parallel to VH then $\frac{HB}{UB} = \frac{VA}{UA}$.

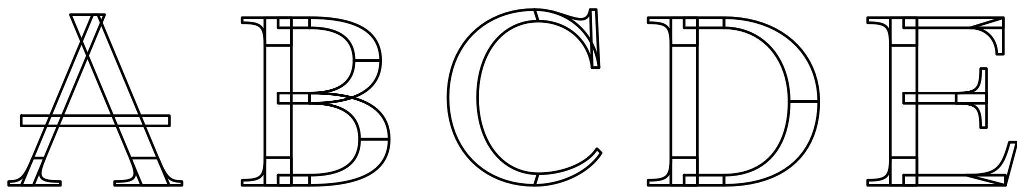
Consider an affined transformation which maps the ellipse to a circle (Fig. C.1):

ellipse -> circle; A->A'; V->V'; B->B', H->H', U->U'

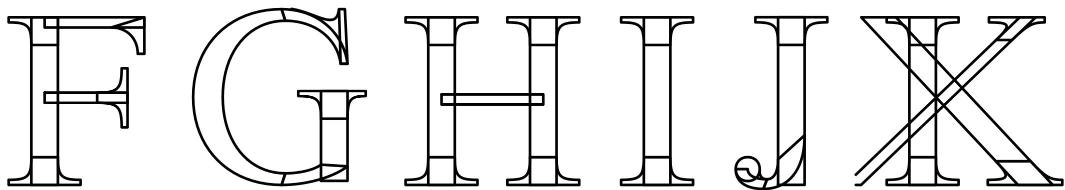
The line pairs (OA, OH) and (OB, OV) are support lines of pairs of conjugate diameters of the ellipse. Corresponding support lines (O'A', O'H') and (O'B', O'V') of the circle are perpendicular. Tangential points V' and H' are symmetric to line O'U'. Since angle V'O'B' is equal to angle A'O'H' (90 degrees), points A' and B' are symmetric in respect to line O'U'. Circle line A'B' is therefore parallel to line V'H'. Since affine transforms map parallel lines into parallel lines, corresponding ellipse lines AB and VH are parallel. □

APPENDIX D Description of characters by components

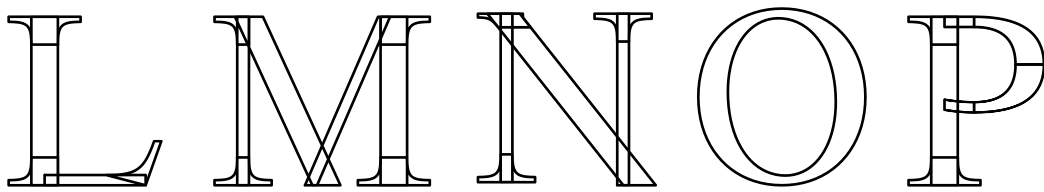
Times Roman, lower-case and upper-case:



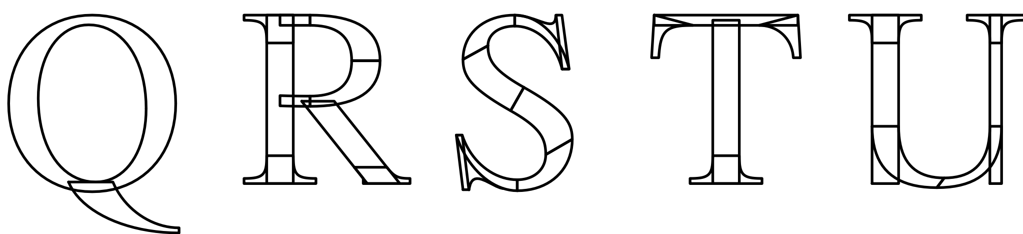
A B C D E



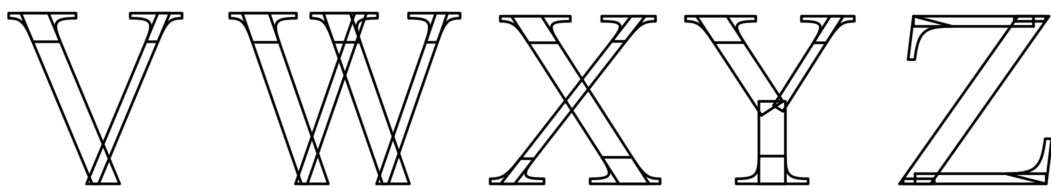
F G H I J K



L M N O P



Q R S T U



V W X Y Z

APPENDIX E Enlarged resynthesized component-based characters

Resynthesized Times Roman lower-case characters:

abcde
fghijk
lmnop
qrstuv
wxyz

Resynthesized Times Roman capitals:

ABCDE
FGHIJK
LMNOP
QRSTU
VWXYZ

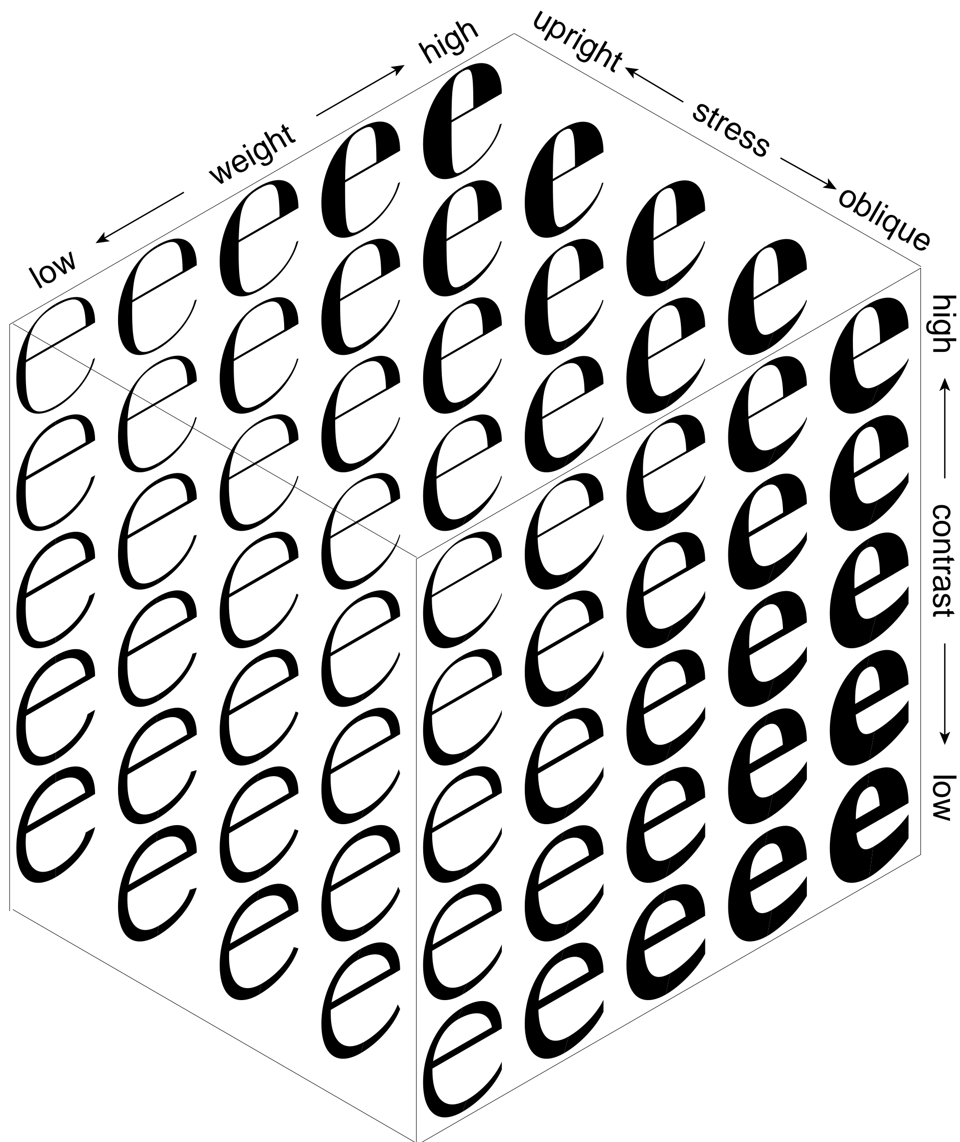
Resynthesized Bodoni lower-case characters:

