National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

Optimization of Bézier outlines and automatic font generation

Sandro Mazzucato

School of Computer Science
McGill University, Montreal

August 1994

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF MASTERS OF SCIENCE.

ISBN 0-612-05596-5

Canada

Abstract

A new method for the approximation of bitmap outlines with Bézier curves is presented in this thesis. This probabilistic approach accurately describes the contours with a small number of C^2 continuous splines. The approximation is refined iteratively using a quality function. The evaluation method is based upon the smoothness of the generated outline and the precision at which it interpolates the original contour. Merge operations of adjacent Bézier splines and spline alterations using an adaptive random search technique are employed for finding an optimal solution.

A program implementing the proposed algorithm was created and may be used to automatically generate PostScript type 1 fonts. The algorithm has shown to be very stable and to converge rapidly. Many new typefaces have been generated with the software and are shown in this thesis.

Résumé

Une nouvelle méthode d'approximation de contour de bitmaps par des courbes de Bézier est présentée dans cette thèse. Cette approche probabilistique décrit avec precision les contours en utilisant un petit nombre de courbes. Une continuité C^2 est guarantie. Une procédure itérative améliore l'approximation en utilisant une fonction de la qualité. La méthode d'évaluation est basée sur l'aspect lisse des contours générés et de l'exactitude de l'interpolation du contour original. Les operations de fusion de courbes de Bézier adjacentes ainsi que les changements apportés aux courbes, en utilisant une technique adaptive de recherche aléatoire, sont utilisés dans la quête d'une optimum.

Un logiciel implantant l'algorithme proposé a été créé et peut être utilisé pour generer automatiquement des polices dans le format type 1 de PostScript. L'algorithme a su démontrer sa stabilité ainsi que sa grande rapidité de convergence. Plusieures familles de polices ont été créé avec ce logiciel et elle sont présentées dans cetter thèse.

Acknowledgment

I would like to acknowledge the School of Computer Science, the office and technical support staff for their assistance in all aspects of my research.

I would like to thank my supervisor, Luc Devroye, for assisting me in my thesis. His constructive comments and suggestions were a source of inspiration well beyond the realm of this research.

Without forgetting my parents and family who's support and encouragements were very appreciated, I thank them.

Special thanks to Isabelle Massarelli, Birgit Devroye, Natasha Devroye, Luc Devroye and Janos Pach who let their pen flow for my thesis. I would also like to thank Jacques André who kindly gave a professional touch to this thesis by supplying the EPODD style file.

Finally I thank all my friends that took some time to share with me during my research.

Contents

1 Introduction

The art of writing is generally understood as the ability of selecting appropriate words to elegantly express ideas and concepts. Several centuries ago, each letter of a word had to be created individually, and the harmonization of adjacent characters in relation with their visual effect was an artwork in itself.

Some of the letters still used in today's western alphabets are direct descendants of those early designs. The evolution of letter shapes through time is a result of the changes in the movement patterns, also know as *ductus*, and of the writing tool employed (Bigelow and Day, 1983).

The creation of books was a very expensive and elaborate process as they were reproduced by hand. Mass production was performed by having a book read aloud while a group of scribes were writing the text. Books were not as omnipresent as today; only rich people could afford owning such a piece of work.

In the early fourteen hundreds, a goldsmith from Mainz turned an important page in typography. Johannes Gutenberg (1398-1468) invented movable types. His knowledge about metal surely helped him in developing his idea of casting letter shapes into metal pieces. These pieces could then be put together to form the words and be used to print the text. Although his invention allowed the production of books at a lower cost, Gutenberg was certainly preoccupied by the results such an instrument should give. He designed some 300 characters to achieve the same quality that was done manually at that time. His masterpiece was the production of a 42-line Bible in which great harmony between the letters and the spatial constraints was present. Gutenberg's work offered a sound balance between industrialization and artwork.

The advent of computer technology in today's lives has modified the working habits of the societies. Typography did not escape this new trend. New techniques may be offered for the creation of typefaces. Following Rubinstein (1988), we may define this new area accordingly:

> Digital typography is the technology of using computers for the design, prepa-

ration, and presentation of documents, in which the graphical elements are organized, positioned, and themselves created under digital control.

As it followed an evolution based on the technological advances in output media, a wide variety of problems needed to be addressed. Rasterization of characters for low resolution output devices is an example of such concerns.

Tools were developed for the creation of digital fonts. These tools were primarily interactive programs that simulated the use of pen and paper designing techniques, (see Adams and André, 1989). In the creation of a typeface, a consistent design throughout all the characters is fundamental.

An entirely mathematical approach was adopted by Donald Knuth in the development of the METAFONT system (Knuth, 1986a). METAFONT is a computer language that allows a designer to describe the important attributes of an entire font. It is mostly suited for raster-based devices such as printers and computer displays. The only knowledge METAFONT has about characters is the position of the characters in the character set. The initial concept behind the METAFONT language was to describe a character with an arbitrary path. The size and shape of the pen used could then be modified at will to create various styles of letters. As the path is defined by the user, METAFONT would then compute the contour curves according to the pen used. Subsequently, METAFONT was improved to permit the user to describe shapes with outlines as well. The METAFONT user does not need to be concerned with all the mathematical and geometrical aspects of the curves. Instead, the METAFONT system chooses under certain constraints the proper curves to use. More exactly the algorithms used by METAFONT for curve drawing were developed by John Hobby. The interpolating splines rely only on the requirement that the resulting curves be aesthetically pleasing, (see Hobby, 1986). The desired curves are obtained through curvature constraints.

The METAFONT system was used for the generation of the COMPUTER MODERN font family by Donald Knuth. Other typefaces were created with METAFONT , such as PANDORA (Billawala 1989).

The PostScript language is an interpreted programming language with powerful graphic capabilities. It can represent arbitrary shapes. Painting operators are available, text may be integrated with graphics and bitmaps can be manipulated by the language, which follows a postfix notation. More details about PostScript may be found in Adobe (1990a) and McGilton and Campione (1992). A PostScript program may be rendered by any PostScript interpreter driving an output device. There is no limit to the kind of output device that may be used. The rasterization process is done by the interpreter taking into account the type of output device used. This is a major difference with METAFONT where a bitmap of a predetermined resolution specified by the user is created.

Since a PostScript program may create any shape and is rendered specifically for the employed device, the PostScript language is quite suitable for creating typefaces. Scalable fonts may be obtained in this manner. Chapter 7 will explore in more detail the generation of fonts in PostScript. For now, a PostScript font may be created by writing a PostScript program describing each character shape. This method of describing a character is more efficient than storing the entire character bitmap. Three font types are available. A type 3 font is one for which the behavior is entirely controlled by the PostScript language procedures defined in the font program. A type 1 font defines the character shapes by using specially encoded procedures (Adobe, 1990b). Both types are known to define *base fonts*. The type 0, on the other hand, is a *composite font* grouped of *base fonts*. A type 3 font is simpler to create than a type 1 but cannot be stored in the printer's memory. Another advantage of type 1 fonts is the small amount of memory required for describing a font.

Both METAFONT and PostScript use cubic parametric curves (or Bézier curves) to describe the outlines. METAFONT and PostScript have shown to be very versatile. They may be used to complement each other as shown in Haralambous (1993), for the creation of font families. Recent developments in digital typography may be found in Karow (1994a, 1994b).

A new standard in font description is available. Somehow similar to PostScript, TRUE-

TYPE fonts technology is not as general since it only used for describing fonts. TRUETYPE, developed initially by the Apple Computer corporation, is now also used in Microsoft's Windows system. Fonts developed earlier and reconstructed in the TRUETYPE format may be found in Bigelow and Holmes (1991). The description of curves is done via the use of quadratic B-splines.

Gutenberg's invention set the base for typography. As he created uniform and reproducible metal types, concerns of consistency and harmony in character shapes, spacing and alignment became some of the guiding principles used in type design. Recently, some attention has been brought to the creation of typefaces that do not entirely follow established conventions, such as *dynamic type* or *dynamic fonts*. In dynamic fonts, one creates typefaces for which each instance of the font creates a new set of character shapes. Knuth investigated this avenue with METAFONT and created the *meta*-font PUNK (Knuth, 1988). This typeface is not entirely dynamic as METAFONT may be seen as a batch processing system. Other research has tried to mimic the behavior of handwriting where two instances of a character are rarely identical. Some dynamic typefaces take advantage of the system for which they were designed. Caching multiple unique instances of a font and selecting the letters from them is a method for simulating dynamic typefaces. This technique is shown in van Blokland (1991). Other techniques also presented in van Blokland (1991), consist of taking advantage of the PostScript semi-random number generator and modifying at print time the outlines of the characters. This method was also investigated in André and Borghi (1989). Calligraphic dynamic typefaces have been created by noting that the movement executed while writing follows a loop-like motion of varying extend. The HELISCRIPT system, Doojies (1989), simulates this behavior by using three-dimensional helical curves written on the surface of a circular cylinder.

The stated methods allow the generation of printed documents that have a more natural or more human feel. Unfortunately they require some typographical—and in some cases programming knowledge—to generate. In this thesis, we will explore a way of easily creating a typeface that may then be used to give a more personalized look at a document.

The resulting typeface will consist of the user's handwritten characters described in the PostScript type 1 language. The generation of the typeface is entirely automatic—no hand tuning is required.

The following chapters explain how such fonts are obtained. First, the mathematical aspects of Bézier curves will be explored. The proposed algorithm for the creation of character outlines will then be described. A font generator program implementing the proposed algorithm has been developed. Its functionality will be described along with the steps necessary in the creation of a typeface. Examples of handwritten typefaces as well as the behavior of the algorithm under different constraints are looked at. Lastly, the interactions between the new fonts and TEX are considered.

2 Bézier Curves

The generation of scalable fonts in the PostScript type 1 format requires the use of Bézier curves in order to describe the contours of a character. Bézier curves were invented independently by P. de Casteljau around 1959 and by P. Bézier around 1962. In both cases the development of such curves resulted from the need to solve some CAD problems; both were working in French car companies, Citröen and Renault respectively. Although de Casteljau developed these functions first, his work was only discovered in 1975 by W. Böhm, (Farin, 1993), whereas Bézier's work was published soon after the creation of the UNISURF system (Böhm, Farin and Kahmann, 1984). Nowadays, these functions are widely used in CAD systems.

2.1 The de Casteljau algorithm

De Casteljau defines Bézier curves algorithmically. It starts from a degree n and Bézier points $b_0, b_1, \ldots, b_n$. A Bézier curve B consists of $\{b_0^n(t); o \leq t \leq 1\}$, where

$$b_i^r(t) = (1-t)b_i^{r-1}(t) + tb_{i+1}^{r-1}(t) \begin{cases} r & = 1, \ldots, n \\ i & = 0, \ldots, n-r \end{cases}$$

and $b_i^0(t) = b_i$.

The following figure shows the de Casteljau algorithm applied to a Bézier curve of third degree with parameter value $t = 0.6$.

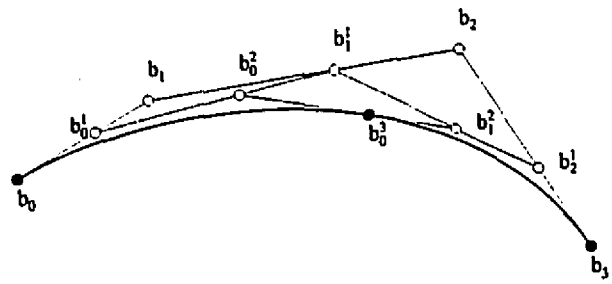


Figure 1. The de Casteljau construction for a Bézier curve of degree three with $t = 0.6$.

By inspection, one will notice that that the intermediate points are all placed in the same relative position, more precisely,

$$\frac{\|b_i^{j+1} - b_i^j\|}{\|b_{i+1}^j - b_i^j\|} = \frac{t}{1-t}$$

for all (i,j) with $0 \leq i, j, i+j \leq n$. Since this construction works for a single value of t, the computation of many such values for a high degree curve may become quite inefficient.

2.2 Bernstein polynomials

Bézier's work, although related to de Casteljau's, takes a different approach. An explicit analytic description of the curves was developed. A mathematical representation of a curve facilitates the understanding and the development of the underlying theory. A Bézier curve $\mathcal{B}$ may be represented in parametric format by

$$\mathcal{B}(t) = \sum_{i=0}^{n} a_i f_i(t), \quad 0 \leq t \leq 1,$$

where $a_i \in \mathcal{V}$ and $\mathcal{V}$ is a vector space. The choice of the functions f_i influences the behavior of $\mathcal{B}$. Hence in order to define the family of f_i's one needs to pay attention to the properties that $\mathcal{B}$ should have. Concerned with the ability to determine endpoints and derivatives at endpoints, Bézier carefully picked his functions f_i. Assume that the initial point of the curve is a_0 and the tangent at that point is parallel to the vector a_1. Similarly assume that the last point must be $\sum_{i=0}^{n} a_i$ while the tangent at that point is parallel to the vector a_n. In the case of surfaces, the tangent planes at the endpoints of the curves are also important, extra restrictions were put at those points of interest. The osculating planes at the first and last points of the curve must be parallel to the vectors a_0, a_1 and a_{n-1}, a_n respectively. The function $\mathcal{B}$ defines a curve according to a certain order. If the order needs to be reversed, then the functions must be symmetric with respect to t and $1-t$. Finally the derivatives f_i^j of order j for the functions f_i must be such that

$$f_i^j(0) \begin{cases} \neq 0 & 3 \leq j \leq n, \ 1 \leq i \leq j \\ = 0 & 3 \leq j \leq n, \ i > j \end{cases},$$

and

$$f_i^j(1)\begin{cases}\neq 0 & 3 \le j \le n,\ n-j < i \le n \\ = 0 & 3 \le j \le n,\ i \le n-j\end{cases}.$$

This ensures that the curve is tangent at its endpoints to the first and last non-zero vectors a_i.

The family of functions that satisfy all the above requirements has the form

$$f_i(t) = \frac{(-t)^i}{(i-1)!}\frac{d^{i-1}\phi_n}{dt^{i-1}}$$

with $\phi_n = (1-t)^n - 1/t$. These functions can be expressed in the form

$$f_{i_n}(t) = (-1)^{p+1}\sum_{p=0}^{n}\sum_{i=0}^{n}\binom{p-i}{p-1}\binom{p}{n}t^p.$$

Although this family of functions gives exactly what was intended, a different notation can be used to simplify the use of Bézier curves. The new notation adopts the endpoints of the vectors instead of the vectors themselves. This is shown in the following figure.

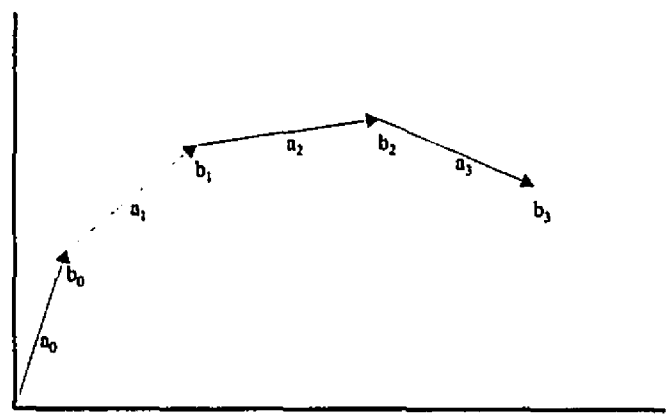


Figure 2. The correspondence between the endpoint notation and the vector notation.

Hence $b_i = \sum_{j=0}^{i} a_j$ for $i = 0, \ldots, n$ and the Bézier curve can be written as

$$B(t) = \sum_{i=0}^{n} b_i g_i(t),$$

where the functions g_i are derived from the f_i's. After some algebraic manipulations, we see that

$$g_i(t) = \binom{n}{i} t^i (1-t)^{n-i}.$$

Hence the Bézier curves are based on Bernstein polynomials. The relation between Bézier's work and Bernstein polynomials was only discovered in 1970 by R. Forrest (see Böhm, Farin and Kahmann, 1984). From here on, $B_i^n(t)$ will be used instead of $g_i(t)$ to denote these polynomials.

Let us mention a few additional properties of Bernstein polynomials. For a given degree n, there are $n+1$ Bernstein polynomials, namely $B_0^n, \ldots, B_n^n$. This family of functions satisfy the recursive relation

$$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) \text{ for } i = 0, \ldots, n\,,$$

with

$$B_0^0(t) \equiv 1 \text{ and } B_j^n(t) \equiv 0 \text{ for } j \notin \{0, \ldots, n\}.$$

In a similar fashion, the derivative of a Bernstein polynomial is

$$B_i^{n\prime}(t) = n(B_{i-1}^{n-1}(t) - B_i^{n-1}(t)) \text{ for } i \in \{0, \ldots, n\}.$$

Furthermore, Bernstein polynomials define a *partition of unity*, as $\sum_{j=0}^{n} B_j^n(t) \equiv 1$. Finally, $B_i^n \geq 0$ on $[0,1]$. The following figure shows the quintic Bernstein polynomials.

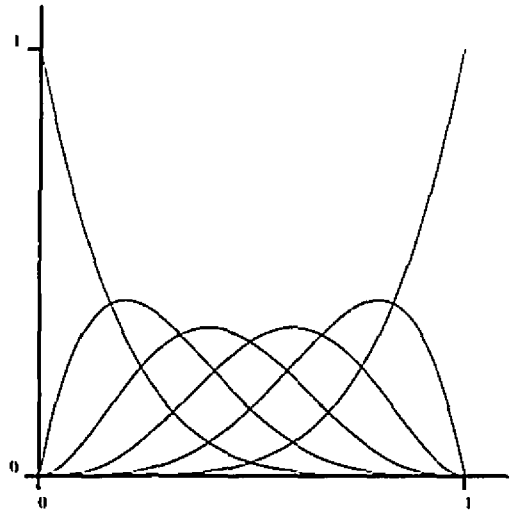


Figure 3. Quintic Bernstein polynomials.

Let us consider $\mathcal{P}_n$, the linear space of polynomials of degree $\leq n$. The dimension of $\mathcal{P}_n$ is known to be $n+1$. Since the $n+1$ Bernstein polynomials are linearly independent, they form a basis of $\mathcal{P}_n$ (Hoffman and Kunze, 1971).

The way Bézier curves were defined makes the behavior of such curves predictable. The curve $\mathcal{B}$ follows a path p from the point b_0 to the point b_n. By joining the endpoints of consecutive vectors b_j and b_{j+1} with line segments, which consist of the sequence $a_1, \ldots, a_n$, we construct a polygon that is called the *characteristic polygon*. There is a close relation between the Bézier curve and the characteristic polygon. For example, Bernstein's approximation theorem shows that given a continuous function f defined over the interval $[0,1]$ and a function $B_n(x) = \sum_{k=0}^{n} f(\frac{k}{n})\binom{n}{k}x^k(1 \cdot x)^{n-k}$ then the function B_n converges uniformly on $[0,1]$ to the function f as $n \to \infty$ (Bartle and Sherbert, 1982). Hence a Bézier curve approximates its characteristic polygon and offers the ability to the user to easily construct the desired curve and to predict its behavior. As mentioned above the Bernstein polynomials form a partition of unity, therefore $\mathcal{B}(t)$ is actually a weighted mean of the vertices b_j of the polygon. Hence the path that $\mathcal{B}(t)$ follows is only dependent on the

vertices of the characteristic polygon. Moreover, the path $B(t)$ must lie entirely inside the convex hull of the characteristic polygon. This fact can be used to perform a preprocessing stage when the intersection of two Bézier curves needs to be computed.

The following figure represents a Bézier curve of degree 5.

