Rasterization Techniques for Chinese Outline Fonts

Kwong-ho WU
The Department of Computer Science
The Chinese University of Hong Kong

A dissertation submitted for the degree of
Master of Philosophy
June 1994

Acknowledgments

Without the kindly help from my dearest friends, this thesis would never have been completed. I would like to thank my supervisor, Dr. Y. S. Moon, for his advice, guidance and encouragement. Without Mr. C. C. Poon's help, I could hardly get my project started. I am also grateful to Mr. C. K. Chen for his advice and technical support throughout this project.

K. H. Wu

Abstract

An outline font gives the shape description of a character with straight lines, arcs and curves. It is now a very popular format for representing Chinese font data because it demands relatively less storage and can be scaled to any size without shape distortion. However, the outline font must be rasterized, i.e. converted to bitmap, before using in raster output devices and the resulting bitmap characters usually look unsatisfactory.

In this thesis, we try to make some improvement on the steps (scan conversion and filling) of the present rasterization method to build a more efficient rasterizer. Also, we are interested in discovering how the rasterization speed would be related to the features (such as stroke count) of the Chinese character.

The other objective is to automatically generate hints for Chinese font so that the resulting bitmap characters would retain most of their features. Font in Ming style is just used for illustration and, in fact, the method can be applied for many other font styles, such as Gothic.

Finally, a series of experiments are done to compare the performances of the suggested methods with the old methods.

Contents

1 Introduction **1**

1.1 Outline Fonts 2

1.1.1 Advantages and Disadvantages 4

1.1.2 Representations 4

1.1.3 Rasterization 5

1.2 Introduction to This Thesis 6

1.2.2 Organization 7

1.2.1 Objectives. 7

2 Chinese Characters Fonts **8**

2.1 Large Character Set 8

2.2 Font Styles 8

2.3 Storage Problems 9

2.4 Hierarchical Structure 10

2.5 High Stroke Count 11

3 Rasterization **13**

3.1 The Basic Rasterization. 13

3.1.1 Scan Conversion 14

3.1.2 Filling Outline 16

3.2 Font Rasterization 17

3.2.1 Outline Scaling 17

3.2.2 Hintings 17

3.2.3 Basic Rasterization Approach for Chinese Fonts 18

3.3 Hintings 20

3.3.1 Phase Control 20

3.3.2 Auto-Hints 21

3.3.3 Storage of Hintings Information in TrueType Font and Postscript Font . 22

4 An Improved Chinese Font Rasterizer **24**

4.1 Floating Point Avoidance 24

4.2 Filling 25

4.2.1 Filling with Horizontal Scan Line 25

4.2.2 Filling with Vertical Scan Line 27

4.3 Hintings 30

4.3.1 Assumptions . . . 30
4.3.2 Maintaining Regular Strokes Width . . . 30
4.3.3 Maintaining Regular Spacing Among Strokes . . . 34
4.3.4 Hintings of Single Stroke Contour . . . 42
4.3.5 Storing the Hinting Information in Font File . . . 49
4.4 A Rasterization Algorithm for Printing . . . 51
4.4.1 A Simple Algorithm for Generating Smooth Characters . . . 52
4.4.2 Algorithm . . . 54
4.4.3 Results . . . 54

5 Experiments . . . **56**
5.1 Apparatus . . . 56
5.2 Experiments for Investigating Rasterization Speed . . . 56
5.2.1 Investigation into the Effects of Features of Chinese Fonts on Rasterization Time . . . 56
5.2.2 Improvement of Fast Rasterizer . . . 57
5.2.3 Details of Experiments . . . 57
5.3 Experiments for Rasterization Speed of Font File with Hints . . . 57

6 Results and Conclusions . . . **58**
6.1 Observations . . . 58
6.1.1 Relationship Between Time for Rasterization and Stroke Count . . . 58
6.1.2 Effects of Style . . . 61
6.1.3 Investigation into the Observed Relationship . . . 62
6.2 Improvement of the Improved Rasterizer . . . 64
6.3 Gain and Cost of Inserting Hints into Font File . . . 68
6.3.1 Cost . . . 68
6.3.2 Gain . . . 68
6.4 Conclusions . . . 69
6.5 Future Work . . . 69

Appendix

Chapter 1

Introduction

A Chinese system has two fundamental functions: inputting and outputting Chinese characters. Actually, a Chinese input method is a mapping from the features (such as shape, structure and pronunciation) of a character to a sequence of key strokes which can be entered to the computer by keyboard or other inputting devices. These key strokes are, in general, represented by roman characters. Then the internal representation code (e.g. Big5 code) corresponding to this sequence of key strokes is found standing for that character in computer (Figure 1.1). Before displaying a character on screen or other output devices, the system must use the internal representation code to access the font data precisely describing the shape of that character, which is then converted into a format for display or printing (Figure 1.2).

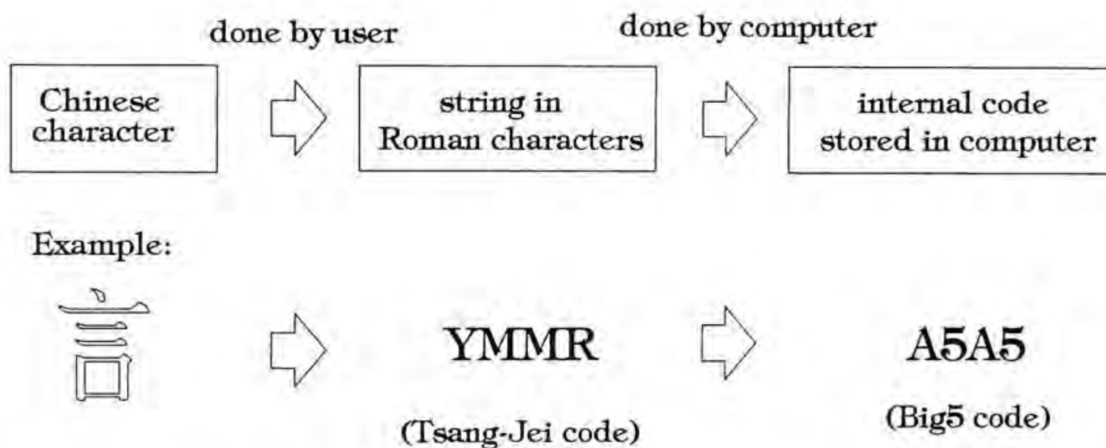


Figure 1.1: Chinese input method.

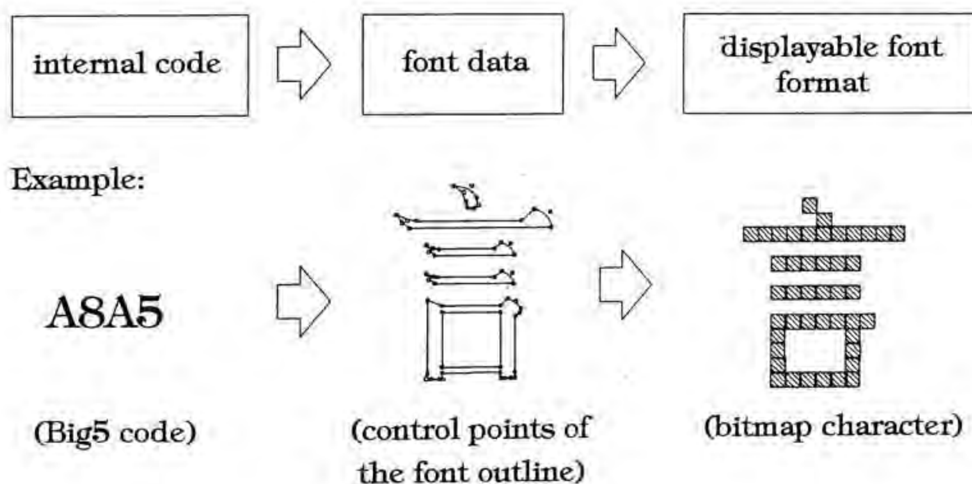


Figure 1.2: Steps of displaying a character.

The input speed depends very much on the design of the input method and the skill of the operator. Many input methods have been proposed to make it easier for the user to retrieve the input code based on the features of the character, shortening the training time for a novice.

The output speed is mainly dependent on three factors: the time for locating the font data, the time for transforming the font data into a displayable font format, and the time for displaying the font data on the output device. The first and the third factors are mainly determined by the data structure of the font file and the design of the output device respectively. Probably, the second factor would be the critical one, especially when a large number of characters are to be generated. If this conversion step can be sped up, the efficiency of the whole system can be much improved.

In the past, bitmap font was not only used for storage but also employed for display and printing. Because bitmap font can be directly used by raster output devices like screen and printer, font conversion was unnecessary. However, since characters in bitmap representation are practically non-scalable and expensive in terms of storage requirement, some scalable and less storage demanding font representations, such as outline font, are developed.

Rasterization is the process to convert an outline character into the corresponding bitmap character. In low-resolution output devices, the bitmaps generated from outline characters usually do not look good enough. Grid-fitting or hinting should be applied to reshape the outline character before a bitmap character with more regular and satisfactory shape can be generated.

1.1 Outline Fonts

An outline font gives the shape description of a character with straight lines, arcs and curves. Originally, this technique is not raster oriented. The earlier output devices, such as plotter and screen, were only able to draw lines and points, so they had to make use of unfilled outline fonts. Before outputting in a bitmap device, these fonts must first be rasterized, that is, the curves and lines are converted into filled bitmap fonts.

If the outline description of the font consists only of straight lines, it is a *vector font* (Figure 1.3). The bitmap characters generated from vector font is satisfactorily smooth as long as the size of the output font is small enough. Clearly, when very large font is generated, corners would be likely to appear at the curve portions.

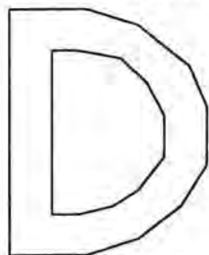


Figure 1.3: Vector font

More exact description of the font outline can be given by a combination of lines and splines (Figure 1.4). In our discussions, outline font is referring to this kind of format.

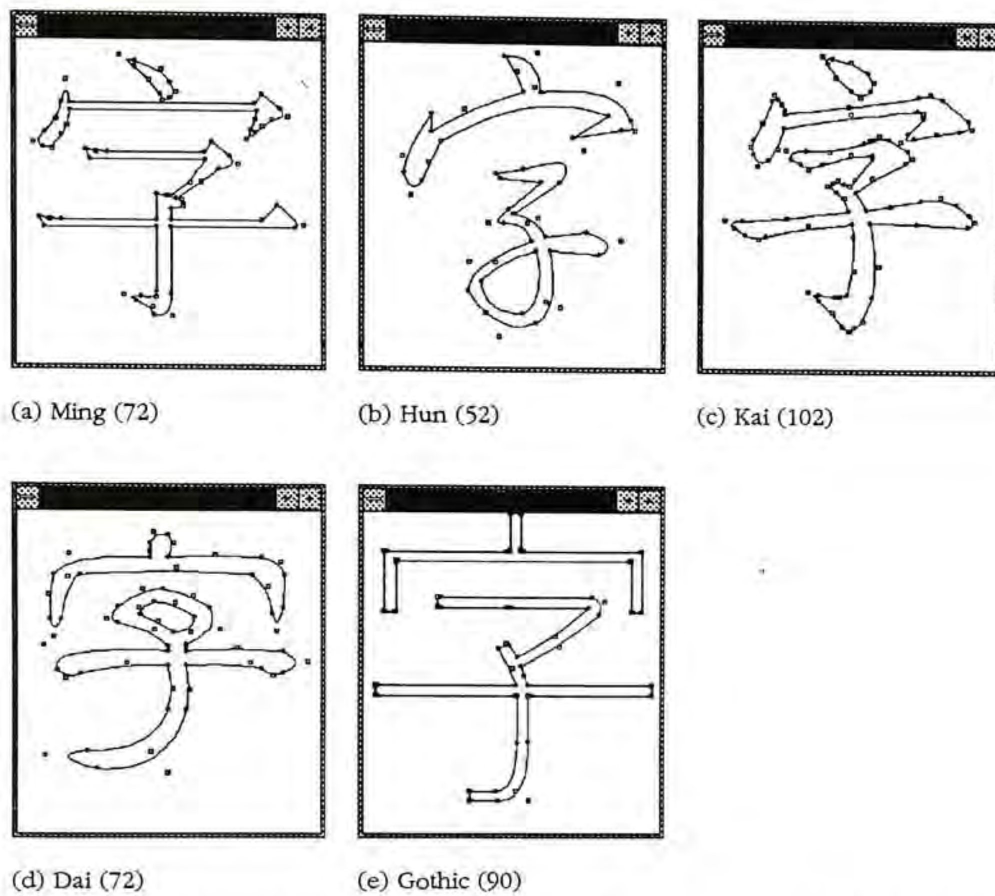


Figure 1.4: Outlines and control points (the number of control points is given in brackets)

A curve segment can be represented by a set of points which precisely control the shape of the curve (Figure 1.5). These points are called *control points*. Their effects on the shape of the curve depend very much on which type the curve is [Foley & van Dam 90, ch. 11]. Outline fonts can be constructed either manually [Rubinstein 88, pp. 134-140] [Kohen 89] or automatically by contour digitization [Moon & Hui 89] [Gonczarowski 91] [Liao & Huang 91].

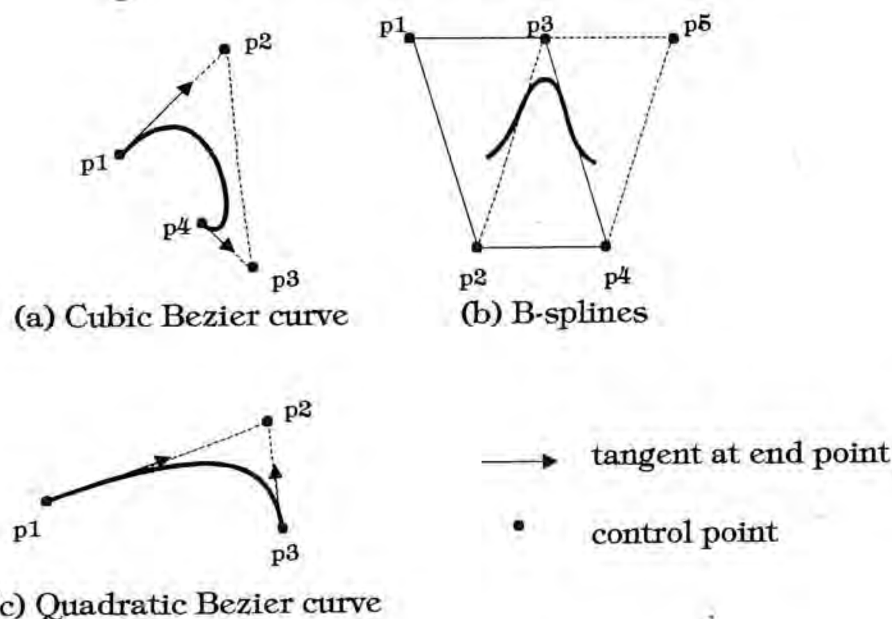


Figure 1.5: Some examples of parametric curves.

1.1.1 Advantages and Disadvantages

Because only the control points are stored in a outline font, fewer data are required, especially when compared to the case of bitmap font. Moreover, by adequately transforming the control points, the outline font can easily be scaled up or down without shape distortion. As a result, just one copy of font data is sufficient for each style to derive fonts in that style in many different sizes.

However, the rasterization process must take some time before the font can be used in raster output devices. The other shortcoming is that when a small raster character is generated from an outline font data, its symmetry and regularity may be partly or totally lost during rasterization.

1.1.2 Representations

In this section, we briefly go over the structures of two common outline font formats, TrueType and PostScript.

TrueType Font

TrueType font [TrueType 90], which is supported by Microsoft Windows, is one of the most popular outline font formats currently in use. We are now going into some details of the font file.

At the beginning of a TrueType font file, headers and tables are found to provide information for the access of font data. Besides containing coordinates of control points and some necessary information, the font data of a character can also include some instructions for grid-fitting so that only desired pixels will be turned on during rasterization.

Shapes of characters are described by lines and parametric cubic (or quadratic) curves. *A contour* is composed of a close series of lines and curves. Lines are defined by two consecutive on curve points. Curves are defined by control points which describe B-splines (e.g. Beziers), and the combination of on and off curve points depends solely on the order of the B-spline in use. For example, if second order Bezier is used, a curve is defined by one off curve point between two on curve points (Figure 1.5c).

Consecutive numbers are assigned to indicate the control points so that if the curve is followed in the direction of increasing point numbers, the filled area will always be to the right. A character can consist of one or many contours and a composite character can be constructed by combining two or more simpler character outlines.

PostScript Fonts

PostScript is a page description language (PDL), which is a common output format allowing documents to be transmitted between different systems for display and printing [Adobe 90] [Holzgang 92] [Moon & Cheang 91]. In fact, PostScript is a true programming language with variables, routines, operators and control structures. Since it is raster oriented, it provides font

facilities to rasterize outline fonts described with lines, arcs and curves into bitmaps of the required sizes. Also, its provision of font cache can raise the efficiency by preventing frequently used characters from being repeatedly rasterized.

Treating text characters as ordinary graphical objects, PostScript handles font with appropriate operators just like any other graphics. For example, from the interpreter's view, a triangle is conceptually equivalent to a character "p". As a result, it causes no trouble to combine text and graphics on a page. PostScript makes use of font dictionaries to describe fonts. A *font dictionary,* which can be referenced by a name literal, provides font data and procedures for the construction of all the characters in that font. This information is then used by the PostScript interpreter to rasterize characters on a string onto the current document being edited.

David A. Holzgang gives a brief summary of the PostScript font in [Holzgang 92]: *"First, PostScript actually does draw each character, using appropriate graphics operations; and second, the PostScript interpreter creates characters through the use of a font dictionary that contains all the information required to produce a given font, including the appropriate procedures for rendering each character."*

1.1.3 Rasterization

Rasterization or *rendering* is the process which converts outline shapes into bitmap images (Figure 1.6). By proportionally scaling the control parameters, bitmap shapes of arbitrary sizes can be generated from a single outline. Then the outline contour is put on a pixel grid and hintings are optionally applied on the outline shape for grid-fitting. Finally, all the internal pixels are turned on.

FUnits
pixel coordinates
pt size 12

Digitize outline with FUnit coordinates in TrueType font file

Scaler converts FUnits to pixel coordinates and scales outline to size requested by application

Outline "sized" to new grid

Interpreter executes instructions associated with glyph "B" and gridfits

Grid-fitted outline

Scan converter decides which pixels to turn on

Bitmap is rendered on raster device

Figure 1.6: Outline font rasterization [TrueType 90].

1.2 Introduction to This Thesis

In this thesis, our focus will be placed on outline font as input and single-bit-per-pixel bitmap fonts as output. So, font design system describing each character with individual program like METAFONT [Knuth 86] and multiple-bit-per-pixel bitmap fonts such as gray scale font [Warnock 80] (Figure 1.7) will not be covered.

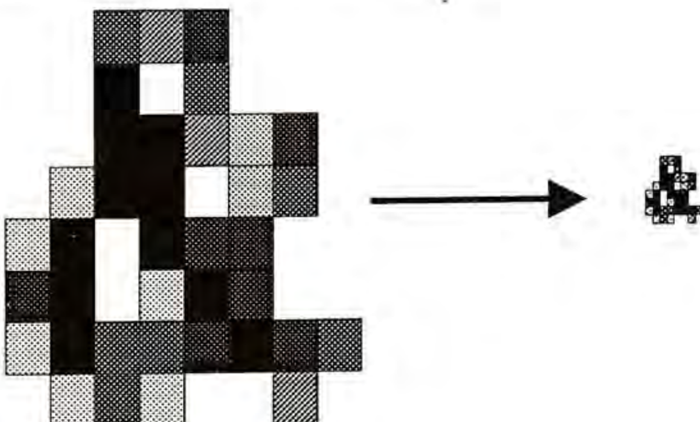


Figure 1.7: Gray scale character.

1.2.1 Objective

Our aims in this thesis are to build a fast Chinese rasterizer and to find ways of automatically adding hints to Chinese font data.

A Fast Rasterizer

We are interested in discovering how the rasterization speed would be related to the features, such as stroke count, of the Chinese character to be generated. Also, we try to make some improvement on the present rasterization method so as to give a more efficient rasterizer.

Automatic Hinting

The other objective is to automatically generate hints for Chinese font so that the resulted bitmap characters would retain most of their regularity and symmetry. Font in Ming style is just used for illustration and, in fact, the method can be applied for many other font styles, such as Gothic.

1.2.2 Organization

Chapter 2 describes the main features of Chinese characters and gives a brief discussion of the problems caused by these features.

In Chapter 3, we first discuss the details in basic rasterization and then we give a brief summary of the currently used hinting methods for European fonts.

In Chapter 4, we explain our techniques for building an improved Chinese font rasterizer. These methods include ways for fast rasterization and hintings of Chinese fonts.

Details of some performance tests on the methods described in Chapter 4 are presented in Chapter 5, and the results are described and analyzed in Chapter 6.

Chapter 2

Chinese Characters Fonts

It is an old legend that Chinese characters were created by Tsang-Jei (倉頡) about 5000 years ago. However, it is now generally believed that Tsang-Jei was not the only creator; just like people in other ancient countries, the ancient Chinese drew pictures to record what they saw and these pictures became the origin of Chinese characters. Collecting all of the characters used at his time, Tsang-Jei rearranged and reorganized them. He also enlarged the character set by making up some new characters.

Collectively, there are six traditional ways, called *Liu-Shu (六書: 象形、會意、形聲、指事、轉注、假借)* [New Image 92, ch. 1], which govern the construction of Chinese characters. We are going to discuss some main features of Chinese characters.

2.1 Large Character Set

There are about 50000 characters in the Chinese character set, of which 2000 to 4000 are currently used. The 1000 most frequently characters can just cover 91% of the characters used in a common Chinese text [Lua 90].

Since 16 bits can represent 65536 different patterns, the Chinese character set can be encoded with two bytes. Two currently used coding standards are GB and Big-5. GB, representing Guo-Bau, is the national standard of the mainland China. It covers about 7000 commonly used characters. Big-5 code is the standard of Taiwan, and it covers 5401 commonly used characters and 7652 less commonly used characters [Liu 87]. If these two standards are merged without duplication, about 18000 characters remain [Moon & Shin 90].