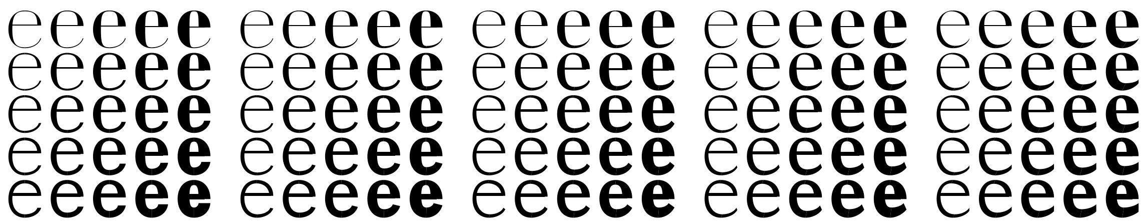
abcde
fghijk
lmnop
qrstuv
wxyz

Resynthesized Helvetica lower-case characters:

abcde
fghijkl
mnop
qrstuv
wxyz

APPENDIX F The variation cube of “e”





APPENDIX G Parameter files of parametrisable Times Roman

The global, group and local parameters files in this appendix contains parameters for the synthesis of a Times Roman normal weight font. These parameter files can also be used to synthesize Times typefaces with different boldness factors by simply modifying the definition of the macro *$BoldnessFactor* in the global parameter file.