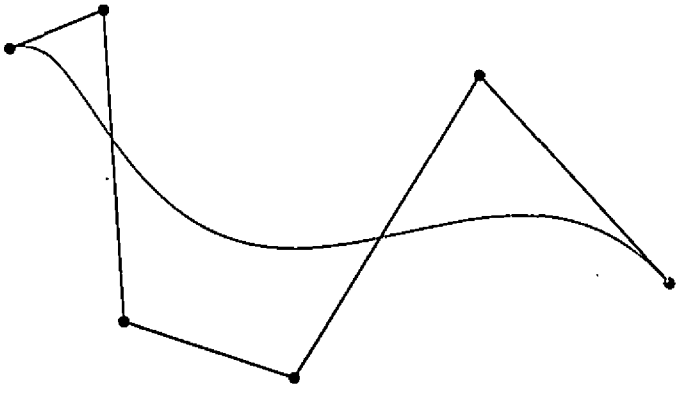


Figure 4. Quintic Bézier curve and its characteristic polygon.

2.3 Bézier derivatives

The derivative of a Bézier curve is

$$\begin{aligned} B'(t) &= \sum_{i=0}^{n} b_i B_i^{n'}(t) \\ &= \sum_{i=0}^{n} b_i n (B_{i-1}^{n-1}(t) - B_i^{n-1}(t)) \\ &= n \sum_{i=0}^{n-1} (b_{i+1} - b_i) B_i^{n-1}(t) \end{aligned} .$$

This shows that the derivative is also based on Bernstein polynomials but it is no longer a Bézier curve. This formula gives rise to a more general one for computing high order derivatives. Let us define the *iterated forward difference* operator Δ^r of a point b_j as $\Delta^r b_j = \Delta^{r-1} b_{j+1} - \Delta^{r-1} b_j$. Note that Δ^r can be written as $\Delta^r b_i = \sum_{j=0}^{r} \binom{r}{j} (-1)^{r-j} b_{i+j}$.

The r-th derivative of a Bézier curve of degree n can then be expressed as

$$\frac{d^r}{dt^r}\mathcal{B}(t) = \frac{n!}{(n-r)!}\sum_{j=0}^{n-r}\Delta^r b_j B_j^{n-r}(t).$$

The r-th derivative formula becomes very simple at $t = 0$ and $t = 1$, reducing to $n!/(n-r)!\Delta^r b_0$ and $n!/(n-r)!\Delta^r b_{n-r}$, respectively. This shows that the r-th derivative at the endpoints of a Bézier curve depends only up on the $r+1$ points adjacent to the endpoint.

2.4 Subdivision

Besides for the computation of a point, the de Casteljau algorithm can also be used to subdivide a Bézier curve into two parts. Given a polygon $b_0, \ldots, b_n$, the construction evaluates in particular the points $b_0^0, b_0^1, \ldots, b_0^n$ and $b_0^n, b_1^{n-1}, \ldots, b_{n-1}^1, b_n^0$ which describes two new polygons. This subdivision process produces two Bézier curves $\mathcal{B}_0$ and $\mathcal{B}_1$ from $\mathcal{B}$. The relation between the curves corresponds to the evaluation point t. Hence if $t = c \in [0,1]$ then $\mathcal{B}_0$ and $\mathcal{B}_1$ correspond to the intervals $[0,c]$ and $[c,1]$, respectively, of $\mathcal{B}$ (Farin, 1993; Su and Liu, 1989).

2.5 Third degree Bézier curves

The Bézier curves of third degree being used by the instruction `curveto` of the PostScript language are described in this section. Given the current point (x_0, y_0), `curveto` takes the three points $(x_1,y_1),(x_2,y_2),(x_3,y_3)$ as parameters. The curve $\mathcal{B}(t) = (x(t), y(t))$ that results from it can be written using the monomial basis as

$$\begin{aligned} x(t) &= a_x t^3 + b_x t^2 + c_x t + x_0 \\ y(t) &= a_y t^3 + b_y t^2 + c_y t + y_0 \end{aligned}$$

where

$$\begin{aligned} a_x &= x_3 - 3(x_2 - x_1) - x_0 & a_y &= y_3 - 3(y_2 - y_1) - y_0 \\ b_x &= 3(x_2 - 2x_1 + x_0) & b_y &= 3(y_2 - 2y_1 + y_0) \\ c_x &= 3(x_1 - x_0) & c_y &= 3(y_1 - y_0) \end{aligned} .$$

Equivalently,

$$\begin{aligned} x(t) &= x_3t^3 + 3x_2t^2(1-t) + 3x_1t(1-t)^2 + x_0(1-t)^3 \\ y(t) &= y_3t^3 + 3y_2t^2(1-t) + 3y_1t(1-t)^2 + y_0(1-t)^3 \end{aligned}$$

in the Bernstein polynomial format. The monomial form of a Bézier curve allows the computations to be performed with Hörner's method. A Bézier curve of the third degree takes one of four possible shapes as shown below.

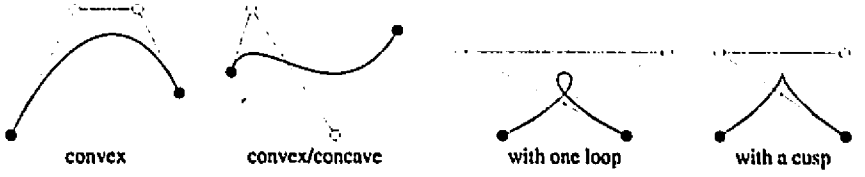


Figure 5. Third degree Bézier curves.

2.6 Bézier splines

A polynomial of degree n can have as many as $n-2$ inflection points while a parametric curve of the same degree can have as many as $2n-4$. Small degree n limits the complexity of a curve. Higher degree curves increase the complexity. One way to overcome this deficiency is to define *piecewise curves* by joining two curves of lesser degree. The junction point of the two segments is known as a *knot* or a *breakpoint*.

A spline S, composed of two adjacent Bézier curves $\mathcal{B}_0$ and $\mathcal{B}_1$ is created. Each curve has its own *local parameter* t while S has a *global parameter* u, where $u \in R$. The knot sequence can be represented in terms of the parameter u, knot i having parameter value u_i. The correspondence between t and u depends on the actual length of each segment, $t = (u-u_i)/(u_{i+1}-u_i)$. We can think of $\mathcal{B}_0$ and $\mathcal{B}_1$ as two independent curves each having a local parameter t ranging from 0 to 1 or we may regard it as two segments of a composite curve with parameter u in the domain $[u_0, u_2]$. Aesthetically pleasing composite curves

are obtained by introducing continuity restrictions and applying smoothness conditions to S (Manning, 1974).

Let us assume that $\mathcal{B}_0$ has Bézier points $b_0, \ldots, b_n$ and $\mathcal{B}_1$ as Bézier points $b_n, \ldots, b_{2n}$. If $\mathcal{B}_0$ and $\mathcal{B}_1$ were the result of a subdivision process then we know that $b_{n+i} = b^i_{n-i}(t)$ would hold for $i = 0, \ldots, n$ and $t = (u_2 - u_0)/(u_1 - u_0)$. If we move the point b_{2n}, the curve changes shape but from the r-th derivative formula the derivatives up to order $n-1$ coincide with the original Bézier derivatives. Since we assumed that $\mathcal{B}_0$ and $\mathcal{B}_1$ occurred after a subdivision, these derivatives coincide at the junction point. The C^r condition for Bézier curves can be stated as: two Bézier curves defined over the intervals $[u_0, u_1]$ and $[u_1, u_2]$ by the polygons $b_0, \ldots, b_n$ and $b_n, \ldots, b_{2n}$ respectively are said to be r times continuously differentiable at $u = u_1$ if and only if

$$b_{n+i} = b^i_{n-i}(t). \qquad i = 0, \ldots, r,$$

where $t = (u_2 - u_0)/(u_1 - u_0)$.

Usually, $r \in \{1, 2\}$. When $r = 1$, Δb_n and Δb_{n-1} must be in the ratio Δ_1/Δ_0, $\Delta_i = u_{i+1} - u_i$, in order to obtain C^1 continuity at $u = u_1$. This condition can be derived from the fact that the derivatives of $\mathcal{B}_0$ and $\mathcal{B}_1$ must coincide at $u = u_1$. Hence

$$\frac{d}{du}s(u) = \frac{1}{\Delta_0}\frac{d}{dt}\mathcal{B}_0(t) = \frac{1}{\Delta_1}\frac{d}{dt}\mathcal{B}_1(t)\,.$$

Since

$$\begin{array}{rcl} \frac{d}{dt}\mathcal{B}_0(1) & = & n\Delta b_{n-1} \\ \frac{d}{dt}\mathcal{B}_1(0) & = & n\Delta b_n \end{array},$$

we get the above desired result.

When $r = 2$, the points $b_{n-2}, b_{n-1}, b_n, b_{n+1}, b_{n+2}$ influence the second derivative at the junction point. If the curve S is C^2 then there must a point d of a polygon b_{n-2}, d, b_{n+2} that describes the same global quadratic polynomial as the 5 points, mentioned above, do. Hence assuming that the curve is already C^1, the following equations must be satisfied in order for d to exist:

$$\begin{array}{l} b_{n-1} = (1 - t_1)b_{n-2} + t_1 d \\ b_{n+1} = (1 - t_1)d + t_1 b_{n+2} \end{array},$$

where $t_1 = \Delta_0/(u_2 - u_0)$. The conditions for C^1 and C^2 curves are shown in the following figure.

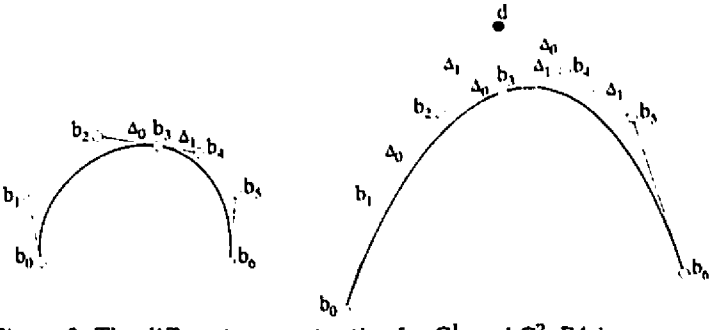


Figure 6. The different segment ratios for C^1 and C^2 Bézier curves.

2.7 Curvature

The curvature of a function describes how the function changes orientation with respect to the distance traveled on that function. More specifically, it measures the rate of change of the angle through which the tangent to the curve turns in moving along the curve. In particular, for a line, the curvature is zero since the angle θ is constant and for a circle the curvature is constant and is inversely proportional to the radius of the circle. Hence if the curvature at a point p of a curve C is κ we can pass a circle of radius $1/\kappa$ through p (Swokowski, 1975).

In the case of parametric equations, the curvature can be calculated as

$$\kappa = \left|\frac{d\theta}{ds}\right| = \frac{\left|\frac{d\theta}{dt}\right|}{\left|\frac{ds}{dt}\right|}.$$

More precisely, for a curve $(x(t), y(t))$, the curvature is defined as

$$\kappa = \frac{\left|x'(t)y''(t) - y'(t)x''(t)\right|}{[(x'(t))^2 + (y'(t))^2]^{\frac{3}{2}}}.$$

For a Bézier curve $\mathcal{B}$ of third degree the curvature $\kappa_{\mathcal{B}}$ at a point t can be calculated as

$$\kappa_{\mathcal{B}}(t) = \frac{\left|6(a_x b_y - a_y b_x)t^3 + 6(a_y b_x - a_x b_y)t^2 + 6(a_y c_x - a_x c_y)t + 2(b_y c_x - b_x c_y)\right|}{\left[(3a_x t^2 + 2b_x t + c_x)^2 + (3a_y t^2 + 2b_y t + c_y)^2\right]^{\frac{3}{2}}}$$

and the total curvature of the Bézier curve $\mathcal{B}$ is

$$\int_0^1 \kappa_{\mathcal{B}}(t)dt.$$

3 From bitmap to outline font

In the development of a new typeface, typographers are concerned with legibility, uniformity among the characters and aesthetics. For handwritten characters, however, the major concern is with the accurate reproduction of the contours.

In order to achieve a satisfactory accuracy a methodology is proposed and developed in this thesis. This section will explore a new method for automatically generating a handwritten typeface.

Different input devices may be used to transform all the information hidden in a single pen stroke into an appropriate computer format. Some have used a stylus with a digitizing tablet for seizing the characteristics of hand drawn images (Pudet, 1993). Handwriting varies according to the instruments utilized. Common tools such as pen and paper are most convenient to use. As scanner technology is perfected and more affordable, it 's a viable solution for collecting the relevant information and transforming it in the desired form. Generally, the software driving a scanner will generate bitmap images. A multitude of bitmap formats, such as TIFF, GIF, EPS, BDF and PPM, are available. Only the shape of characters is pertinent to our problem. The different bitmap formats are, de facto, equivalent. Many programs for converting from one format to another are also available—see, e.g., PBMPLUS package. Written by Jef Poskanzer, PBMPLUS is a comprehensive format conversion and image manipulation package. We will describe in chapter 4, the entire process required for generating a typeface. For now, assume that we are given a 2-dimensional array of black/white pixels for a given character (see figure 7 below).

A new representation is required in order to generate a truly scalable font. Bézier curves may be used to approximate the outlines of the original characters. As indicated in Schneider (1990), many kinds of interpolating curves have been used in curve-fitting problems. In general, multiple curves are necessary for approximating a desired shape. They define an interpolating set. The number of such curves may vary between characters of a same typeface and is not, a priori, known. Also, as the scanner resolution may be selected by the user, the number of pixels describing a character is not fixed. The set of

points necessary to determine the interpolating functions is also unknown.

The interpolation of points by curves has been the subject of some extensive research. Not only restricted to the realm of typography, curve-fitting problems were investigated before the advent of computer graphics. For example, in the early eighteen hundreds, ship designers were concerned with the manual generation of smooth curves that go through a set of points. The creation of characters for a new typeface is a similar type of problem.

Some criteria that measure the accuracy of the interpolation must be established. For example, such criteria might take into consideration the difference between the original and generated character. As fonts represented in the PostScript type 1 format may be stored in the printer's memory, it is also preferable to minimize the size of the typeface's description. The process of scanning the handwritten characters may induce imperfections that must be removed. Within the aforementioned restrictions, we may wish to obtain a font that minimizes the number of curve segments in a character's description and the discrepancy between the generated and original characters, while at the same time preserving smooth pleasing outlines. Some of these criteria may conflict with each other. For example, a small number of curve segments can induce large differences between the source and target characters. These constraints form the basis of a quality function Φ that may be used to evaluate the interpolation. The optimal interpolation occurs when Φ is minimum. The choice of Φ is entirely subjective and reflects the font designer's taste as well as the user's demands.

The design of a new typeface is based around a group of specific criteria set by the typographer. Hence one may view a typographer's typeface as a weighted optimum, each different typeface having its own weight. For handwriting, the precise weight is not known a priori. Minimizing Φ may be achieved iteratively. Roughly, we have:

```
for each character do
    extract contours
    create initial splines
    perform the following n times
        modify part of a spline
        compute the quality function Φ
        if modification is acceptable
        then
            accept modification
        else
            reject modification
```

We will deal with each aspect of this algorithm separately.

3.1 Contour extraction

As a symbol is represented by a bitmap, many *black* pixels are required to form the character. Some pixels are "interior" and others constitute the "contour" of the characters. Since the objective is to use curves for expressing the character outlines, the contour pixels are more important and need to be differentiated from the others.

Extraction of contour pixels or points may be achieved in different ways. Some algorithms are established according to the type of bitmap. For grey-level images, Avrahami and Pratt (1991) developed a contour extraction algorithm. This algorithm was modified and used in Itoh and Ohno (1993). A different contour-tracing algorithm derived from algorithms designed to verify connectedness of components (Minsky and Papert, 1969) has been employed by Gonczarowski's algorithm (Gonczarowski, 1991). Algorithms performing contour extraction are commonly used in the area of pattern analysis and recognition. A simple algorithm, that works for all bitmap images, known as Moore's tracing algorithm, can be found in the literature (Pavlidis, 1982).

A contour pixel can be defined as having at least one *white* pixel as a neighbor in a 4-connected representation. Neighbors are sometimes referred to by their relative position, **north, east, south, west** or simply **N, E, S, W**. A pixel can be part of more than

one contour. More than one contour may be present in a character. The following figure illustrates the contour pixels of a mock handwritten character.

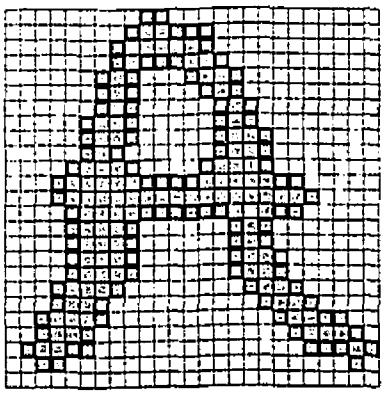


Figure 7. Contour pixels of a character.

The following simple algorithm finds the contour pixels in an $m \times n$ bitmap in time $\Theta(mn)$.

```
/* Proceed horizontally */
for i = 1 to m do
    for j = 1 to n do
        if pixel (i,j) is black and
           (pixel (i,j-1) is white or pixel (i,j+1) is white)
           then
                pixel (i,j) is a contour pixel

/* Proceed vertically */
for j = 1 to n do
    for i = 1 to m do
        if pixel (i,j) is black and
           (pixel (i-1,j) is white or pixel (i+1,j) is white)
           then
                pixel (i,j) is a contour pixel
```

The above algorithm finds all the contour pixels, in matrix format. It is sometimes more convenient to have a representation in which contour pixels are linked together in

a chain or chains. In general, the pixels of a contour form a closed path. For the more degenerate cases, such as contour sections having width 1, the pixels can be considered twice in order to form the closed path. The linearization of a contour relies on the relations between neighboring contour pixels. Each contour pixel has at least one white pixel as a neighbor. Two neighboring contour pixels share at most two common white pixels as a neighbor in a 8-connected representation. These two facts are sufficient for defining a set of white pixels bordering the path of contour pixels. Each white pixel of the set is the neighbor of at least one contour pixel of the considered contour. The set of white pixels defines also a path which is composed of only 4-connected pixels, as shown by a partial set in part (a) of the following figure.

The use of contour pixels in conjunction with the path of white pixels mentioned above allows the construction of an algorithm that builds the desired ordering. Some earlier contour-following algorithms (Duda and Hart, 1973), do not create the correct ordering for some 8-connected images. The result of such algorithms is shown in part (b) of the following figure, while part (c) shows the correct chain.

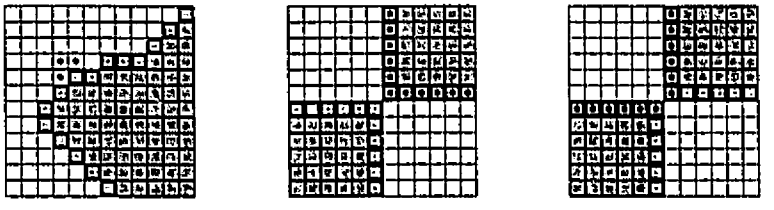


Figure 8. Ordering of contour points.

The algorithm for linearization is given below

```
find a contour pixel p
let w_p be a white pixel that is a 4-connected neighbor of p
let Flag = TRUE
while Flag == TRUE do
   let Flag = FALSE
   while all 8-connected contour pixel neighbors n of p
         are not yet tested and Flag == FALSE do
      while all 4-connected white pixels neighbor w_n of n
            are not yet tested and Flag == FALSE do
      if there is a 4-connected path of white pixels not yet
         visited from w_p to w_n
      then
         mark path w_p to w_n as visited
         let the pixel n be the current pixel p
         let the white pixel w_n be the current pixel w_p
         let Flag = TRUE
```

The above linearization algorithm traces a single outline and is linear in terms of the number of pixels in the contour. To guarantee that all outlines are found, the visited pixels are marked and the search for another outline can be started by considering unvisited pixels. The search may simply be done by scanning the bitmap in an up-down, left-right fashion. The algorithm for the linearization of all contours of a bitmap is thus linear with respect to the number of pixels in the bitmap, and resembles in some respect depth-first search (Cormen, Leiserson and Rivest, 1990).