It is straightforward to input an article written in English to computer since each word is a linear combination of characters and there is a one-to-one correspondence between the 26 English characters and the keystrokes on keyboard. On the contrary, the case for Chinese characters is much complicated. Each Chinese word is 2-dimensionally composed of radicals (see section 2.4) and the presence of over 200 radicals poses problems for inputting [Cao & Suen 87].

2.2 Font Styles

A style or typeface is *"a set of visually related symbols"* [Robinstein 88]. The characters in distinct styles can differ in stroke width, stroke shape, serif shape, the ratio of horizontal stroke width to vertical stroke width, and even the structure of the whole character.

There are two main branches of styles: one for writing and the other for printing. About 5000 years ago, when the primitive Chinese characters were made, they were actually some

drawings of what people saw, so some characters were very complicated and it would take a quite long time to "draw" one character. For practical purpose, the characters went into several stages of simplification, regularization and rearrangement. Each version of the character set corresponding to a conversion stage becomes one style. The Dai Style, Kai Style and Hun Style came out in this way.

Printing is one of the "Four Great Inventions" of Chinese. Since a mould of document must be sculptured before a page can be printed, it would be easier and faster for the mould-maker to have characters in more regular shapes. Hence some new styles like Ming style and Gothic style were then made up for printing.

The main task for defining a Chinese style is to give the shapes of the basic strokes and the whole character set can then be derived from these strokes [NewImage 92, ch. 5-13]. Currently, there are over 20 styles in daily use and 5 common examples of them are shown in Figure 2.1. The characters in various styles usually look very different. For instance, the characters in Ming Style have many straight line segments whereas the characters in Hun Style have curve segments only.

明體	行書	楷書	隸書	黑體
Ming Style	Hun Style	Kai Style	Dai Style	Gothic Style

Figure 2.1: 5 common typefaces

2.3 Storage Problems

In the past, bitmap fonts were adopted by many major micro-computer Chinese systems, such as Eten Chinese System and Kuo-Kiu Chinese System, for font storage, display and printing. Keeping fonts in bitmap representation is very expensive in terms of storage.

The size of memory necessary for holding k bitmap characters of n by n pixels can be calculated by the formula:

$$memory_required = (INT(\frac{n+7}{8}) * n * k) bytes$$

For example, the storage required to hold 24 by 24 bitmap fonts for the 13053 characters in the Big-5 code set is INT((24+7)/8)*24*5401 bytes or 379.76 KB. Table 2.1 summarizes the storage requirements of some commonly needed bitmap font sizes for these 13053 characters.

size	16 x 16	24 x 24	32 x 32	48 x 48	64 x 64
storage	168.78 KB	379.76 KB	675.13 KB	1.48 MB	2.64 MB

Table 2.1: summary of storage for different font size

In a 150 dpi (dots per inch) printer, a 64 by 64 bitmap corresponds to a 64/150*72 or 31 point font and, in a 300 dpi, it corresponds to a 15 point font. These point sizes are commonly used.

Obviously, the price will be even higher if fonts of many different styles have to be available. The emergence of some scalable font formats like outline font resolves this problem to a certain extent.

2.4 Hierarchical Structure

Writing brush and ink were used together as the popular writing tools for the Chinese until the 20th Century. Just like painting, when a writing brush soaked with ink runs on paper, it leaves an ink trace on the paper. Each trace of the brush tip is called a *stroke*, which is the elementary component of a character. Some examples of strokes are shown in Figure 2.2.

Figure 2.2: Examples of basic strokes

The total count of basic strokes varies from literature to literature, depending on how the researchers interpret the similarities between the strokes. For instance, [Yang 86] claims that there are 7 basic strokes while [Cao & Suen 87] states that there are 11. Nonetheless, by modifying a basic stroke or composing several basic strokes together, the other kinds of strokes can be derived. A detailed classification of the stroke shapes is presented in [Yang 86]. Some systems for recognizing Chinese characters are also built by systematic and structural analysis of strokes [Hsu & Cheng 85], [Morishita, Ooura & Ishii 88], [Chen, Li & Chang 88].

Several strokes can be adequately combined to produce an intact component called *radical*. Rules have been developed to govern how the strokes can be joined, overlapped or put together to form regular radicals. 202 radicals are listed in [Cao & Suen 87], 350 radicals are claimed in [Chou & Tsai 91] and, in the Tsang-Jei Input Method, 24 major radicals and 75 minor radicals are used for building characters [Liu 87].

When one or more radicals are properly put together, a character is formed. Thousands of characters can be obtained in this way (Figure 2.3).

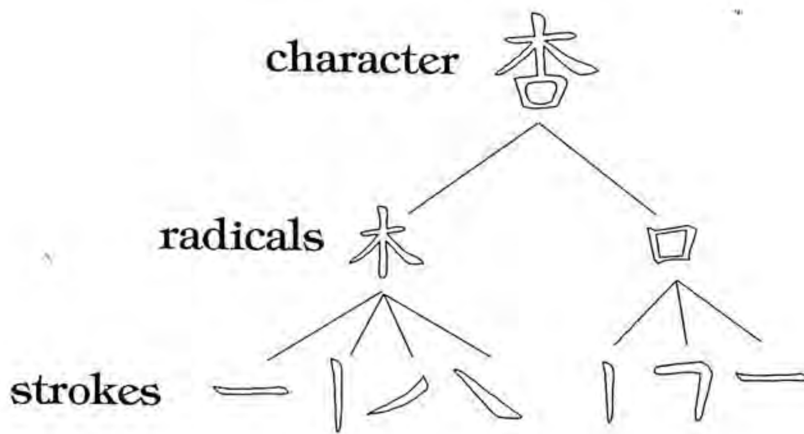
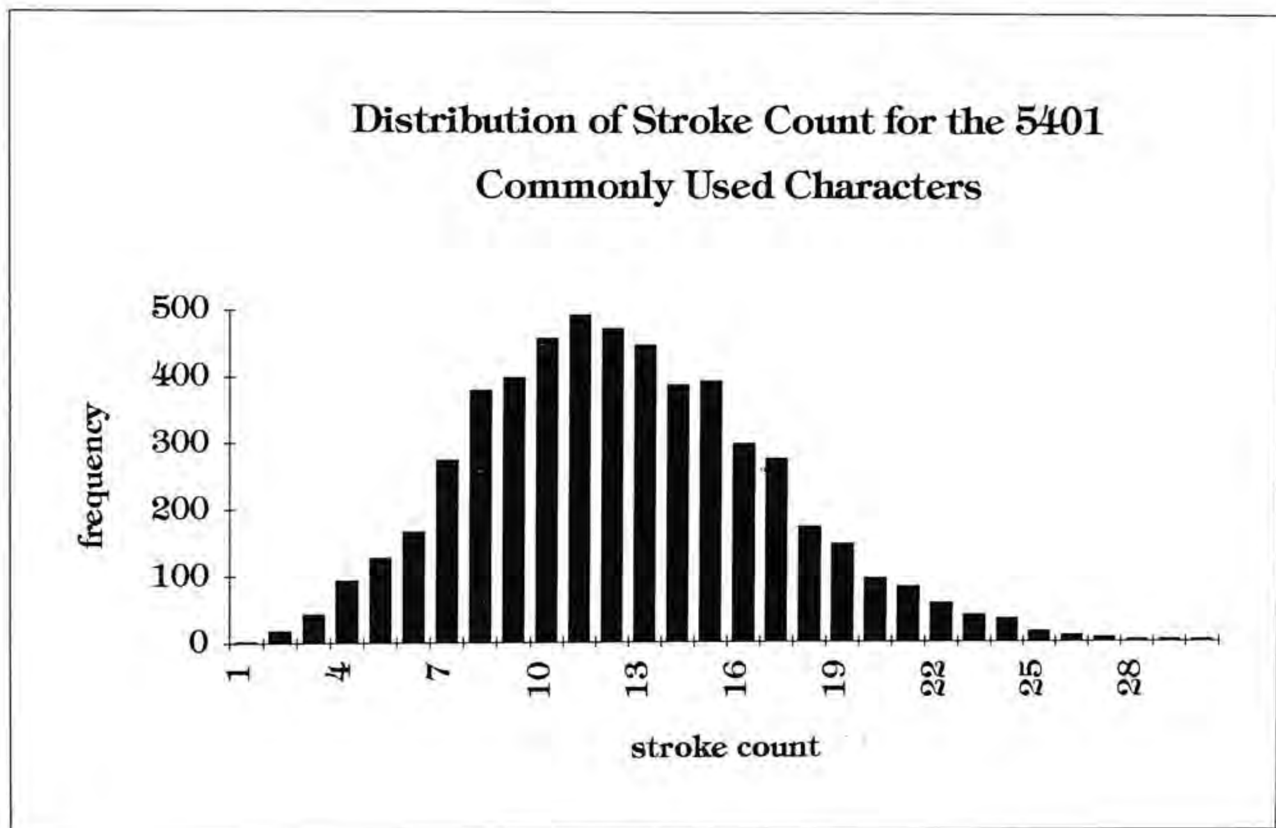


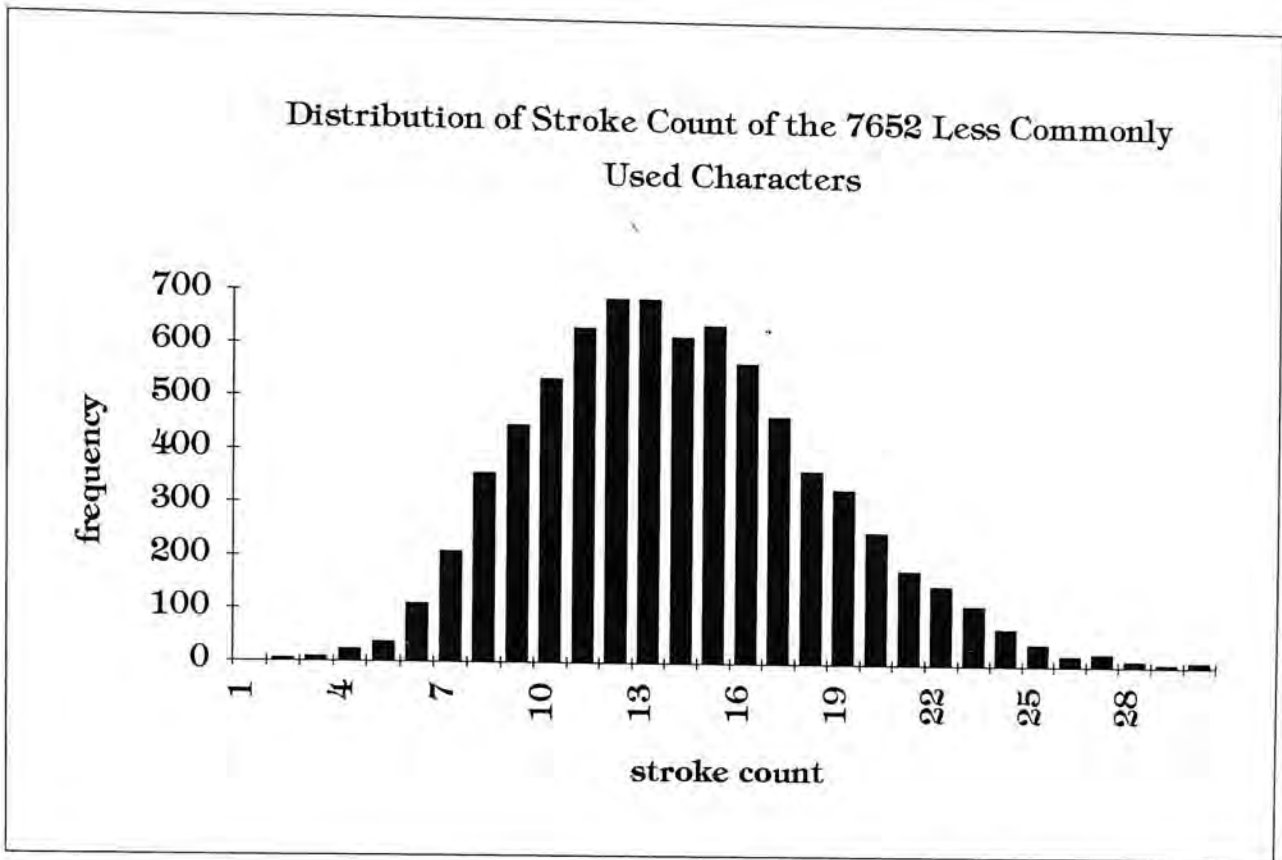
Figure 2.3: Composition of a character

2.5 High Stroke Count

It is common that a Chinese character can be composed of a fairly large number of strokes. Based on the 5401 commonly used characters in the Big-5 Chinese character set [Liu 87], we found that the average stroke count for a character is 12.358 and, based on the 7652 less commonly used characters, the average stroke count is 13.98 (Figure 2.4).



(a)



(b)

Figure 2.4: Distributions of stroke counts for (a) the commonly used characters and (b) the less commonly used characters in the Big-5 Chinese character set.

Chapter 3

Rasterization

As mentioned before, *rasterization* or *rendering* is the process which converts outline shapes into bitmap images. This conversion process is a general process implemented in many graphical application programs. However, in this thesis, we usually use the term *rasterization* to specify the font rendering process.

In this chapter, we first discuss the steps involved in general rasterization process and font rasterization process. Then we describe our approach for rendering Chinese outline fonts and the general hinting methods for European fonts.

3.1 The Basic Rasterization

Basically, shape rendering involves two steps: scan conversion and filling. These steps are originated from the flag fill algorithm [Ackland 81]. A pixel is considered as an *interior pixel* if over 50% of its surface is covered by the shape master. Since shape boundaries are relatively smooth, any pixel with center lying within the continuous border would probably be an interior pixel. Roughly, we can approximate the area of a pixel covered by the shape master by an area which is bounded by a straight line segment perpendicular to the scan line (Figure 3.1). However, it is true only if the pixels are fine relative to the size of the shape to be rendered. A counter-example is shown in Figure 3.2.

Now we are going into the details of the scan conversion and filling processes.

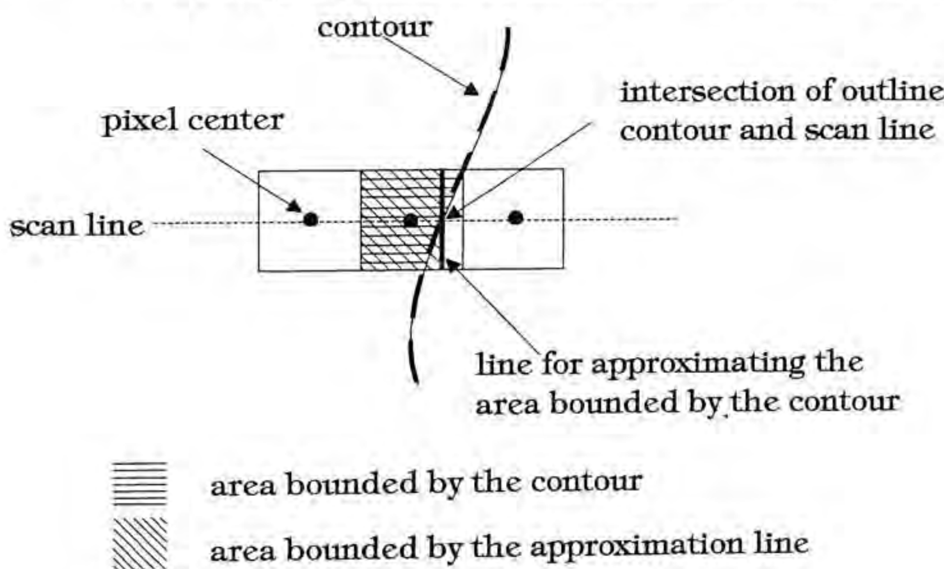


Figure 3.1: Area bounded by the contour can be approximated by that bounded by a line.

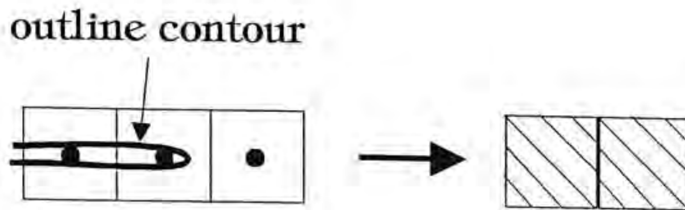


Figure 3.2: Counter-example of interior pixel

3.1.1 Scan Conversion

Scan conversion is the process which indicates the pixels at the boundaries of the shape outline. *Scan line* is the line which joins the centers of the pixels in the same row or column in a dot matrix. We now briefly discuss some common approaches of scan conversion.

Solving Equations

A simple method is to compute the intersections of the curves and the scan lines by explicitly solving the equations involved. For example, we can make use of the following set of equations to find the intersection points of a quadratic parametric curve and a horizontal scan line y=k.

$$y = a + b \cdot t + c \cdot t^2$$
$$x = p + q \cdot t + r \cdot t^2$$
$$y = k$$
$$t = \frac{-b \pm \sqrt{b^2 - 4c \cdot (a-k)}}{2 \cdot c}$$

where a, b, c, p, q, r and k are constants, and t is ranging between 0 and 1.

Although this method is straightforward, it takes time to do the calculations or special computation hardware is required to speed up the process [Fahlander 89].

Modified Polygon Scan Conversion

To render a polygon, the edges of the polygon are first sorted in ascending order in the y direction. Then each scan line is examined one by one from top to bottom or from bottom to top, and an active list is kept to indicate which edges are intersecting with the current scan line. The intersections of each scan line and the members in the active list are computed to determine which pixels are at the boundaries [Rogers 85, ch. 2].

This algorithm can be extended to render shape which is composed solely of straight lines and curves monotonic along the x or y direction [Pavlidis 85]. Obviously, when there are many curves segments in a contour (which is the case for Chinese outline fonts), the cost for breaking the curves into monotonic ones and sorting the segments would be very high.

Forward Differencing

Based on the position of a point on a curve, this approach can quite accurately estimate the position of an adjacent point. After locating one point on the curve, we can recurrently repeat this process to render the whole curve.

Suppose the coordinates of the points on a quadratic curve are described by the formulas:

$$x(t) = a + b \cdot t + c \cdot t^2$$
$$y(t) = p + q \cdot t + r \cdot t^2$$

where a, b, c, p, q, r are constants which control the shape of the curve, and t is between 0 and 1. The 2 recurrent formulas for computing the forward differences of x coordinates are:

$$\Delta x_1(t) = x(t+h) - x(t) = (bh + ch^2) + (2ch)t$$
$$\Delta x_2(t) = \Delta x_1(x+h) - \Delta x_1(t) = 2ch^2$$

which can be rearranged as:

$$x(t+h) = x(t) + \Delta x_1(t)$$
$$\Delta x_1(t+h) = \Delta x_1(t) + \Delta x_2(t)$$
$$\Delta x_2(t+h) = 2ch^2$$

Since the last term is independent of t, once the step value has been given and the values $x(0), \Delta x_1(0), \Delta x_2(0)$ have been calculated, x(0+h) can be estimated from x(0). The value of y(0+h) can be computed similarly. Repeatedly, the points on the curve (i.e. (x(0),y(0)), (x(h),y(h)), (x(2h), y(2h)), ..., (x(1), y(1))) can be found.

It sounds attractive that the position of a new point can be found with just 2 additions. But if the first derivative of x(t) (or y(t)) changes abruptly in the range [0,1], many points may coincide. Therefore, Jokob Gonczarowski [Gonczarowski 89] suggested a modified forward differencing algorithm which adaptively adjusts the step value.

Recursive Subdivision

Given the formula describing a curve, if we can find a point other than the two end points on the curve, we can divide the curve into two shorter curves. When this subdivision process is recursively repeated so that the resulted curve segments are flat enough to be approximated by line segments, the curve can approximated by a series of short lines which is easily to be rendered.

The exact way for subdividing a curve is dependent on the type of the curve. Figure 3.3 shows the bi-division of a quadratic Bezier curve (i.e. the curve is subdivided with t=0.5).

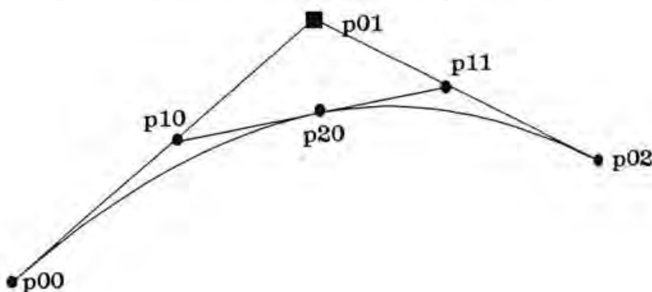


Figure 3.3: Bi-division of a quadratic Bezier curve.

where (p00, p01, p02) are the control points of the original curve while (p00, p10, p20) and (p20, p11, p02) are the two sets of control points of the subdivided curves. The coordinates of the control points can be computed by the simple formulas:

level 1: p10.x = (p00.x + p01.x)/2, p10.y = (p00.y + p01.y)/2,

p11.x = (p01.x + p02.x)/2, p11.y = (p01.y + p02.y)/2;

level 2: p20.x = (p10.x + p11.x)/2, p20.y = (p10.y + p11.y)/2;

The above subdivision is quite efficient as it takes only 1 addition and 1 shift to give a new control point. The disadvantage here is that it would generate a number of straight lines, which complicates the problem.

In conclusion, all these methods are able to scan convert an outline shapes into unfilled bitmap images.

3.1.2 Filling Outline

Among the filling algorithms are edge fill, flood fill, seed fill and parity fill [Rogers 85, ch. 2] [Foley & van Dam 90, ch. 19]. Edge fill is suitable for filling polygons and, provided that at least one interior pixel is found in each individual boundary, both flood fill and seed fill can be used for filling close pixel boundaries. We are now going into the details of the parity fill, the most straightforward and usually the most efficient way of filling.

Parity Fill

By parity fill, a pixel is turned on if a horizontal line originating form it intersects the outline contours an odd number of times, i.e. an odd parity [Pavlidis 79] [Pavlidis 81]. The algorithm can be summarized as:

```
{
input: bitmap with boundary pixels turned on
output: filled bitmap
}
for each scan line do
        set count=0;
        for each pixel in a scan line do
                if the pixel is on then
                        increment count by 1;
                else
                        if count is indivisible by 2 then
                                turn the pixel on;
                        end {if}
                end {if}
        end {for}
end {for}
```

Algorithm 3.1

It does the filling correctly only if all the interior pixels have odd parity. Unfortunately, this may not be the case if some boundary pixels coincide. Therefore, instead of representing the scan

converted points as pixels, we keep a table of span extrema for each scan line (Figure 3.4). This table gives the exact positions of the intersection points and hence guarantees the parity filling correct. We turn on the pixels with centers falling in the odd parity span. The remaining problem is that local maxima, local minima and any horizontal line may meet a scan line (Figure 3.5).