1. The global parameter file for the parametrisable Times font together with a definition of the boldness factor:

```
/* TimesBoldness.glb - the global parameter file for Times ***************/
/* Created: 3/sep/1997 CHU                                                */
/* Modified: 31/aug/1998, CHU, boldness variation by macro def&ref.       */
/*----------------------------------------------------------------------*/
/*     This is a global parameter file written to prepare the global      */
/* parameters for the component-based font synthesizing system. This      */
/* global parameter file has a $BoldnessFactor defined for generating     */
/* typefaces in Times family by varying boldness.                         */
/*     The global parameter file is readable, editable and programmable.  */
/* However, any modification to this file should be clearly noticed at    */
/* this file header.                                                      */
/*          -------- Copyright 1997, 1998, LSP/EPFL --------              */
/************************************************************************/

/****================ begin of font information =================****/
sFontname = "TimesBoldness";        /* name of the font */
sFontfamily = "Times";              /* family name of the font */
sCreator = "CHU";                   /* author name */
sVersion = "2.0";                   /* version munber */
sDescription = "variation of boldness"; /* brief description of the font */
sCopyright = "LSP/DI/EPFL 1998";    /* copyright info */
sEncodingStandard = "ASCII";        /* encoding standard */
iNumberOfCharacters = 26;           /* from first to last continously */
iFirstCharCode = 0x61;              /* first code, from 'a' */
iLastCharCode = 0x7A;               /* lart code, to 'z' */
fScaleFactorU = 1.0;                /* scale factor */
fScaleFactorX = 1.0;                /* scalex */
fScaleFactorY = 1.0;                /* scaley */
/****================ end of font information ==================****/

/*********************** style variation factors ********************/
//  To change boldness to X (for example 120%):
//    - vertical stem and diagonal bar width increased X
//    - horizontal bar width increased to 1 + (X - 1) * 0.5
//    - vertical serif width reduced to 1 - (X - 1) * 2.0 if X > 130%
//
$BoldnessFactor = 1.00;             /* good through [80%, 150%] */
$BV = $BoldnessFactor;              /* for vertical stems and diagolal bars */
$BH = 1.0 $BoldnessFactor 1.0 sub 0.5 mul add; /* for horizontal bars */
$BS = $BoldnessFactor 1.3 gt 1.0 $BoldnessFactor 1.3 sub 2.0 mul sub 1.0 ifelse;
                                    /* for serif shrinking */

/*********************** global earmark styles *********************/
/* serif style ---------------------------------------------------- */
iSerifStyle = 1;                // 1 Times serif, 2 sans-serif
iSerifSupportType = 1;          // 1 smooth, 2 angled, 3 straight, 4 none
iSerifEndType = 1;              // 1 bracket (butt), 2 round
iSerifFaceType = 1;             // 1 flat, 2 concave
iTopSerifSupportType = 1;       // 1 smooth, 2 angled, 3 straight, 4 none
iTopSerifEndType = 1;           // 1 bracket (butt), 2 round
iTopSerifFaceType = 1;          // 1 coved, 2 flat
iHorSerifSupportType = 1;       // 1 smooth, 2 angled, 3 straight, 4 none
iHorSerifEndType = 1;           // 1 bracket (butt), 2 round
iHorSerifFaceType = 1;          // 1 flat, 2 concave
iHorTopSerifSupportType = 1;    // 1 smooth, 2 angled, 3 straight, 4 none
iHorTopSerifEndType = 1;        // 1 bracket (butt), 2 round
iHorTopSerifFaceType = 1;       // 1 coved, 2 flat
pSerifRoundCorrectionH = 0;     // height of round correction, not used
pSerifConcaveCorrectionH = 0;   // height of concave correction, not used

/*********************** global parameters *************************/
pBaseLine   = 0;                /* base line position */
```

```
pXheight    = 445;              /* x-height line position */
pCaps       = 659;              /* caps-height line position */
pNumbers    = 659;              /* height of numbers and symbols */
pAscender   = 679;              /* ascender line position */
pDescender  = -220;             /* descender line position */

/* beta & eta ------------------------------------------------------*/
pBestBeta = 0.59;               /* other betas may be related to this */
pStdBetaXS = 0.54;              /* extra flat curve */
pStdBetaS  = 0.59;              /* round loop */
pStdBetaM  = 0.67;              /* normal flat curve */
pStdBetaL  = 0.85;              /* sharp curve */
pStdBetaXL = 0.95;              /* extra sharp curve */

pSerifBeta = 0.67;              /* serif support */
pLoopExternalBeta = $pBestBeta; /* loop external */
pLoopInternalBeta = $pBestBeta; /* loop external */

pStdLoopEta = 0.13;             /* global eta of loops */

/* stem ------------------------------------------------------------*/
pcVerStemW = 104 $BV mul;       /* standard vertical stem width list */
pmVerStemW = 86 $BV mul;
pdVerStemW = 78 $BV mul;
pcHorStemW = 40 $BH mul;        /* standard horizontal stem width list */
pmHorStemW = 34 $BH mul;
pdHorStemW = 76 $BH mul;
pcNarrowVerStemW = 46 $BV mul;  /* standard narrow vertical stem width */
pmNarrowVerStemW = 20 $BV mul;
pdNarrowVerStemW = 0;
pcNarrowHorStemW = 0;           /* standard narrow horizontal stem width */
pmNarrowHorStemW = 31 $BH mul;
pdNarrowHorStemW = 66 $BH mul;
pcVerCurveW = 116 $BV mul;      /* standard vertical curve width */
pmVerCurveW = 93 $BV mul;
pdVerCurveW = 97 $BV mul;
pcHorCurveW = 38 $BH mul;       /* standard horizontal curve width */
pmHorCurveW = 33 $BH mul;
pdHorCurveW = 29 $BH mul;
pcDiagStemW = 99.67 $BV mul;    /* standard diagnal stem width */
pmDiagStemW = 79.0 $BV mul;
pdDiagStemW = 89.6 $BV mul;
pcNarrowDiagStemW = 43.33 $BV mul; /* standard narrow diagnal stem width */
pmNarrowDiagStemW = 37 $BV mul;
pdNarrowDiagStemW = 51.13 $BV mul;

/* character width -------------------------------------------------*/
pcRoundLetterW = 655;           /* standard round letter width RLW */
pmRoundLetterW = 440;
pdRoundLetterW = 452;
pcStemStemW = 0;                /* standard SSW, for h, n, m, u */
pmStemStemW = 266;
pdStemStemW = 0;
pcStemCurveW = 0;               /* standard SCW, for b, d, p, q */
pmStemCurveW = 312;
pdStemCurveW = 0;

/* serif -----------------------------------------------------------*/
pcVerSerifW = 89 $BS mul;       /* standard vertical serif width */
pmVerSerifW = 76 $BS mul;
pdVerSerifW = 95 $BS mul;
pcVerSerifD = $pcVerSerifW;     /* standard vertical serif depth */
pmVerSerifD = $pmVerSerifW;
pdVerSerifD = $pdVerSerifW;
pcDiagOuterSerifW = 56 $BS mul; /* standard diagnal outer serif width */
pmDiagOuterSerifW = 49 $BS mul;
pdDiagOuterSerifW = 0;
pcDiagOuterSerifD = $pcDiagOuterSerifW; /* standard diagnal outer serif depth */
pmDiagOuterSerifD = $pmDiagOuterSerifW;
pdDiagOuterSerifD = $pdDiagOuterSerifW;
pcDiagInnerSerifW = 100 $BS mul; /* standard diagnal inner serif width */
pmDiagInnerSerifW = 64 $BS mul;
pdDiagInnerSerifW = 0;
pcDiagInnerSerifD = $pcDiagInnerSerifW; /* standard diagnal inner serif depth */
pmDiagInnerSerifD = $pmDiagInnerSerifW;
pdDiagInnerSerifD = $pdDiagInnerSerifW;
pcVerSerifH = 20;               /* standard vertical serif height */
pmVerSerifH = 15;
pdVerSerifH = 18;
pcHorSerifW = 96;               /* standard horizontal serif width */
pmHorSerifW = 105;
```

```
pdHorSerifW = 80;
pcHorSerifD = $pcHorSerifW;     /* standard horizontal serif depth */
pmHorSerifD = $pmHorSerifW;
pdHorSerifD = $pdHorSerifW;
pcHorSerifH = 23;               /* standard horizontal serif height */
pmHorSerifH = 20;
pdHorSerifH = 18;
pcTopSerifDeg = -7;             /* standard slanted serif orientation */
pmTopSerifDeg = 13 $BS mul;
pdTopSerifDeg = 0;

/* others ----------------------------------------------------------*/
pcDotR = 47 $BV mul;            /* standard dot radius */
pmDotR = 52 $BV mul;
pdDotR = 0;
pcOptCor = 16;                  /* standard optical correction */
pmOptCor = 12;
pdOptCor = 16;

/* spacing ---------------------------------------------------------*/
pCapitalSpacing = 187;          /* capital ideal optical spacing */
pSmallSpacing = 130;            /* small ideal optical spacing */
pCapitalStemSpacing = 217;      /* capital NN spacing */
pSmallStemSpacing = 147;        /* small nn spacing */
pCapitalCurveSpacing = 67;      /* capital OO spacing */
pSmallCurveSpacing = 60;        /* small oo spacing */
pCapitalStemToCurveSpacing = 142; /* capital ON spacing */
pSmallStemToCurveSpacing = 104; /* small on spacing */
pCapitalMinimalSpacing = 43;    /* capital minimum spacing */
pSmallMinimalSpacing = 29;      /* small minimum spacing */
```