3.2 Interpolation

Once the contour pixels are determined, a set of interpolating curves is defined. We first review the relevant literature. As knots define the endpoints of curves, dynamic programming methods may be used to find a good knot partitioning as in Plass and Stone (1983). A modified version of it, presented in Schneider (1990), consists of replacing the heuristic by a subdivision process that breaks the curve where the maximal error occurs. Other approaches perform first corner detection to define an initial set of knots. Corner detection

consists of interpreting the bitmap to find locations where the contour changes direction abruptly. Between two consecutive corners, a certain number of knots may be defined. An iterative approach, used in Gonczarowski (1991), consists of finding the longest curve from a given point such that it approximates the desired section of the bitmap with a user-specified threshold. As mentioned in Itoh and Ohno (1993), the precise detection of contour points is a very hard problem. The algorithm of Itoh and Ohno uses the estimated corner points for defining segments. Approximate curves for a given segment are found by a reparameterization process from a starting interpolating curve. The reparameterizations occur until a satisfactory interpolation is found.

The approximation of pixels contours by curves is sometimes referred as *auto-tracing*. Some commercial packages perform such an operation. Some of the better known software products are Adobe Illustrator, Fontographer and a freely available package called `fontutils` that may be found on some UNIX systems.

The method used in this algorithm is quite different from the ones mentioned above. The pixels composing a contour may be grouped to define a sequence of sections. These sections will be used in the construction of the approximating Bézier curves. The sections are derived from the approximation of the contour with lines. Each section corresponds to a line defined by the centers of two contour pixels as endpoints. Moreover, the section divides the black pixels from the white pixels of the bitmap, and is thus an approximation with zero error. The way the line segments are determined relies on a simple fact about pixels.

> **Pixel coverage property:** A pixel is said to be black if at least half of it is covered. Moreover if a line goes through the middle point of a pixel, the pixel is divided into two parts of same area.

The METAFONT system uses this property during the curve rasterization process, (see Knuth, 1986a). With this particularity, a line can be drawn on a portion of the contour without altering the color of the pixels. Section i, starting at point d_i (which is the center of a black pixel), can be delimited by a point d_{i+1} creating the longest line segment $d_i - d_{i+1}$

of such kind. That is, adding one more pixel would result in an imperfect approximation. Hence once all the sections are determined, the lines define a perfect representation of the characters in terms of the pixel colors.

Figure 9. Section partitions of an outline portion.

The smallest possible section is delimited by two adjacent pixels. The construction of section i is done iteratively from this base case. At each step of the the process a longer line is tested until no line satisfying the restrictions can be found. We will see that only one such line need to be verified and the incremental test can be done in $O(1)$ worst-case time. Hence the overall time is bounded by the number of contour pixels.

The simple case with two adjacent pixels is shown in part a of the figure 10.

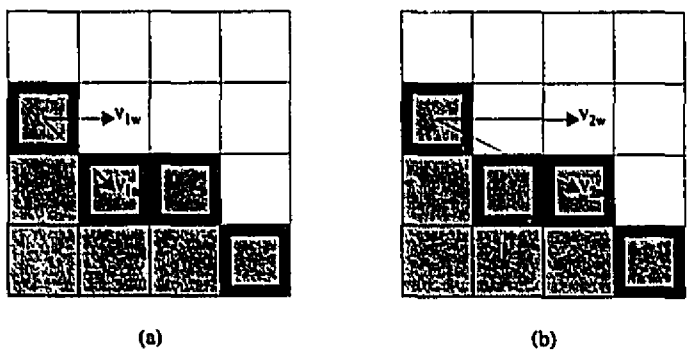


Figure 10. Initial steps of the sectioning algorithm.

Two vectors v_{1n},v_{1w}, are formed from the point d_i, one with the adjacent black pixel n_1 and the other with the white pixel w_1 bordering n_1. Assuming that there exists a longer line, the new line segment must be from d_i to n_2 where n_2 is the neighbor of n_1. Furthermore the line segment d_i-d_{i+1} must be between v_{1n} and v_{1w}, otherwise the line would inadequately separate the border of white and black pixels found so far between pixels d_i and n_1. With a valid line, the vectors can be recalculated as follows:

$$\begin{aligned} v_{2b} &= n_2 - p_i \\ v_{2w} &= \begin{cases} w_2 - p_i & \text{if } \|v_{1n} \times (w_2 - p_i)\| = \|(w_2 - p_i) \times v_{1w}\| \\ v_{1w} & \text{otherwise.} \end{cases} \end{aligned}$$

The result of a vector recalculation example is shown in part (b) of figure 10. The angle between the two vectors gets smaller as the process continues. The algorithm stops when no longer line can be found. Here is the algorithm.

```
/* Define a section from point d_i */
let n be the next neighbor of d_i in the linear representation
let w be a white pixel adjacent to d_i and n
let δ be the relative direction of w from n (N,E,S,W)
compute v_n and v_w
let flag = TRUE
while ( flag == TRUE ) do
    let ν be the next contour pixel of n
        in the linear representation
    if ||v_n × (ν − d_i)|| == ||(ν − d_i) × v_w|| then
       compute v_n and v_w
       let n = ν
    else flag = false
let the endpoint d_{i+1} be pixel n.
```

The above sectioning algorithm is linear in terms of the number of contour pixels.

3.3 C^2 spline construction

The continuity conditions enumerated in section 2.6 allow us to determine if a sequence of Bézier splines is C^2 continuous or not. Böhm presented an algorithm that produces such

splines (Böhm, 1977).

Before looking at Böhm's construction, we introduce some notation. Let the spline S have knots at $u = u_i$ for $i = 0, \ldots, m$ with respect to the global parameter $u \in R$. Let the Bézier points of a spline be written in the form b_{3i+j} where $j = 0, 1, 2$ and i is the Bézier section number. Hence the four Bézier points of the i-th Bézier curve are given by b_{3i}, b_{3i+1}, b_{3i+2} and b_{3i+3}. Given a set of points d_i for $i = 0, \ldots, m$, Böhm's algorithm determines the location of the Bézier points such that the C^2 constraints are satisfied. The resulting spline does not necessarily go through the points d_i. These points are used for the construction of the spline. Initially, the algorithm divides the line segments joined by d_i and d_{i+1} for $i = 0, \ldots, m-1$. The segments are divided by the Bézier points b_{3i+1} and b_{3i+2} to create three parts that are in the ratio $\Delta_i : \Delta_{i+1} : \Delta_{i+2}$, where $\Delta_i = u_i - u_{i-1}$. Once this step is performed, all the inner Bézier points are placed. The endpoints need a bit of care. The point b_{3i} is placed such that the line b_{3i-1}, b_{3i+1} is divided in the ratio $\Delta_i : \Delta_{i+1}$. This last step corresponds to the conditions enumerated previously for having a C^1 continuity at the knots. The first step ensures C^2 continuity.

If the points d_i describe an open polygon, that is $d_0 \neq d_m$, the notation needs to be modified in order for the algorithm to work at the endpoints. The points will be relabeled as $d_{-1}, \ldots, d_n$, where $n = m - 1$. And the endpoint conditions become $b_0 = d_{-1}$, $b_1 = d_0$, $b_{3(n-1)} = d_n$ and $b_{3(n-2)-2} = d_{n-1}$. Note that the resulting curve starts and ends at a point d_i, as shown in figure 11. Alternatively, if the described polygon is closed, $d_0 = d_m$, the operations simply need to be performed in modulo m. In this case the curve doesn't necessarily pass through the point d_0. By changing a point d_i, the resulting curve gets altered. Moreover, the modifications on the curve are local. The sections i and $i-1$ get modified and so does the associated Bézier curves of the sections $i-2$, $i-1$, i and $i+1$.

The Bézier spline can be entirely described in terms of the d_i's and Δ_i's. The points d_i correspond to the section endpoints defined in the previous section. Examples of sequences open-ended and closed polygons are shown below.

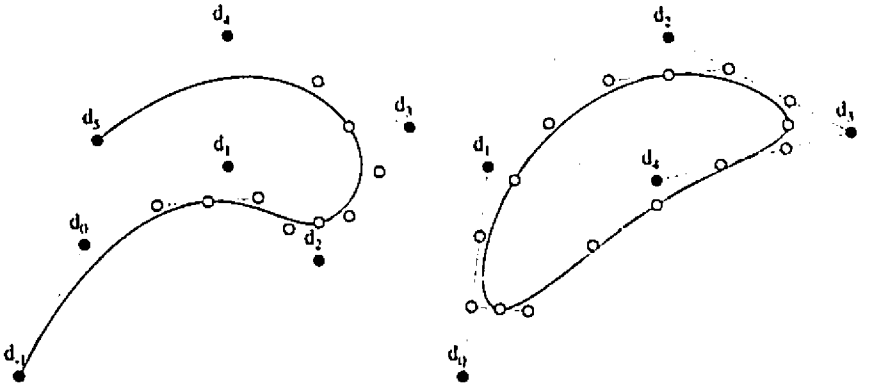


Figure 11. Böhm's C^2 construction algorithm. The b_i's are represented by white circles.

A parameterization can be adopted for the spline by selecting the values of the u_i's. If $u_i = i$ for all i, then $\Delta_i \equiv 1$, and this parameterization is said to be *uniform* or *equidistant* (Farin, 1993). The above figure 11 uses such a parameterization. A different type of parameterization takes into account the distance between the data points. This parameterization is called *chord length* and can be expressed as

$$\frac{\Delta_i}{\Delta_{i+1}} = \frac{\|\Delta S(u_i)\|}{\|\Delta S(u_{i+1})\|},$$

where $\|\Delta S(u_i)\|$ represent the distance between the points corresponding to the parameter values u_{i-1} and u_i. The chord-length parameterization is used in Schneider (1990), Itoh and Ohno (1993) and Plass and Stone (1983).

The uniform parameterization is used in the proposed algorithm as well as for the figures 12 and 13. This simple construction creates a Bézier spline whose shape resembles that of the polygon formed by the d_i's. In some cases, the distance between the polygon and the Bézier spline can be excessive as shown in figure 12a. A possible solution consists of duplicating some of the points d_i, creating some sections of zero length. We may replace the sequence $\{d_j\}$ by a sequence in which one or more of the d_j's are repeated. For example figures 12b and 12c show the Bézier splines for $d_0, d_0, d_1, d_1, d_2, d_2, \ldots$ and $d_0, d_0, d_0, d_1, d_1, d_1, \ldots$ respectively. While the splines approach the initial polygon, we may

no longer have C^2 continuity.

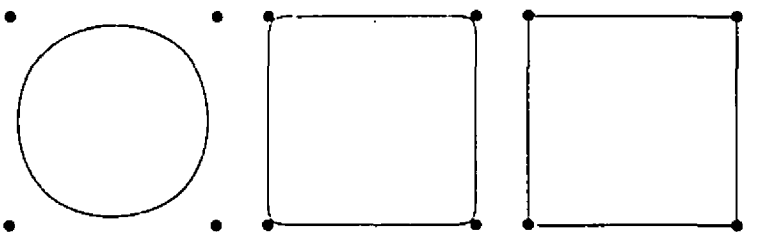


Figure 12. C^2 construction with multiplicities of 1,2 and 3 respectively.

By cutting a Bézier section into two parts, we may obtain new interesting splines. In figure 13b, each section of a Bézier spline was divided into three parts, in the proportions 0.3, 0.4 and 0.3. For figure 13c, the ratios are 0.1, 0.8, 0.1. Although this operation increases the number of Bézier sections we will see later that this operation is worthwhile.

As mentioned above, some algorithms perform corner detection. Our method overcomes the necessity of interpreting pixel configurations for finding the corners. Also, the spline obtained by section splitting may be a better interpolant. Also, more freedom is available in the interpolation process.

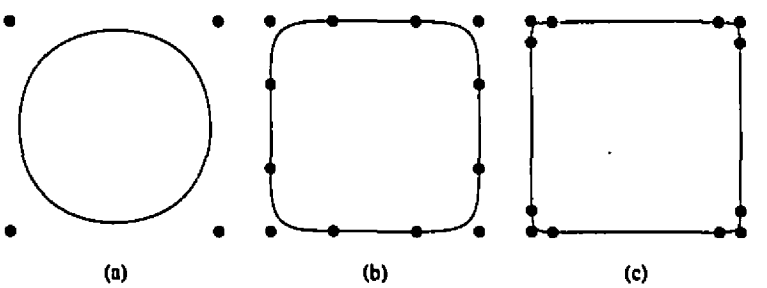


Figure 13. C^2 construction with section splitting.

3.4 The merge and move operations

A *merge* operation consists of replacing two adjacent Bézier curve segments by a single one. Since the C^2 spline constraint is always present, the merge is executed by replacing two adjacent sections s_j and s_{j+1} with one, namely s, simply by defining the new section with the endpoints of the polysegment s_j-s_{j+1}. The total number of sections in the contour is thus decreased by one. The sequence of sections, $s_0, \ldots, s_{j-2}, s_{j-1}, s_j, s_{j+1}, s_{j+2}, \ldots, s_n$ for a given contour becomes after the merge $s_0, \ldots, s_{j-2}, s_{j-1}, s, s_{j+2}, \ldots, s_n$. This new sequence of sections may be used to recompute the corresponding Bézier curves. Figure 14 shows different section configurations and how the Bézier spline changes shape as the merge operation is completed, the dotted curves representing the achieved result. The white section knot represents the junction point of sections s_j and s_{j+1}.

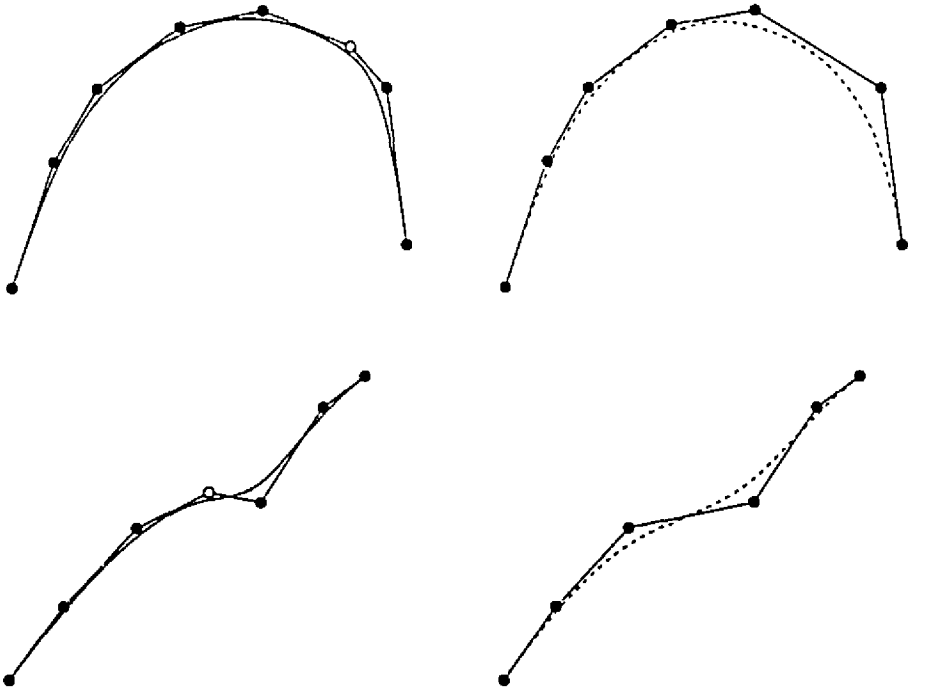


Figure 14. The modifications resulting from a merge operation on two different section polygons. The black dots represent the section knots. The white section knot corresponds to the junction point of the two section that will be merged.

A *move* operation moves a section endpoint a certain distance away from its current location, causing two sections to be modified. Figure 15 shows how a move can affect the Bézier spline. The new and old curves are represented in the same manner.

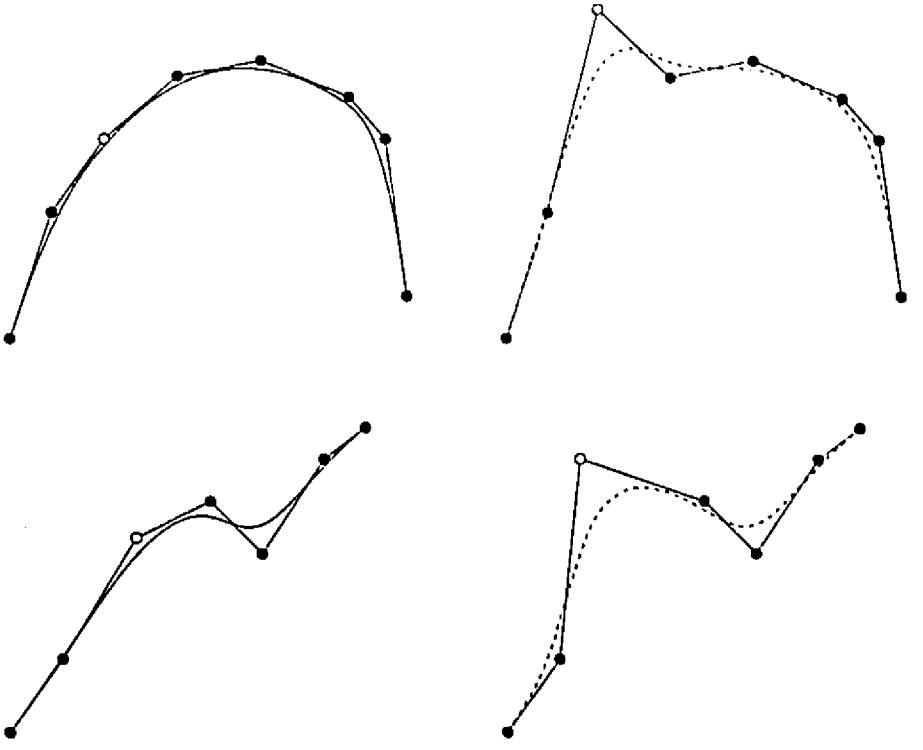


Figure 15. The modifications resulting from a move operation on two different section polygons. The black dots represent the section knots. The white dot corresponds to the point considered for the move operation.

The modification that results from the merge and move operations is according to Böhm's construction. Let us consider the Bézier curve segments $\mathcal{B}_0$, $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$, $\mathcal{B}_4$ of a spline S and see how they change with each operation. If $\mathcal{B}_2$ and $\mathcal{B}_3$ are merged into $\mathcal{B}$, then only the curves associated to $\mathcal{B}_1$, $\mathcal{B}_4$ and $\mathcal{B}$ need to be recomputed. Similarly if a move is performed on the junction point of sections $\mathcal{B}_2$ and $\mathcal{B}_3$, the affected curve segments are $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$, and $\mathcal{B}_4$. These operations ensure the locality of the modifications on a spline S.

A merge operation can perturbate the curve significantly (see figure 14). As it lowers the number of sections in a character, the sections become longer. It allows the removal of small imperfections introduced during the input process. The merge is most efficient

on consecutive sections that do have more or less the same orientation. It will force long segments of consecutive pixels to be approximating by very few curves whereas pixels describing a curve will be approximated with more Bézier curves.

3.5 Optimization of the outline

After having scanned and linearized the bitmap, we have for each character a description of the character in the form of line segments. An example of such a description is shown in the following figure on the mock handwritten character from section 3.