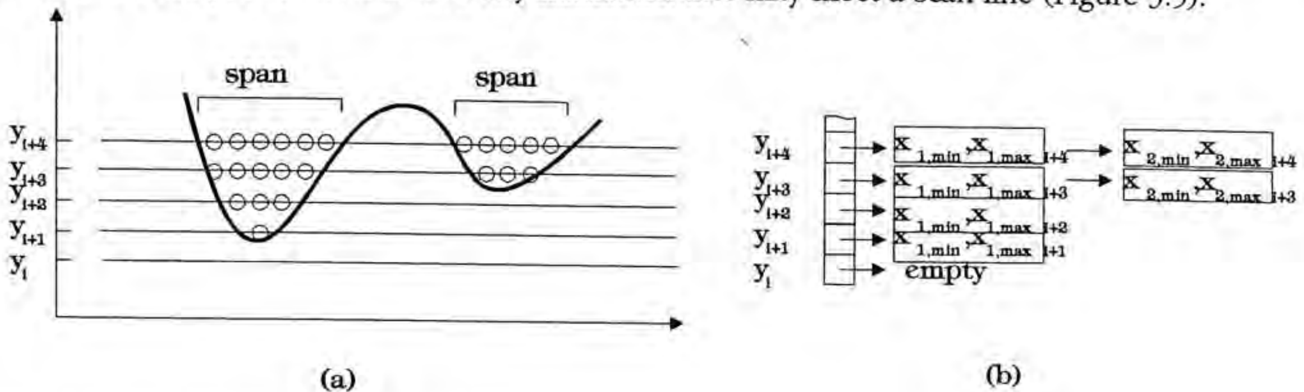


Figure 3.4: (a) Each horizontal span is stored with its extrema. (b) Span table.

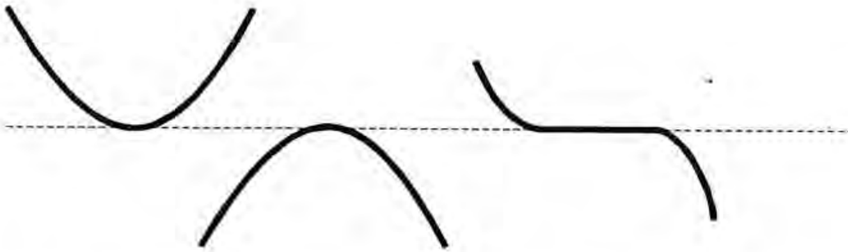


Figure 3.5: local maxima, local minima and horizontal coincide with a horizontal scan line.

For filling polygons, this problem can be resolved by performing sign tests for the two end points of each line segment [Pavlidis 85] [Andler 90].

3.2 Font Rasterization

Besides the scan conversion and the filling processes mentioned, font rasterization in general includes two more steps, outline scaling and grid-fitting.

3.2.1 Outline Scaling

In general, the outline font data are measured in an arbitrary scale to describe the shape of the character in great detail and then even the subtle curve parts can be exactly represented. For instance, 1000 and 1024 are two common choices of size of the bounding box for font description. Therefore, before turning into bitmap, the font data must be scaled to the desired size.

3.2.2 Hintings

When outline fonts are rendered in low resolution devices, some characteristics like regularity, thickness and uniform appearance of the characters may lose and hintings are the rules used to adjust the outlines to keep as much features of the characters as possible (see section 3.3).

3.2.3 Basic Rasterization Approach for Chinese Fonts

Since the above rasterization approaches are either for graphical shapes or for European characters, they are not very adequate for rendering Chinese outline fonts. Our elementary approach is based on [Cheang 90, appendix] and is briefly described below (Figure 3.6).

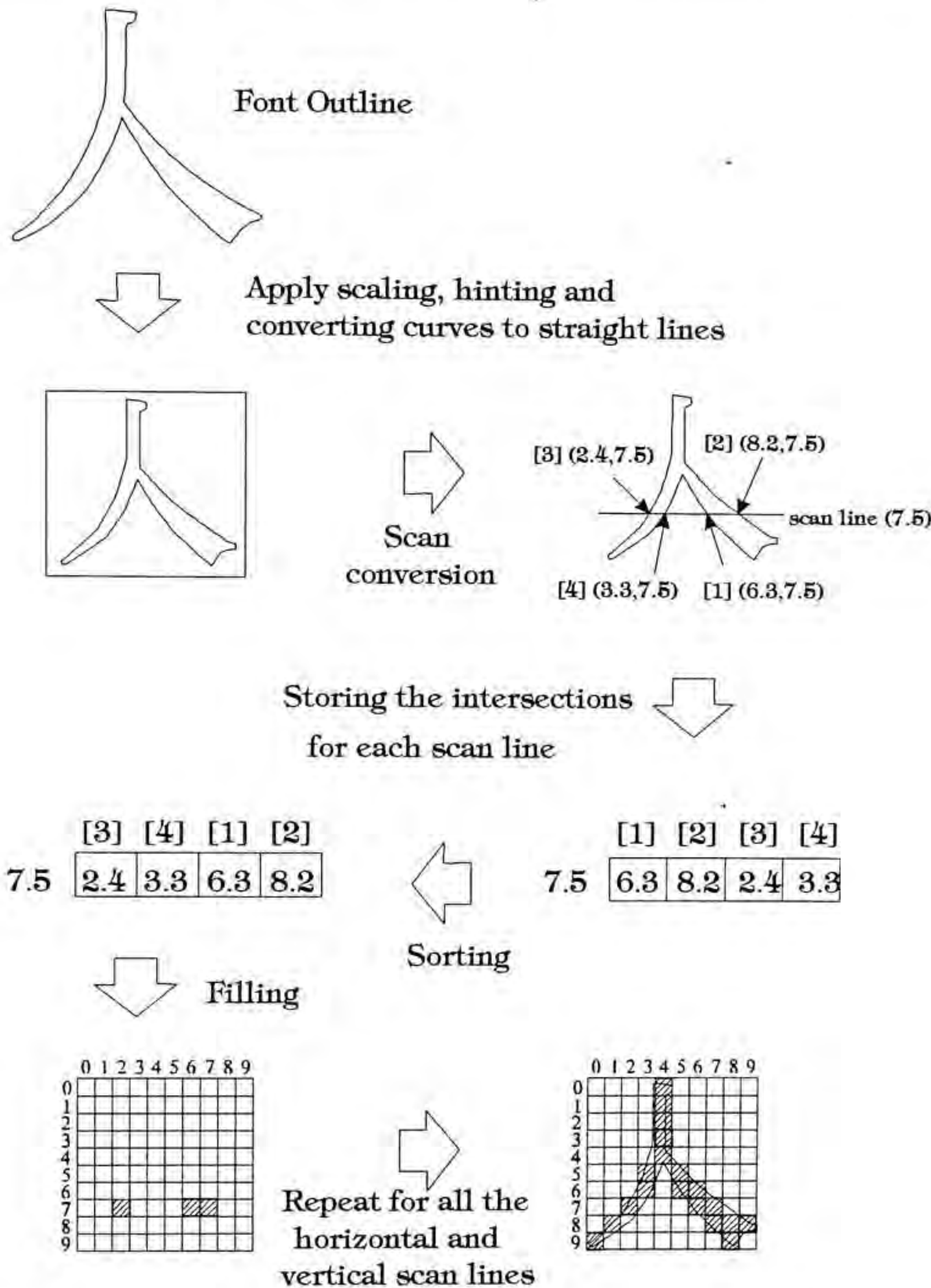


Figure 3.6: Our rasterization approach.

Outline Scaling and Hinting

Suppose the outline of a character is composed only of straight lines and Bezier curves. These lines and curves are first scaled up or scaled down to the size required. Subsequently, if necessary, grid fitting or hintings are applied to the outline, making sure that the main features (e.g. equally wide strokes) of the character are maintained.

Scan Conversion

By recursive division, the Bezier curves of the outline are converted into sets of straight lines. Then the outline is no more than a set of straight lines, just like a vector font. Except the horizontal lines, all these lines go through the scan conversion process, in which intersection points with horizontal scan lines are found and stored. When the end point of a line meets any scan line, the intersection for the upper end point is considered but that for the lower end point is ignored (Figure 3.7). This ensures the parity fill can be done correctly.

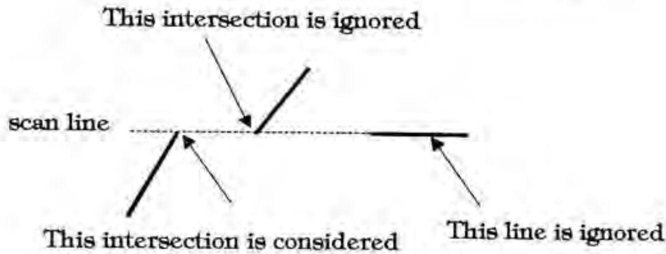


Figure 3.7: End points of line segments coincide with scan line.

Parity Filling

The font is then filled by parity filling. For each scan line, we turn on all the pixels with centers in between alternative pair of consecutive intersection points for each scan line. Even if a span does not include any pixel center, the nearest pixel will be activated to avoid dropouts (Figure 3.8). Finally, the scan conversion and this filling process are repeated for the vertical scan lines, further preventing dropout pixels and ensuring better looking bitmap characters can be generated.

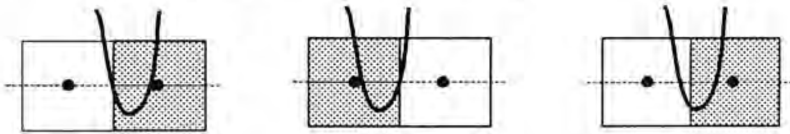


Figure 3.8: If a scan line cuts the outline, at least one pixel will be turned on to avoid drop-out.

Algorithm

The algorithm of the rasterizer can be summarized as:

```
{
input: control points of quadratic Bezier curves of the character outline
output: bitmap image of the character
}
Scale all the control points to the size of the output bitmap;
Apply hintings to the control points;
Convert the quadratic Bezier curves into straight lines by recursive sub-division;
for each straight line segment of the outline do
        for each horizontal scan line intersecting the straight line segment but not touching the lower end point of the
        segment do
                Compute and store the intersection point of the straight line segment and the scan line;
        end {for}
end {for}
for each horizontal scan line do
        Sort the intersection points in ascending order;
        for each alternative pair of consecutive intersection points do
                for each pixel with center falling between the two intersection points do
                        turn the pixel on;
                end {for}
                if no pixel center falls between the intersection points then
                        Turn on the pixel with center closer to the two intersection points;
                end {if}
        end {for}
end {for}
Repeat the process for vertical scan lines.
```

Algorithm 3.2

In this algorithm, only simple computations are involved and, as the scan conversion is done both horizontally and vertically, drop-out can be avoided and the resultant characters would look better when compared to that generated with unidirectional scan lines. On the contrary, considerable memories are required for holding the line segments and intersection points during rasterization.

3.3 Hintings

Even if high-quality outlines are available, the character shapes should be adjusted so as to work well on a grid of uniformly placed pixels, especially when the type size gets smaller. At lower resolution, the goals of hintings include getting all near-similar weight strokes in a character to appear as equal weights and making sure that necessary curves are visible, smooth and symmetric [Seybold 92]. The rules which govern these shape adjustments are called *hints*. We now briefly go over some common hinting methods for European fonts.

3.3.1 Phase Control

Phase control or *grid fitting* is a way to adjust the character outline to a particular resolution by displacing control points in the character outline to the nearest mid-point between pixels or pixel center [Hersch 87] [Hersch 88] [Hersch 89] [Betrisey & Hersch 89] [Betrisey & Hersch 91a]. Figure 3.9 shows how a bitmap character can have a better look after grid fitting.

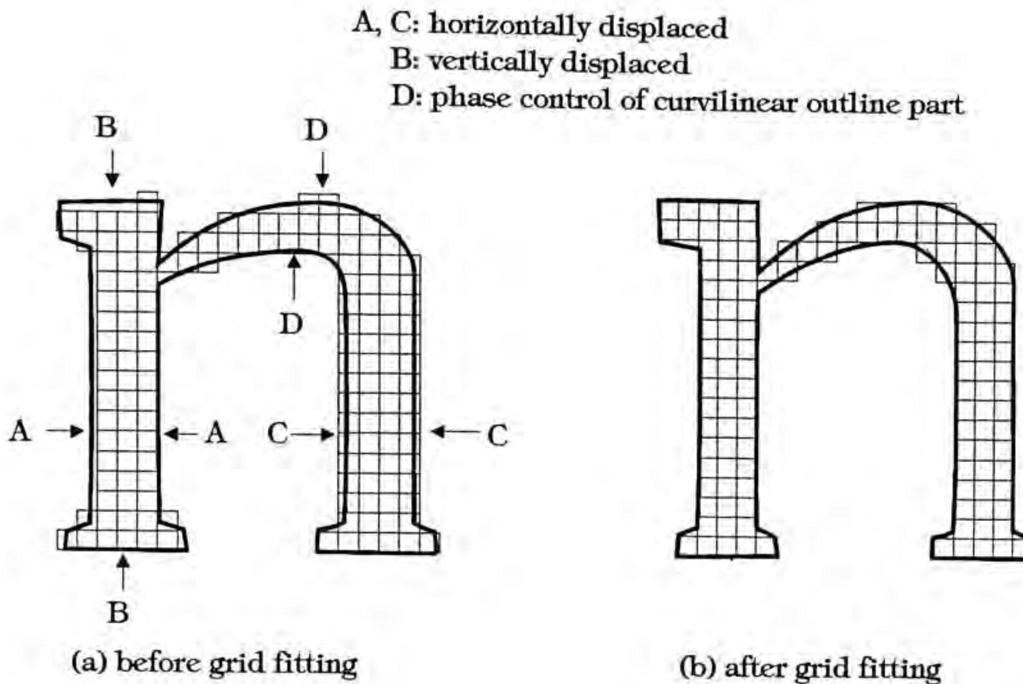


Figure 3.9: Grid fitting [Hersch 89].

Commonly, a *constraint description* is used to specify the actions that should be taken to modify the coordinates of the character outline. Such a description should provide two kinds of information: a constraint qualifier and a constraint application part.

A *constraint qualifier* indicates how to compute the horizontal or vertical displacement parameters by which certain characteristic points within the shape outlines are migrated. The constraint qualifier includes some basic constraints which relate to the references lines such as base line, x-height line and caps line. These reference lines should be moved to the nearest grid lines and, accordingly, horizontal or vertical stems are controlled individually. On the other hand, bowls and curved character parts are adjusted by appropriately keeping the phase of the vertical or horizontal extremity of arcs within a given phase range, controlling the flatness of the produced discrete arc.

A *constraint application part* specifies on which parts of a character the current displacement must be applied. These parts are specified by pairs of starting and ending control points. The computed displacements are not equally applied to the whole outline: some parts are moved according to the displacement parameters; some parts are just proportionally deformed; and some parts may even remain fixed at their original locations.

3.3.2 Auto-Hints

In the above method, manual insertion of the constraint descriptions (hinting information) to the font data is necessary and it would be very time consuming. Therefore, some researchers make their effort to develop algorithms for automatically addition of hinting information without manual intervention [Karow 89] [Andler 90]. The main idea of these methods is to detect the characteristic points which control the shapes of the outline character and the associated reference lines for

determining the adequate displacements of the characteristic points. They are then converted into parameters adding to the font data.

In [Betrisey & Hersch 91b], a different approach for auto-hints called model-based matching is suggested. In this approach, a topological model characterizing the main features of the shape found in European character is first established. This model provides sufficient information for matching its characteristic points to the corresponding points in an input font (Figure 3.10). The model has a table of hinting information for automatic hints generation. After matching the input shape to the model, hints which can be applied to that font are taken from the table and inserted to the outline description. In addition, the model can include an extra structural description about typographic structural parts, such as stems, serifs and bowls, using characteristic model points (Figure 3.11).

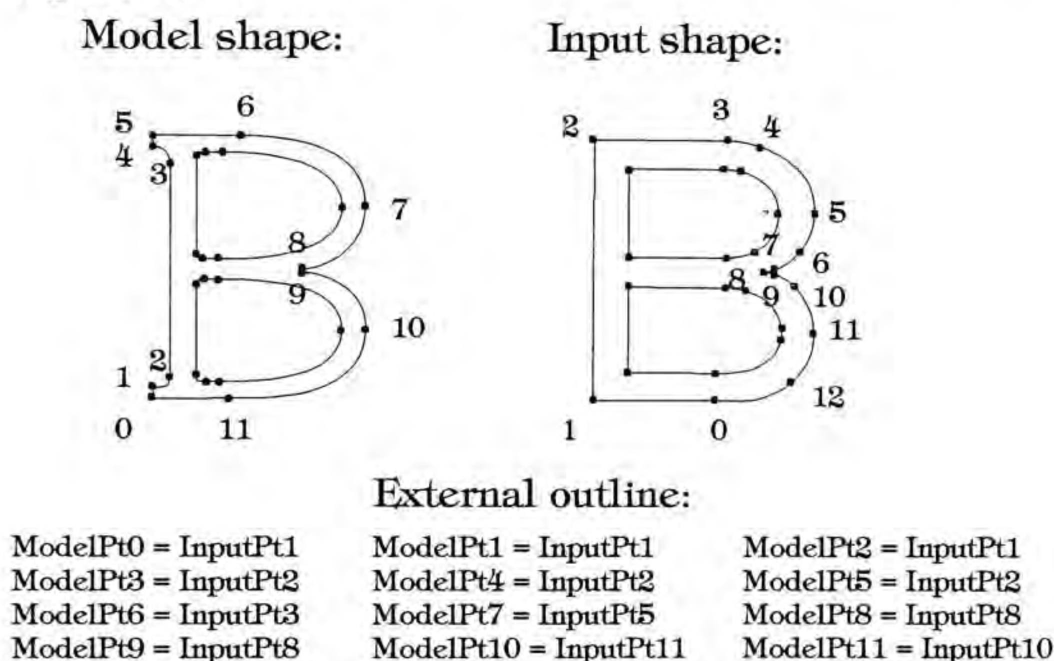


Figure 3.10: Matching of characteristic points of an input font to the model [Betrisey & Hersch 91].

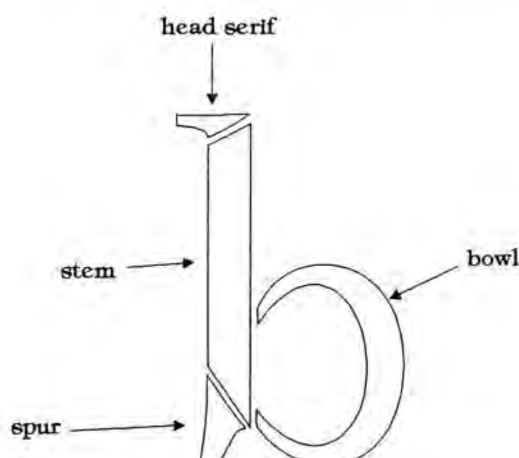


Figure 3.11: Structural letter parts [Betrisey & Hersch 91]

3.3.3 Storage of Hintings information in TrueType Font and PostScript Font

TrueType font stores the hintings information in the form of instruction statements as part of font data for each character. These instruction statements specify a sequence of actions and contain the

related parameters for hintings. If the instruction statements are absent, hintings will be performed solely by the rasterizer or no hintings will be done at all.

In a similar way, PostScript keeps the hintings information with specific operations. For instance, the operator *setstrokeadjust* changes a boolean value in the graphics state which indicates whether stroke adjustment will be done during subsequent stroke and related operators.

Chapter 4

An Improved Chinese Font Rasterizer

In this chapter, we first describe the technique of avoiding floating point arithmetic in section 4.1. Then, in section 4.2, we present a fast filling algorithm originated from parity fill. In section 4.3, some hintings techniques for Chinese outline fonts are described and, in section 4.4, an improved rasterization algorithm for printing is presented.

4.1 Floating Point Avoidance

In general, we do the computations in the rasterization process with floating point arithmetic because:

Accuracy: Floating point numbers can accurately represent the data values in the computation steps. Otherwise, rounding floating point data to integers may introduce round off errors, which may seriously distorts the shapes of characters generated.

Simplicity: It is straightforward to do the calculations with floating point numbers.

Nonetheless, computation involving floating point arithmetic is relatively complicated and hence slower than integer arithmetic. Moreover, the input data, control points of the outline, are usually represented by integers, so it would be faster and more convenient to do the computations with integer arithmetic.

Before integer arithmetic can be used, we must handle the round-off errors caused by rounding real numbers to integers. This can be done by scaling up all data before rounding them to integers.

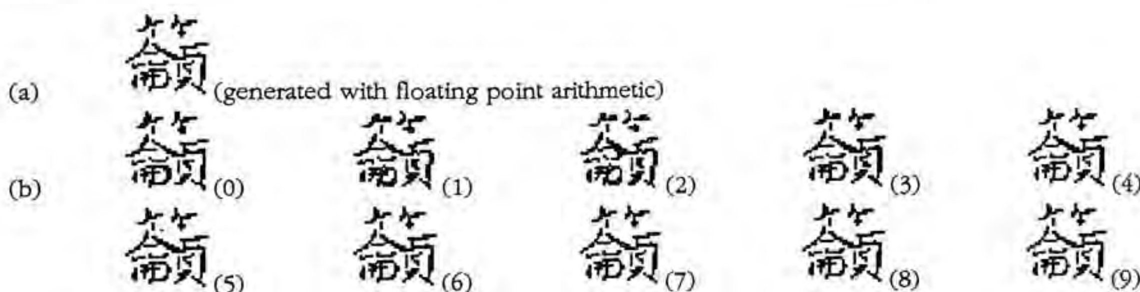


Figure 4.1: (a) Bitmap characters generated with floating point arithmetic; (b) Bitmap characters generated with integer arithmetic and the number in brackets is related to the scaling factor, e.g. 3 means shifting the data to the left by 3 bits before rounding.

In order to determine how large the scaling factor should be, we generate some bitmap characters making use of integer arithmetic with different scaling factors, and we carefully compare them with the corresponding character generated with floating point arithmetic to find the least shifting parameter which would give the best results. This comparison step is repeated for many characters in various styles, one example of which is shown in Figure 4.1. We found that if the scaling factor is set to 256 or 2^8, there is no difference between the bitmap characters generated with floating point and integer arithmetic.