2. The group parameter definition file for the parametrisable Times font:

```
/* TimesBoldness.grp - the group parameter file for Times ****************/
/* Created: 3/sep/1997 CHU                                               */
/* Modified: 31/aug/1998, CHU, boldness variation by macro def&ref.      */
/*-----------------------------------------------------------------------*/
/*     This is a group parameter definition file for preparing the group */
/* parameters for the component-based font synthesizing system. This     */
/* group parameter file contains macro names of local parameter groups   */
/* which will be refered to in the local parameter file.                 */
/*     Rules of macro definitions:                                       */
/*     - a macro definition is a "name = value" pair, in which           */
/*     - a name is a string begin with letter '$',                       */
/*     - value is a float value readable by the C/C++ function scanf (). */
/*     Naming convensions:                                               */
/*     - the first letter after the '$' is a letter c/m/n indicating     */
/*       capital/minuscule/number.                                       */
/*     Macro definitions may be typeface dependent, hence group names    */
/* are user defined. Local parameter grouping is our method of enabling  */
/* coherent local feature modification amongst several characters.       */
/*     The group parameter file is readable, editable and programmable.  */
/* However, any modification to this file should be clearly noticed at   */
/* this file header.                                                     */
/*          -------- Copyright 1997, 1998, LSP/EPFL --------             */
/*************************************************************************/

/* group parameters' macro definition for Times  ************************/

/*****************************/
/* standard words shortening */
/*****************************/
//    Terminal    --> Term
//    Junction    --> Junc
//    Horizontal  --> Hor
//    Vertical    --> Ver
//    Position    --> Pos
//    External    --> Ext
//    Internal    --> Int
//    Width       --> W
//    Height      --> H
//    Depth       --> D
//    Radius      --> R
//    Xdirection  --> X
//    Ydirection  --> Y

/* macros for minuscules -----------------------------------------------*/

// typical character groups and their representative characters:
```

```
//    b/d/p/q     --> b            same loop, junction
//    h/m/n/u     --> n            same arch
//    c/e/o       --> e            round letter
//    k/v/w/x/y/z --> x            slanted stem
//    f/j/r/y     --> f            pear terminated
//    i/j         --> i            same dot
//

/* style */
$iSerifStyle = 1;              // this overrides the globle parameter
$mBendedTermStyle = 1;         // a/c/f/j/r/y, 1 - dot/pear, 2 - butt, 3 - long
$mSweepStemJuncStyle = 1;       // a/b/d/g/h/m/n/p/q/r/u, 1 - angled, 2 - smoothed

/* loop */
$mLoopCenterXPosExt = 0.50;   // b/d/p/q
$mLoopCenterXPosInt = 0.45;   // b/d/p/q
$mLoopEtaExt = $pStdLoopEta;  // b/d/p/q
$mLoopEtaInt = 0.00;          // b/d/p/q

/* pos */
$mArmPos = 0.84;              // h/m/n/u/r
$mEBarPos = 0.64;             // e
$mABarPos = 0.68;             // a

/* narrow/wide */
$mHorCurveWnarrow = 0.79;     // a/f
$mHorCurveWwide = 1.70;       // a/b/c/d/e/g/h/m/n/p/q/t/u
$mDotRnarrowa = 0.87;         // a/c/f/j/r/y
$mDotRnarrowb = 1.00;         // a/c/f/j/r/y
$mVerCurveWnarrow = 0.75;     // e/g/s
$mVerCurveWwide = 1.05;       // e
$mVerSerifWnarrow = 0.84;     // h/k/m/n/u,
$mVerSerifWwide = 1.11;       // f
$mDiagSerifOuterWnarrow = 0.76; // v/w/y
$mDiagSerifInnerWwide = 1.25; // k/w/y
$mHorSerifWnarrow = 0.80;     // z

/* connecting sweep A */
$mLetterbLowerSweepA = 0.83;  // b/d/p/q
$mLetterbUpperSweepA = 0.60;  // b/d/p/q

/* arch */
$mLetternArchOuterTop = 0.61;      // h/m/n/u
$mLetternArchInnerTop = 0.60;      // h/m/n/u
$mLetternArchOuterStart = 0.76;    // h/m/n/u
$mLetternArchInnerStart = 0.72;    // h/m/n/u
$mLetternArchJuncSweepA = 0.45;    // h/m/n/u

/* special earmarks */
$mLetterbLowerJuncStyle = $BoldnessFactor 1.4 lt 3 $mSweepStemJuncStyle ifelse;
                                   // b, 3 - meet, if bold<1.4, otherwise spur
$mLetterbUpperJuncPos = $mArmPos;  // b/d/p/q/g/a
```