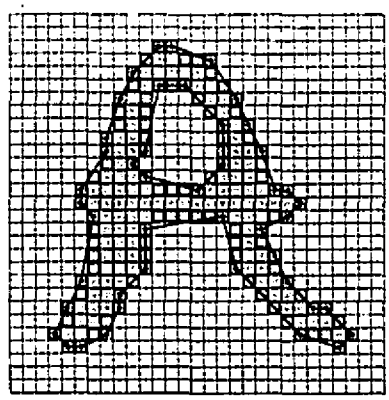


Figure 16. The character A approximated by line segments. The black dots corresponding to the section knots.

The number of line segments necessary for interpolating a character is dependent on the size of the bitmap. For the generated typefaces that are shown in chapter 5, this description often consists of 800 to 3000 linear segments.

We may also take the vertices in the polygon as points d_i of a Böhm Bézier spline (as described in the previous section).

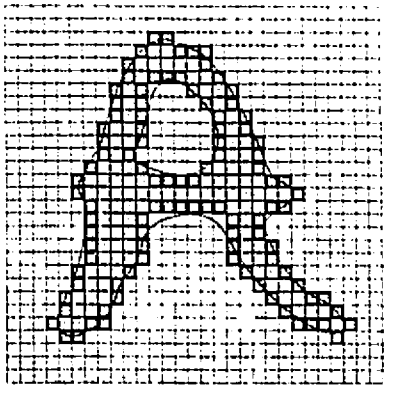


Figure 17. The character A approximated by Bézier splines. The splines are constructed by Böhm's algorithm using the sections defined by the linearization process.

This introduces some error with respect to the bitmap. Also, the number of sections is exorbitant. We then apply an optimization algorithm that minimizes our quality function Φ (see section 3.6). As Φ is possibly multimodal and is defined on a variable number of parameters, it seems best to use a random search method for this purpose (see Törn and Žilinskas (1989), Zhigljavsky (1991), Männer and Schwefel (1991) or Rinnooy Kan and Timmer (1987a, 1987b) for surveys).

Random search has the advantage that it converges in all circumstances to a global optimum, and that it finds acceptable solutions relatively quickly. The basic outline is as follows:

{start}
(Φ is a quality function described later)
($\mathcal{S}$ is a Böhm-type Bézier spline, initially given to us
where $\Pi_{\mathcal{B}}$ is the curvature $\kappa_{\mathcal{B}}$ of $\mathcal{B}$
for all Böhm-type Bézier spline $\mathcal{B}$ of $\mathcal{S}$)
Create a heap $\mathcal{H}$ with all sections $\mathcal{B}$ of $\mathcal{S}$,
where the key is $\Pi_{\mathcal{B}}$
(the smallest $\Pi_{\mathcal{B}}$ is at the top of the heap).

{search}
$\delta_1 \leftarrow$ initial step (initial step size for random search)
(δ_1 is integer)
$\pi_1 \leftarrow 0$ (Initial penalty value)

for n = 1 to N do (N is the number of iterations)
with probability p_n do: (p_n may be varying or constant)
$\mathcal{B} \leftarrow$ top($\mathcal{H}$)
$\mathcal{B}' \leftarrow$ shortest section neighboring $\mathcal{B}$ on $\mathcal{S}$
$\mathcal{S}' \leftarrow \mathcal{S}$ with $\mathcal{B}$ and $\mathcal{B}'$ merged
(and adjacent sections modified)

otherwise do:
select d uniformly and at random from
the junction points of $\mathcal{S}$
set $d' \leftarrow d + \delta_n U$, where U is uniformly distributed
on the unit circle (so $\|d' - d\| = \delta_n$)
$\mathcal{S}' \leftarrow \mathcal{S}$ with d replaced by d'
(and adjacent sections modified)

if $\Phi(\mathcal{S}') < \Phi(\mathcal{S})$,
then {a success}:
$\delta_{n+1} \leftarrow \delta_n + \gamma$ ($\gamma \in \{1,2,3\}$ is a user parameter)
$\mathcal{S} \leftarrow \mathcal{S}'$
update $\mathcal{H}$ by removing obsolete sections
and/or altering the key value of update sections
(Note: maximally 4 sections are involved,
for each one do $\Pi_{\mathcal{B}} \leftarrow \kappa_{\mathcal{B}}$)
else {a failure}:
$\delta_{n+1} \leftarrow max\{1, \delta_n - \gamma\}$
$\Pi_{\mathcal{B}} \leftarrow 1.1\, max\{\Pi_{\mathcal{B}'}\}$ for all
neighboring spline sections as explained in section 3.4

As seen in the above algorithm, the location of the new point in a move operation is determined randomly by restricting the distance d by which the point can move. The distance may be fixed to a given value throughout the optimization process. This is known as a *fixed step size optimization*. Although the probability of improvement while using a small step size is high, the improvement is small, as noted in Schumer and Steiglitz (1968). On the other hand, if the step size is big, the probability of improvement becomes quite small as the the modification will overshoot the minimum. The optimum step size is between those two extremes. Since the optimum step size is unknown, *adaptive step size random search* algorithms were created. The distance d varies according to the past experience. The basic principle behind these adaptive algorithms is to try bigger steps as an improvement occurs and to reduce the step on unsuccessful trials (Matyas, 1965). Each algorithm uses a different variant. For example, the algorithm *Adaptive Step Size Random Search* (ASSRS) found in Schumer and Steiglitz (1968) tries two step sizes, d and $d(1+a)$ where $0 < a < 1$, and waits a certain number of consecutive unsuccessful trials before reducing the step size d. If on the other hand the attempt succeeds, the value, d or $d(1+a)$, that creates the best improvement is taken as the next value for d.

A more general adaptive procedure presented in Devroye (1972), tries to combine a random search with a non-random direct search. The *compound random search algorithm* (CRSA) basically inspects a deterministic modification to the approximation as well as a controlled random variation. The interesting factor, here, is the control of the step size. If d_j represents the distance used at iteration j then

$$d_{j+1} = \begin{cases} d_j(1+\alpha) & \text{if the trial is successful} \\ d_j(1-\beta) & \text{otherwise,} \end{cases}$$

where with $\alpha > 0$, $0 < \beta < 1$ and d_0 arbitrary, we have

$$P_{\text{success}} \approx \frac{\beta}{\alpha+\beta} < 0.5$$

Hence if we choose $P_{\text{success}} = 0.2$ then we must have $\alpha = 4\beta$. The values for α and β have an indirect influence on the performance of this algorithm. It is recommended though that

P_{success} be kept between 0.15 and 0.35.

Unfortunately, this method cannot be employed naively. The PostScript type 1 format requires that the different instruction parameters be integer values (see Adobe, 1990b).

A reasonable scheme to overcome the deficiency is simply given by

$$d_{j+1} = \begin{cases} d_j + \gamma & \text{if the trial is successful} \\ d_j - \gamma & \text{if the trial is not successful and } d_j - \gamma \geq 1 \\ d_j & \text{otherwise,} \end{cases}$$

where $\gamma \in \{1, 2, 3\}$. The values for γ are derived from the following Bernstein polynomials property. The maximum of a Bernstein polynomial B_i^n is attained at $t = i/n$ making the change reasonably predictable. As a rule of thumb, the maximum variation of each B_i^n is roughly $\frac{1}{3}$. Thus a change of a control point by three units changes the curve by one unit (Farin, 1993).

3.6 Quality function

The most used quality function Φ is based upon the least-squares criterion (Plass and Stone, 1983) (Itoh and Ohno, 1993) (Gonczarowski, 1991) (Schneider, 1990). It evaluates the distance between the contour pixels and their corresponding locations on the interpolating curve. It requires the computation of a mapping between the pixels and the local parameter t. Different methods are used to perform the approximation mapping.

The presented method here does not require any such mapping. As the curvature of the interpolating curves is used, a different quality function results. The error in pixels and the curvature make up the components of the quality $\Phi(\mathcal{S})$ of a spline $\mathcal{S}$:

$$\Phi(\mathcal{S}) = \alpha \text{ error}(\mathcal{S}) + \beta \text{ curvature}(\mathcal{S}).$$

Here the weights α and β satisfy $\alpha + \beta = 1$, $\alpha \geq 0$, $\beta \geq 0$. The curvature may be evaluated by Simpson's rule (Davis and Rabinowitz, 1984). The pixel error error($\mathcal{S}$) is explained in the next section.

3.7 Pixel error evaluation

The quality of a C^2 Bézier spline can be evaluated by the precision at which it approximates a certain portion of the bitmap. The number of pixels erroneously colored measures the quality of the approximation. A pixel of the generated image is said to be in error if its color does not coincide with its color in the original bitmap. This approach considers the area in error instead of distances.

Pixel color determination can be done in many ways using filling algorithms. A simple method known as the *flood-fill* algorithm consists of filling recursively a bounded area with a given color. This would require that the Bézier spline be rendered in order for the bordering pixels to be determined. Instead a scan-line algorithm based upon the *even-odd* rule (Foley et al, 1992), will be used. It colors a pixel by drawing an imaginary line, usually vertical or horizontal, between a pixel and some other distant point for which the color is known. If the number of times the line intersects the polylines or curves is odd then the pixel lies inside and can be colored black. Otherwise it is white. The following figure shows the horizontal line determining the number of crossing with a spline.

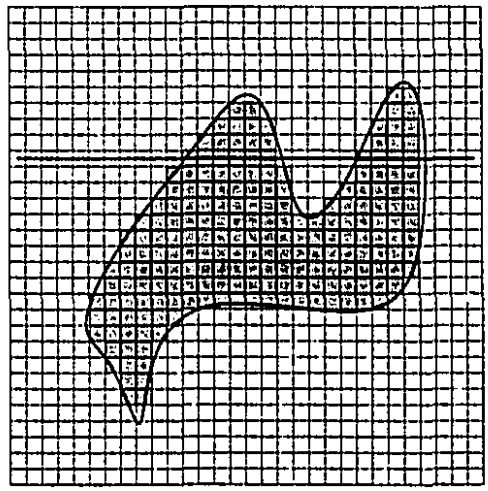


Figure 18. The even-odd algorithm.

The convex hull property of Bézier curves ensures the locality of the modifications. If a modification ξ is applied to a Bézier curve $\mathcal{B}_1$ to produce another Bézier curve $\mathcal{B}_2$, the region of the bitmap for which pixels might change color is delimited by the convex hulls of $\mathcal{B}_1$ and $\mathcal{B}_2$. Note that the two cannot be disjoint since $\mathcal{B}_2$ must be attached to the portion of the spline that $\mathcal{B}_1$ was connected to initially. For simplicity, a bounding box BB_ξ can be used to enclose the two convex hulls. With a spline that gets modified at each step of the generation process, the computations are kept to a minimum by the locality property of Bézier curves.

The even-odd rule dictates that the intersections between the curves and the pixel rows and columns be found prior to its use. Hence for a Bézier curve $\mathcal{B}$, with control points b_0,b_1,b_2,b_3 and a bounding box BB the intersections of $\mathcal{B}$ with the rows and columns of BB need to be computed. An intersection for $\mathcal{B}$ is calculated by solving the cubic roots of one of the two polynomials of the Bézier monomial form

$$\begin{array}{rcl} a_x t^3 + b_x t^2 + c_x t + x_0 & = & x_l \\ a_y t^3 + b_y t^2 + c_y t + y_0 & = & y_l \end{array},$$

depending if a vertical line at $x = x_l$ or a horizontal line at $y = y_l$ is considered. Note that x_l and y_l are contained in the bounding box BB.

Without loss of generality, let us consider the case of a vertical line at $x = x_l$. We call a root t_0, t_1, t_2 valid, if it is real and falls in the range $[0, 1]$. Let t_j be such a valid root. Then the curve intersects the line at point $(x_l, \mathcal{B}(t_j))$. Let $y_b = \lfloor \mathcal{B}(t_j) \rfloor$. If the total number of curves passing between the points (x_l, y_b) and $(x_l + 1, y_b)$ is odd then the color of the two points (x_l, y_b) and $(x_l + 1, y_b)$ is different.

If the considered bitmap is of size $m \times n$, then knowing the color of pixel (x, y), $0 \leq x < m, 0 \leq y < n$, as well as the number of times the line (x, y) $(x + 1, y)$ gets intersected by curves is sufficient to determine the color of the pixel $(x + 1, y)$. The only information that must be kept for a pixel (x, y) is thus the number of intersections between (x, y) and $(x+1, y)$. The information that must be retained is the set of intersection points with the horizontal and vertical lines.

3.8 Bushfire algorithm

If we define the error to be the number of incorrectly colored pixels, then each pixel has the same weight in the criterion. Experiments show that it is preferable to give more weight to pixels that are far away from the contour. This, in effect, creates better-fitting splines.

The distance from the outline is the length of the shortest adjacent-pixel-path (in which only north, south, east, west moves are allowed) starting at the pixel to reach a pixel of the appropriate color. As this is a shortest path problem, ordinary breadth-first-search may be performed to find all distances in time proportional to the number of pixels (Cormen, Leiserson and Rivest, 1990). By analogy with a bush fire, we coin this the bushfire algorithm—the name was taken from G.T. Toussaint's Computational Geometry course at McGill University. More details may be found on page 254 of Preparata and Shamos (1985). The algorithm is given below.

```
S ← set of contour pixels
Q ← ∅ (Q is a queue)
∀s ∈ S do: mark s as visited
           set value(s) ← 0
           enqueue(s,Q)

while not empty(Q) do:
      s ← dequeue(Q)
      for all 4-connected neighbors e of s do
          if e has not been visited then
             mark e visited
             set value(e) ← value(s) +1
             enqueue(e,Q)
return the array "value", which contains the distances
      to the contour pixels
```

The pixel error may now be defined by

$$\sum_{\text{white pixels } p} W(\text{value}(p)) + \sum_{\text{black pixels } p} W(\text{value}(p)) ,$$

where W is an increasing penalty function such as

$$W(U) = \begin{cases} U & \text{(linear penalty)} \\ U^2 & \text{(quadratic penalty)} \\ U^3 & \text{(cubic penalty)} \\ UI_{U\leq\alpha} + \infty I_{U>\alpha} & \text{(linear penalty, } \infty \text{ beyond } \alpha) \\ U^2 I_{U\leq\alpha} + \infty I_{U>\alpha} & \text{(quadratic penalty, } \infty \text{ beyond } \alpha) \\ U^3 I_{U\leq\alpha} + \infty I_{U>\alpha} & \text{(cubic penalty, } \infty \text{ beyond } \alpha) \end{cases}$$

As mentioned above, many existing algorithms use the least-squares method. Roughly speaking, these correspond to picking $W(U) = U$ as the sum of penalties $1, 2, 3, \ldots, k$ for a pixel at distance k is $k(k+1)/2$.

4 Generating process

The different steps necessary for the creation of a handwritten typeface, using the developed software, will be explored in this chapter. The process consists of four distinct stages. Initially the desired character set is written on paper and read by a scanner. Once a bitmap is obtained from the scanner's software, it is converted to the desired format. The next step creates a bitmap font in the PostScript type 3 format. Finally, this new font will be used as input for the creation of the handwritten typeface. The result is a file in the PostScript type 1 format. This process is depicted in figure 19. Each stage of the process is explained in more details in the following sections.

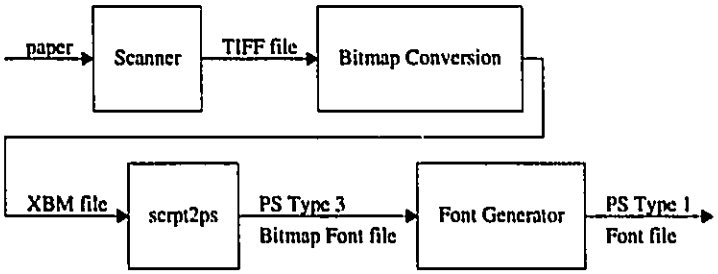


Figure 19. The typeface generation process.

4.1 Scanning

Although flexibility is one of our goals, we introduce some restrictions on how the characters must be presented.

Two types of characters may be considered. One is the set of characters that are composed of two horizontal parts such as the *dieresis* (¨), blank space dividing the different pieces. The other group is composed of all the horizontally unbroken pieced characters, like "a","b" and "c". Note that characters such as "i" may be part of the first group if they are written in an extreme slanted manner. In order to distinguish characters of the first type from adjacent characters of the second type, some noticeable space should separate

distinct letters.

The height characteristics of letters may be described by four lines, the baseline, the descender line and the lower case and upper case lines. They guarantee some consistency among the characters of a typeface. The following figure shows the mentioned lines for letters of the Times-Roman typeface.

Figure 20. Example of the guidelines for characters in the Times-Roman font.

These lines may be used as guidelines when writing the letters. As the four lines must not appear in the bitmap, blue ink may be used to draw them. The position of the baseline is fundamental to ensure that all the letters of a typeface be correctly aligned when printed. Two black line segments are required on the left and right sides of a string of characters to indicate the baseline. A set of characters with the baseline indicators constitute an input strip. A character set may require multiple bitmaps to be entirely represented. The input strips are scanned individually. By partitioning a page in multiple sections, the amount of memory required is lowered. Scanning an entire page at high resolution may require several megabytes of storage. An example of a input strip is shown below.

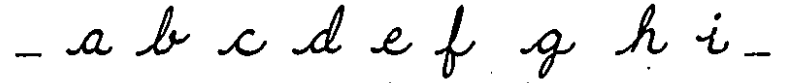


Figure 21. Example of an input strip. The two black line segment at the extremes indicate the location of the baseline.

4.2 Bitmap conversion

Depending on the scanner software, the bitmap may be saved in different formats. In our case, the generated bitmap is in the TIFF format designed by the Aldus Corporation. It

is an independent and extensible description format.

The next stage of the process expects as input a file in the XBM format. The term XBM stands X bitmap of the X-Windows system. The format is such that the bitmap may be incorporated in any C language program. The following corresponds to the mock handwritten letter A of figure 7 in the XBM format.

```
#define B_width 25
#define B_height 25
static char B_bits[] = {
   0x00, 0x06, 0x00, 0x00, 0x00, 0x3f, 0x00, 0x00,
   0x80, 0x3f, 0x00, 0x00, 0x80, 0x7f, 0x00, 0x00,
   0xc0, 0xf1, 0x00, 0x00, 0xc0, 0xe1, 0x00, 0x00,
   0xc0, 0xc1, 0x01, 0x00, 0xe0, 0xc1, 0x01, 0x00,
   0xe0, 0xc1, 0x03, 0x00, 0xe0, 0xc0, 0x03, 0x00,
   0xf0, 0xe1, 0x03, 0x00, 0xf8, 0xff, 0x0f, 0x00,
   0xf8, 0xff, 0x1f, 0x00, 0xf0, 0xff, 0x0f, 0x00,
   0xf0, 0x81, 0x03, 0x00, 0xf0, 0x81, 0x03, 0x00,
   0xf0, 0x81, 0x07, 0x00, 0xf0, 0x81, 0x07, 0x00,
   0xf8, 0x00, 0x0f, 0x00, 0x78, 0x00, 0x1e, 0x00,
   0x7c, 0x00, 0x7c, 0x00, 0x3c, 0x00, 0xf8, 0x00,
   0x3e, 0x00, 0xf0, 0x01, 0x0c, 0x00, 0x80, 0x00,
   0x00, 0x00, 0x00, 0x00};
```

Note that if the width of the bitmap is not a multiple of 8 then the bitmap is artificially padded with zeros. The conversion from TIFF to XBM can be achieved by using the PBMPLUS package. The conversion of a file `bitmap.tiff` into `bitmap.xbm` may be achieved with the command line:

```
tifftopnm bitmap.tiff | pbmtoxbm > bitmap.xbm
```

Both programs, `tifftopnm` and `pbmtoxbm` are part of PBMPLUS.