With this scaling factor, we can shift a number to the left by 8 bits for scaling up and shift it to the right by 8 bits for scaling down. If integers are represented by 4 bytes or 32 bits, there are 24 bits left for the mantissa, which is large enough for handling many practical cases. If only 16 bits are available for representing an integer, a scaling factor of 16 or 2^4 can be used to give rather satisfactory results.

Some adjustments to the parameters must be done:

Threshold for sub-division: This is the value that determines whether a curve is flat enough to be approximated by a straight line. This value must be multiplied by the scaling factor, otherwise extra sub-divisions of the curves would reduce the efficiency of the rasterizer.

Width between adjacent scan lines: scan lines should appear at every 256-th line rather than every line.

4.2 Filling

The simplest way to store a bitmap character is to store it as an array of bytes. Each bit in a byte represents one pixel (Figure 4.2). If a bit is set to 1, the corresponding pixel is on, otherwise, the pixel is off [Lee 92] [Rosenberg 91]. Given the bitmap character is represented in this way, we try to speed up the horizontal and vertical filling processes respectively.

```
00000000 0000000 00000000
00000000 0000000 00000000
00000001 1111111 00000000
00000011 1100011 11000000
00000111 1000001 11100000
00001111 0000000 11110000
00011110 0000000 01111000
00111100 0000000 00111100
00111111 1111111 11111100
00111100 0000000 00111100
00111100 0000000 00111100
00111100 0000000 00111100
00111100 0000000 00111100
00111100 0000000 00111100
00000000 0000000 00000000
00000000 0000000 00000000
```

Figure 4.2: Font data of character 'A' in bitmap form

4.2.1 Filling with Horizontal Scan Line

In the horizontal filling process of Algorithm 3.2 in Chapter 3, the pixels or bits are turned on one after another. This filling process can be accelerated by filling one byte rather than one bit each time. Supposing we want to fill all the pixels between pixel 10 to pixel 30 of any horizontal scan line, referring to Figure 4.3, we are to fill the first byte with 8 zeroes, the second byte with 2 zeroes followed by 6 ones, the third byte with 8 ones and the fourth byte with 7 ones followed by 1 zero.

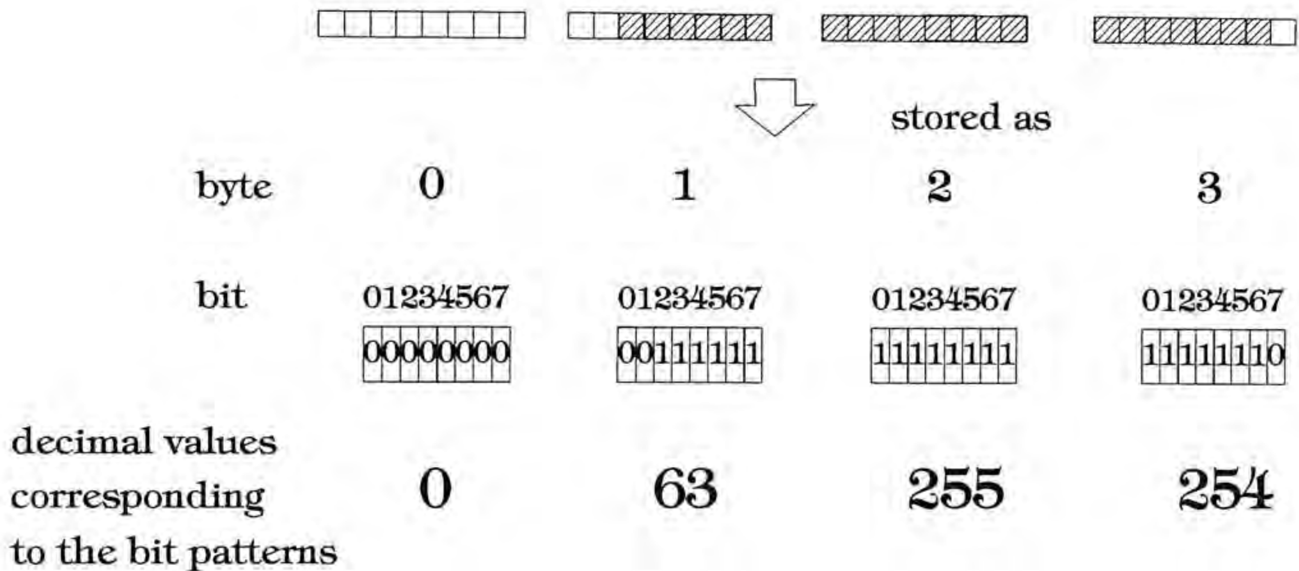


Figure 4.3: Decimal values correspond to pixel patterns.

Once we know the range of contiguous bits to be turned on for a byte, we can turn them all on immediately. For instance, if we want to turn on bit 2 to bit 7 of a byte, we are setting a bit pattern of 00111111 to that byte, which corresponds to the decimal value 63. Therefore we can fill this byte by storing this decimal value to the byte. Some more examples are shown in Table 4.1.

byte pattern	bits (pixels) to be filled	decimal value
00111111	bit 2 to bit 7	63
01111110	bit 1 to bit 6	126
11111100	bit 0 to bit 5	252

Table 4.1

To save time, we can store the decimal values representing the patterns in a two dimensional array to form a look-up table. The formula for initializing the content of the look-up table is:

```
for i=0 to 7 do
        for j=0 to 7 do
                if i<j then
                        pattern [i][j] = 0;
                else
                        pattern [i][j] = 2^(8-i) - 2^(7-j);
                end {if}
        end {for}
end {for}
```

Algorithm 4.1

Then, pattern [2][7] = 63, pattern [1][6] = 126 and pattern [0][5] = 252. Filling the bit p to bit q of a byte, we can set the value of that byte to pattern[p][q]. Algorithm 4.2 helps us to do horizontal filling with the look-up table, filling up to 8 pixels in a horizontal scan line each time rather than 1 pixel each time.

```
{
input: coordinates of start and end pixels in scan line k for filling
output: filled bitmap
}
Determine the first byte (first_byte) and the first bit (first_bit) in that byte to be filled;
Determine the last byte (last_byte) and the last bit (last_bit) in that byte to be filled;
if first_byte = last_byte then
        first_byte of scan line k = pattern [first_bit][last_bit];
else
        first_byte of scan line k = pattern [first_bit][7];
        for all the internal bytes of scan line k do
                byte = pattern [0][7];
        end {for}
        last_byte of scan line k = pattern [0][last_bit];
end {if}
```

Algorithm 4.2

4.2.2 Filling with Vertical Scan Line

The purpose of doing vertical scan conversion is to complement the horizontal scan conversion. Horizontal scan lines may not hit the outline of a thin square (Figure 4.4), as a result, such a square may totally vanish in the final bitmap image. Unfortunately, a horizontal stroke in a Chinese character is conceptually similar to such a thin square. The vertical scan conversion not only prevents the horizontal strokes from disappearing, but also makes the resultant bitmap characters look better.

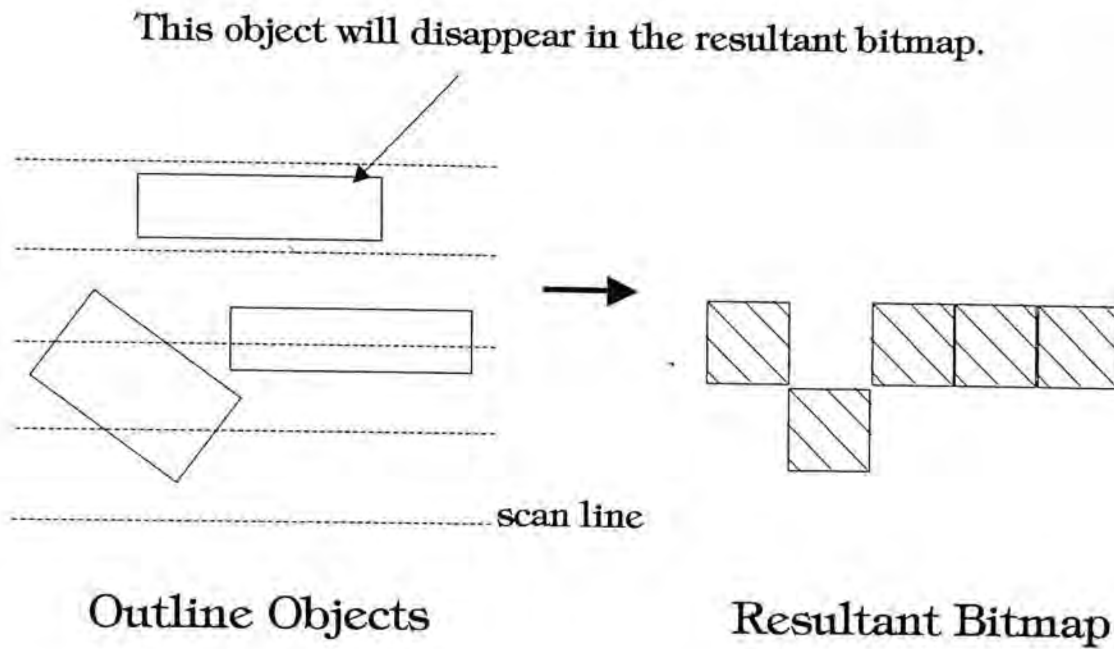


Figure 4.4: Thin object may disappear if only horizontal scan lines are involved in scan conversion.

Since the horizontal scan conversion has accounted for most area of the bitmap character, we need not completely do the steps involved in vertical filling but partly perform it to generate the same output bitmap. We only have to turn on the unfilled pixels instead of all pixels in each vertical span.

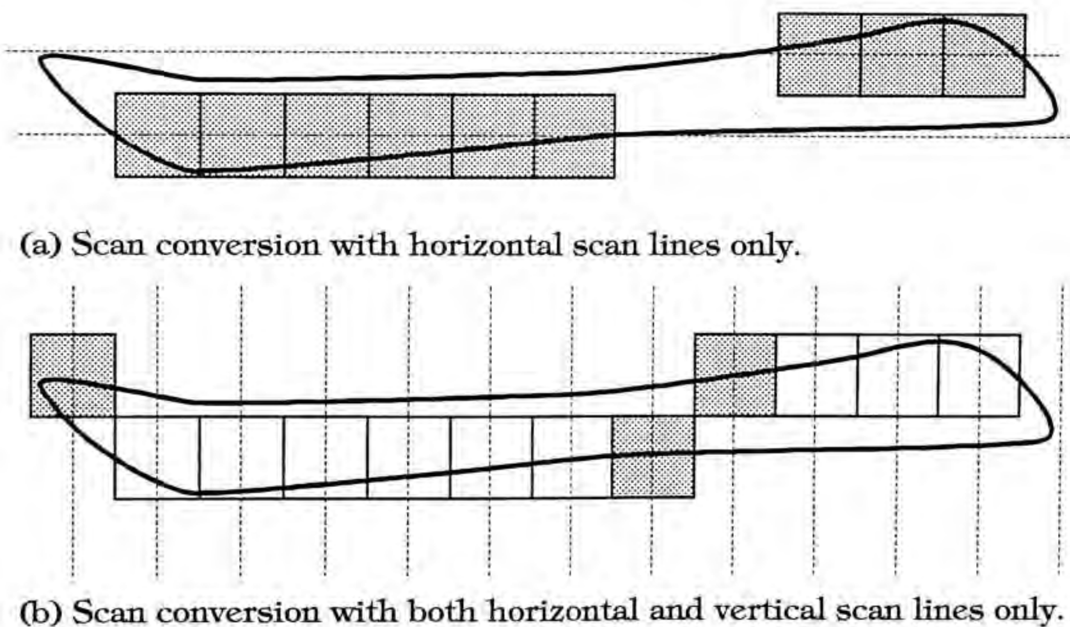


Figure 4.5: Comparison of bitmap images generated with and without vertical scan lines.

Suppose we want to turn on the pixels vertically between pixel (3,2) and pixel (3,9), and the pixels between (3,4) and (3,7) have been blackened in the horizontal filling process (Figure

4.6). First we start from the end pixel (3,2) and turn on the pixels below it until a filled pixel or the other end pixel (3,9) is met. Then, we start from the end pixel (3,9) and turn on the pixels above it until a filled pixel or the other end pixel (3,2) is encountered. The filling algorithm is summarized in algorithm 4.3.

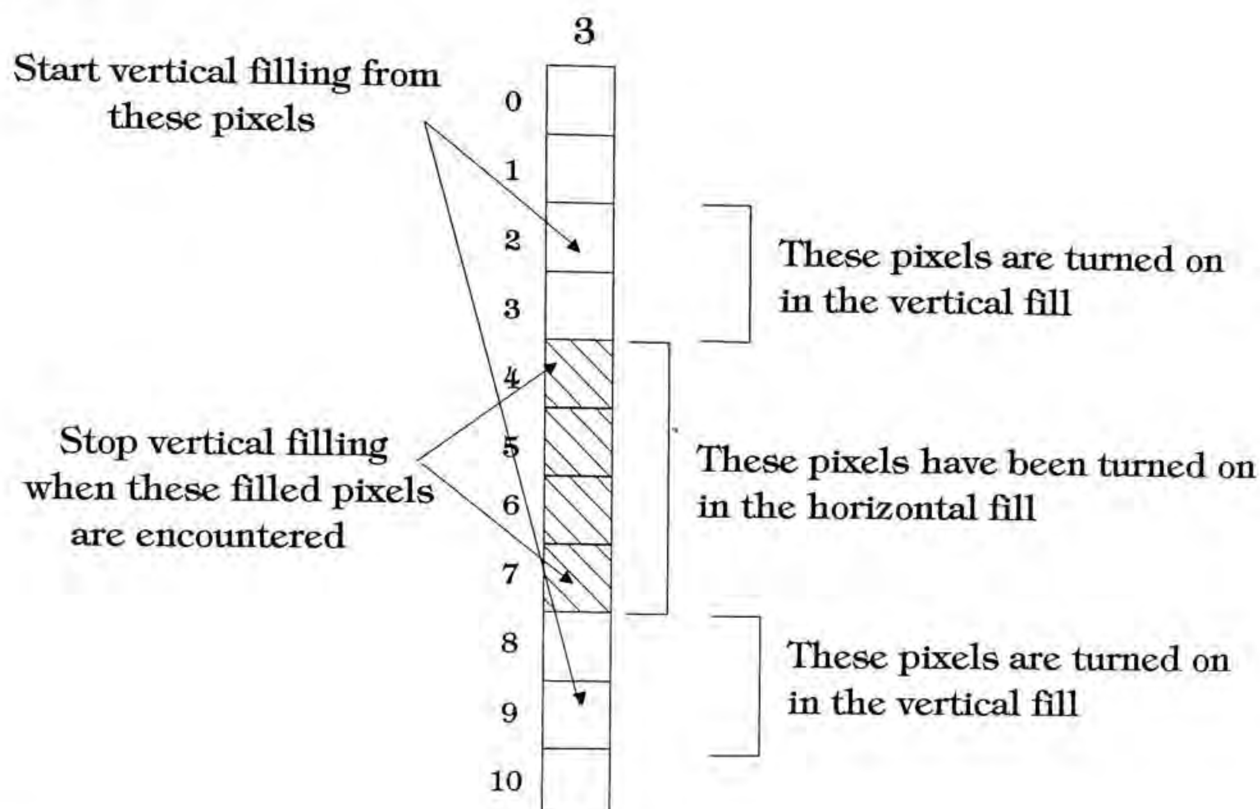


Figure 4.6: Illustration of vertical filling after horizontal filling.

```
{
purpose: fill cells vertically between cell[x][y_start] and cell[x][y_end], where k>=j
}
set y=y_start;
while y<=y_end and cell[x][y] is unfilled do
        turn on cell[x][y];
end {while}
set y=y_end;
while y>=y_start and cell[x][y] is unfilled do
        turn on cell[x][y];
end {while}
```

Algorithm 4.3

4.3 Hintings

Usually, the appearances of bitmap characters directly generated from outlines by rasterization are not satisfactory. The regularity and harmony of the bitmap characters may be partly destroyed. The problem includes stroke width order not preserved and space width order between adjacent strokes not conserved. These topics will be discussed in detail in section 4.3.2 and 4.3.3. In section 4.3.3, the case of single curve stroke is considered.

4.3.1 Assumptions

We make some assumptions here:

1. Each contour is a non-intersecting curve and is composed of straight lines and quadratic Bezier curves only.
2. A segment can consist of the control points of a Bezier curve or the two end points of a straight line.
3. The sequence of the control points in a contour is specified in *clockwise order*; holes are specified in *counterclockwise order*.
4. There exist threshold values to decide whether a stroke is horizontal, vertical or other.

4.3.2 Maintaining Regular Stroke Width

The order of thickness of the strokes in a character is not necessarily preserved in the generated bitmap character. For instance, as shown in Figure 4.7, two equal wide strokes of an outline character may be converted to bitmap strokes with unequal widths.

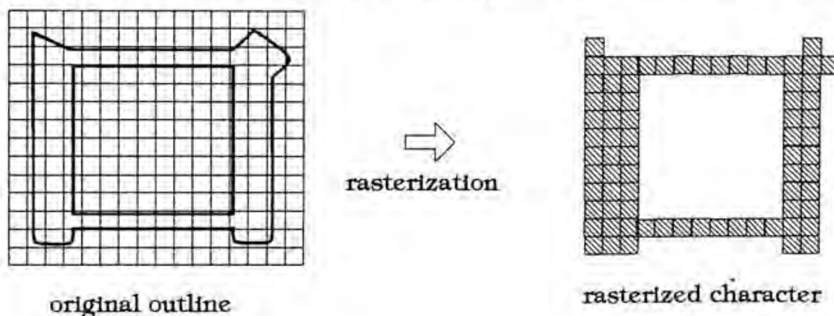


Figure 4.7: The generated character has unequal wide strokes.

The above problem can be resolved by displacing the vertical strokes of the master character as shown in Figure 4.8. The resulted bitmap character has equal wide vertical strokes.

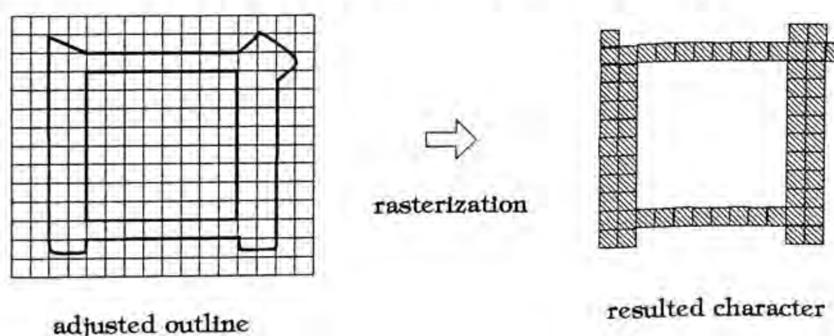


Figure 4.8: The character generated from adjusted outline has equal wide strokes.

Preserving Stroke Width Order

There are 3 steps for keeping the stroke width order of a character:

1. Identify the horizontal and vertical strokes in a character.
2. Determine the width and center line of each stroke.
3. Based on the width and position of center line of each stroke, apply centering or anti-centering process.

For the sake of simplicity, we limit our discussion to horizontal strokes only. The case for vertical strokes can be handled by similar technique.