3. The local parameter file for the parametrisable Times font:

```
/* TimesBoldness.lcl - the local parameter file of Times *****************/
/* Created: 3/sep/1997 CHU                                                */
/* Modified: 31/aug/1998, CHU, boldness variation by macro def&ref.      */
/*-----------------------------------------------------------------------*/
/*     This is a local parameter file written for preparing every local  */
/* parameters for the component-based font synthesizing system. This     */
/* local parameter file contains local parameters of each characters.    */
/* Some of the parameters are grouped to ensure coherent local feature   */
/* modification.                                                          */
/*     Local parameter names may not be understandable even with the     */
/* comments. Fortunately, they are not necessary for users who wants to  */
/* derive new typeface just by varing global typeface features, such as  */
/* wight, width, contrast, stress and height proportion. To better       */
/* understand local parameters, one should read the charXX.cp files.     */
/*     The local parameter file is readable, editable and programmable.  */
/* However, any modification to this file should be clearly noticed at   */
/* this file header.                                                      */
/*         -------- Copyright 1997, 1998, LSP/EPFL --------              */
/*************************************************************************/

char61                             // "a"
{
    variation = 1;                 // 1 - double storey; 2 - single storey
```

```
    pdx1 = 0.54;                     // main width, belly to stem
    pdx2 = 0.50;                     // secondary width, bulb to stem
    Tr1c1 = $mBendedTermStyle;       // 1 dot/pear, 2, 3 butt
    Tr2c2 = 1;                       // 1 pointed, 2 butt down, 3 butt right,
                                     // 4 sans-serif, 5 slab-serif
    Jr1c2 = 1;                       // 1 angled, 2 smoothed, 3 loop
    Jr2c2 = $mSweepStemJuncStyle;    // 1 angled, 2 smoothed
    dotAngle = -10;                  // Degree
    headWidth = 0.60;                // in proportion to StdVerStemW
    crr1 = 0.03;
    ppp1 = 0.50;                     // top arc center
    ppp2 = 0.71;                     // top arc start
    ppp3 = 0.77;                     // dot center height
    ppp4 = 0.15;                     // tail bottom-most x
    ppp5 = 0.45;                     // tail right-most x
    ppp6 = 0.12;                     // tail right-most y
    ppp7 = 0.60;                     // tail curve A.x
    ppp8 = 0.25;                     // belly left-most height
    ppp9 = 0.60;                     // s6 curve A.x
    pppA = 0.45;                     // belly bottom-most x
    pppB = 0.42;                     // belly inner bottom-most x
    pppC = 0.35;                     // s8 curve A.x
    pppD = 0.85;                     // s2 curve A.x
    pppE = 0.18;                     // tail start y
    pppG = 1 $mArmPos sub;           // s8 arrival y, = 1 - ArmPos
    pppH = $mABarPos;                // a-bar position
    pppJ = 0.45;                     // smoothing edge1, related to dx1
    pppK = 0.10;                     // smoothing edge2, related to pXheight
    pppL = 0.15;                     // smoothing tang1, related to dx1
    pppM = 0.10;                     // smoothing tang2, related to pXheight
    narrow1 = $mHorCurveWnarrow;     // narrow minuscule horizontal curve width
    narrow2 = $mDotRnarrowa;         // narrow minuscule dot width a
    narrow3 = $mDotRnarrowb;         // narrow minuscule dot width b
    wide1 = $mHorCurveWwide;         // wide minuscule horizontal curve width
}
char62                               // "b"
{
    Tr1c1 = $iSerifStyle;            // 1 has topserif, 2 no topserif
    Tr2c1 = 1;                       // 1 spur, 2 no spur, 3 half foot serif
    Jr1c1 = $mSweepStemJuncStyle;    // 1 angled, 2 smoothed
    Jr2c1 = $mLetterbLowerJuncStyle; // 1 angled with spur, 2 smoothed,
                                     // 3 angled and meet
    eta1 = $mLoopEtaExt neg;         // loop eta external, - 0.15
    eta2 = $mLoopEtaInt;             // loop eta internal
    ppp1 = $mLoopCenterXPosExt;      // center1.x of the loop
    ppp2 = $mLoopCenterXPosInt;      // center2.x of the loop
    ppp3 = 0.13;                     // junction Tr2c1 height
    ppp4 = 0.16;                     // junction Tr2c1 internal height
    ppp5 = $mLetterbLowerSweepA;     // beta-curve control A of sweep s3
    ppp6 = $mLetterbUpperSweepA;     // beta-curve control A of sweep s4
    ppp7 = $mLetterbUpperJuncPos;    // arm position of Jr1c1, b/d/p/q
    spurAngle = 13;                  // Degree
    wide1 = $mHorCurveWwide;         // wide minuscule hor curve width, a/b/n
}
char63                               // "c"
{
    pdx1 = 0.88;                     // main width, half-loop to tail
    pdx2 = 0.84;                     // secondary width, half-loop to bulb
    Tr1c1 = $mBendedTermStyle;       // 1 dot, 2, 3 butt
    Tr2c1 = 1;                       // 1 butt
    eta1 = -0.02;                    // loop external, non-grp
    eta2 = 0.10;                     // loop internal, non-grp
    dotAngle = 10;                   // Degree
    tailAngle = 30;                  // Degree
    tailWidth = 0.28;                // tail width
    ppp1 = 0.50;                     // loop center external
    ppp2 = 0.55;                     // loop center internal
    ppp3 = 1.30;                     // bottom arc width correction
    ppp4 = 0.78;                     // dot center y
    ppp5 = 0.85;                     // s2 curve A.x
    ppp6 = 0.35;                     // tail y
    ppp7 = 0.65;                     // tail s3 curve A.x
    pppJ = 0.40;                     // smoothing edge1, related to dx1
    pppK = 0.10;                     // smoothing edge2, related to pXheight
    pppL = 0.15;                     // smoothing tang1, related to dx1
    pppM = 0.10;                     // smoothing tang2, related to pXheight
    wide1 = $mHorCurveWwide;         // wide minuscule hor curve width a/b/c/n
    narrow1 = $mDotRnarrowa;         // narrow minuscule dot radius a
    narrow2 = $mDotRnarrowb;         // narrow minuscule dot radius b
}
char64                               // "d"
```

```
{
    pdx1 = 1.0;                     // main width, half-loop to stem
    Tr1c1 = $iSerifStyle;           // 1 has topserif, 2 no serif
    Tr2c1 = $iSerifStyle;           // 1 has serif, 2 no serif
    Jr1c1 = $mSweepStemJuncStyle;   // 1 angled link, 2 smoothed link
    Jr2c1 = $mSweepStemJuncStyle;   // 1 angled link, 2 smoothed link
    eta1 = $mLoopEtaExt neg;        // loop eta external
    eta2 = $mLoopEtaInt;            // loop eta internal
    ppp1 = $mLoopCenterXPosExt;     // center1.x of the loop
    ppp2 = $mLoopCenterXPosInt;     // center2.x of the loop
    ppp3 = 0.83;                    // s3 Pd.y
    ppp4 = 0.70;                    // s3 Qd.y
    ppp5 = $mLetterbLowerSweepA;    // s3 curve A.x
    ppp6 = $mLetterbUpperSweepA;    // s4 curve A.x
    ppp7 = 1 $mLetterbUpperJuncPos sub; // arm position of Jr2c1, b/d/p/q
    spurAngle = 13;                 // bottom serif slant, never zero.
    wide1 = $mHorCurveWwide;        // wide hor curve width, a/b/c/n
}
char65                              // "e"