4.3 Scrpt2ps

The program `scrpt2ps`, developed by Luc Mikiszko at McGill University during the summer of 1993, creates a PostScript type 3 bitmap font from a series of bitmap strips.

Modifications were made for increased flexibility.

This piece of software requires some hints on the supplied characters as it is does not automatically perform pattern recognition. As the name of each character of a bitmap strip is unknown to the program, the names need to be provided. For a better understanding of the restrictions imposed on the layout of the characters, let us take a closer look at how `scrpt2ps` works.

For each bitmap strip, the following operations are done. The baseline position is determined from the marks on the extremities of the bitmap. As the thickness of the marks may vary, the center of mass of each one of them is taken into consideration to construct the baseline. As the line of characters may not have been exactly horizontal during the scan, this fictive line, traversing the bitmap, helps recalibrate the position of the baseline for each character.

The character locations need to be determined. Sections of consecutive columns having some black pixels are first delimited. Each section may correspond to a possible valid character. Noise may be present in the bitmap and must be removed. As noise is simply a group of black pixels that does not represent a character, the number of pixels composing an entity in the bitmap may be used to differentiate noise from valid character pieces. Below a certain threshold, a set of connected black pixels is considered as noise and is removed. Note that a set of white pixels may create an undesirable hole in a region of black pixels in which case they constitute noise also. The same concept may be used to remove noise generated by the white pixels.

The bounding box for each character is then computed and the corresponding bitmap is then used to generate the font. Some characteristics like the size and baseline position are also generated.

The command line for using `scrpt2ps` is of the form:

```
scrpt2ps [options] [input_files]
```

It may read input files from the standard input channel as well as from specified filenames. The output is sent to the standard output channel. The different options that may be used

with `scrpt2ps` are explained below.

`-h:` displays a brief description of the program options.

`-r:` specifies the scanner resolution. For example, `-r 600` indicates that the bitmap was scanned at 600 dpi. The resolution option is used for converting distances in terms of number of pixels. It is also used to determine the threshold in noise detection. The default resolution is 300 dpi.

`-s:` specifies the space between adjacent characters. The parameter value represents the minimal space that must be present between two characters to consider them individually. The value is expressed in inches. The default value is half an inch (0.5).

`-l:` specifies the filename containing the list of names present in the supplied bitmap. The filename containing the list of names must also be specified on the command line. The format for the list file is very simple, one name per line. Hence the file for the bitmap band shown in figure 21 would be

```
/a
/b
/c
/d
/e
/f
/g
/h
/i
```

The reserved name `/undefined` may be used to indicate that a character should be skipped.

`-d:` specifies the filename containing the font encoding. Once all the bitmap strips are processed by `scrpt2ps`, the font can be generated. The desired encoding must be specified. A brief example of the Computer Modern Roman encoding shows the required format for the dictionary file.

```
0 /Gamma
1 /Delta
2 /Theta
3 /Lambda
4 /Xi
5 /Pi
6 /Sigma
7 /Upsilon
```

The following simple shell script shows how `scrpt2ps` may be used on six input strips. The characters were scan at a resolution of 600 dpi and the minimal distance separating adjacent characters was a quarter of an inch. Initially each input strip, in TIFF format, is converted to XBM and processed by `scrpt2ps`. When all six temporary PostScript files are created, they are united into the file `bitfont.ps`.

```
#!/bin/csh
foreach f ( 1 2 3 4 5 6 )
   tifftopnm bit$f.tiff | pbmtoxbm >! bit.xbm
   scrpt2ps -r 600 -s 0.25 -l list$f < bit.xbm > bit$f.ps
end
scrpt2ps -r 600 -d dictfile bit*.ps > bitfont.ps
```

An example of what the resulting PostScript file looks like is shown in Appendix A.

4.4 Font generator

Once the bitmap font is created, the typeface generator program FONT may be used to generate the typeface in the type 1 PostScript format.

The command line for FONT is

```
font [options] font_file
```

The different options that may be used are explained below. The options are grouped according to their relation. The options that start with a lower case letter may be used to

alter the approximation process.

`-h:` displays a description of the program options.

`-a:` governs the probability of a merge to be performed. The default value is 0.5.

`-i:` specifies the number of optimization iterations to perform on each character, the default value being 1000.

`-r:` may be used to specify the resolution of the written characters. By selecting this option the scaling between scanned pixels and the character space may be computed. If the option `-r` with parameters 150 and 500 is used, then the program knows that a character occupying 150 pixels in height in the original bitmap should occupy has a height of 500 in the character space, a metric box of 1000 x 1000. This option overides the options `-L` and `-LS`. May be used to developed and entire font family and keeping consistency among the members.

`-va:` indicates that a variable merge probability be used.

`-w:` specifies the value of β in the quality function Φ, α being $1 - \beta$. The default value of β is 0.5. β weighs the curvature component in Φ and α the pixel error component. In many cases, values like $\beta = 0.9, 0.99, 0.999$ or even 0.9999 may be used to obtain increasingly smooth (but less realistic) outlines.

`-ss:` specifies the initial step size to use in a move operation. The default step size is one pixel.

`-gs:` stands for global step. As by default, each Bézier section has its own step size that varies as move operations are performed on the section. A global step size may be used by this switch.

`-BF:` indicates which bushfire "value" function $W(U)$ to use, see section 3.8. Six functions are available, and can be selected with parameter values between 0 and 5. The parameters values 0, 1, 2 are for linear, quadratic and cubic functions respectively. The parameter values 3, 4, 5 are identical except that past a certain distance, the penalty becomes infinity.

`-BFD:` specifies the distance at which the bushfire algorithm decides to set the pixel

weight as ∞. Note that this parameter may only be used when the option -BF is used with parameter values 3, 4 or 5. The default maximum distance is 3.

-B: specifies the number of layers of pixels to be added to the characters to make them bold. The bushfire algorithm is used here also to determine the pixels that are at a distance less than the value supplied from a contour.

-SP: specifies the width of the space character. The value should be between 0 and 1000. This is only necessary when defining a typeface that explicitly describes the space character. The default width is 500.

-SB: may be used to override the default side bearings width. The default is 60 represented in the character space of 0 to 1000.

-L: specifies the name of a character for which as reference height may be used. The default character is the letter "x".

-LS: specifies the height of the reference character. The height must be given in terms of the 1000 by 1000 metric box. The default value is 430 representing the letter "x" of the Computer Modern Roman font.

-NP: specifies that the generated font should be monospaced. The widest character determines the width of all the characters.

-SL: specifies the single character on which the font generation should be performed. The name of the single character is written out in text form just as in the encoding vector. The name of the character "4" is thus "four".

The different fields that are required in a type 1 font dictionary may be supplied by the user. The options pertaining to font specifications all start with the letter F. Once a PostScript type 1 font is generated, the font file may be edited to change the values passed by these options.

-FN: specifies the name of the font. By default the name is Mine. It is also used to specify the name of the output file in which the font will be created. Hence if -FN MyFont, the created font will be stored in MyFont.ps

-FF: specifies the font family name. The default name being Mine.

-FFN: specifies the font full name, here again the default name is `Mine`.

-FV: indicates the version number of the generated font. The value 0.0 being the default version number.

-FI: specifies the angle to use for italic letters. Normal letters being generated by default, the default is 0.

-FID: specifies the UniqueID value to use for the generated font. The default ID is 4000000.

-FW: indicates the weight of the font. The default value is medium.

-FP: may be used to specify if stroked or filled characters are desired. A value of 0 indicates filled and 2 stroked characters. By default, the generated font characters are filled.

Some of the options that are available to the user were introduced to gather information on the interpolation process. Statistics on the success of the operations as well as the average error and curvature during the interpolation of the entire typeface can be accumulated.

-S: indicates that a stats file named `stats` be created. The information recorded to the average state of the optimization at each iteration of the process. As the information is stored in binary, to save space, the program `stats` may be used to create an ASCII file that can be used by `gnuplot` for plotting the behavior of each recorded attribute.

-H: allows a more complete description to be recorded in a history file. The file keeps the information for each iteration and character. The resulting file may become quite large. The created file is named `history`. Once again to save space, the information is stored in binary format. The program `history` may be used to generate input for the `gnuplot` program.

-O: allows the creation of a visual history. A PostScript file is created and is named `out.ps`.

-OF: may be used to specify the name of the visual history file.

-OI: indicates the number of iterations between two successive records of the current

character in the output file.

`-OP:` indicates the percentage of the box specified by the `-OS` option to use. The default value is 0.80, so the 1000 x 1000 metric box will be scaled so that it occupies a square box of 160 points if the option `-OS 200` is used.

`-OS:` indicates the maximum size a character may occupy on the output page. The default values defines a square box of 100 points.

Here are some examples of how the `font` program may be used.

The command

```
font -i 1000 -a 0.9 -w 0.99 -gs -FN Flamingo -FFN Flamingo -Fv 2.0 -FF
Flamingo Flamingo-Bitmap.ps
```

creates the version 2.0 of the typeface "Flamingo". The number of iterations used for the generation is 1000 and ninety percent of the time a merge is performed. The curvature is 99 times more important than the error in pixels when the quality function Φ is used. Also a global step size is used and the input file is `Flamingo-Bitmap.ps`. The output file will be `Flamingo.ps`

The command

```
font -i 1000 -3L two -O -OI 50 -OS 200 -OP 1.0 Bitmap.ps
```

creates a font named "Mine" including only one character, namely the letter "2". The number of iterations applied to the character is once more 1000 and at every 50 iterations the state of the character is output in the PostScript file `out.ps`. The 1000 x 1000 metric box of "2" will occupy a square box of 200 points. The input file is `Bitmap.ps`.

The command

```
font -i 100 -H Bitmap.ps
```

creates also a font named `Mine` but this time all the characters included in the input file `Bitmap.ps` are processed. A history file is also created. By using 100 as the number of iterations to be performed for each character, the size of the history file is controlled.

5 Handwritten fonts

In this section, we will illustrate some of the fonts that were generated by the mentioned algorithm. They are briefly shown in figures 23, 24 and 25 using a format similar to Wallis (1990).

An example of a the letter "W" from the Isabella2 is shown in the next figure. The black dots represent the Bézier curve endpoints. The bounding box and the 1000 x 1000 metric box are also shown. One will notice how the Bézier curves are distributed along the contours. More curves are used in sections with high curvature.

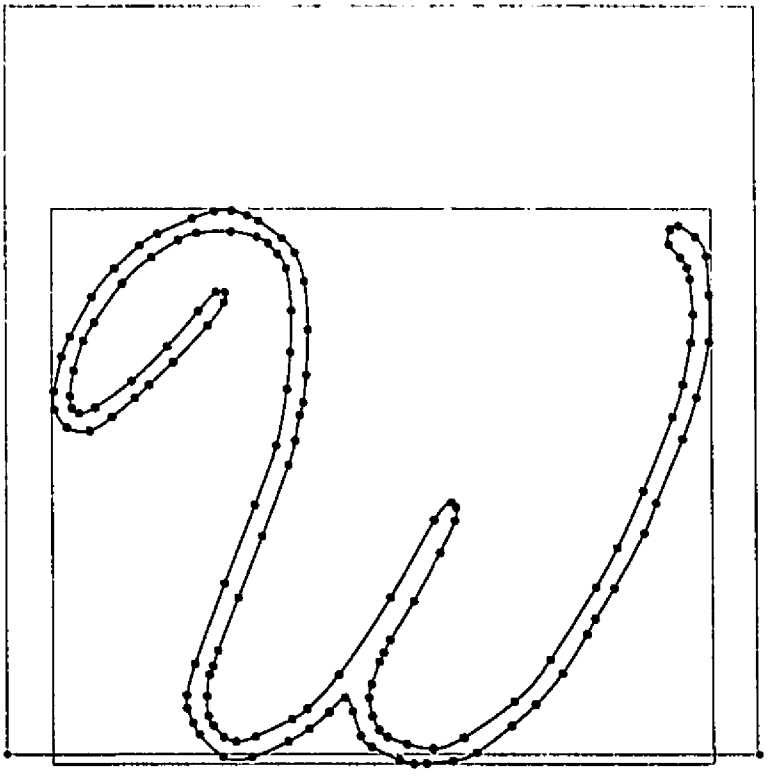
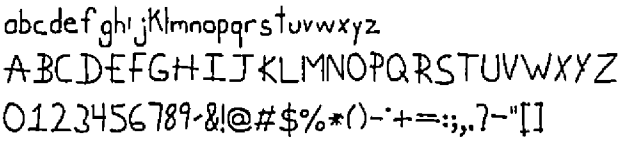


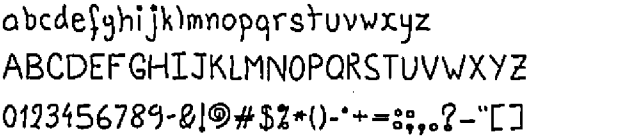
Figure 22. Isabella2's letter W created with the parameters -i 4000 -a 0.90 -w 0.99 -gs. The bounding box and the metric space are shown. The two black dots define the width of the character. The Bézier spline endpoints are shown as white dots.

Handwritten fonts may be used for a myriad of applications. To attract customers in

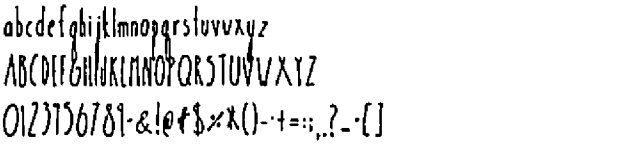
Bazooka



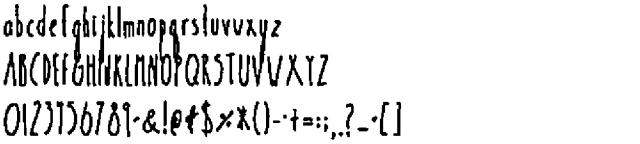
Cacographic-Roman



Flamingo



Flamingo-Bold



Flamingo-Black

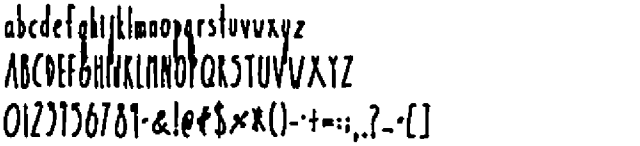


Figure 23. Newly created handwritten typefaces. Birgit Devroye created the Bazooka typeface. The Cacographic-Roman typeface is written by Luc Devroye, and the Flamingo typeface is from Natasha Devroye's hand.

Isabella1

abcdefghi jklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789-&!@#$%*()-'+=:;,.?-"[]

Isabella2

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789-&!@#$%*()-'+=:;,.?-"[]

Isabella3

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789-&!@#$%*()-'+=:;,.?-"[]

Isabella4

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789-&!@#$%*()-'+=:;,.?-"[]

Isabella5

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789-&!@#$%*()-'+=:;,.?-"[]

Figure 24. Newly created handwritten typefaces. The Isabella series was produced by Isabelle Massarelli.

Isabella6

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789-&!@#$%*()-'+=:;,.?-"[]

Isabella6-Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789-&!@#$%()-'+=:;,.?-"[]*

Pach

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789-&!@#$%*()-'+=:;,.?-"[]

TropDePoils

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789 &!@#$%*()-_+=:;,.?|/[]

TropDePoils-Mono

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789 &!@#$%*()-_+=:;,.?|/[]

Figure 25. Newly created handwritten typefaces. Janos Pach wrote the Pach typeface and the TropDePoils typeface is due to Luc Devroye.

a warm ambiance, restaurant menus are often written by hand or with a typeface that resembles someone's handwriting. Blurbs in commercials or comic strips require simulated handwriting as well. And of course, there are the personalized letters and the old-style handwritten mathematics lecture notes.

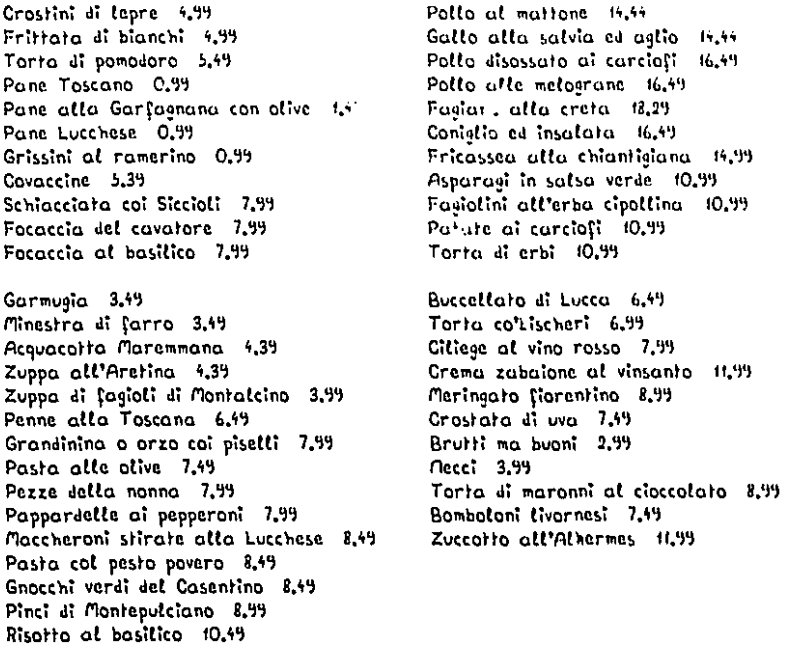


Figure 26. An Italian menu in the TropDePoils typeface.

Different sizes of the Isabella4 typeface are shown below:

To each fontsize a different purpose.
To each fontsize a different purpose.
To each fontsize a different purpose.
To each fontsize a different purpose.
To each fontsize a different purpose.

New typefaces for use with TEX may also be obtained. To illustrate this, the Cacographic family has been created. Since TEX fonts need to be handled differently than

PostScript fonts, only the Cacographic-Roman is shown here and the remainder of the family is presented in chapter 8 along with the necessary modifications that are needed for the font to be correctly used by TEX .

These lines are written with the Cacographic-Roman type-face. The word Cacographic comes from the Greek word for "bad handwriting".

One of the particularities of TEX is the way foreign letters are handled. Accented letters, like é,è,ê, are generated by composition of two characters. The newly created typeface has no problems conforming to it as shown in the following examples.

Bazooka:

La peinture est un poésie qu'on voit au lieu de l'entendre,
la poésie une peinture qu'on entend au lieu de voir.

Léonard De Vinci

Flamingo:

C'est dans les mathématiques que réside le principe vraiment créateur. En un certain sens, donc, je tiens pour vrai que la pensée pure est compétente pour comprendre le réel, ainsi que les Anciens l'avaient rêvé.

Albert Einstein

Isabella1:

J'écris parce que j'ai l'impression ou le sentiment que le monde est inachevé, comme si Dieu, qui a créé le monde en six jours et qui s'est reposé le septième, n'avait pas eu le temps de tout faire. Je trouve le monde trop petit, la vie trop courte, le bonheur pas assez bonheur. J'écris pour achever le monde, pour ajouter à la création le huitième jour.

Antonine Maillet

No attempt was made here to properly kern the letters—all the inter-character spacings are those obtained by the new usage of TEX's TFM files, without kerning information. The nature and the purpose of the TFM files will be examined in section 8.

As the approximation evolves through time, the characters converge towards an optimal representation. The following figure shows the visual history of the approximation of the letter "y" in the Flamingo typeface. To appreciate the evolution of the character, the stroked letter is shown instead of the filled one. The Bézier curve endpoints are also shown. We clearly see the the number of Bézier curve segments rapidly decreases.