Extracting Horizontal Strokes

We extract horizontal strokes by the y-coordinates and directions of the horizontal line segments (Figure 4.9). If two horizontal line segments with the same y-coordinate and direction, and the shortest interval between them fall inside the black area, these lines will be combined to form one single line. Two adjacent lines with opposite directions will compose a stroke if the interval between them falls inside the black area and the width of the interval is close to the width of a stroke. The algorithm is:

```
{
purpose: extract horizontal strokes from contours
input: contours in a character
output: set of individual horizontal strokes
}
identify horizontal lines in the outline character and their directions, right or left;
sort the lines according to descending order of their y-coordinates;
group the lines with nearly the same y-coordinates together;
for each group of line do
        when the interval between two adjacent lines is entirely falls inside the black area, merge the two lines;
end {for}
for each line (l0) do
        for each line below the current line (l1) do
                if l0 and l1 constitute a stroke then
                        group these lines as a stroke;
                end {if}
        end {for}
end {for}
for each identified stroke do
        include the segments which constitute serifs of that stroke;
end {for}
```

Algorithm 4.4

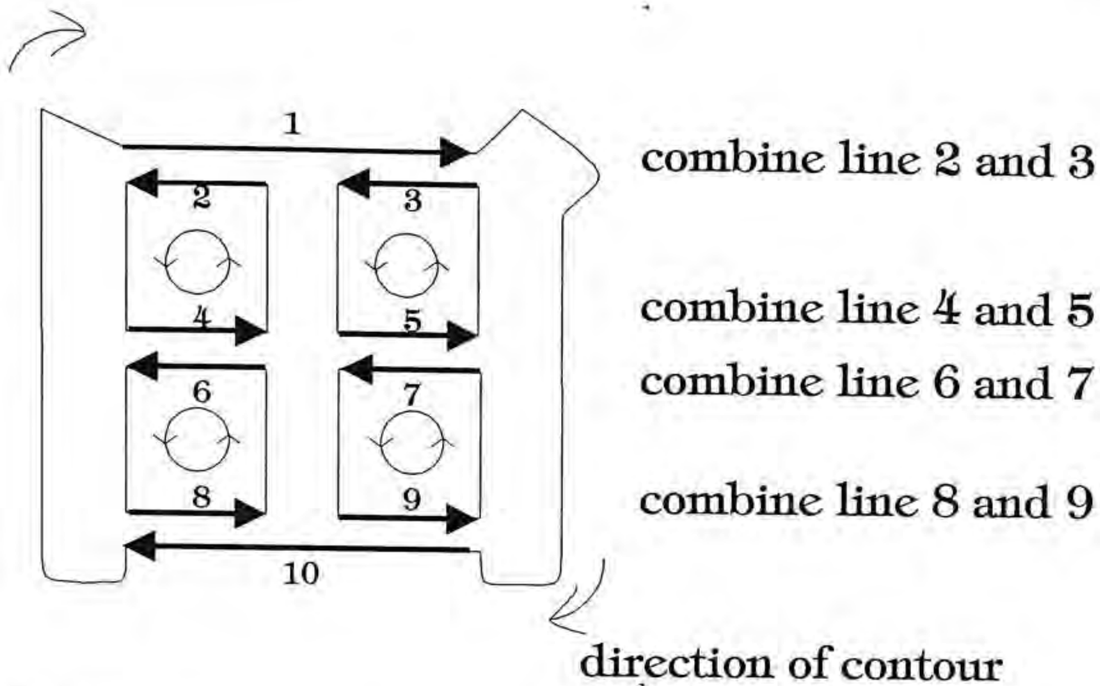


Figure 4.9: Combination of line segments

As illustrated in Figure 4.8, the way to determine whether a line is wholly falling inside the black area is based on parity check:

```
{
purpose: determine whether a line is wholly inside the filled area of the character
input: the line to be tested and contours of the character
output: indicator which shows that the line is inside or outside the filled area of the character
}
select the left end point of the line and store it as p;
draw a horizontal line, l, from p to the left;
set count=0;
for each of the line and curve segment of the outline character do
        if the segment intersect l do
                if the segment is not a horizontal line do
                        if the segment is going upwards do
                                increase count by 1;
                        else
                                decrease count by 1;
                        end {if}
                end {if}
        end {if}
end {for}
if count==0 then
        l is outside black area;
else
        l is inside black area;
end {if}
```

Algorithm 4.5

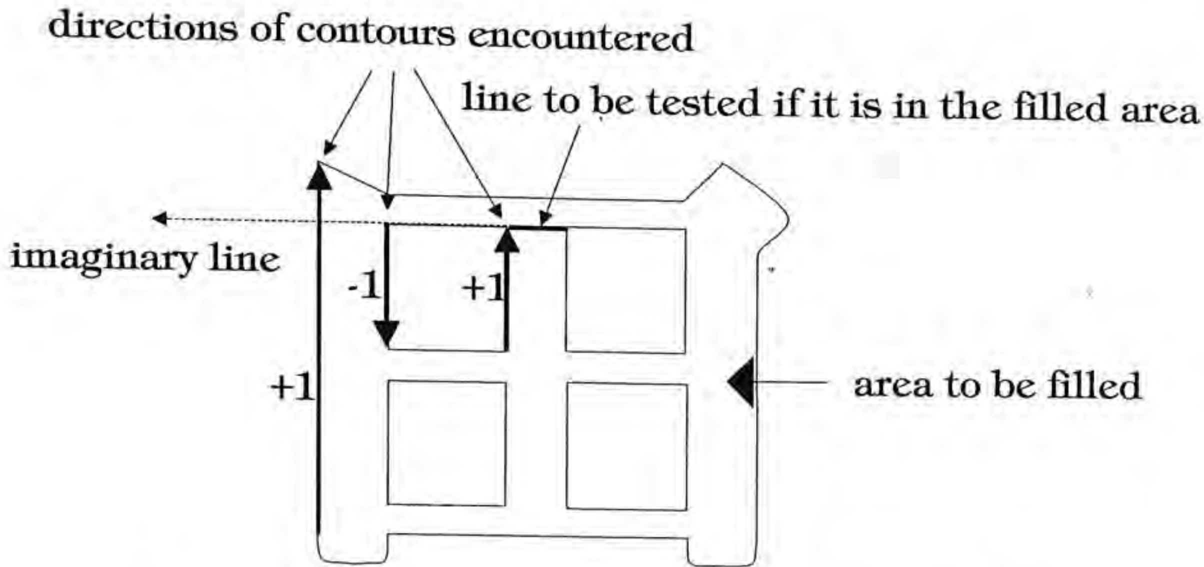


Figure 4.10: Determine if a line is in the filled area.

The conditions which determine whether two lines constituting a stroke are (Figure 4.11):

1. The lines are overlapping (i.e. the center point of the shorter line falling between the endpoints of the longer line).
2. They are in opposite direction.
3. The distance between the lines is not much greater than the width of a stroke.

After the strokes are identified, the segments constituting the serifs of these strokes are also extracted.

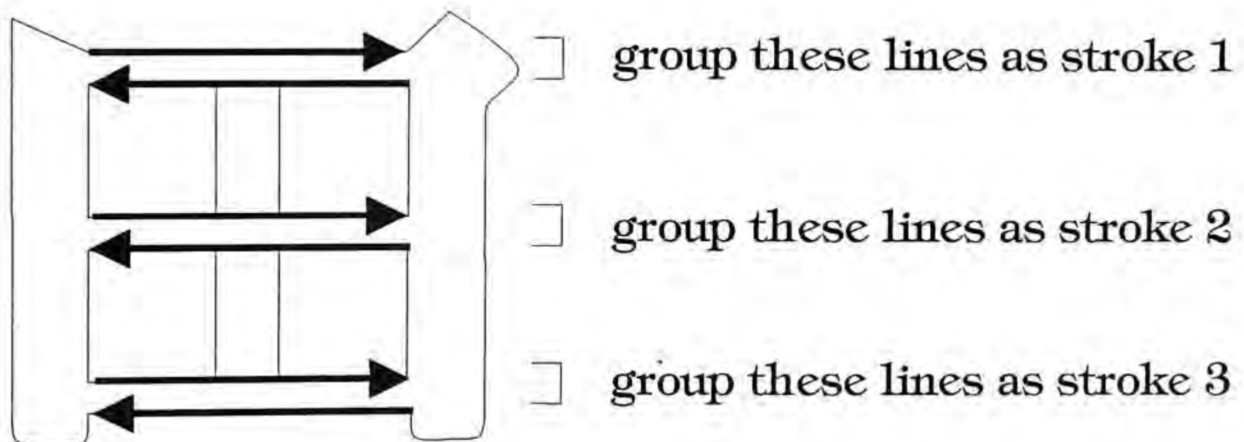


Figure 4.11: Grouping of lines to form strokes

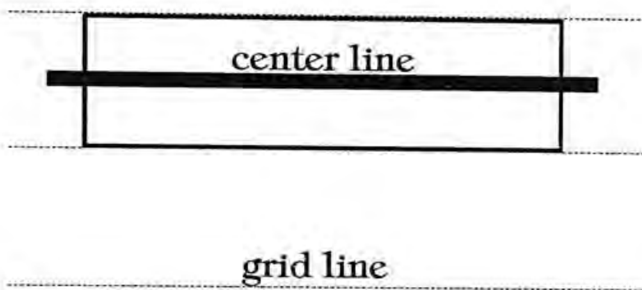
Determining Width and Center Line of Stroke

The *width of a stroke* is the shortest distance between the two main lines constituting the stroke. The *center line* of a stroke is the line which falls halfway between the two main lines of the stroke.

Centering or Anti-Centering of Stroke

The scaled outline strokes are displaced according to their widths. There are two ways of displacing the stroke: centering and anti-centering. If *centering* is applied on a stroke, it will be moved so that its center line is positioned at the nearest middle point of grid lines (Figure 4.12a). If *anti-centering* is operated on a stroke, it will be migrated so that its center line falls on the nearest grid line (Figure 4.12b). To determined which displacement method should be used, we have the simple rule:

```
if round(stroke_width)==0 or (round(stroke_width))mod(2)==1 then
        set displacement=centering;
else
        set displacement=anti-centering;
end {if}
```



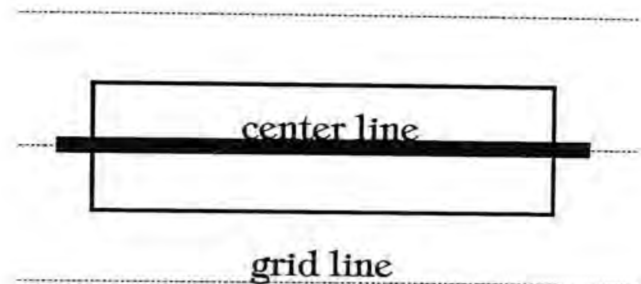


Figure 4.12: Centering and anti-centering

The aim of this process is to keep the order of stroke width by making as few modifications to the master character as possible.

4.3.3 *Maintaining Regular Spacing Among Strokes*

The hinting process described in the above section can only regulate the widths of strokes, but it does not care about the spacing among adjacent strokes. So, in many occasions, the hinted character would have irregular spacing between adjacent strokes, which would greatly downgrade the appearance of the character.

An example of the undesirable cases is shown in Figure 4.13. There are two white space groups of horizontal strokes in the character. In the original outline, the two space groups are of equal width. If the character is only hinted for preserving the order of strokes' thickness, the resulted bitmap character may have horizontal strokes apart unevenly.

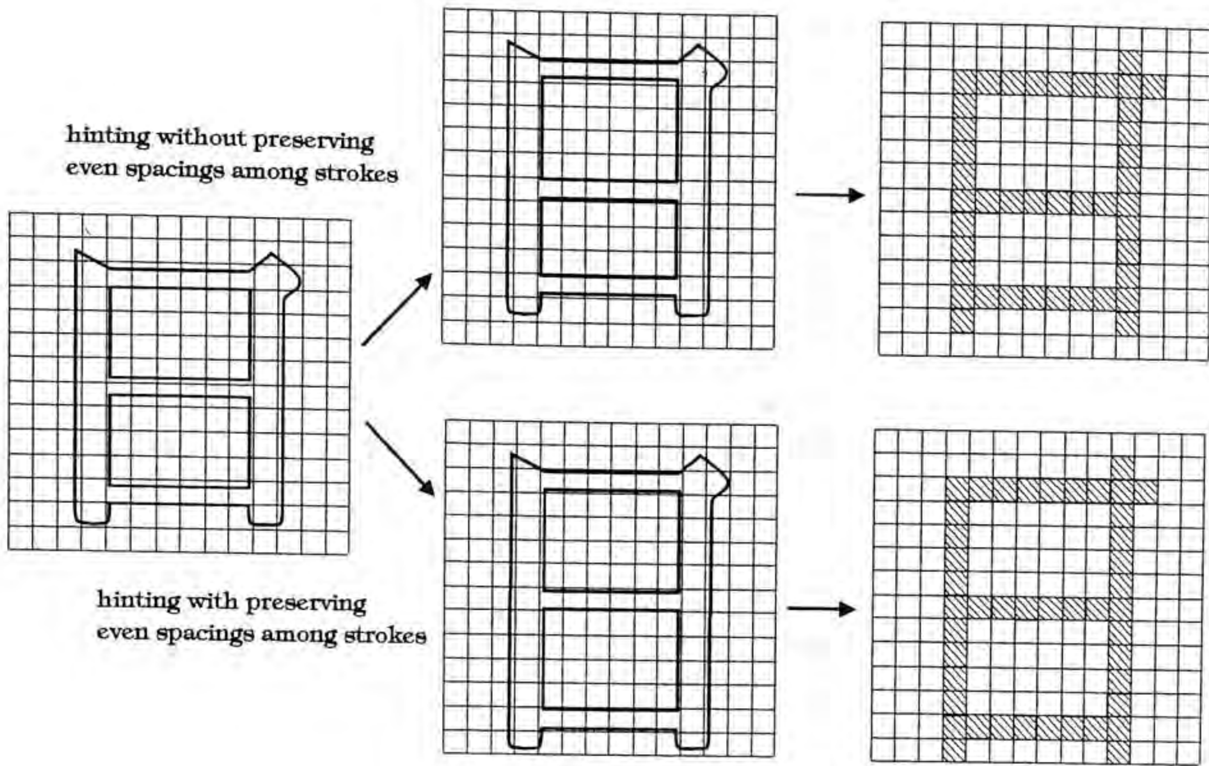


Figure 4.13: Hinting with and without preserving proper spacing

Therefore, besides the information about preserving widths of strokes, we have to acquire some more information to regulate the spaces. For example, we can measure the distances between every adjacent pair of straight strokes in a character and, based on these extra figures, the strokes can be further adjusted to give a nice looking bitmap character.

In the following discussion, we represent a stroke with its pair of main lines with the serifs removed.

Space Group

If the center of the shorter stroke falls between the two end points of the longer one, these two strokes are *overlapping* (Figure 4.14). Two strokes are said to be *adjacent* if (i) they are overlapping and (ii) there is no other stroke overlapping with and present between these strokes. For example, in Figure 4.15a and 4.15b, both stroke 1 and stroke 2 are adjacent to stroke 0; however, in Figure 4.15c, only stroke 1 is adjacent to stroke 0.

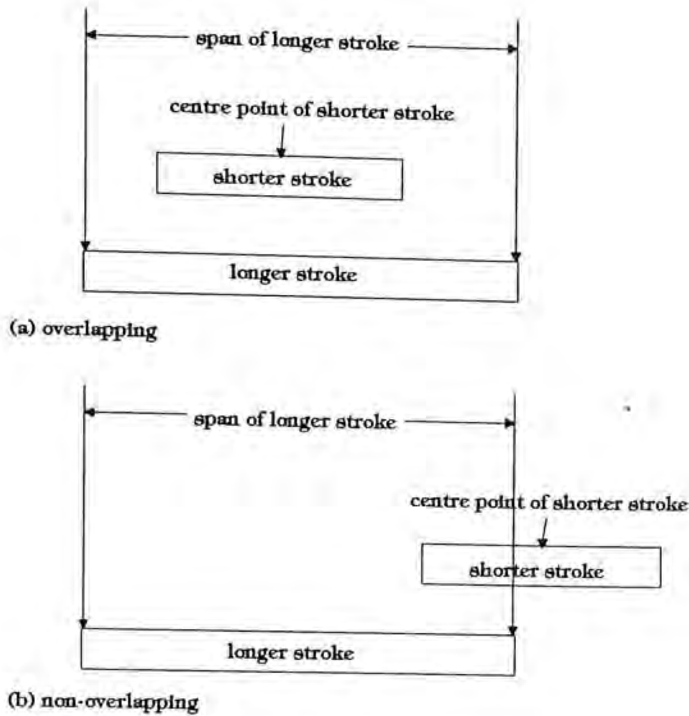


Figure 4.14: Overlapping and non-overlapping strokes

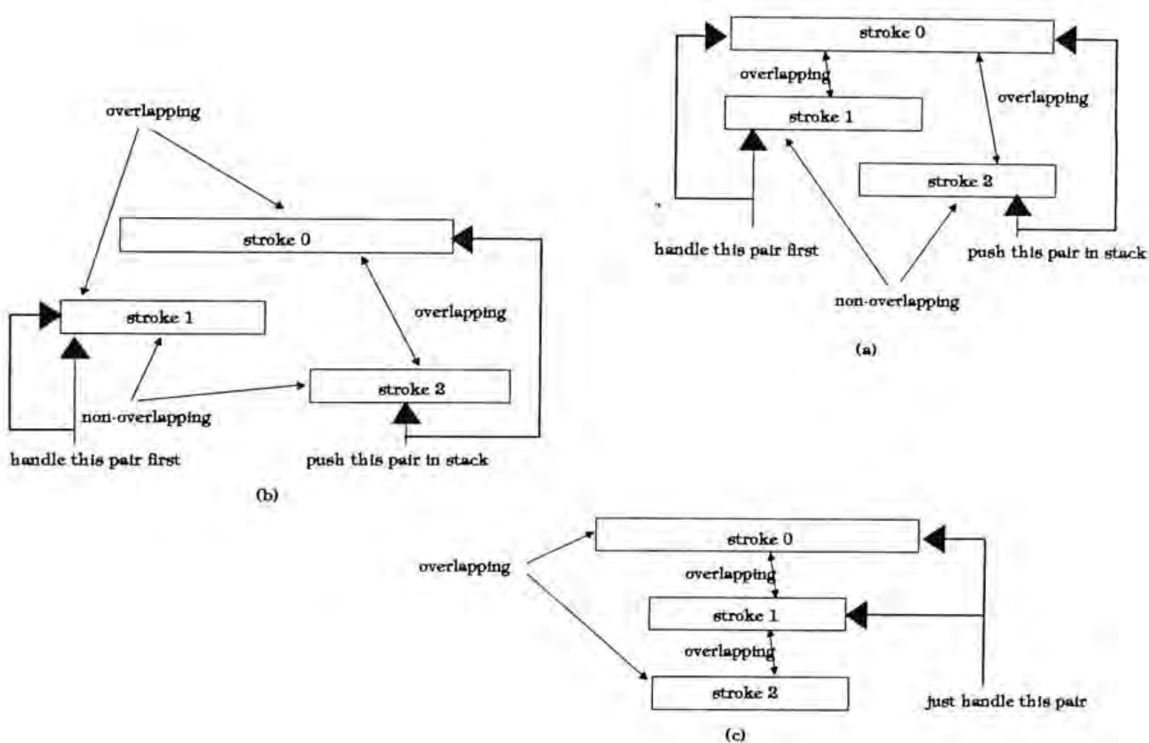
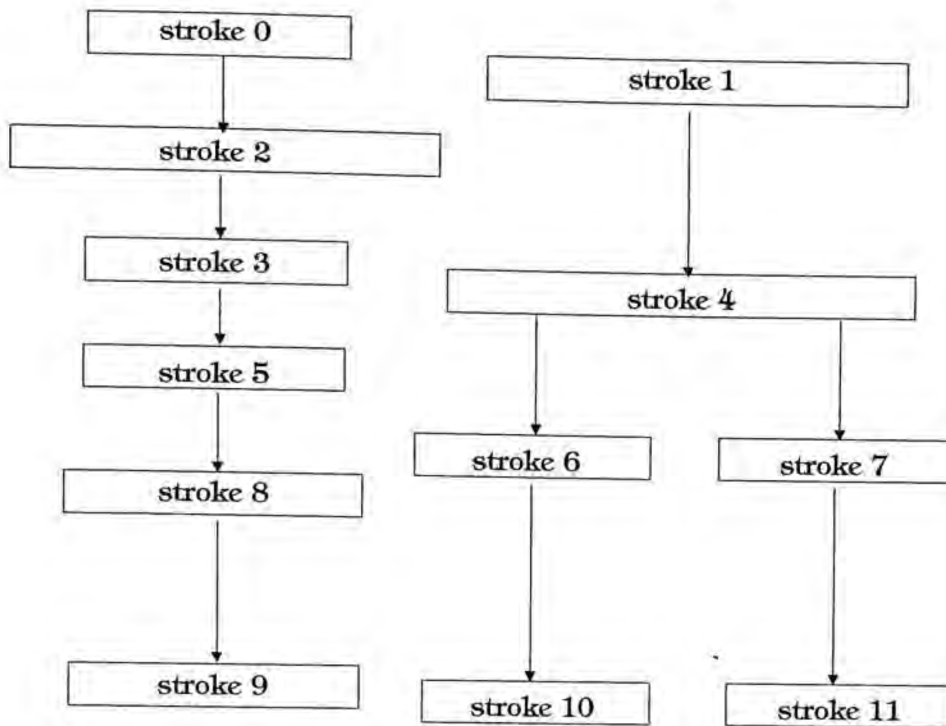


Figure 4.15: A stroke overlaps with more than one strokes: in (a) and (b), both stroke 1 and stroke 2 are adjacent to stroke 0; in (c) only stroke 1 is adjacent to stroke 0.

The unfilled region between two adjacent strokes is called a *space*. Before being capable of measuring the widths of the spaces, of course, we must know which two strokes form an adjacent pair in a character. A list of strokes in which every consecutive pair of strokes is an adjacent pair is called a *space group* (Figure 4.16). Obviously, it is probable that a Chinese character has more than one space group.



space group 0: stroke 0, 2, 3, 5, 8, 9
space group 1: stroke 1, 4, 6, 10
space group 2: stroke 4, 7, 11

Figure 4.16: 3 space groups.

After finding all space groups in a character, the distances between the center lines of every pair of adjacent strokes are measured. These values are used to compute how far two adjacent strokes should be separated.

Then we assign an elasticity to each space. A space can be classified into elastic or inelastic. The width of an *inelastic space* is maintained under any circumstances while the width of an *elastic space* will not be adjusted given that its change is within a tolerated value. We will illustrate elasticity later.

The algorithm for finding the space groups of a character is:

```
{
purpose: search for the space groups for horizontal strokes in a character
input: horizontal strokes of a character
output: space groups
}
sort all the horizontal strokes in descending order of centerline;
i=0;
stack ={empty};
set={empty};
for each stroke in the sorted list (s0) do
        s1=s0;
        while s1 is not in any stroke group or stack is not empty do
                if s0 is not in any stroke group then
                        add s1 to stroke group i;
                else
                        i=i+1;
                        s1 =pop(stack);
```

```
                add s1 to stroke group i;
                s1=pop(stack);
                add s1 to stroke group i;
        end {if}
        for each of the stroke not in any stroke group (s2) do
                if s2 is not in any stroke group and s2 overlaps with s1 and none of other strokes which is
                between s1 and s2 overlaps both with s1 and s2 then
                        add s2 to set;
                end {if}
        end {for}
        if set is not empty do
                if there is only 1 stroke in set then
                        s3=the stroke in set;
                else
                        s3=stroke which is closest to s0 in set;
                        for each of the remaining stroke in set (s4) do
                                push (stack, s4);
                                push (stack, s1);
                        end {for}
                end {if}
                add s2 to stroke group i;
                s1=s3;
        end {if}
    end {while}
end {for}
```

Algorithm 4.6

In this algorithm, a stack is used to temporarily hold the strokes to be handled later. It is not uncommon that one stroke may overlap with several strokes which are non-overlapping (Figure 4.15). Therefore, we select the closest one to handle first and store the rest in a stack. After tackling the first one, we pop a stroke from the stack, treat it together with the original stroke as elements of a new space group and carry out the space group construction procedure again. This process is continued until the stack is empty and all strokes have been dealt with.

Elasticity of Space

The aim of regulating spaces in a character is to ensure that two originally equal wide spaces would result in equal wide spaces in the resulted bitmap character. In fact, if the character does not have any two spaces of identical width, no alternation of the spaces or strokes' positions would be required at all. The resultant bitmap character would be the best approximate discrete image of the original outline character. In other words, we highly prefer to change the outline as little as possible.