{
    pdx1 = 0.91;                    // main width, half-loop to tail
    pdx2 = 0.86;                    // secondary width, half-loop to corner
    Tr2c2 = 1;                      // 1 butt tail
    Jr1c1 = 1;                      // 1 perpendicular link
    Jr1c2 = 1;                      // 1 squared corner
    eta1 = 0.00;                    // loop eta external
    eta2 = 0.19;                    // loop eta internal
    tailAngle = 30;                 // Degree
    tailWidth = 0.28;               // tail width
    ppp1 = 0.50;                    // loop center external
    ppp2 = 0.51;                    // loop center internal
    ppp3 = 1.30;                    // bottom arc width correction
    ppp4 = 0.35;                    // tail right-most y
    ppp5 = 0.65;                    // tail s3 curve A.x
    ppp6 = $mEBarPos;               // e-bar position
    narrow1 = $mVerCurveWnarrow;    // narrow minuscule vertical curve width
    wide1 = $mVerCurveWwide;        // wide minuscule vertical curve width
    wide2 = $mHorCurveWwide;        // wide munuscule horizontal curve width
}
char66                              // "f"
{
    pdx1 = 0.41;                    // main width, stem to bulb
    Tr1c1 = $mBendedTermStyle;      // 1 dot, 2 butt, 3 long squared
    Tr2c1 = $iSerifStyle;           // 1 serif, 2 no serif
    dotAngle = 30;                  // Degree
    ppp1 = 0.19;                    // bar left-most
    ppp2 = 0.27;                    // bar right-most
    ppp3 = 0.90 $BV 1.0 sub 0.08 mul sub; // dot center y, adjusted to boldness
    ppp4 = 0.50;                    // top arc internal extreme x
    ppp5 = 0.65;                    // top arc external extreme x
    ppp6 = 0.85;                    // top arc start internal y
    ppp7 = 0.69;                    // top arc start external y
    ppp8 = 0.65;                    // top arc s5 curve A.x
    pppJ = 0.40;                    // smoothing edge1, related to dx1
    pppK = 0.07;                    // smoothing edge2, related to pXheight
    pppL = 0.15;                    // smoothing tang1, related to dx1
    pppM = 0.05;                    // smoothing tang2, related to pXheight
    narrow1 = $mHorCurveWnarrow;    // narrow minuscule horizontal curve width
    narrow2 = $mDotRnarrowa;        // narrow minuscule dot ridus a
    narrow3 = $mDotRnarrowb;        // narrow minuscule dot ridus b
    wide1 = $mVerSerifWwide;        // wide minuscule vertical serif width/depth
}
char67                              // "g"
{
    variation = 1;                  // 1 - double storey; 2 - single storey
    Tr1c1 = 1;                      // 1 bar, 2 ear (sw + dot + smooth);
    x0 = 245;                       // orginal
    /* if variation == 1 */
    eta1 = 0.00;                    // loop eta external
    eta2 = 0.19;                    // loop eta internal
    pdx1 = 0.28;                    // main width, loop center to loop right
    pdx2 = 0.50;                    // secondary width, to belly right
    pdx3 = 0.50;                    // third width
    ppp1 = 0.66;                    // loop center y
    ppp2 = 0.90;                    // currection of curve width of loop s0
    ppp3 = 0.90;                    // bar height
    ppp4 = 0.15;                    // s2 Pd.x
    ppp5 = 0.10;                    // s2 Qd.x
    ppp6 = 0.15;                    // s2 Pa.y
    ppp7 = 0.68;                    // currection of curve width of s2.PaQa
    ppp8 = 0.20;                    // s2 Qa.y
```

```
    ppp9 = 0.47;                      // s2 curve A.y
    pppA = 0.10;                      // s3 Pa.x
    pppB = 0.12;                      // s3 Qa.y --- old y1
    pppC = 0.30;                      // s3 Qa.x
    pppD = 1.00;                      // s3 curve A.y
    pppE = 0.32;                      // curreection of curve width of s4.PaQa
    pppF = 0.32;                      // s4 Pa.y
    pppG = 0.22;                      // s4 Qa.y
    pppH = 1.00;                      // s4 curve A.y
    pppI = 0.00;                      // s5 Pa.x
    pppJ = 0.20;                      // s5 Qa.x
    pppK = 0.78;                      // curreection of curve width of s6.PaQa
    pppL = 0.42;                      // s6 Pa.y
    pppM = 0.52;                      // s6 Qa.y
    pppN = 0.15;                      // s7 Pa.x
    pppO = 0.22;                      // s7 Qa.x
    pppP = 0.40;                      // s7 curve A.y
    narrow1 = $mVerCurveWnarrow;      // narrow minuscule vertical curve width
    narrow2 = $mHorCurveWnarrow;      // narrow minuscule horizontal curve width
    wide1 = $mHorCurveWwide;          // wide munuscule hor curve width
}
char68                                // "h"
{
    pdx1 = 1.0;                       // main width, stem to stem
    Tr1c1 = $iSerifStyle;             // 1 has topserif, 2 sans-serif
    Tr2c1 = $iSerifStyle;             // 1 has serif, 2 sans-serif
    Tr2c2 = $iSerifStyle;             // 1 has serif, 2 sans-serif
    Jr1c1 = $mSweepStemJuncStyle;     // 1 angled link, 2 smoothed link
    ppp1 = $mLetternArchOuterTop;     // arc top outer x
    ppp2 = $mLetternArchInnerTop;     // arc top inner x
    ppp3 = $mLetternArchOuterStart; // s5 Pd.y
    ppp4 = $mLetternArchInnerStart; // s5 Qd.y
    ppp5 = $mLetternArchJuncSweepA; // s6 curve A.x
    ppp6 = $mArmPos;                  // arm position of Jr1c1
    wide1 = $mHorCurveWwide;          // wide minuscule hor curve width
    narrow1 = $mVerSerifWnarrow;      // narrow minuscule vertical serif width
}
char69                                // "i"
{
    pdx1 = 0.02;                      // main width, stem to dot center
    Tr1c1 = 1;                        // 1 round dot, 2 square dot
    Tr2c1 = $iSerifStyle;             // 1 has topserif, 2 sans-serif
    Tr3c1 = $iSerifStyle;             // 1 has serif, 2 sans-serif
    narrow1 = $mDotRnarrowa;          // narrow minuscule dot radius a
}
char6A                                // "j"
{
    pdx1 = 0.40;                      // main width, stem to tail bulb
    pdx2 = 0.02;                      // secondary width, stem to dot center
    Tr1c1 = 1;                        // 1 round dot, 2 square dot
    Tr2c1 = $iSerifStyle;             // 1 has topserif, 2 sans-serif
    Tr3c1 = $mBendedTermStyle;        // 1 dot/pear, 2 butt, 3 long

    dotAngle = 45 $BV 1.0 sub 10 mul sub; // bulb, Degree, adjusted to boldness
    ppp1 = 0.40;                      // tail left start s2 Pd.y
    ppp2 = 0.15;                      // tail right start s2 Qd.y
    ppp3 = 0.50;                      // tail bottom s2 Pa.x
    ppp4 = 0.65;                      // tail bottom s2 Qa.x
    ppp5 = 0.73 $BV 1.0 sub 0.2 mul sub; // bulb center y, adjusted to boldness
    ppp6 = 0.66;                      // tail s4 curve A.x
    pppJ = 0.40;                      // smoothing edge1, related to dx1
    pppK = 0.07;                      // smoothing edge2, related to pXheight
    pppL = 0.10;                      // smoothing tang1, related to dx1
    pppM = 0.05;                      // smoothing tang2, related to pXheight
    narrow1 = $mDotRnarrowa;          // narrow minuscule dot radius a
    narrow2 = $mDotRnarrowb;          // narrow minuscule dot radius a
}
char6B                                // "k"
{
    pdx1 = 0.63;                      // main width, stem to right-bottom
    pdx2 = 0.62;                      // secondary width, to right-up