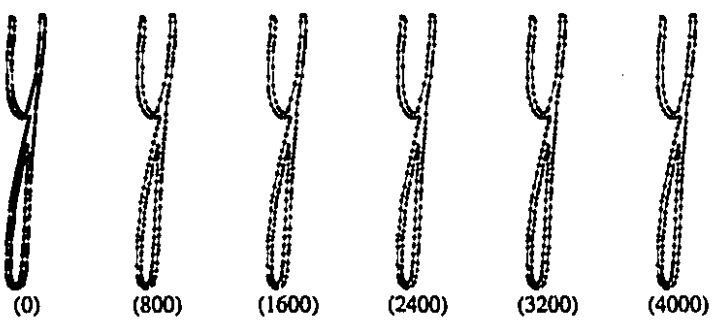


Figure 27. The evolution of Flamingo's y during the generation of the typeface. The numbers represent the iteration number and the black dots are the Bézier curve endpoints.

Following is the visual history of the dollar sign of the typeface Bazooka.

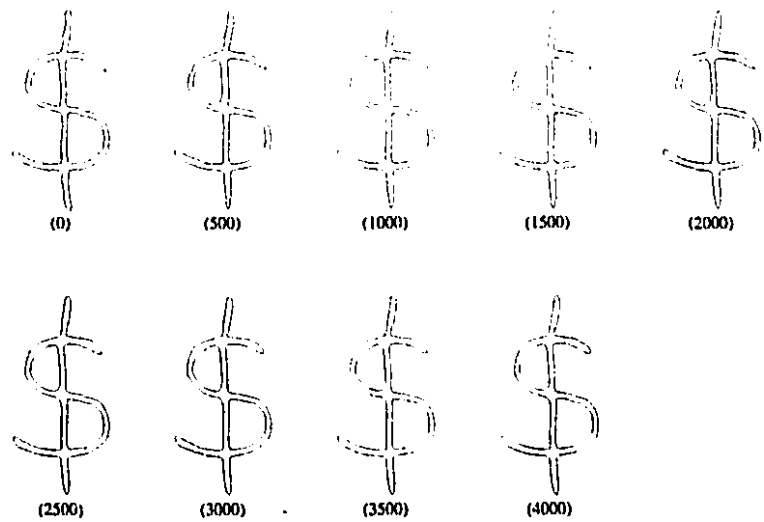


Figure 28. The evolution of Bazooka's $ during the generation of the typeface. The numbers represent the iteration number.

Monospaced fonts may be used to display algorithms or lists. The following example gives the binary search algorithm using the monospaced version of the Cacographic-TypewriterType typeface.

```
BSEARCH(ptr,key)
   if ptr == NIL or ptr->val == key
      return ptr
   if key < ptr->val
      then return BSEARCH(ptr->left,key)
      else return BSEARCH(ptr->right,key)
```

The following comparative example demonstrates the creation of a bold typeface. The Flamingo typeface is used as the base font, and the bushfire algorithm is applied to increase the thickness of each character. It is performed by adding extra layers of black pixels.

Words written with a bold font are more noticeable to the eye.
Words written with a **bold** font are **more** noticeable to the eye.

Bold fonts may also be achieved with a wider nib. Here is an example of the Pach typeface.

A calligrapher uses a wide nib with grace and fluidity to create elegant glyphs.

Italic characters may be produced by slightly changing some parameters in the PostScript font file as explained in section 7. The following shows the typeface Isabella6 with the derived italic version.

Italic words attract also the attention of the eye *without* disturbing the reading flow.

6 Algorithm evaluation

This section will take a look at the behavior of the algorithm under the various parameters for the generated typefaces shown in the previous section.

As mentioned above, one may view a typographer's work as a reference of optimality. One may evaluate an algorithm by considering how well it works with an existing typeface. The following figure shows the approximated letter "A" of the Optima typeface. The gray area represents the scaled bitmap of the original character. The Bézier curve endpoints are shown.

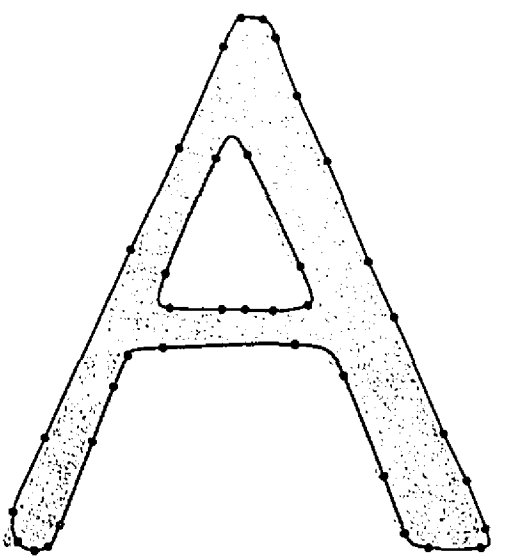


Figure 29. The letter "A" of Optima using the optimizing algorithm. The parameters used are: -i 2000 -a 0.90 -w 0.99 -gs.

6.1 Curvature versus time

Through the approximation process, the curvature varies according to its associated weight in the quality function Φ. The following figure shows how the curvature changes for the typeface Isabella5. Two curves are depicting the behavior. One has a curvature weight of 0.99 while the other has a weight of 0.4. In other words the curvature is considered to be

99 times more important than the pixel error for the first case. Inversely, the pixel error is only 1.5 times more important than the curvature in the second case.

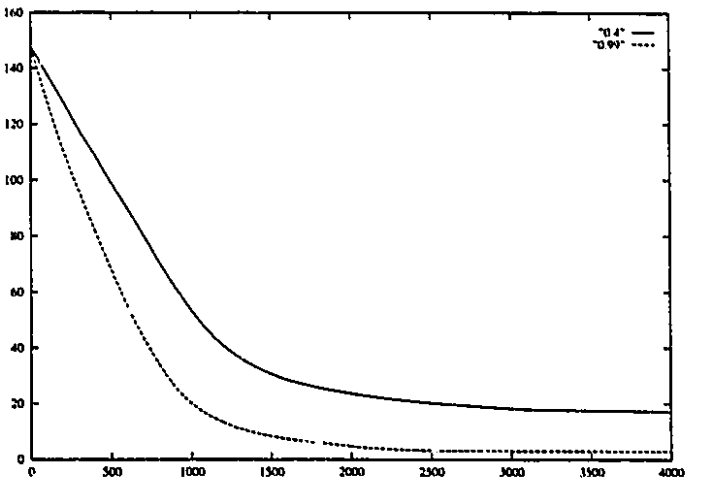


Figure 30. Curvature versus iteration number for the typeface Isabella5. The curvature weights are 0.99 and 0.4.

By using a high curvature weight, the average curvature of the typeface drops rapidly in the first 1000 iterations.

6.2 Pixel error versus time

The second component of the quality function Φ measures of the accuracy of the approximation, and depends upon the used bushfire function (see section 3.8). As mentioned in section 3, the curvature and pixel error are two conflicting criteria. This is shown in the next figure. By using a high curvature weight, hence a low pixel error weight, the algorithm favors the introduction of pixel errors in the first few iterations of the character generation. Figure 31 shows the average error for the typeface Cacographic-MathExtension at each iteration of the generation. With an error weight of 0.01 we see that the curvature is initially favored and the pixel error is subsequently minimized. A totally different behavior is seen when the pixel error weight carries more weight.

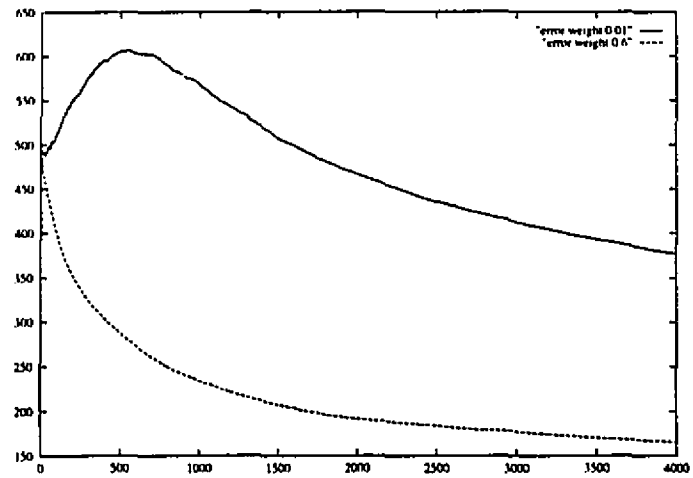


Figure 31. Pixel error for the typeface Cacographic-MathExtension versus iteration number, with two different weights.

6.3 Number of sections versus time

The number of sections depends upon the number of successful merges. The curvature weight impacts heavily on the final number of Bézier sections. This effect is shown in figure 32. The curvature weight thus has a direct influence on the size of the generated typeface.

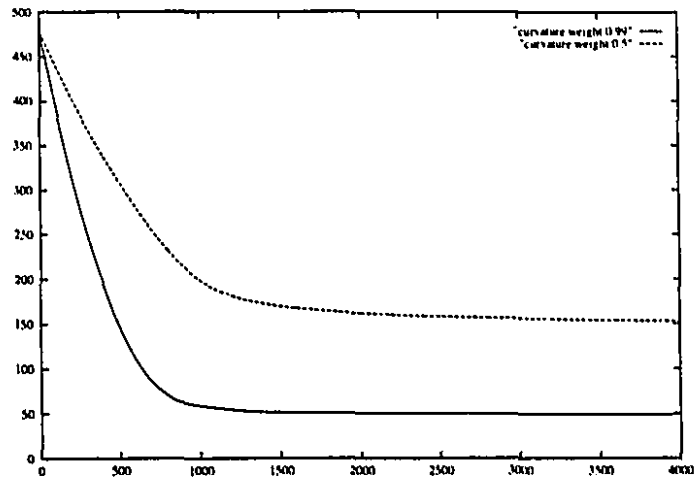


Figure 32. The average number of Bézier sections for two different curvature weights as a function of the number of iterations for the typeface Isabella4 and two different weights.

6.4 Adaptive operation selection

The adaptive random search technique CRSA (Devroye, 1972) may be used in the optimization process. It is triggered by the option -va, and requires parameters α and β (see section 3.5). For a P_{success} of 0.2, we have that $\alpha = 4\beta$. We see in the next graph that the variations stabilize around 0.70—probabilistically speaking, 70 percent of the time a merge operation will be performed.

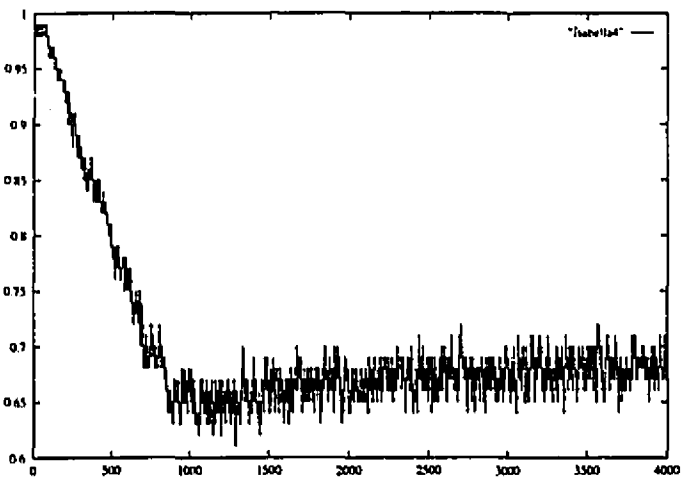


Figure 33. The frequency of the merge operation for the typeface Isabella4 versus the number of iterations. The parameters are -i 4000 -a 0.90 -w 0.99 -va -gs.

Lower curvature weight or a lower initial merge operation weight do not seem to change the final result. This behavior is shown in the next figure. The rate of convergence is affected, however.

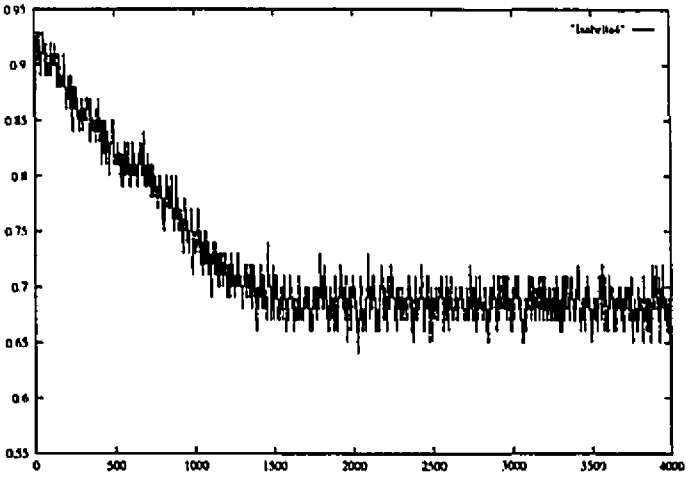


Figure 34. The frequency of the merge operation for the typeface Isabella4 versus the number of iterations. The parameters are -i 4000 -a 0.5 -w 0.5 -va -gs.

6.5 Step size versus time

The average step size use through a generation process is shown in the following graph. We see that initially the step size increases and then slowly diminishes. As the number of performed iterations increases, the optimization process reaches a point where the rate of successful operations drops. Once this point is reached, the average step size starts decreasing.

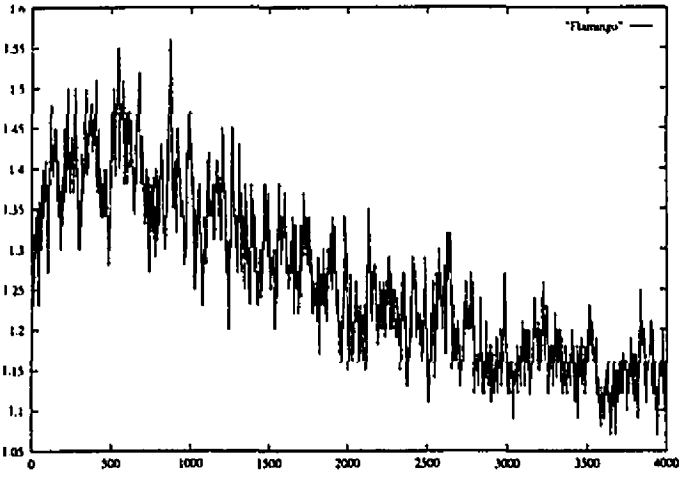


Figure 35. The average step size during the creation of the typeface Flamingo. The parameters are -i 4000 -a 0.90 -w 0.995 -gs.

6.6 Pixel errors

In this section, we show two figures for the raw pixel error (number of incorrectly colored pixels) versus the iteration number. In each case, the Bazooka typeface was constructed with two different bushfire parameters, namely `-BF0` and `-BF 5`. We notice that if the curvature weight is high, the more restrictive penalty controls the maximum number of pixels colored incorrectly.

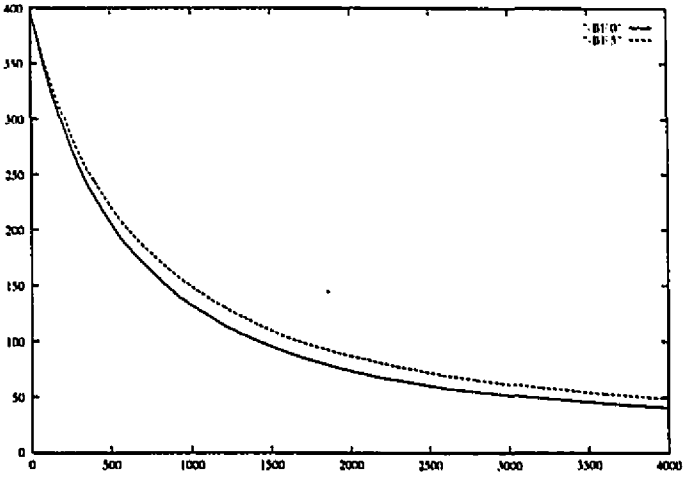


Figure 36. The variation of the number of pixels wrongly colored versus the iteration number for the typeface Bazooka with two bushfire parameters. The parameters are -i 4000 -a 0.1 -w 0.1 -gs.

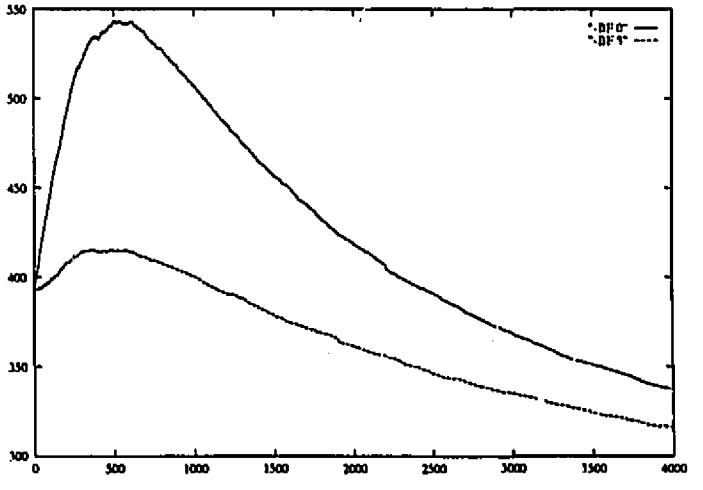


Figure 37. The variation of the number of pixels wrongly colored versus the iteration number for the typeface Bazooka with two bushfire parameters. The parameters are -i 4000 -a 0.9 -w 0.99 -gs.

7 PostScript

As seen in chapter 1, multiple PostScript formats may be used to define a typeface. We will describe in this chapter the PostScript type 1 format and how a typeface may be created.

The type 1 format was designed for storage efficiency as well as accuracy. Two kinds of file may be defined, an ASCII and a binary version. The convention used in naming the font file for each kind is as follows: the file extension `.pfa` is used for the ASCII version, while `.pfb` is used for the binary one. We will consider `.pfa` files.

In Appendix B, an example of a type 1 font is shown for the typeface TropDePoils. A type 1 font file may be divided into three parts. The header describes global characteristics about the font while the body contains the character descriptions. Although the next few sections will examine the different parts of the font, a more elaborate description of the format may be found in Adobe (1990b).

7.1 The header

Following the declarative prologue of the font file is some global font information stored in dictionaries. The dictionary `FontInfo` holds mainly the full name of the font and its family name as well as the version number. The font's name is designated by the keyword `FontName`. The entry `FontType` is always 1. The `FontMatrix` corresponds to the transformation matrix that is applied to every character before it is generated. In general this entry is used to indicate how the scaling should be done and often looks like:

```
/FontMatrix [0.001 0 0 0.001 0 0 ] readonly def
```

This transforms the character space into the user space before the appropriate scaling is applied. If we follow Böhm's construction, the Bézier control points may not be always integer values. For efficiency and accuracy the parameters used by type 1 instructions must be integer values. The size of the character space is thus modified in our work to be 6000 x 6000. Therefore the `FontMatrix` entry for our generated typefaces is as follows

```
/FontMatrix [1 6000 div 0 0 1 6000 div 0 0 ] readonly def
```

The `FontMatrix` may be modified for creating italics.

```
/FontMatrix [ 1 6000 div 0 1 6000 div θ sin θ cos div mul 1 6000 div 0 0]
                                    readonly def
```

The above `/FontMatrix` performs a shear transformation of θ degrees. The `/UniqueID` entry is used to reference a font via a number. This identification number should be unique. The last important entry in the header is the encoding vector.

7.2 The body

The body of the font starts with the instruction `currentfile eexec`. The remainder of the body consists of an encrypted string describing the character shapes. The encryption algorithm is also described in Adobe (1990b).

When a typeface is generated, no encrypted section is produced. Instead the character descriptions are produced using the standard type 1 instructions. Appendix C shows an example of the typeface TropDePoils with the type 1 instructions. From the generated TropDePoils.ps file, the file TropDePoils.pfa may be created with the program `t1asm`. This program is part of a freely available package for manipulating type 1 fonts and was created by Lee Hetherington.