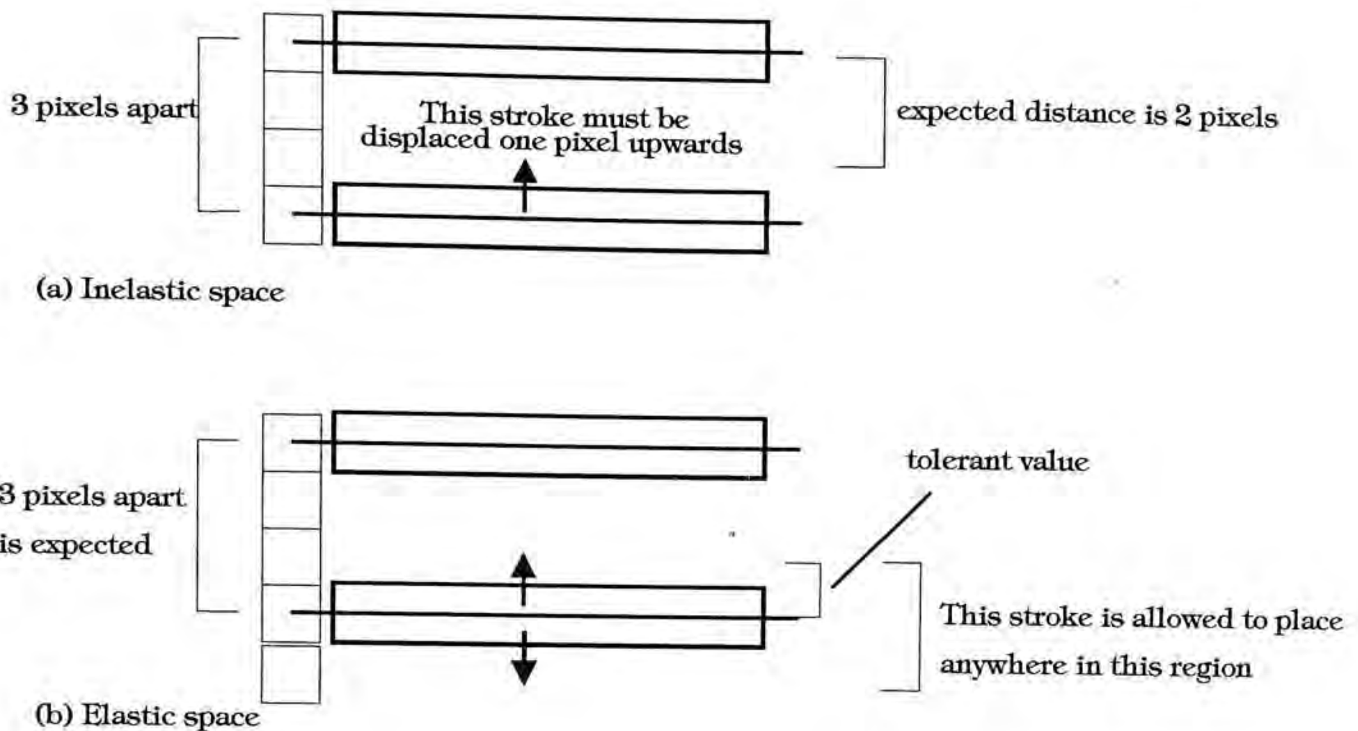


Figure 4.17: Comparison of the movement of stroke just below an inelastic space and elastic space.

The elasticity of a space can help us to determine how the relevant stroke should be displaced. The stroke instantly below an inelastic space has to be completely adjusted according to the expected space width computed from the figures stored in the space group (Figure 4.17a). On the other hand, as shown in Figure 4.17b, the stroke just below an elastic space need not be further relocated if its distance from the upper stroke does not exceed a region which is determined by the expected distance and a tolerant value. If the stroke has been displaced out of this region, it must be moved back into it.

The following rules are used to determine the elasticity of a space:

1. If 5 consecutive spaces in a space group are equal in width, set sp0, sp4 to elastic and set sp1, sp2, sp3 to inelastic (Figure 4.18).
2. If 4 consecutive spaces in a space group are of equal width, then we have to consider 2 cases (Figure 4.19):

   case 1: if s0 is the longest stroke, set sp0 to elastic and set sp1, sp2 and sp3 to inelastic.

   case 2: if s3 is the longest stroke, set sp3 to elastic and set sp0, sp1 and sp2 to inelastic.
3. If 3 consecutive spaces in a space group list are equal in width (Figure 4.20), then we have to consider 3 cases:

   case 1: if s0, s1 are in the same part and s2, s3 are in the same part (see section 4.3.3 for the details of determining the part of a character), or s0, s1, s2 and s3 are in 4 different parts, then set sp0, sp2 to inelastic and set sp1 to elastic;

   case 2: if s0, s1, s2 are in the same part and s3 is in another part, then set sp0 and sp1 to inelastic and set sp2 to inelastic;

   case 3: if s1, s2, s3 are in the same part while s0 is in another part, then set sp0 to elastic and set sp1, sp2 to inelastic.

4. If 2 consecutive spaces in a space group list are equal in width, set both of their elasticity to inelastic (Figure 4.21).
5. Set the elasticity of the rest to elastic.

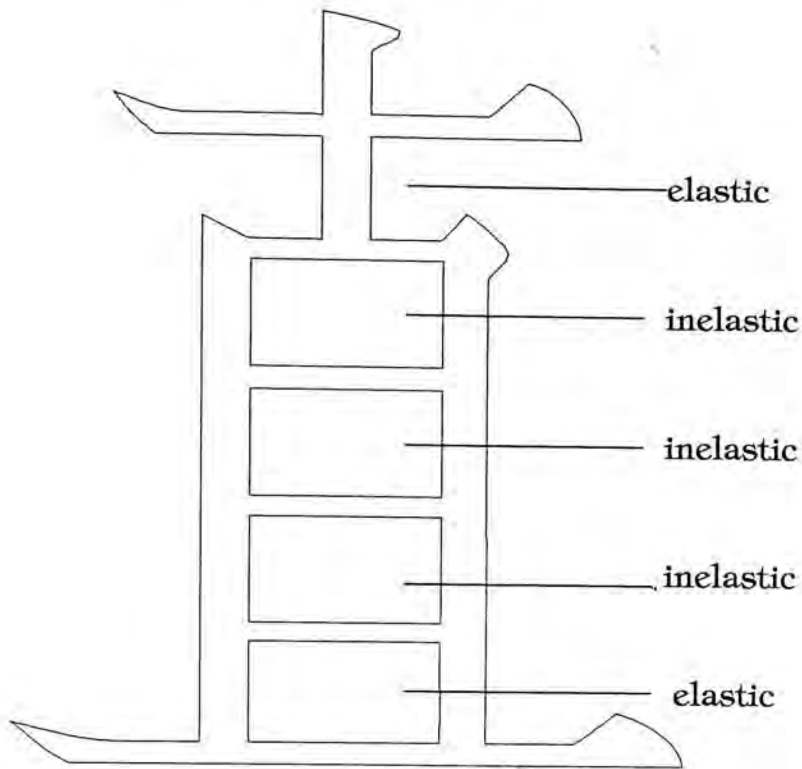


Figure 4.18: 5 consequent equal wide spaces.

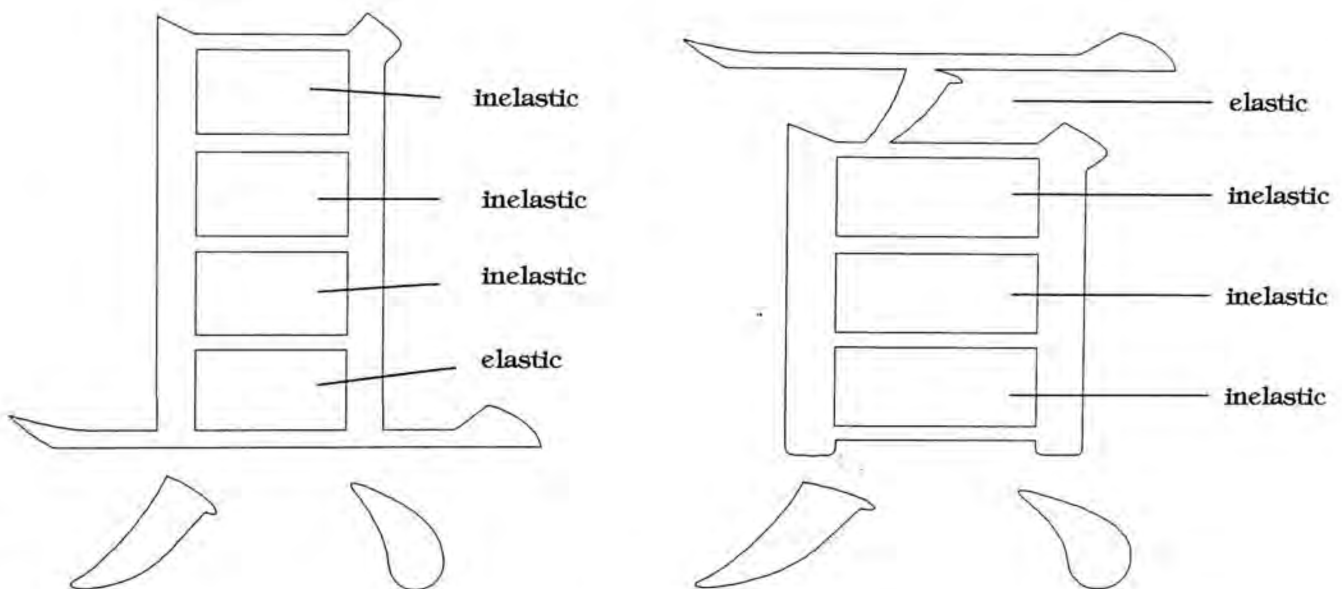


Figure 4.19: Two characters with 4 consequent equal wide spaces.

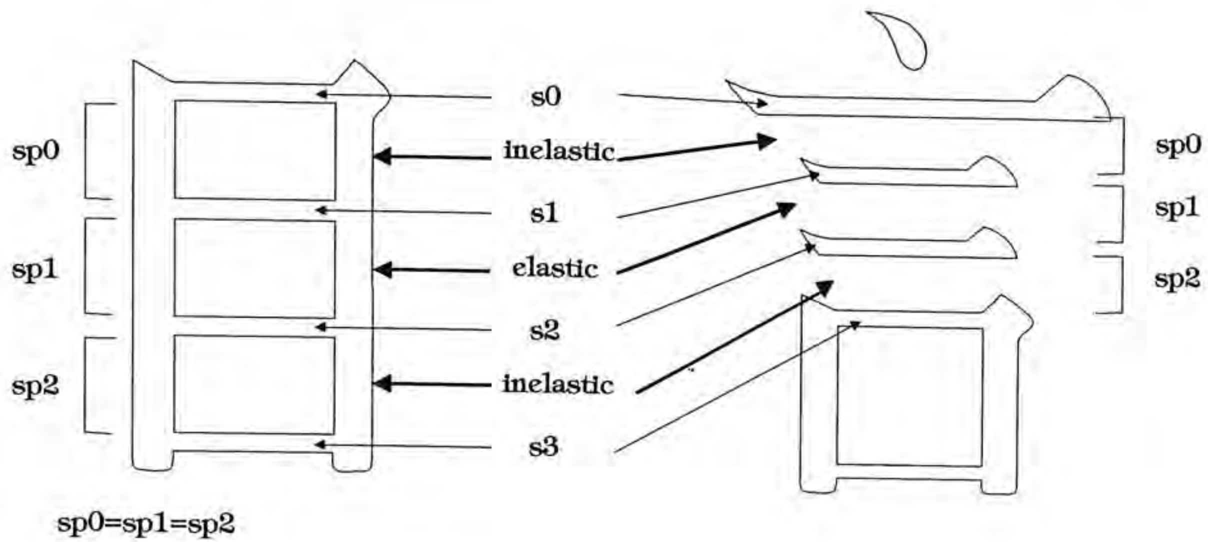


Figure 4.20: 3 consecutive spaces with equal widths.

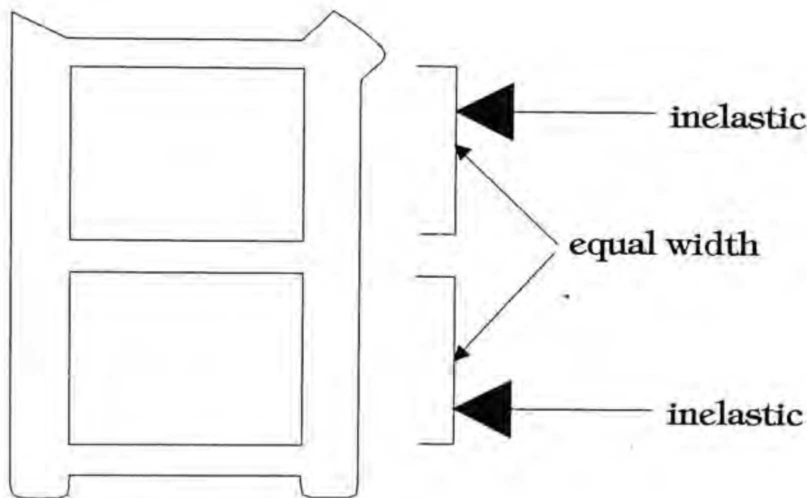


Figure 4.21: 2 consecutive spaces equal in width.

Adjustment of the Spaces

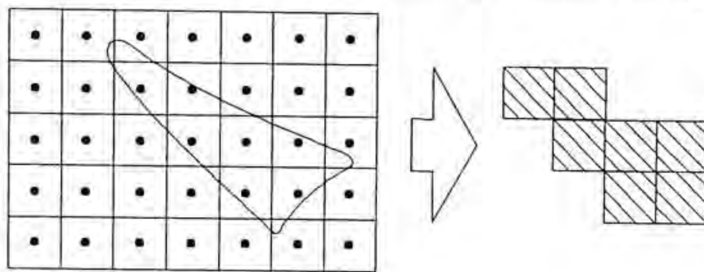
First of all, we calculate the displacement of each stroke in a character for preserving stroke width with the techniques described in section 4.3.1. Then, with reference to the information stored in each space group, these displacements are adequately adjusted to regulate the spaces with the techniques discussed above. The displacement parameters are actually adjusted one by one starting from the top stroke down to the bottom stroke in each space group. Finally, the displacement values are added to the control points of the corresponding strokes.

4.3.4 Hintings of Single Stroke Contour

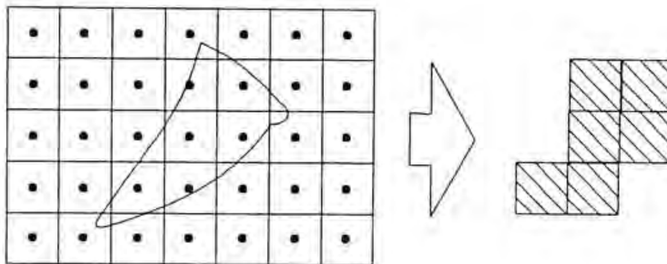
Figure 4.22 shows some rasterized bitmaps of single strokes with features not very well matching with the original outline. The serifs and the curve shape of the strokes are not well preserved and these deficiencies may reduce the readability of the character. Therefore, in addition to maintaining stroke width and spacing, we have to keep up some more features, such as the serifs and the round bow curve of dot stroke. Otherwise, even the same single outline strokes would give very different bitmap images.

The outline should be adjusted so that:

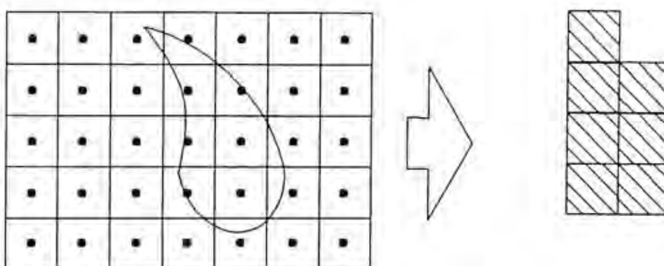
1. the serifs of a stroke are conserved;
2. the round shape of the dot stroke is conserved;
3. similar curve strokes outline give similar bitmap strokes.



(a) right slanted stroke



(b) left slanted stroke



(c) dot stroke

Figure 4.22: Some unsatisfactory bitmaps of single strokes.

As illustrated in Figure 4.23, if we can appropriately align two extrema of the outlines to the grid lines, the resulted bitmap strokes would look better with retaining some significant features. In the following sections, we will discuss how we can do these adjustments for single stroke outline.

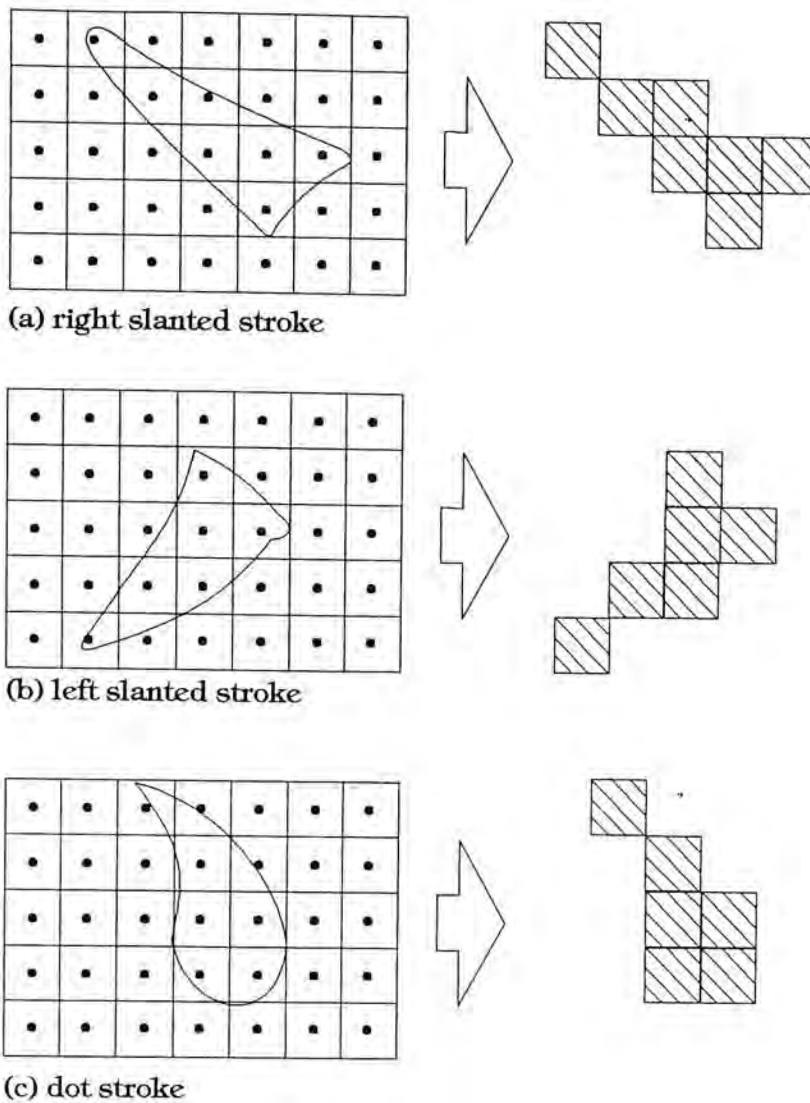


Figure 4.23: Bitmaps generated from outlines fitted to grid lines

Steps for Grid Fitting of Single Stroke

The grid fitting process includes 5 steps:

1. Classify the contours into different parts.
2. Extract single stroke contours.
3. Identify the type of the single stroke contours.
4. Find extrema of the identified curve strokes for each part.
5. Add hints to the outline.

Determining parts

A character can be composed of more than one disjoint part. Each part may consist of one contour or more contours with inner contours inside an outer contour (Figure 4.24). Before the contours can be classified into different parts, we must determine the geometric relationships among the contours. We can handle each part of the character individually.

For European fonts, one part cannot be found inside another part, therefore a contour can be found inside at most one another contour. However, for Chinese fonts, one part can be found inside another parts (Figure 4.24), so a contour can be enclosed by more than one contours.

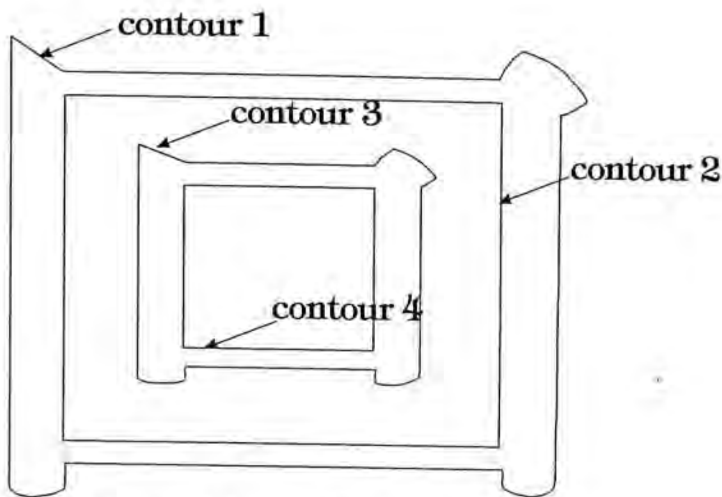


Figure 4.24: One part is found inside another part

To classify contours into parts, we have the following algorithm:

```
{
purpose: to classify contours into parts
input: all contours
output: contours with appropriate part number
}
set part_number = 0;
for each contour i do
        compute n = number of contours enclosing contour i;
        if n == 0 or a multiple of 2 then {contour i does not belong to any part of the outer contours}
                if contour i has not been assigned a part number then
                        set the part number of contour i to part_number;
                        increment part_number;
                end {if}
        else {contour i forms a part with one of the outer contours}
                find contour j = one of the outer n contours which is found inside the rest n-1 contours;
                if contour i or contour j is already assigned a part number then
                        assign this part number to the other contour;
                else
                        set the part numbers of contour i and j to part_number;
                        increment part_number;
                end {if}
        end {if}
end {for}
```

Algorithm 4.7

Under assumption 3 stated in section 4.3.1, a contour i is detected inside contour j if a point on contour i falls inside contour j. Parity check can be used to determine whether a point is inside an contour.

Extracting Single Stroke Contour

A *single stroke contour* is a closed contour description which represents one single stroke of a character. A *multiple strokes contour* is a closed contour description which represents more than one single strokes of a character (Figure 4.25).

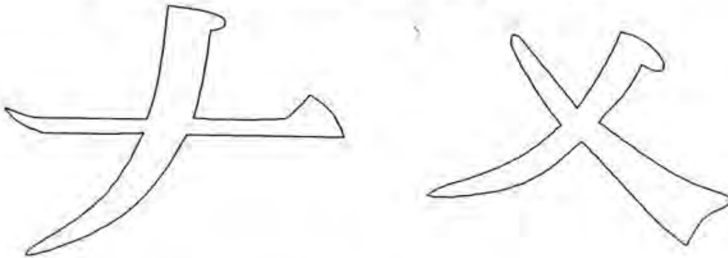


Figure 4.25: Examples of multiple strokes contour

Here are 4 rules for identifying single stroke contour:

Rule 1: If a contour encloses another contour, it is not a single stroke contour.