    Tr1c1 = $iSerifStyle;             // 1 has topserif, 2 sans-serif
    Tr1c2 = $iSerifStyle;             // 1 has serif, 2 sans-serif
    Tr2c1 = $iSerifStyle;             // 1 has serif, 2 sans-serif
    Tr2c2 = $iSerifStyle;             // 1 has serif, 2 sans-serif
    Jr1c1 = 1;                        // 1 singla junction, 2 double junction
    ppp1 = 0.85;                      // junction center x
    ppp2 = 0.60;                      // junction center y
    wide1 = $mDiagSerifInnerWwide;  // wide diag serif inner width
    narrow1 = $mVerSerifWnarrow;      // narrow minuscule vertical serif width
}
```

```
char6C                          // "l"
{
    Tr1c1 = $iSerifStyle;       // 1 has topserif, 2 sans-serif
    Tr2c1 = $iSerifStyle;       // 1 has serif, 2 sans-serif
}
char6D                          // "m"
{
    pdx1 = 1.0;                 // main width, stem to stem
    Tr1c1 = $iSerifStyle;       // 1 has topserif, 2 sans-serif
    Tr2c1 = $iSerifStyle;       // 1 has serif, 2 sans-serif
    Tr2c2 = $iSerifStyle;       // 1 has serif, 2 sans-serif
    Tr2c3 = $iSerifStyle;       // 1 has serif, 2 sans-serif
    Jr1c1 = $mSweepStemJuncStyle;  // 1 angled link, 2 smoothed link
    Jr1c2 = $mSweepStemJuncStyle;  // 1 angled link, 2 smoothed link
    ppp1 = $mLetternArchOuterTop;  // arc top outer x
    ppp2 = $mLetternArchInnerTop;  // arc top inner x
    ppp3 = $mLetternArchOuterStart; // s5 Pd.y
    ppp4 = $mLetternArchInnerStart; // s5 Qd.y
    ppp5 = $mLetternArchJuncSweepA; // s6 curve A.x
    ppp6 = $mArmPos;            // arm position of Jr1c2/Jr1c2
    wide1 = $mHorCurveWwide;    // wide minuscule hor curve width
    narrow1 = $mVerSerifWnarrow;   // narrow minuscule vertical serif width
}
char6E                          // "n"
{
    pdx1 = 1.0;                 // main width, stem to stem
    Tr1c1 = $iSerifStyle;       // 1 has topserif, 2 sans-serif
    Tr2c1 = $iSerifStyle;       // 1 has serif, 2 sans-serif
    Tr2c2 = $iSerifStyle;       // 1 has serif, 2 sans-serif
    Jr1c1 = $mSweepStemJuncStyle;  // 1 angled link, 2 smoothed link
    ppp1 = $mLetternArchOuterTop;  // arc top outer x
    ppp2 = $mLetternArchInnerTop;  // arc top inner x
    ppp3 = $mLetternArchOuterStart; // s5 Pd.y
    ppp4 = $mLetternArchInnerStart; // s5 Qd.y
    ppp5 = $mLetternArchJuncSweepA; // s6 curve A.x
    ppp6 = $mArmPos;            // arm position of Jr1c1
    wide1 = $mHorCurveWwide;    // wide minuscule hor curve width
    narrow1 = $mVerSerifWnarrow;   // narrow minuscule serif width h/n/m/u
}
char6F                          // "o"
{
    eta1 = 0.00;                // loop eta external
    eta2 = $pStdLoopEta 0.02 add;  // loop eta internal, a little stress
}
char70                          // "p"
{
    pdx1 = 1.0;                 // main width, stem to curve
    Tr1c1 = $iSerifStyle;       // 1 has topserif, 2 no serif
    Tr2c1 = $iSerifStyle;       // 1 has serif, 2 no serif
    Jr1c1 = $mSweepStemJuncStyle;  // 1 angled link, 2 smoothed link
    Jr2c1 = $mSweepStemJuncStyle;  // 1 andled link, 2 smoothed link
    eta1 = $mLoopEtaExt neg;    // loop eta external
    eta2 = $mLoopEtaInt;        // loop eta internal
    ppp1 = $mLoopCenterXPosExt;    // loop center1.x
    ppp2 = $mLoopCenterXPosInt;    // loop center2.x
    ppp3 = 0.17;                // s3 Pd.y
    ppp4 = 0.25;                // s3 Qd.y
    ppp5 = $mLetterbLowerSweepA;   // s3 curve A.x
    ppp6 = $mLetterbUpperSweepA;   // s4 curve A.x
    ppp7 = $mLetterbUpperJuncPos;  // arm position of Jr1c1, b/d/p/q/...
    spurAngle = 13;             // top serif slant
    wide1 = $mHorCurveWwide;    // wide minuscule hor curve width
    narrow1 = $mVerSerifWnarrow;   // narrow minuscule serif width h/n/m/u
}
char71                          // "q"
{
    pdx1 = 1.0;                 // main width, stem to curve
    Tr1c1 = $iSerifStyle;       // 1 has topserif, 2 no serif
    Tr2c1 = $iSerifStyle;       // 1 has serif, 2 no serif
    Jr1c1 = $mSweepStemJuncStyle;  // 1 angled link, 2 smoothed link
    Jr2c1 = $mSweepStemJuncStyle;  // 1 andled link, 2 smoothed link
    eta1 = $mLoopEtaExt neg;    // loop eta external
    eta2 = $mLoopEtaInt;        // loop eta internal
    ppp1 = $mLoopCenterXPosExt;    // loop center1.x
    ppp2 = $mLoopCenterXPosInt;    // loop center2.x
    ppp3 = 0.85;                // s3 Pd.y
    ppp4 = 0.70;                // s3 Qd.y
    ppp5 = $mLetterbLowerSweepA;   // s3 curve A.x
    ppp6 = $mLetterbUpperSweepA;   // s4 curve A.x
    ppp7 = 1.80;                // top spur height correctness of Opt
    ppp8 = 1 $mLetterbUpperJuncPos sub; // arm position of Jr2c1, b/d/p/q/h/n/m
```

```
    spurAngle = 20;                 // top spur slant, Degree
    wide1 = $mHorCurveWwide;        // wide minuscule hor curve width
}
char72                              // "r"
{
    pdx1 = 0.39;                    // main width, stem to bulb center
    Tr1c1 = $iSerifStyle;           // 1 has topserif, 2 sans-serif
    Tr2c1 = $iSerifStyle;           // 1 has serif, 2 sans-serif
    Tr1c2 = $mBendedTermStyle;      // 1 dot/pear, 2 butt, 3 long
    Jr1c1 = $mSweepStemJuncStyle;   // 1 angled link, 2 smoothed link
    dotAngle = 30;                  // Degree
    ppp1 = $BV 1.0 ge 0.98 $BV 1.0 sub 0.04 mul sub 0.98 ifelse;
                                    // dot y, adjusted to boldness
    ppp2 = 0.60;                    // arc s4 curve A.x
    ppp3 = $mArmPos;                // arm position of Jr1c1
    pppJ = 0.60;                    // smoothing edge1, related to dx1
    pppK = 0.07;                    // smoothing edge2, related to pXheight
    pppL = 0.10;                    // smoothing tang1, related to dx1
    pppM = 0.10;                    // smoothing tang2, related to pXheight
    wide1 = $mHorCurveWwide;        // wide minuscule hor curve width
    narrow1 = $mVerSerifWnarrow;    // narrow minuscule ver serif width h/n/m/u
    narrow2 = $mDotRnarrowa;        // narrow minuscule dot radius a
    narrow3 = $mDotRnarrowb;        // narrow minuscule dot radius b
}
char73                              // "s"
{
    pdx1 = 0.52;                    // main width, left curve to right curve
    pdx2 = 0.57;                    // secondary width, left tail to right curve
    pdx3 = 0.51;                    // secondary width, left curve to right tail
    Tr1c1 = $iSerifStyle;           // 1 serif-like, 2 butt
    Tr2c1 = $iSerifStyle;           // 1 serif-like, 2 butt
    crr1 = 0.12;                    // special curved stroke stress
    ppp1 = 0.35;                    // y1, right tail height
    ppp2 = 0.38;                    // y2, right curve height
    ppp3 = 0.31;                    // y3, right tail height
    ppp4 = 0.36;                    // y4, left curve height
    ppp5 = 0.81;                    // s1 curve A.x
    ppp6 = 0.62;                    // s2 arrival, s3 start
    ppp7 = 0.60;                    // s4 arrival, s5 start
    ppp8 = 0.81;                    // s6 curve A.x
    pppJ = 1.00;                    // smoothing edge1, related to pXheight
    pppK = 0.43;                    // smoothing edge2, related to dx1
    pppL = 0.02;                    // smoothing tang1, related to pXheight
    pppM = 0.15;                    // smoothing tang2, related to dx1
    tailAngle = 30;                 // Degree, for sans-serif
    narrow1 = $mVerCurveWnarrow;    // narrow minuscule vertical curve width