Some other declarations may be seen in the unencrypted version of the font. The font bounding box is defined by the entry `FontBBox`. An array of subroutines is defined and named `Subrs`. The four subroutines are required to be present in the font, as shown in Appendix C. Subroutines may be used to execute common instructions for certain characters. This construct is useful when a typeface is manually designed with precise characteristics shared among the characters. The dictionary `CharStrings` contain the description for every character of the font.

7.3 The trailer

The trailer of a type 1 font file simply consists of multiple zeros used for padding, and of the instruction `cleartomark`.

7.4 Type 1 instructions

One of the difficulties in writing a type 1 font is that a new set of specific instructions must be learned. We will look at the ones that are used in the generated typefaces.

Similar to the instructions `curveto` and `rcurveto`, Bézier curves may be created with the instruction `rrcurveto`. The difference between all three instructions is how the parameters are interpreted. For `curveto`, the six parameters correspond to the last three points of a Bézier curve. In `rcurveto` the parameters are relative to the current point while in the instruction `rrcurveto` the parameters are relative to the last indicated point. the following gives an example of how each instruction may be used to describe the same curve with starting point (100,100).

```
150 150 200 150 250 100 curveto
50 50 100 50 150 0 rcurveto
50 50 50 0 50 -50 rrcurveto
```

Displacements are achieved by the instruction `rmoveto` that acts identically to the native PostScript instruction. Similarly, the instruction `closepath` acts as its PostScript homologue. The end of a character description is indicated by the instruction `endchar`.

In order to define the size of a character as well as the sidebearings, the instruction `hsbw` may be used. Using

```
sbx wx hsbw
```

sets the width vector to `(wx,0)` and the sidebearing point at `(sbx,0)`. Note that the point `(sbx,0)` becomes the current point without defining a point on the path. The instruction `rmoveto` must be used to define the initial point of the character.

8 TEX and mathematical writing

The creation of a typeface that may be subsequently used by TEX requires some attention. First of all the character ordering (or encoding vector) employed by TEX is different from the one that PostScript normally uses. No explicit "space" character is defined in TEX. The accented characters are created by glyph composition while in PostScript the letters need to be explicitly defined. Since TEX is a typesetting system, it is not surprising that it does not work internally with characters but with a different representation. Rectangular boxes of diverse sizes are used in which characters live. In fact, TEX only juggles with empty boxes and does not worry about the actual characters that will later be put in those boxes.

When typesetting a document, TEX decides how a sequence of words will be broken into individual lines. The horizontal position of each character is decided via some appearance criteria. In the case of accented letters, the accent's box is positioned directly on top of the letter's box. The accent box is centered with respect to the width of the letter's box. The following questions arise. How does TEX know about the sizes of the boxes and how does one inform TEX about them? The answer is through font metric files.

8.1 Font metrics

A font metric file contains information regarding the entire font and about individual characters. The actual characters are not stored in a metric file. The Adobe corporation developed a portable font metric format known as the Adobe Font Metric format (`afm`). More on the actual format may be found in Adobe (1990c). The information is stored in ASCII making this format machine independent and extensible.

TEX uses a similar information file. The TEX Font Metric, referred to by (`tfm`), may be obtained from the font `afm` file. A computer program `afm2tfm` available with the TEX package may be used for this purpose. For example,

```
afm2tfm foo
```

creates the `tfm` file for the font named `foo`. Unlike the `afm` format, the `tfm` file is in binary

format. The commands

```
afm2tfm foo -v foo
```

and

```
vptovf foo.vpl foo.vfm foo.tfm
```

sequentially create a virtual property list and a virtual font file along with the TEX font metric file. The virtual property list file is a complete description of the font metrics in ASCII format. Excerpts from TEX's `cmr10` font is shown in Appendix D. The `vpl` and `tfm` files are equivalent in all respects.

The font global information is listed at the beginning of the file. Most important is the font dimension section, denoted by the keyword `FONTDIMEN`. The `XHEIGHT` element indicates the height of the letter "x", also known as the font x-height. The dimensions are expressed on the $[0,1]$ interval.

Information for each character is also indicated. The width and height of characters are specified as well as the depth. The keywords are `CHARWD`, `CHARHT` and `CHARDP` respectively. The height of a character is not the height of the bounding box but the distance between the highest pixel and the baseline.

8.2 TEX font families

The Computer Modern family was developed specifically for TEX by Knuth (1986c). Each typeface of the family serves a different purpose. The most common of these is the Computer Modern Roman font, also known as `cmr`. The following table shows the entire character set of `cmr10`. The octal values indicate the character codes.

	0	1	2	3	4	5	6	7
00x	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ
01x	Φ	Ψ	Ω	ff	fi	fl	ffi	ffl
02x	ı	ȷ	`	´	ˉ	˘	¯	°
03x	¸	ß	æ	œ	ø	Æ	Œ	Ø
04x	-	!	”	#	$	%	&	’
05x	(	)	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7
07x	8	9	:	;	¡	=	¿	?
10x	@	A	B	C	D	E	F	G
11x	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W
13x	X	Y	Z	[	“	]	^	˙
14x	‘	a	b	c	d	e	f	g
15x	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w
17x	x	y	z	–	—	˝	~	¨

The basic fonts of TEXmay be divided into two categories. The first one is for text and also include the typefaces `cmbx`, `cmsl` and `cmti`. The abbreviations stand for Computer Modern Bold Extended, Computer Modern Slanted and Computer Modern Text Italic. The second group of typefaces in the family is mainly used for mathematical writing. These include `cmtt`, `cmmi`, `cmsy` and `cmex`—Computer Modern Typewriter Type, Computer Modern Math Italic, Computer Modern Math Symbols and Computer Modern Math Extension. The typeface `cmtt` is a monospaced font.

The Computer Modern family may be extended with other fonts. For example, the `cmcsc` font corresponding to a font with small capitals letters is often used in TEX. An example of the Cacographic-SmallCaps font and the `cmcsc10` font follows.

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.

THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.

8.3 Special characters

Earlier, we mentioned that TEX uses boxes to decide the location of each character. The `tfm` file does not contain any bounding box information. For a character such as "d", the dimensions of the letter, including the side bearings, may be computed from the width,

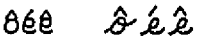
the height and the depth parameters. In the case of accents, the dimensions of the letter cannot be calculated so easily. TEX uses the x-height of the font as a reference point. It assumes that an accent can be placed over the letter x without any translation of the accent character. Hence to compose an accented letter with a base character that is smaller than the x-height, TEX would need to raise the box of the acute character by an amount of (CHARHT of base character - XHEIGHT) units

In the case of the cedilla, the character has no height. Hence TEX uses the difference between the baseline and the depth of a character to figure out by how much the cedilla needs to be moved.

With theses requirements in mind, a newly generated font may need to be slightly adjusted. A program, named `fixtexfont`, was developed to make the appropriate modifications to a TEX font. For modifying a font file `font.ps`, it is invoked by the command line

```
fixtexfont [-s distance] font.ps
```

The optional parameter `-s` allows the user to specify extra space between the accents and the letters. The distance supplied to the `-s` parameter corresponds to the distance in the 1000 x 1000 character space. The following examples show the typeface Isabella1 and the typeface Isabella2 with extra space added between the letters and the accents.



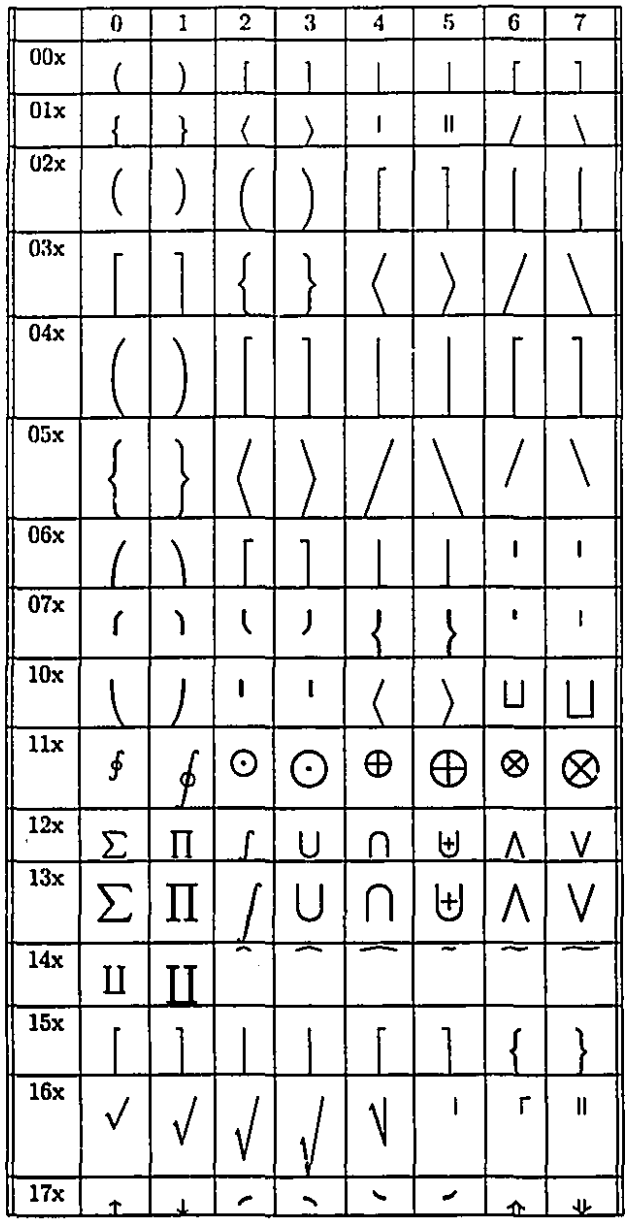
8.4 Mathematics

TEX is primarily geared to scientific writing. It is important to understand how TEX works with mathematical formulas in order to generate typefaces for the simulation of handwritten mathematics.

As mentioned earlier, special typefaces are used for mathematics. TEX math mode uses by default the `cmmi` typeface. The following table shows the character set of the `cmmi10` font.

	0	1	2	3	4	5	6	7
00x	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ
01x	Φ	Ψ	Ω	α	β	γ	δ	ϵ
02x	ζ	η	θ	ι	κ	λ	μ	ν
03x	ξ	π	ρ	σ	τ	υ	ϕ	χ
04x	ψ	ω	ε	ϑ	ϖ	ϱ	ς	φ
05x	$\leftharpoonup$	$\leftharpoondown$	$\rightharpoonup$	$\rightharpoondown$	$\triangleright$	$\triangleleft$	$\triangleright$	$\triangleleft$
06x	0	1	2	3	4	5	6	7
07x	8	9	.	,	<	/	>	$\star$
10x	∂	A	B	C	D	E	F	G
11x	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W
13x	X	Y	Z	$\flat$	$\natural$	$\sharp$	$\smile$	$\frown$
14x	ℓ	a	b	c	d	e	f	g
15x	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w
17x	x	y	z	$\imath$	$\jmath$	$\wp$	$\vec{}$	$\hat{}$

The typefaces cmmi, cmsy and cmtt are not conceptually any different from cmr. On the other hand, the cmex typeface which consists of math extended characters is quite different. The following table shows the entire set.



We notice that some characters seem to be incomplete. With those character sections, TEX may build characters of arbitrary height without altering their visual aesthetics. The parentheses in the following example are actually composed of three parts, mainly the top, extension and bottom parts. The extension section may be repeated as often as needed to reach the desired character height. The sections for the left parenthesis are the characters

060, 102 and 100 in the octal representation.

$$A = \begin{pmatrix} x-\lambda & 1 & 0 \\ 0 & x-\lambda & 1 \\ 0 & 0 & x-\lambda \end{pmatrix}.$$

These characters are more complicated to handle as the different sections need to coincide perfectly in the character space metric. Moreover extra parameters in the `vpl` file need to be defined in order for TEX to typeset the formulas correctly. More information on these parameters can be found in Appendix G of the TEXbook (Knuth, 1986b).

8.5 Simple mathematics

If we restrict ourselves to simple formulas, the generated Cacographic family may be used without too many modifications. TEX has two mathematical modes, text and display. Hence certain characters such as the summation sign ($\sum$) and the integral ($\int$) need to be created in two distinct sizes. The special token `NEXTLARGER` in the `vpl` file—see Appendix D—indicates the character that has the next bigger size.

Here again a small program was written. The command line

```
fixvpl font.vpl
```

fixes the `vpl` file `font.vpl`. It adds the appropriate tokens to the characters of a Math Extension typeface. The `vpl` file may then be used to create the correct `tfm` file.

Some examples of simple mathematics are shown below.

Bernoulli's inequality may be expressed as: if $x > -1$, then

$$(1+x)^n \geq 1+nx \text{ for all } n \in N.$$

The natural logarithmic function is

$$\log x = \int_1^x \frac{1}{t}dt, \quad x > 0.$$

$$2x \cup 3y \vee 4z = 8a \odot 3b * 7c$$

$$\Phi(\alpha) = \int_{-\infty}^{\infty} e^{-x^2/2}dx$$

$$\cos^2(\theta) + \sin^2(\theta) = 1$$

$$\phi(\ell) = \ell^2 + 2\ell + 1$$

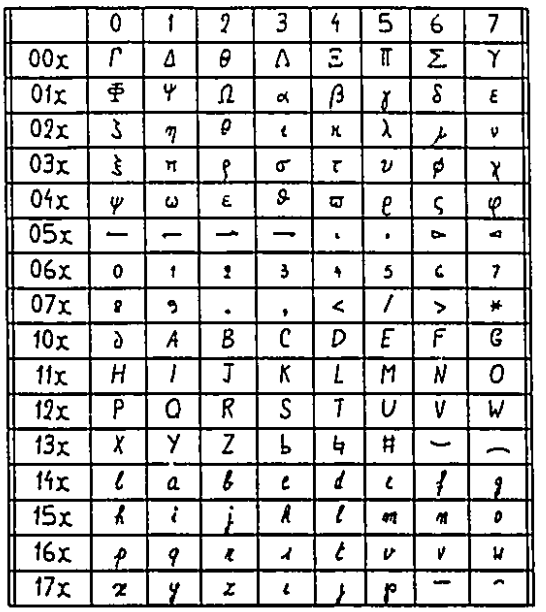
$$\lim_{\beta \to \infty} \frac{1}{\beta} = 0$$

The following tables show the generated mathematical typefaces.

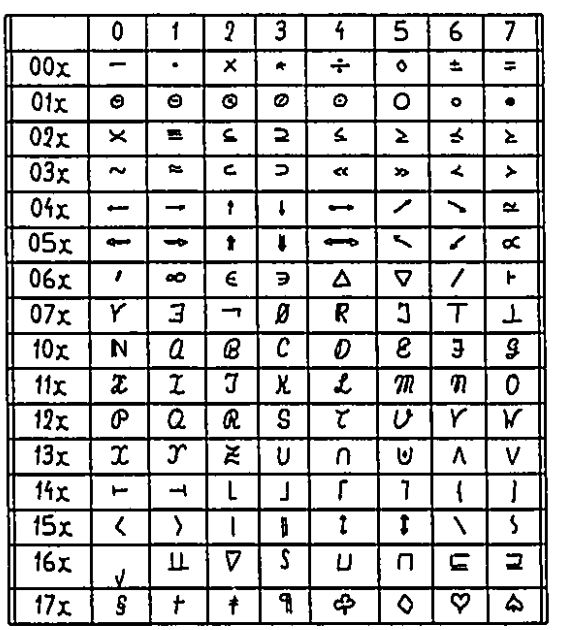
Cacographic-Roman:

	0	1	2	3	4	5	6	7
00x	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ
01x	Φ	Ψ	Ω	ff	fi	fl	ffi	ffl
02x	ı	ȷ	`	´	ˇ	˘	¯	°
03x	¸	ß	æ	œ	ø	Æ	Œ	Ø
04x	-	!	"	#	$	%	&	'
05x	(	)	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7
07x	8	9	:	;	¡	=	¿	?
10x	@	A	B	C	D	E	F	G
11x	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W
13x	X	Y	Z	[	"	]	^	·
14x	'	a	b	c	d	e	f	g
15x	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w
17x	x	y	z	–	—	¨	~	¨

Cacographic-MathItalic:



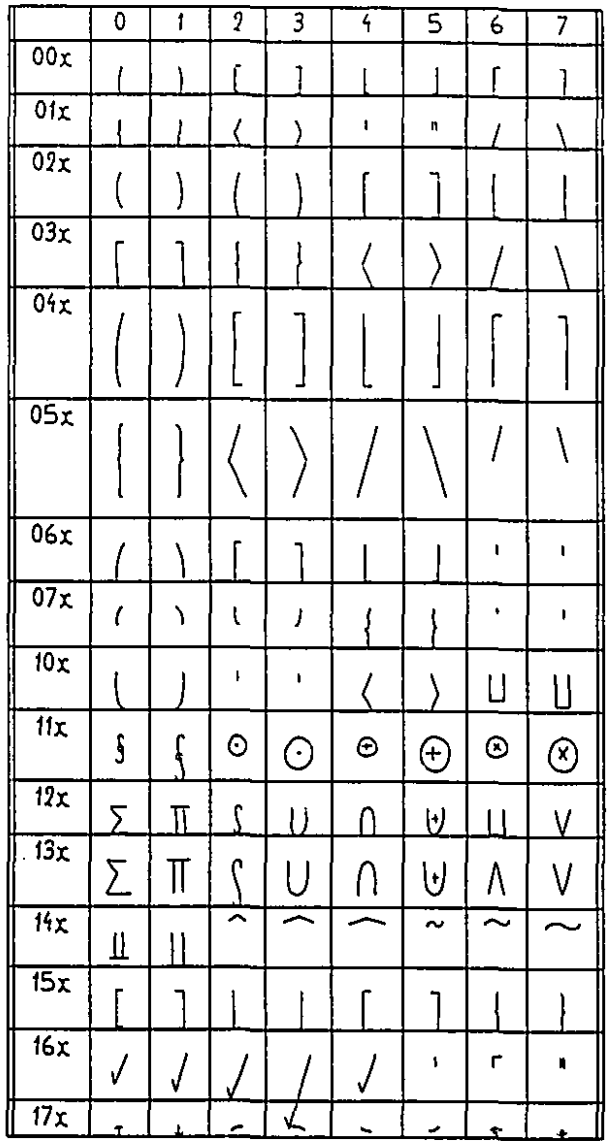
Cacographic-Symbol:



Cacographic-SmallCaps:

	0	1	2	3	4	5	6	7
00x	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ
01x	Φ	Ψ	Ω	ı	ȷ	`	´	ˇ
02x	˘	¯	˚	¸	˛	˜	˝	˙
03x	‚	ß	æ	œ	ø	Æ	Œ	Ø
04x	¿	¡	ª	º	§	%	¶	†
05x	(	)	[	]	.	-	.	/
06x	0	1	2	3	4	5	6	7
07x	8	9	:	;	¡	=	¿	?
10x	@	A	B	C	D	E	F	G
11x	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W
13x	X	Y	Z	[	"	]	^	·
14x	`	a	b	c	d	e	f	g
15x	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w
17x	x	y	z	–	—	˝	~	¨

Cacographic-MathExtension:



Cacographic-TypewriterType:

	0	1	2	3	4	5	6	7
00x	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ
01x	Φ	Ψ	Ω	↑	↓	'	¡	¿
02x	ı	ȷ	`	´	˘	ˇ	¯	˚
03x	¸	ß	æ	œ	ø	Æ	Œ	Ø
04x	␣	!	"	#	$	%	&	'
05x	(	)	*	+	,	-	.	/
06x	0	1	2	3	4	5	6	7
07x	8	9	:	;	<	=	>	?
10x	@	A	B	C	D	E	F	G
11x	H	I	J	K	L	M	N	O
12x	P	Q	R	S	T	U	V	W
13x	X	Y	Z	[	\	]	^	_
14x	`	a	b	c	d	e	f	g
15x	h	i	j	k	l	m	n	o
16x	p	q	r	s	t	u	v	w
17x	x	y	z	{	|	}	~	¨

9 Conclusion

In this thesis we have presented a new method for approximating outlines with Bézier curves. The generated outlines are guaranteed to be C^2 continuous. The method employs different optimization techniques to achieve the desired result. The derived algorithm may be employed for the generation of typefaces. It has been shown to be very flexible and to converge rapidly.