Rule 2: If it is geometrically smooth at the conjunction point of two segments, the two segments belong to the same stroke.

Rule 3: If a segment is turning right relatively to the precious segment, these two segments belong to the same stroke.

Rule 4: If the length of one or both of two contiguous segments is smaller than a threshold value, the two segments belong to the same stroke.

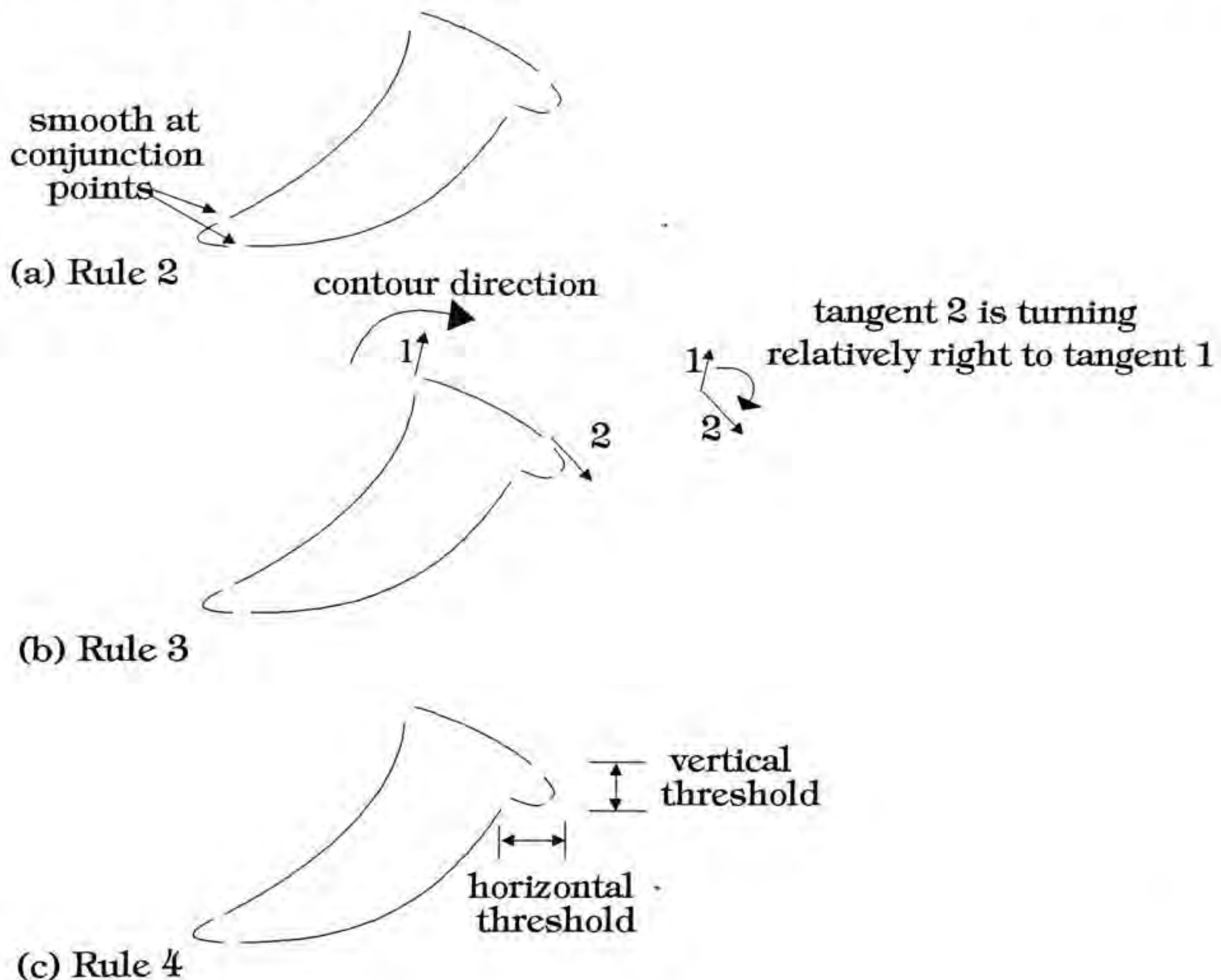


Figure 4.26: Illustration of Rule 2, 3 and 4.

Rule 1 is trivial. Rule 2 means that the first derivatives of the two segment curves are approximately the same at the conjunction point. This ensures that the whole stroke looks smooth (Figure 4.26a). Rule 3 comes from the property of a close curve. Since the contours are running in clockwise direction, the subsequent segment should turn relatively right to the previous segment so as to form a close curve (Figure 4.26b). The turning direction can be determined by the cross product of tangent 1 and tangent 2 as follow:

```
{
purpose: to determine the relative turning direction of two tangents
input: tangents of the second end points of two segments
output: relative turning direction
}
compute crosspdt=CrossProduct (tangent1, tangent2);
If crosspdt==0 then
        turning_direction = NIL;
else if crosspdt>0 then
        turning_direction = right;
else
        turning_direction = left;
end {if}
```

Algorithm 4.8

Rule 4 handles the case that Rules 2 and 3 are violated solely due to the serifs. Actually, the threshold value can be replaced by one horizontal and one vertical threshold value so that the actual length of the curve is not required to calculate (Figure 4.26c).

The algorithm for determining a single stroke is:

```
{
purpose: to determine whether a contour is a single stroke contour
input: all contours of a character outline
output: status of each contour
}
for each contour do
        if the contour encloses another contour then
                set single_stroke = FALSE;
        else
                set single_stroke = TRUE;
                for each subsequent pair of segments of the contour do
                        if the two segments does not belong to the same stroke (determined by Rules 2, 3 and 4) then
                                set single_stroke = FALSE;
                                break;
                        end {if}
                end {for}
        end {if}
end {for}
```

Algorithm 4.9

Identifying the Extracted Strokes

Suppose we have extracted single strokes from a character outline. The following guide lines can help us to distinguish the type of the stroke:

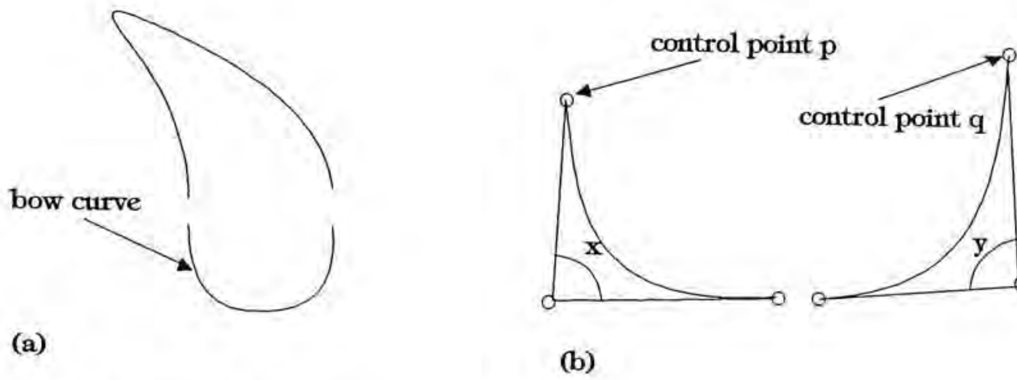


Figure 4.27: Bow curve of dot stroke

Dot stroke: The significant feature of a dot is a bow curve (Figure 4.27a). A bow curve is a curve with great curvature and, usually, such a bow curve is represented by two consecutive quadratic Bezier curves. With reference to Figure 4.27b, the conditions for detecting a bow curve are: (1) $x>45°$, $y>45°$, (2) $150°<=x+y<=200°$ and (3) the curve is smooth at control points p and q. The direction that the sharp end slants determines the inclination orientation of the stroke (Figure 4.28c, d).

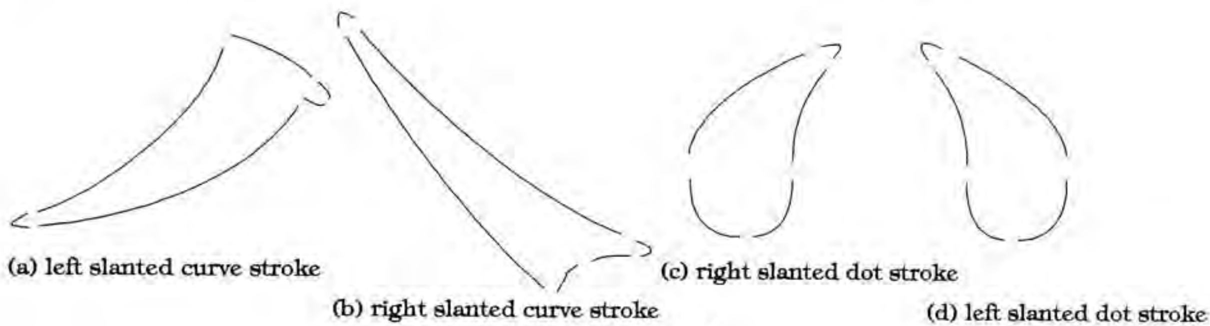


Figure 4.28: 4 types of curve strokes

Left Slanted Stroke: It is recognized by a pair of long curves with positive first derivatives (Figure 4.28a). Long curves refer to the two longest curve segments which compose the single contour. Moreover, they must converge in the down-left direction and diverge in the up-right direction.

Right slanted stroke: It is recognized by a pair of long curves with negative first derivative, and they converge in the up-right direction but diverge in the down-left direction (Figure 4.28b).

Finding Extrema of Single Stroke

For simplicity, let us limit our discussions on the case of right slanted curve stroke. The other cases can be treated in a similar manner.

Referring to Figure 4.22, we can see that the serif of the stroke disappears because the corners of the serif do not overlap the scan line. So, if we can shift the stroke such that the corners of the serif fitted to the grid line, the serif will appear again (Figure 4.23).

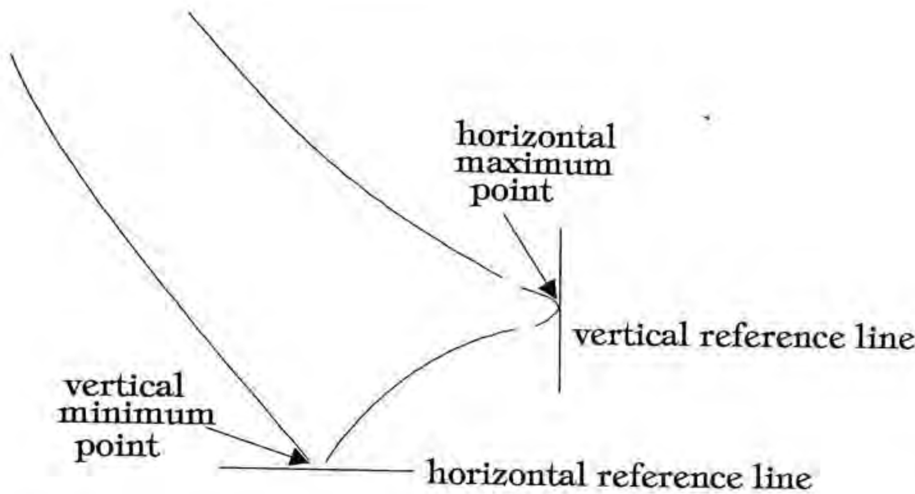


Figure 4.29: Reference lines of a right slanted stroke.

We can find the corners of a stroke contour by locating its extremum points. For instance, considering a right slanted curve stroke, the minimum point in y and the maximum point in x on the contour have to be found (Figure 4.29). These extrema help to determine the vertical and horizontal offsets of the corner points from the grid lines. One should note that an extremum point of a stroke is probably not a control point.

Here is the algorithm used to search for the lowest point of a contour:

```
{
purpose: to find the lowest point of a curve stroke
input: an individual curve stroke outline
output: the lowest point
}
set y_min = A_LARGE_VALUE
for each segment s of the stroke contour do
        if s is a straight line then
                if the y-coordinate of the lower end point of s is smaller than y_min then
                        set y_min = y-coordinate of the lower end point of s
                end {if}
        else { s is a Bezier curve }
        { Let p0 and p2 be the two end points of the Bezier curve and p1 be the off line control point }
                if at least one control point of s has a y-coordinate is smaller than y_max then
                        if p1.y<p0.y and p1.y<p2.y then { p1 is the lowest point }
                                find p, the minimum point in y of s;
                                if p.y < y_min then
                                        set y_min = p.y;
                                end {if}
                        else if p0.y<=p2.y then {p0 is the lowest point }
                                set y_min = p0.y;
                        else { p2 is the lowest point }
                                set y_min = p2.y;
                        end {if}
                end {if}
        end {if}
end {for}
```

Algortihm 4.10

For a quadratic Bezier curve, the minimum point in y direction is found at the point where the first derivative is zero and the second derivative is positive. It is simple to compute these

derivatives for a quadratic equation. If such an extremum point does not exist on the Bezier curve, the local minimum must be the lower end point. A similar technique can be used to find the horizontal maximum point.

Inserting Hints in the Outline

Vertical extrema are transformed to horizontal reference lines and horizontal extrema are transformed to vertical reference lines. These reference lines together with the set of affected points are saved as hints in the outline description.

At run time, distances of the reference lines from their nearest grid lines are computed. These offsets are then added to all the related control points of the single strokes. If the coordinates of a control point are specified relative to the previous control point, a hint can apply to all the affected points by merely adding the offset to the first affected point and adding the reversal of the offset just after the last affected point.

Application to the Other 3 Types of Single Strokes

The same procedure can be applied to the other 3 types of curve stroke (left slanted curve stroke, right slanted dot and left slanted dot), with the following amendment:

1. Left slanted curve stroke: vertical maximum point and horizontal maximum point are searched;
2. Right slanted dot: vertical minimum point and horizontal minimum points are searched;
3. Left slanted dot: vertical minimum point and horizontal maximum points are searched (same as that of right slanted curve).

4.3.5 Storing the Hinting Information in Font File

In the above discussion, we do the hinting process at run time. However, it is very time consuming to do so. The efficiency can be raised a lot if we attach the hinting information to the end of each original font data.

Data Structure

To test how good the performance of a font file with hinting information could be, we attach hints to a font file with the following data structure:

Type	**Name**	**Description**
BYTE	numVertStroke	Number of vertical strokes in the character.
STROKE_TYPE	vertStroke [*n*]	Array of information of each vertical stroke; *n* is the number of vertical strokes.
BYTE	numHoriStroke	Number of horizontal strokes in the character.

STROKE_TYPE	horiStroke [*n*]	Array of information of each horizontal stroke; *n* is the number of horizontal strokes.
BYTE	numVertStrokeList	Number of vertical stroke group in the character.
STROKE_LIST_TYPE	vertStrokeList [*n*]	Array of information of each vertical stroke group; *n* is the number of vertical stroke groups.
BYTE	numHoriStrokeList	Number of horizontal stroke group lists in the character.
STROKE_LIST_TYPE	horiStrokeList [*n*]	Array of information of each horizontal stroke group; *n* is the number of horizontal stroke group.
BYTE	numSingleContour	Number of single contour in the character.
SINGLE_CONTOUR_TYPE	singleContour [*n*]	Array of information of each single contour.

where the definitions of structures and the meanings of the variables are:

STROKE_TYPE:

Type	**Name**	**Description**
BYTE	order	Order of the stroke (the first stroke is of order 0, the second is of order 1, and so on.).
BYTE	width	Width of the stroke.
BYTE USHORT	centerLine	Position of the center line of the stroke; its leftmost bit indicates its type: if that bit is set, it is of type USHORT with the rest 15 bits storing the value; otherwise, it is of type BYTE with the rest 7 bits storing the value.
BYTE	numControlPointPairs	Number of pairs of control points; each pair contains a starting control point and a ending control point, and the intermediate control points are in consequent positions
BTYE	flags[*n*]	Array of flags for each pair of control points; *n* is the number of pairs of control points.
BYTE USHORT	startControlPoint [*n*]	Array of starting control points, their types are indicated by flags; *n* is the number of pairs control points.
BYTE	endControlPoint [*n*]	Array of ending control points; the first number is relative to 0; others are relative to previous number; *n* is the number of pairs of control points.

Each flag is a single byte. Their meanings are shown below.

Flags	Bit	Description
Type of starting point	0	If set, the type of the starting control point is USHORT; otherwise, the type of the starting control point is BYTE.
Existence of ending control point	1	If set, there is a corresponding ending control point; otherwise, control point there is no corresponding ending control point.
Offset of ending control point	2	If set, the relative offset of the ending point from the corresponding starting point is 1; otherwise, the relative offset is 0.
Reserved	3-7	These bits are reserved. Set them to zeroes.

STROKE_LIST_TYPE:

Type	Name	Description
BYTE	order	Order of the stroke group (the first stroke group is of order 0, the second is of order 1, and so on.)
BYTE	numStrokeInList	Number of strokes in the group.
BYTE	strokeOrder [*n*]	Array of strokes in the group; *n* is the number of strokes in the group.
BYTE USHORT	gap [*n-1*]	Array of gaps between consecutive strokes; its leftmost bit indicates its type: if that bit is set, it is of type USHORT with the rest 15 bits storing the value; otherwise, it is of type BYTE with the rest 7 bits storing the value; *n* is the number of strokes in the group.
BYTE	elasticity [*n*]	Array of elasticity for each gap; *n*=int((NumStrokeInList-1)/8)+1.

SINGLE_CONTOUR_TYPE:

Type	Name	Description
BYTE	order	Order of the single contour (the first single contour is of order 0, the second is of order 1, and so on.)
USHORT	vertRefLine	Reference line for vertically adjusting the single contour.
USHORT	horiRefLine	Reference line for horizontally adjusting the single contour.

The experimental results of this hinted file is presented in Chapter 5 and 6.

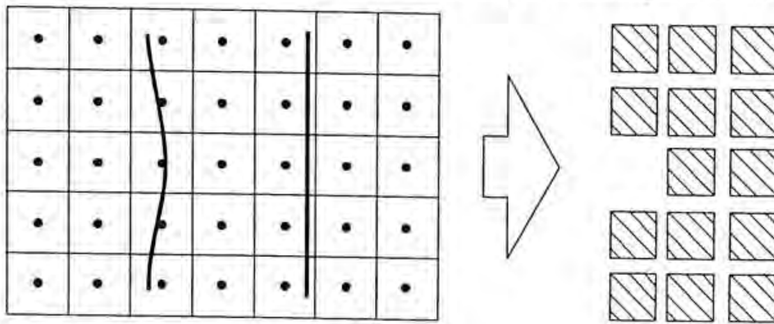
4.4 A Rasterization Algorithm for Printing

We can restate the general rules for determining internal pixels of the output bitmap based on the character outline as follows:

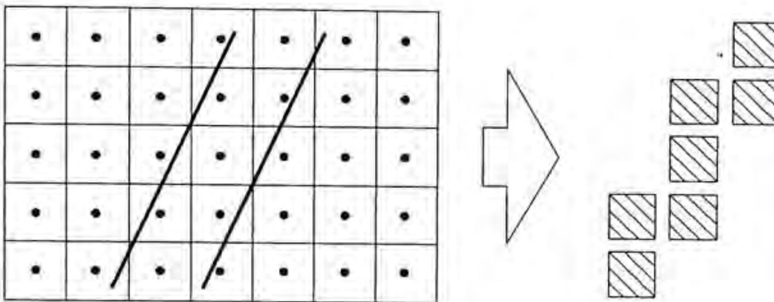
Rule 1: *If a pixel's center falls within the character outline, that pixel is turned on.*

Rule 2: *If a pixel's center coincides with the character outline, that pixel is turned on*

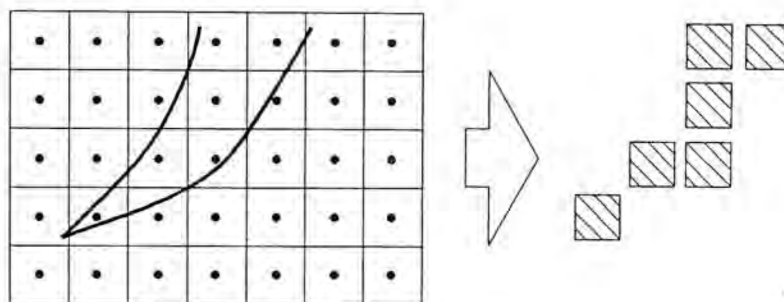
Unfortunately, the geometric properties of a slanted or curve stroke may not be preserved by these simple rules. Figure 4.30 shows 3 examples of the undesired cases.



case 1: not smooth stroke



case 2: uneven width



case 3: destroying geometric property

Figure 4.30: 3 undesired cases of the simple algorithm

In these cases, the bitmap strokes are not smooth and this may destroy the regularity and harmony of the characters composed of these strokes. The root of this problem is that, when a character with continuous boundaries is converted into a discrete image, some information of the shape gets loss. Nonetheless, we can make some adjustments to the rasterization process to lessen this deficiency, especially when printing.

4.4.1 A Simple Algorithm for Generating Smooth Characters

With reference to the rules for determining internal pixels mentioned above, if a resultant bitmap image is of 2 pixels wide, the original outline span can be only wide enough to cover two pixel centers or just not wide enough to cover three pixel centers. In other words, the width of the outline span can range between 1 pixel and 3 pixels (Figure 4.31). So, the maximum error of this

algorithm is 1 pixel. We now present a rasterization method which gives maximum error of 0.5 pixel.

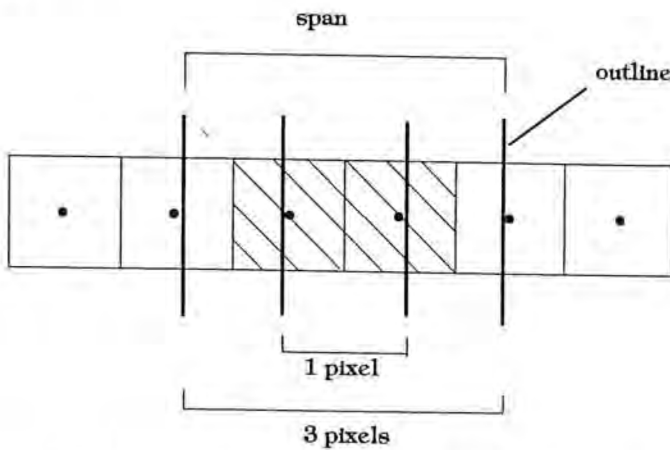


Figure 4.31: A stem with width ranging from 1 to 3 pixels may give a 2 pixels wide image.