    narrow2 = $mHorCurveWnarrow;    // narrow minuscule horizontal curve width
}
char74                              // "t"
{
    Tr1c1 = 1;                      // 1 concave head, 2 cross bar,
                                    // 3 cross bar with possibly slanted end
    Tr2c1 = 1;                      // 1 pointed tail, 2 butt
    tailAngle = 45;                 // Degree
    tailWidth = 0.50;               // tail width
    ppp1 = 0.22;                    // bar right-most
    ppp2 = 0.28;                    // tail right-most
    ppp3 = 0.13;                    // bar left-most
    ppp4 = 0.29;                    // stem top extension
    ppp5 = 0.15;                    // tail left-most y
    ppp6 = 0.20;                    // tail start s2 Pd.y
    ppp7 = 0.25;                    // tail start s2 Qd.y
    ppp8 = 0.45;                    // tail bottom most external x
    ppp9 = 0.41;                    // tail bottom most internal x
    pppA = 0.40;                    // tail s3 curve A.x
    pppB = 0.55;                    // top correction left
    pppC = 0.80;                    // top correction top
    pppD = 0.62;                    // top correction S4 curve A
    wide1 = $mHorCurveWwide;        // wide minuscule hor curve width
}
char75                              // "u"
{
    pdx1 = 1.0;                     // main width, stem to stem
    Tr1c1 = $iSerifStyle;           // 1 has serif, 2 sans-serif
    Tr1c2 = $iSerifStyle;           // 1 has serif, 2 sans-serif
    Tr2c2 = $iSerifStyle;           // 1 has topserif, 2 sans-serif
    Jr2c2 = $mSweepStemJuncStyle;   // 1 angled link, 2 smoothed link
    ppp1 = $mLetternArchOuterTop;   // arc bottom outer x
    ppp2 = $mLetternArchInnerTop;   // arc botton inner x
    ppp3 = $mLetternArchOuterStart; // s3 Pd.y
    ppp4 = $mLetternArchInnerStart; // s3 Qd.y
```

```
    ppp5 = $mLetternArchJuncSweepA 1 0.1 iplt;
                 // s4 curve A.x, iplt ($, 1, 0.1), bigger than that in 'n'
    ppp6 = 1 $mArmPos sub;         // arm position of Jr2c2
    wide1 = $mHorCurveWwide;       // wide minuscule hor curve width
    narrow1 = $mVerSerifWnarrow;   // narrow minuscule serif ver width h/n/m/u
}
char76                             // "v"
{
    pdx1 = 0.73;                   // main width, left to right
    Tr1c1 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Tr1c2 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    ppp1 = 0.58;                   // s0 start x
    ppp2 = 0.44;                   // s1 start x
    narrow1 = $mDiagSerifOuterWnarrow; // narrow diag serif outer width, v/w/y
}
char77                             // "w"
{
    pdx1 = 0.71;                   // main width, similar to v
    pdx2 = 0.15;                   // secondary width, overlap of two 'v'
    Tr1c1 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Tr1c2 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Tr1c3 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Jr1c2 = 1;                     // not used, equivalent to Tr1c2
    ppp1 = 0.56;                   // s0 start x
    ppp2 = 0.42;                   // s1 start x
    narrow1 = $mDiagSerifOuterWnarrow; // narrow diag serif outer width, v/w/y
    wide1 = $mDiagSerifInnerWwide; // wide diag serif inner width
}
char78                             // "x"
{
    pdx1 = 0.74;                   // main width, bottom left to right
    Tr1c1 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Tr1c2 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Tr2c1 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Tr2c2 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    crr1 = 0.15;                   // optical crr for bottom-left, stem s6
    crr2 = 0.45;                   // crr for top-left, s1
    crr3 = 0.05;                   // crr for top-right, s0
}
char79                             // "y"
{
    pdx1 = 0.77;                   // main width, top left to right
    pdx2 = 0.0;                    // secondary width, left to tail bulb center
    Tr1c1 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Tr1c2 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Tr2c1 = $mBendedTermStyle;     // 1 dot/pear, 2 butt, 3 long
    dotAngle = 75;                 // Degree
    ppp1 = 0.61;                   // stem s0 start x
    ppp2 = 0.42;                   // stem s1 start x
    ppp3 = 0.35;                   // stem s1 start y
    ppp4 = 0.78 $BV 1.0 sub 0.2 mul sub; // dot center y, adjusted to boldness
    ppp5 = 0.05;                   // tail bottom-most x
    ppp6 = 0.20;                   // tail s6 curve A.x   new !!!
    pppJ = 0.30;                   // smoothing edge1, related to dx1
    pppK = 0.10;                   // smoothing edge2, related to pXheight
    pppL = 0.15;                   // smoothing tang1, related to dx1
    pppM = 0.10;                   // smoothing tang2, related to pXheight
    narrow1 = $mDiagSerifOuterWnarrow; // narrow diag serif outer width, v/w/y
    narrow2 = $mDotRnarrowa;       // narrow minuscule dot radius a
    narrow3 = $mDotRnarrowb;       // narrow minuscule dot radius b
    wide1 = $mDiagSerifInnerWwide; // wide diag serif inner width
}
char7A                             // "z"
{
    pdx1 = 0.78;                   // main width, bottom left to bottom right
    pdx2 = 0.02;                   // secondary width, bottom left to top left
    pdx3 = 0.09;                   // secondary, bottom right to top right
    Tr1c1 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Tr2c2 = $iSerifStyle;          // 1 has serif, 2 sans-serif
    Jr1c2 = 1;                     // no use for Times
    Jr2c1 = 1;                     // no use for Times
    angle1 = -5;                   // topBeakAngle Degree
    angle2 = -5;                   // bottom Beak Angle Degree
    ppp1 = 0.05;                   // top MiterLimit, in ppp to StdDiagStemW
    ppp2 = 0.05;                   // bottomMiterLimit, in ppp to StdDiagStemW
    narrow1 = $mHorSerifWnarrow;   // narrow hor serif width
}
```

Curriculum Vitae

I was born on June 10, 1966, and I hold the Chinese nationality.

Since Jul. 1995, I have been working as a research assistant as well as a Ph.D candidate at the Peripheral Systems Laboratory (LSP) of EPFL. My main research effort was focussed on the design and development of a new parametrisable font synthesis system.

From Oct. 1994 to Jun. 1995, I worked as a technical consultant at the HuaGuang Electronics Group Inc. (Beijing) for the development of a Chinese photocomposing system. I was in charge of the high-quality high-speed Chinese character rasterization technology.

From Sep. 1992 to Aug. 1994, I was a research assistant at the Computer Science Department of Nanjing University (Nanjing, China). I took part in a national research project related to the flexible application of Chinese outline fonts. I designed and developed an automatic hinting system for Chinese outline fonts based on automatic stroke separation.

From Sep. 1991 to Aug. 1992, I was a teaching assistant at the Computer Science Department of Tsinghua University (Beijing).

In 1991, I received a Master degree in Computer Science and Application from Nanjing University. For the thesis, I developed a prototype PostScript interpreter with an extension to describe and print Chinese outline characters.

In 1988, I received my Bachelor degree in Computer Science from Nanjing University.