Many issues still remain open. The weights used in the quality function Φ depend upon the resolution employed and on the size of the drawn characters. The derivation of a resolution-independent method as well as an adaptive approach for evaluating the weight values is the biggest issue to solve.

Dynamic fonts derived from handwritten characters pose a serious challenge. Also, TEX font families require more attention for mathematical writing. Methods for automatically handling glyphs of the extended character set need to be derived.

10 References

D. Adams and J. André, "New trends in digital typography," in: *Raster Imaging and Digital Typography: Proceedings of the International Conferences, Ecole Polytechnique Fédérale, Lausanne, Switzerland, October 1989*, ed. J. André and R. D. Hersch, pp. 14—21, Cambridge University Press, Cambridge, 1989.

Adobe Systems Inc. , *PostScript Language Reference Manual*, Addison-Wesley, Reading, MA, 1990a.

Adobe Systems Inc. , *Adobe Font Metric Files Specification Version 3.0*, Adobe Systems Inc., 1990c.

Adobe Systems Inc. , *Adobe Type 1 Font Format*, Addison-Wesley, Reading, MA, 1990b.

J. André and B. Borghi, "Dynamic fonts," in: *Raster Imaging and Digital Typography*, ed. J. André and R. D. Hersch, pp. 198—204, Cambridge University Press, Cambridge, 1989.

G. Avrahami and V. Pratt, "Sub-pixel Edge Detection in Character Digitization," in: *Raster Imaging and Digital Typography II - Papers from the second*, ed. R. A. Morris and J. André, pp. 54—64, Cambridge University Press, Cambridge, 1991.

R. G. Bartle and D. R. Sherbert, *Introduction to Real Analysis*, John Wiley & Sons, New York, 1982 .

C. A. Bigelow and D. Day, "Digital typography," *Scientific American*, vol. 249, pp. 106—119, 1983.

C. A. Bigelow and K. Holmes, "Notes on Apple 4 fonts," *Electronic Publishing*, vol. 4, pp. 171—181, 1991.

N. Billawala, "Pandora: an experience with METAFONT," in: *Raster Imaging and Digital Typography: Proceedings of the International*, ed. J. André and R. D. Hersch, pp. 34—53, Cambridge University Press, Cambridge, 1989.

W. Böhm, "Cubic B-Spline curves and surfaces in computer-aided geometric design," *Computing*, vol. 19, pp. 29—34, 1977.

W. Böhm, G. Farin, and J. Kahmann, "A survey of curve and surface methods in CAGD," *Computer aided Geometric Design*, vol. 1, pp. 1—60, 1984.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, MA, 1990.

P. J. Davis and P. Rabinowitz, *Methods of Numerical Integration*, Academic Press, Orlando, FL, 1984.

L. Devroye, "The compound random search algorithm," *Proceedings of the International Symposium on Systems Engineering*, pp. 105—110, Lafayette, IN, 1972.

E. H. Doojies, "Rendition of quasi-calligraphic script defined by pen trajectory," *Raster Imaging and Digital Typography*, in: *Raster Imaging and Digital Typography: Proceedings of the International*, ed. J. André and R. D. Hersch, pp. 251—260, Cambridge University Press, Cambridge, 1989.

R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, John Wiley & Sons, New York, 1973.

G. Farin, *Curves and Surfaces for CAGD, A practical guide*, Academic Press, New York, 1993 .

J. D. Foley, A. van Dam, S. Feiner, and J. Hughes, *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, Reading, MA, 1992.

J. Gonczarowski, "A fast approach to auto-tracing (with parametric cubics)," in: *Raster Imaging and Digital Typography*, ed. R. A. Morris and J. André, vol. 2, pp. 1—15, Cambridge University Press, Cambridge, 1991.

Y. Haralambous, "Parametrization of PostScript fonts through METAFONT —alternative to Adobe multiple master fonts," *Electronic Publishing*, vol. 6, pp. 145–157, 1993.

J. D. Hobby, "Smooth, easy to compute interpolating splines," *Discrete Computational Geometry*, vol. 1, pp. 123—140, 1986.

K. Hoffman and R. Kunze, *Linear Algebra*, Prentice Hall, Englewood Cliffs, NJ, 1971.

K. Itoh and Y. Ohno, "A curve fitting algorithm for character fonts," *Electronic Publishing*, vol. 6, pp. 195–205, 1993.

P. Karow, *Digital Typefaces*, Springer-Verlag, Berlin, 1994a.

P. Karow, *Font Technology*, Springer-Verlag, Berlin, 1994b.

D. E. Knuth, *The* METAFONT *book*, Addison-Wesley, Reading, MA, 1986a.

D. E. Knuth, *The TEXbook*, Addison-Wesley, Reading, Mass, 1986b.

D. E. Knuth, *Computer Modern Typefaces*, Addison-Wesley, Reading, Mass, 1986c.

D. E. Knuth, "A punk meta-font," *TUGboat*, vol. 9, pp. 152–168, 1988.

J. R. Manning, "Continuity conditions for spline curves," *The Computer Journal*, vol. 17, pp. 181–186, 1974.

J. Matyas, "Random optimization," *Automation and Remote Control*, vol. 26, pp. 244—251, 1965 .

H. McGilton and M. Campione, *PostScript by Example*, Addison-Wesley, Reading, MA, 1992.

M. Minsky and S. Papert, *Perceptrons, an Introduction to Computational Geometry*, MIT Press, Harvard, MA, 1969.

R. Männer and H.-P. Schwefel, "Parallel Problem Solving from Nature," vol. 496, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1991.

T. Pavlidis, *Algorithms for Graphics & Image Processing*, Computer Science Press, Rockville, MD, 1982.

M. Plass and M. Stone, "Curve-fitting with piecewise parametric cubics," *Computer Graphics*, vol. 17, pp. 229—239, 1983 .

F. P. Preparata and M. I. Shamos, *Computational Geometry, an Introduction*, Springer-Verlag, New York, NY, 1985.

T. Pudet, "Dessin à main levée et courbes de Bézier," Digital, Paris Research Laboratory, 1993.

A. H. G. Rinnooy Kan and G. T. Timmer, "Stochastic global optimization methods part I: clustering methods," *Mathematical Programming*, vol. 39, pp. 27–56, 1987a.

A. H. G. Rinnooy Kan and G. T. Timmer, "Stochastic global optimization methods part II: multi level methods," *Mathematical Programming*, vol. 39, pp. 57–78, 1987b.

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

P. J. Schneider, "An algorithm for automatically fitting digitized curves," in: *Graphics Gems*, ed. A. S. Glassner, pp. 612–626, Academic Press, San Diego, CA, 1990.

M. A. Schumer and K. Steiglitz, "Adaptive step size random search," *IEEE Transactions on Automatic Control*, vol. 13, pp. 270—276, 1968.

B.-Q. Su and D.-Y. Liu, *Computational Geometry—Curve and Surface Modeling*, Academic Press, Boston, 1989.

E. W. Swokowski, *Calculus with Analytic Geometry*, Prendle Webes & Schmidt, Boston, MA, 1975.

A. Törn and A. Žilinskas, *Global Optimization*, Lecture Notes in Computer Science, vol. 350, Springer-Verlag, Berlin, 1989.

E. van Blokland and J. van Rossum, "Different approaches to lively outlines," in: *Raster Imaging and Digital Typography II*, ed. R. A. Morris and J. André, pp. 28—33, Cambridge University Press, Cambridge, 1991.

L. W. Wallis, *Modern Encyclopedia of Typefaces, 1960-90*, Van Nostrand Reinhold, New York, NY, 1990.

A. A. Zhigljavsky, *Theory of Global Random Search*, Kluwer Academic Publishers, Hingham, MA, 1991.

APPENDIX A PostScript bitmap font

Example of a PostScript font file created by the program `scrpt2ps`. The header and one character of the Bazooka font as well as the trailer part of the file are shown. The encoding is also not entirely shown.

```
%!
%Creator: Scrpt2ps

9 dict dup begin
  /FontType 3 def
  /FontMatrix [1 0 0 1 0 0] def
  /Encoding 256 array def
0 1 255 {Encoding exch /.notdef put} for
Encoding
dup 0 /Gamma put
dup 1 /Delta put
              .
              .
              .
dup 126 /tilde put
dup 127 /dieresis put
   255 /.notdef put
  /BuildChar
    { 0 begin
        /char exch def
        /fontdict exch def
        /charname fontdict /Encoding get char get def

        /charinfo fontdict /CharData get charname get def
        /wx charinfo 0 get def
        /charbbox charinfo 1 4 getinterval def
        wx 0 charbbox aload pop setcachedevice
        charinfo 5 get charinfo 6 get true

        fontdict /imagemaskmatrix get
        dup 4 charinfo 7 get put
        dup 5 charinfo 8 get put
        charinfo 9 1 getinterval cvx
        imagemask
      end
    } def
/BuildChar load 0 6 dict put
```

```
/imagemaskmatrix [406 0 0 -406 0 0 ] def
/CharData 256 dict def
CharData begin

 /dieresis [  0.20   0.00  0.37  0.20  0.45  82  33 -1.5 182.5
<000000000000001fc00000
000000000000007ff00000
00000000000001fffc0000
00000000000007fffe0000
007f000000000fffff0000
07ff800000001fffff8000
1fffc00000001fffff8000
3fffe00000001fffffc000
7fffe00000003fffffe000
7fffe00000003fffffe000
fffff00000003ffffff000
fffff00000003ffffff800
fffff00000003ffe3ffc00
fffff80000003ffe3ffc00
fffff80000007ffe7fe000
ffffffc0000003ffefff000
ffffffc0000001ffffff000
7ffffe0000001ffffff800
7ffffe0000000ffffffc00
7ffffe00000007fffffc00
7fffffc00000003fffffe00
7fffffc00000001fffffe00
7fffffc00000000ffffff80
3fffffc00000000fffffffc0
3fffffc00000000fffffffc0
3fffffc000000007ffffe00
1fffffc000000007ffffe00
1fffffc000000001ffffe00
1fffffc000000000007ffc00
0ffff8000000000003fe000
0ffff0000000000001f0000
03ffc00000000000000000
007e000000000000000000
>] def
/space [ 0.24 0 0 0 0 1 1 0 0 <> ] def
/.notdef [.24 0 0 0 0 1 0 0 <>] def
end
/FontBBox [ 0.91 0.64 0.00 -0.30 ] def
/UniqueID 2 def
end
/Bitfont exch definefont pop
```

APPENDIX B Type 1 format

Here is an example of a type 1 PostScript font. The vector encoding is not entirely shown.

A small portion of the encoded body is also shown.

```
%!FontType1-1.0: TropDePoils 1.0
%%CreationDate: Mon Aug 15 09:12:23 1994
% Parameter Used: -i 4000 -a 0.90 -v 0.99 -S -gs -FN TropDePoils
% -FF TropDePoils -FFN TropDePoils -SP 500 -FV 1.0 all.ps
11 dict begin
        /FontInfo 8 dict dup begin
        /Version (1.0) readonly def
        /FullName (TropDePoils) readonly def
        /FamilyName (TropDePoils) readonly def
        /Weight (medium) readonly def
        /ItalicAngle 0 def
        /isFixedPitch false def
        /UnderlinePosition -98 def
        /UnderlineThickness 54 def
end readonly def
/FontName /TropDePoils def
/PaintType 0 def
/FontType 1 def
/FontMatrix [1 6000 div 0 0 1 6000 div 0 0 ] def
/Encoding 256 array
0 1 255 { 1 index exch /.notdef put } for
dup 32 /space put
dup 33 /exclam put

                        .
                        .
                        .

dup 251 /germandbls put
readonly def
/FontBBox{      6       6    5994    5994} def
/UniqueID 4829400 def
currentdict end
currentfile eexec
D9D66F633B846A989B9974B0179FC6CC445BC7C8A959A39A32E9DCE7FAEF17EE
3BEC9F50CF7269C04E6A63459A9211B2F19CE3B85116D00C0080DED86FE95FED
7261DA657C3D97CA8E203F5D840114FCDF36CC39021D69046B9667441E78D684
7B79AFD6950C60389C705C3F0325B605745CFC39E5EE5C5EB4DBD13BE4BE149E
400AC7B22119FA56DDC4CD856E7693259B48BD11C76F33BFB6C8B2EEB7FF534C
9807EC0FA9DDC963278FA1A136C4C48CCEC5BFC1E247D87F40262F1201BFA09F
B1E29FD65573AB793C4326085BEBF71D421AAED9BDB4E8A39EA39C9F5918009B
```

943E4A90D995DB851A07B6EE53B2CAC7328982DF620429308D57253B9E456985

.
.
.

05FCE9C56436216C3A3E6ABB98D13D0C4493E1AAA3BBAB36C1317CD71092A3CF
477C4971E3F4ABD42EBF2F0FF521D0BD916E16759DEE4EE82B1BEF0228AE6C09
C133623F2366AFCFABB0D19C446A9F73188B9CC726FE457F83F24723909EF7A3
AE0FA5893E19596872D4F4CD8882D5432CD0357F46746F02FC67FFAB867BAEFB
332A878B063DCAE2ABC4FCDFBA07A1AA12BD842C0884128587FAECD0660C466C
0CA8850C400E6A659CBC07BD
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
cleartomark

APPENDIX C Non-encrypted type 1 format

An example of a generated type 1 font file before the encryption.

```
%!FontType1-1.0: TropDePoils 1.0
%%CreationDate: Mon Aug 15 09:12:23 1994
% Parameter Used: -i 4000 -a 0.90 -w 0.99 -S -gs -FN TropDePoils
% -FF TropDePoils -FFN TropDePoils -SP 500 -FV 1.0 all.ps
11 dict begin
        /FontInfo 8 dict dup begin
        /Version (1.0) readonly def
        /FullName (TropDePoils) readonly def
        /FamilyName (TropDePoils) readonly def
        /Weight (medium) readonly def
        /ItalicAngle 0 def
        /isFixedPitch false def
        /UnderlinePosition -98 def
        /UnderlineThickness 54 def
end readonly def
/FontName /TropDePoils def
/PaintType 0 def
/FontType 1 def
/FontMatrix [1 6000 div 0 0 1 6000 div 0 0 ] def
/Encoding 256 array
0 1 255 { 1 index exch /.notdef put } for
dup 32 /space put

                        .
                        .
                        .

dup 250 /oe put
dup 251 /germandbls put
readonly def
/FontBBox{      6       6    5994    5994} def
/UniqueID 4829400 def
currentdict end
currentfile eexec
dup /Private 10 dict dup begin
/|-{readonly def} def
/password 5839 def
/|{readonly put} def
/UniqueID 4829400 def
/-|{string currentfile exch readhexstring pop} def
/BlueValues[] def
/-|{string currentfile exch readstring pop} def
/MinFeature{16 16} def
```

```
 dup 0 {
return
} |
 dup 1 {
return
} |
 dup 2 {
return
} |
 dup 3 {
return
} |
readonly def
2 index /CharStrings 256 dict dup begin

                    .
                    .
                    .

/hyphen {
    127    1907 hsbw
   1074    2621 rmoveto
155 0 177 -7 95 -4 rrcurveto
95 -4 14 0 14 -32 rrcurveto
14 -32 14 -64 -4 -32 rrcurveto
-4 -32 -21 0 -120 -4 rrcurveto
-120 -4 -219 -7 -254 -11 rrcurveto
-254 -11 -290 -14 -159 0 rrcurveto
-159 0 -28 14 -11 32 rrcurveto
-11 32 7 49 14 28 rrcurveto
14 28 21 7 42 7 rrcurveto
42 7 64 7 95 7 rrcurveto
95 7 127 7 131 7 rrcurveto
131 7 134 7 155 0 rrcurveto
endchar
} |-

                    .
                    .
                    .

/.notdef {
      0    3000 hsbw
endchar
} |-
/space {
      0    3000 hsbw
```

```
endchar
} |-
end
end
readonly put
readonly put
dup /FontName get exch definefont pop
mark currentfile closefile
```

APPENDIX D Virtual property list

Example of the virtual property list for the CMR10 font.

```
(FAMILY CMR)
(FACE O 352)
(CODINGSCHEME TEX TEXT)
(DESIGNSIZE R 10.0)
(COMMENT DESIGNSIZE IS IN POINTS)
(COMMENT OTHER SIZES ARE MULTIPLES OF DESIGNSIZE)
(CHECKSUM O 11374260171)
(FONTDIMEN
   (SLANT R 0.0)
   (SPACE R 0.333334)
   (STRETCH R 0.166667)
   (SHRINK R 0.111112)
   (XHEIGHT R 0.430555)
   (QUAD R 1.000003)
   (EXTRASPACE R 0.111112)
   )


(CHARACTER O 22 (comment grave)
   (CHARWD R 0.500002)
   (CHARHT R 0.694445)
   )
(CHARACTER O 23 (comment acute)
   (CHARWD R 0.500002)
   (CHARHT R 0.694445)
   )

(CHARACTER O 30 (comment cedilla)
   (CHARWD R 0.444446)
   (CHARDP R 0.170138)
   )


(CHARACTER C c
   (CHARWD R 0.444446)
   (CHARHT R 0.430555)
   (COMMENT
      (KRN C h R -0.027779)
      (KRN C k R -0.027779)
      )
   )
(CHARACTER C d
   (CHARWD R 0.555557)
```

```
   (CHARHT R 0.694445)
   )
(CHARACTER C e
   (CHARWD R 0.444446)
   (CHARHT R 0.430555)
   )
```

The following is an excerpt of the CMEX10 vpl file. We notice that extra parameters in the FONTDIMEN field are present.

```
(FAMILY CMEX)
(FACE O 352)
(CODINGSCHEME TEX MATH EXTENSION)
(DESIGNSIZE R 10.0)
(COMMENT DESIGNSIZE IS IN POINTS)
(COMMENT OTHER SIZES ARE MULTIPLES OF DESIGNSIZE)
(CHECKSUM O 37254272422)
(FONTDIMEN
   (SLANT R 0.0)
   (SPACE R 0.0)
   (STRETCH R 0.0)
   (SHRINK R 0.0)
   (XHEIGHT R 0.430555)
   (QUAD R 1.000003)
   (EXTRASPACE R 0.0)
   (DEFAULTRULETHICKNESS R 0.039999)
   (BIGOPSPACING1 R 0.111112)
   (BIGOPSPACING2 R 0.166667)
   (BIGOPSPACING3 R 0.2)
   (BIGOPSPACING4 R 0.6)
   (BIGOPSPACING5 R 0.1)
   )
(CHARACTER O 0
   (CHARWD R 0.458336)
   (CHARHT R 0.039999)
   (CHARDP R 1.160013)
   (NEXTLARGER O 20)
   )
(CHARACTER O 1
   (CHARWD R 0.458336)
   (CHARHT R 0.039999)
   (CHARDP R 1.160013)
   (NEXTLARGER O 21)
   )
(CHARACTER O 2
```

```
   (CHARWD R 0.416669)
   (CHARHT R 0.039999)
   (CHARDP R 1.160013)
   (NEXTLARGER O 150)
   )

(CHARACTER O 60
   (CHARWD R 0.875003)
   (CHARHT R 0.039999)
   (CHARDP R 1.760019)
   (VARCHAR
      (TOP O 60)
      (BOT O 100)
      (REP O 102)
      )
   )
```