Our principle is very simple: the number of pixels to be turned on for a span is determined by rounding the width of that span. For instance, a 2.3 pixel wide outline span will give a 2 pixel wide bitmap image, and a 2.8 pixel wide outline span will result in a 3 pixels wide bitmap image.

Suppose the width of a span is between 1.5 pixels and 2.5 pixels, then 2 pixels (*ROUND(x)=2 for 1.5<=x<2.5*) must be turned on. We have to determine which two pixels are to be turned on. Depending on the width and position of the span, it may overlap 2, 3 or 4 pixel columns:

1. If it overlaps 2 pixel columns, the 2 pixel columns are turned on (Figure 4.32a).
2. If it overlaps 3 pixel columns, the middle pixel and the end pixel with greater coverage are turned on (Figure 4.32b, 4.32c).
3. If it overlaps 4 pixel columns, the middle 2 pixels are turned on (Figure 4.32d).

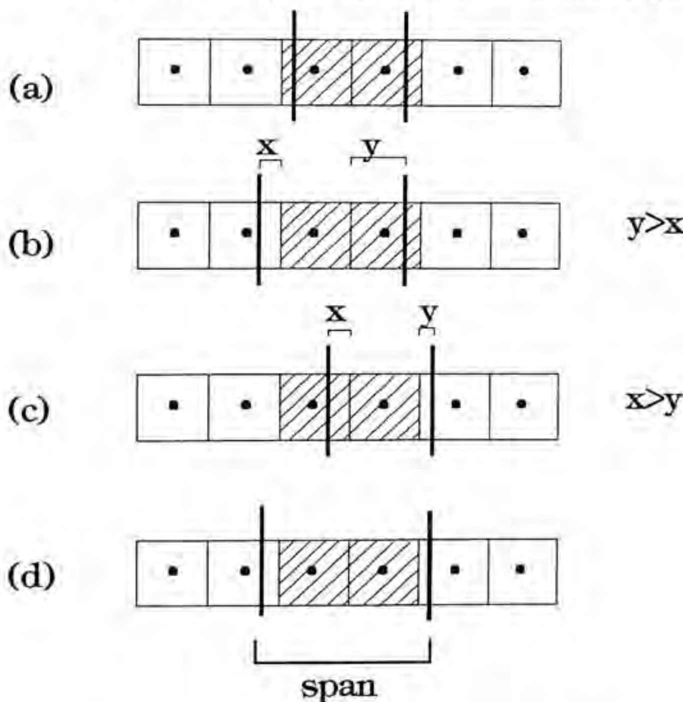


Figure 4.32: Illustration of the algorithm

To illustrate the second case in more detail, we consider one more example. Suppose a 2.2 pixel wide span overlaps 3 pixel columns with coverage of (0.55, 1.0, 0.65). Since ROUND(2.2)=2, two pixels must be blackened. Undoubtedly, the middle pixel with 1.0 coverage must be turned on and, as 0.65 > 0.55, the right pixel is also turned on.

The characters generated by this technique sometimes have ragged edges. If these characters are displayed in low resolution output device such as screen with 80 dpi, they may not look better than those generated by the traditional algorithm. However, when the bitmap characters are output to a printer with medium resolution (say 200 dpi or 300 dpi), blank regions around the characters' boundaries will take on the color of the dots adjacent to them, just like the case of half-bitting [Rubinstein 90, p.80] [Maag 89]. This would increase the qualities of the characters printed.

4.4.2 Algorithm

The algorithm is listed below:

```
{
purpose: to determine which pixel should be turned on
input: intersection points of the outline with each horizontal scan line
output: bitmap image of the outline
}
for each horizontal scan line do
    for each alternative pair of intersection points do
        calculate the distance between the two points, d;
        compute w=ROUND(d);
        count the number of pixels overlapped by the extent of the two points, n;
        if w=n then
            turn the n pixels on;
        else {the middle n-2 pixels are turned when w=n+1 or w=n+2 and n >= 2}
            turn the middle n-2 pixels on;
            if w=n+1 then
                if the left end pixel with more coverage than the right end pixel then
                    turn the left end pixel on;
                else
                    turn the right end pixel on;
                end {if}
            end {if}
        end {if}
    end {for}
end {for}
```

Algorithm 4.11

4.4.3 Results

Figure 4.33 shows the bitmaps that would be generated by the algorithm mentioned above. The deficiencies shown in Figure 4.30 can be corrected to preserve the main features of the characters. Figure 4.34 and Figure 4.35 shows some examples of bitmaps generated by the traditional method and the above algorithm. We can see that the characters generated by the above algorithm is satisfactorily smooth.

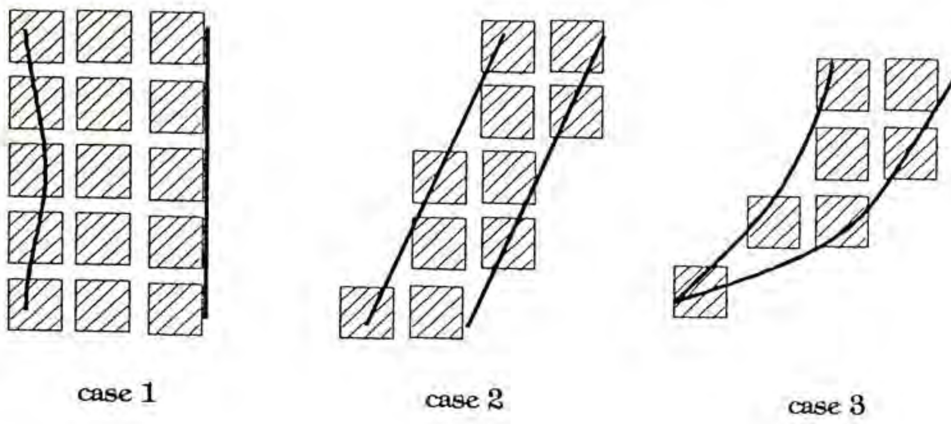


Figure 4.33: Resulted bitmaps generated by the smooth algorithm.

戈火父瓜多狹

戈火父瓜多狹

(a)

戈火父瓜多狹

戈火父瓜多狹

(b)

Figure 4.34: (a) bitmap characters generated by traditional algorithm and (b) bitmap characters generated by the above algorithm in a low resolution printer.

戰火連三月家書抵萬金

戰火連三月家書抵萬金

戰火連三月家書抵萬金

戰火連三月家書抵萬金

Figure 4.35: Upper 2 rows are bitmap characters generated by traditional algorithm and the lower two rows are bitmap characters generated by the above algorithm in a higher resolution printer.

Chapter 5

Experiments

By conducting some experiments, we want to test if the algorithms described in Chapter 4 could do well. We now describe the details of the experiments in this chapter and an analysis of them will be given in Chapter 6.

5.1 Apparatus

Performance measurements are made on an IBM-PC 80486/50 with 8M RAM. Font files are in TrueType format where curve segments are described by quadratic Beziers. For fair comparisons among various fonts, we have carefully examined the font files to ensure their qualities are comparable, i.e. roughly the same number of quadratic Bezier segments is used to describe curves with similar shapes.

5.2 Experiments for Investigating Rasterization Speed

Besides investigating the performance of the new algorithm, we are also interested in seeing the effects of Chinese characters' features, such as stroke count and font style, on the time required for rasterization.

5.2.1 Investigation into the Effects of Features of Chinese Fonts on Rasterization Time

Among the properties of Chinese characters discussed in Chapter 2, we expect that two of them may affect the rasterization speed: the stroke count and the kind of style of the Chinese characters generated.

Stroke Count

The complexity of a Chinese character may probably be reflected by the number of strokes it has. The more strokes a character contains, the more curves would be in the outline and the more complex the character would be.

So, the time required for rasterization is expected to be an increasing function of the stroke count. There is some more information we would like to know: Is the effect of stroke count significant? How does the time for rasterization relate to the stroke count?

Font Style

The style of a character determines the shapes of the basic strokes and the relationship between strokes, and hence the whole impression of the character. We have chosen 5 common kinds of typefaces, namely Ming, Gothic, Hun, Kai and Dai styles (see Figure 2.1 in Chapter 2). They are selected not only due to their popularity in use, but also because of their great varieties in style and structure.

5.2.2 *Improvement of Fast Rasterizer*

The performance of two rasterizers, Rasterizer 1 implementing algorithm 3.1 (the original method) and Rasterizer 2 implementing the techniques discussed in Chapter 4 (the improved method), are compared.

In addition, we want to find out the influences of (i) style and (ii) stroke count of the output bitmap character on the performance. The varying factors are:

1. Font style: Dai style (隸書), Hun style (行書), Kai style (楷書), Ming style (明體) and Gothic style (黑體);
2. Number of strokes: 1 to 30.

5.2.3 *Details of Experiments*

We take the 5401 most commonly used characters of the Big5 Chinese character set from each of the 5 sorts of typefaces mentioned above. The size of the output bitmap characters are fixed at 150 by 150 pixels which is so large that hintings are not necessary.

The characters are then divided into groups according to their stroke count. Rasterization times are recorded for all members of each group, where time includes character generation only and displaying time is excluded. Then the mean times for each group are computed.

5.3 Experiments for Rasterization Speed of Font File with Hints

Since the objective of the above experiment is to measure the improvement of the suggested rasterization algorithm without doing hintings, we do one more experiment to investigate the gain and loss of rasterizing font file with hints.

By simple random sampling (see Appendix), we select 100 characters from the set of most commonly used characters (with 5401 members). Then we extract the data of these characters from the font file in Ming style and save these data in a new file. Inserting hints into this font file using the techniques and data structure introduced in section 4.3 of Chapter 4 produces another new font file with hints. By carefully comparing aspects of these font files, we want to see if it is worthwhile to insert hints to the file.

Chapter 6

Results and Conclusions

In this chapter, we present and analyze the results obtained from performing the experiments described in Chapter 5. Finally, we conclude our work and give some directions for future research.

6.1 Observations

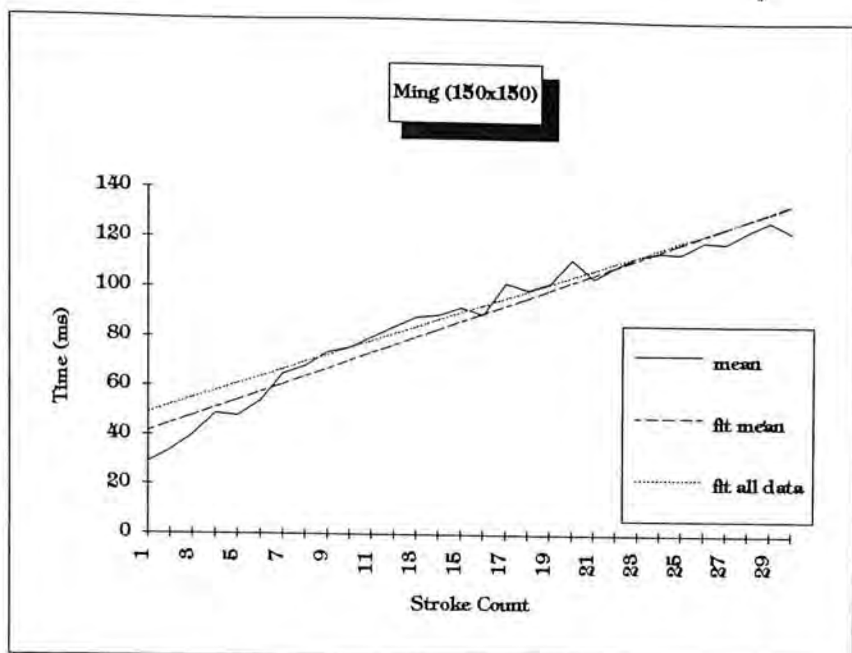
Based on the results of our experiments, we get some interesting observations about the relationship between the time required for rasterization and the features of the Chinese characters.

6.1.1 Relationship Between Time for Rasterization and Stroke Count

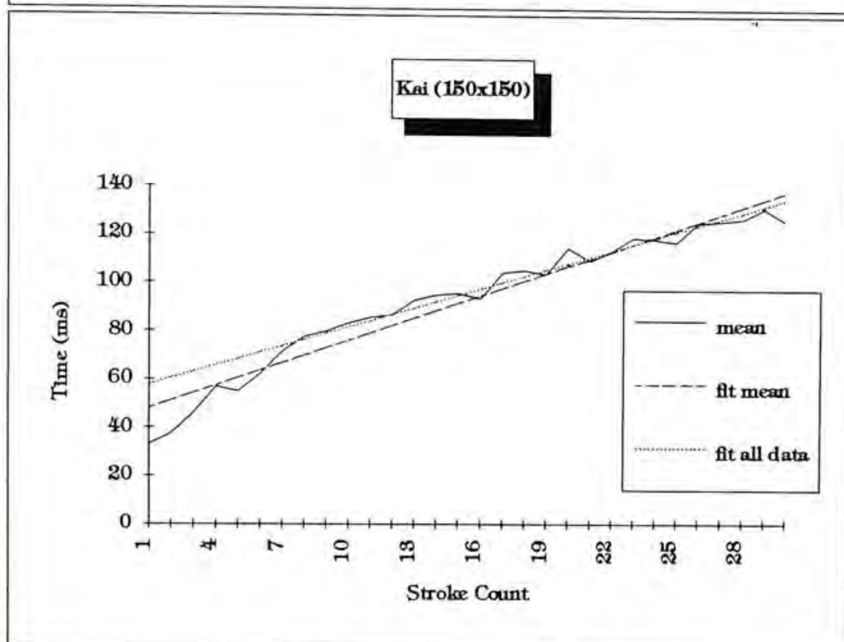
Figure 6.1 shows the plots of average time for rasterization against stroke count for the 5 font styles respectively. It is likely that we can describe the relationships shown in the figures with straight line models. To test this conjecture, we fit linear regression models to the plots. The fitted values of the model are shown in each of the figures as dotted lines, and we found that the model can explain over 90% of the changes in observed data (Table 6.1). Since the slopes of the fitted lines, displayed in Table 6.1, are significantly greater than 0, we can conclude that the time is linearly dependent on the number of strokes.

Typeface	Dai	Hun	Kai	Ming	Gothic
R^2	0.96	0.92	0.97	0.95	0.97
slope	3.20	2.33	3.09	3.22	3.56
vertical intercept	39.18	37.92	45.4	38.22	33.94

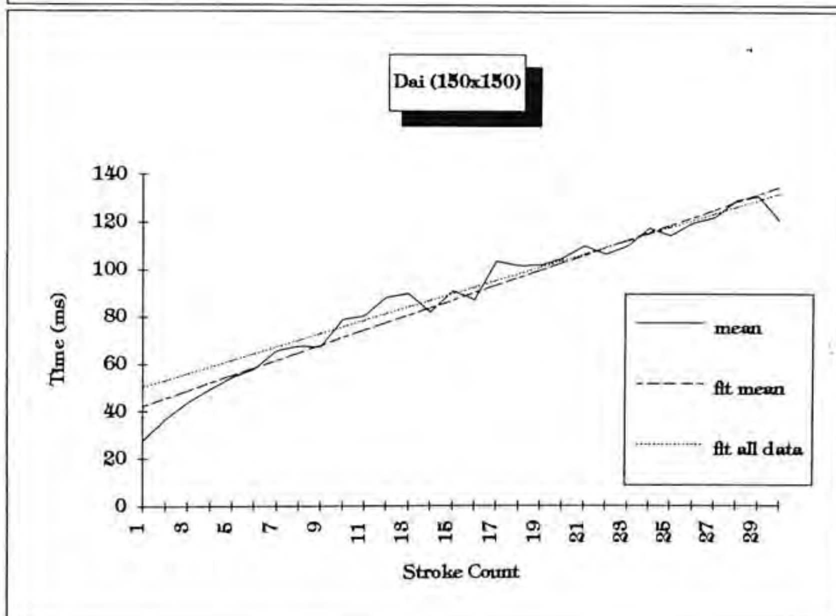
Table 6.1: A summary of the linear regression models fitted to the mean times.



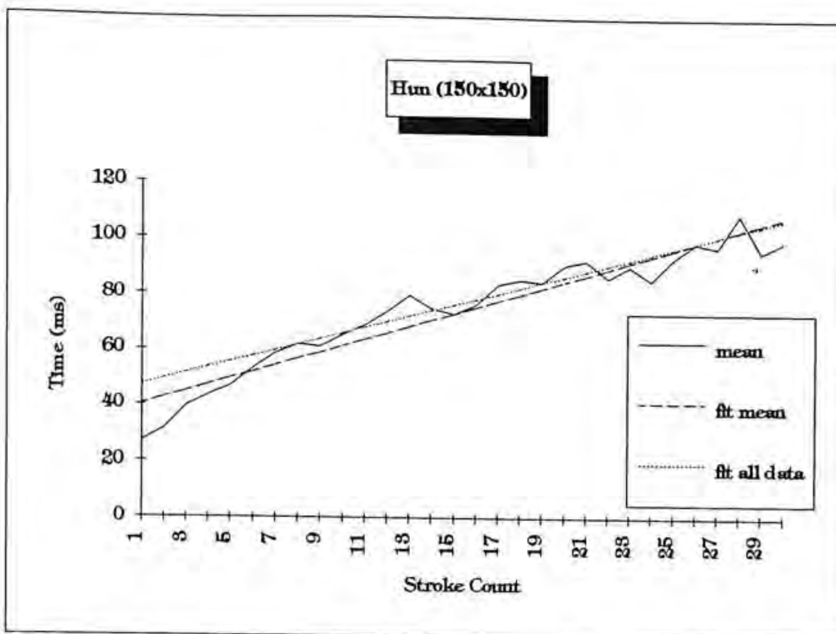
(a)



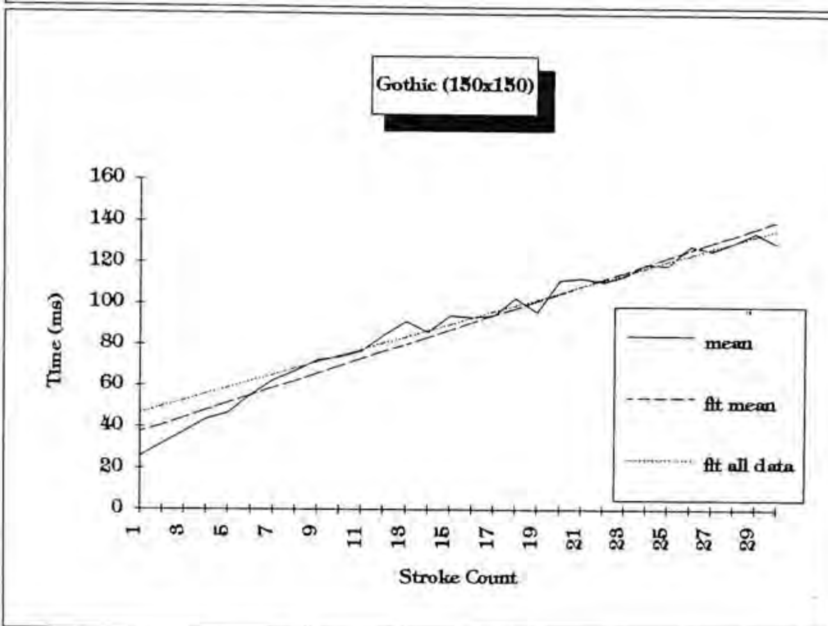
(b)



(c)



(d)



(e)

Figure 6.1: Plots of time against number of strokes for the 5 typefaces

Since there are repeated observations for each stroke count, we can perform a lack of fit test to verify whether the straight-line model is fit to the data [Bhat & Johnson 77; ch.11]. Table 6.2 summarizes the results for the tests based on all data rather than just mean times. The computed F statistics are very large and the corresponding p-values are close to 0, which means the straight-line models can completely account for the changes in data.

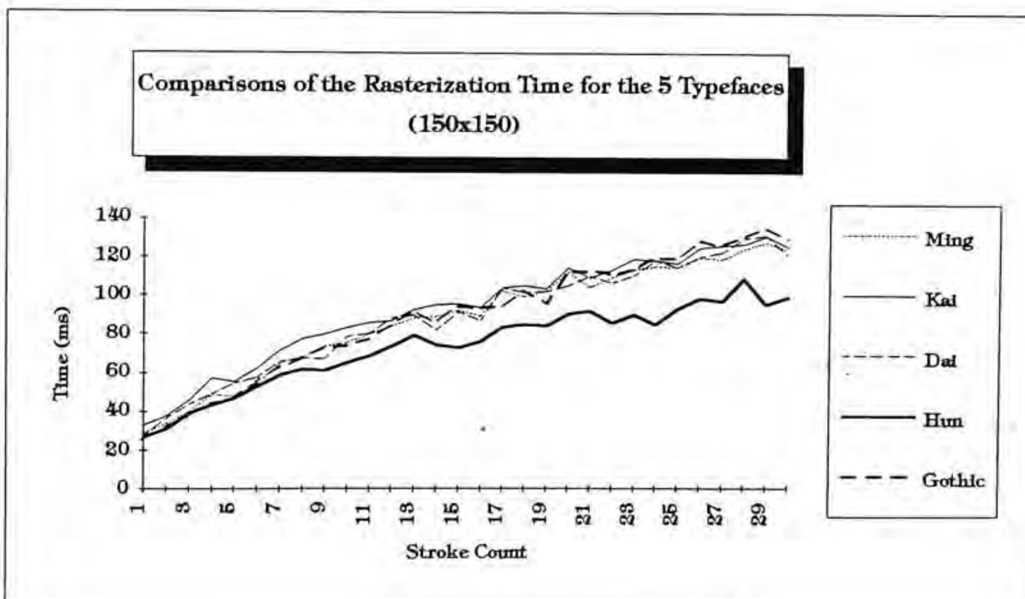
Typeface	Dai	Hun	Kai	Ming	Gothic
R^2	0.96	0.92	0.97	0.95	0.97
slope	2.82	2.06	2.64	2.93	3.08
vertical intercept	47.56	45.22	55.10	46.26	43.75
F-value	1196	488	919	1103	1343
d.f.	28, 5371	28, 5371	28, 5371	28, 5371	28, 5371
p-value	approx. 0	approx. 0	approx. 0	approx. 0	approx. 0

Table 6.2: A summary of the lack of fit test based on all data.

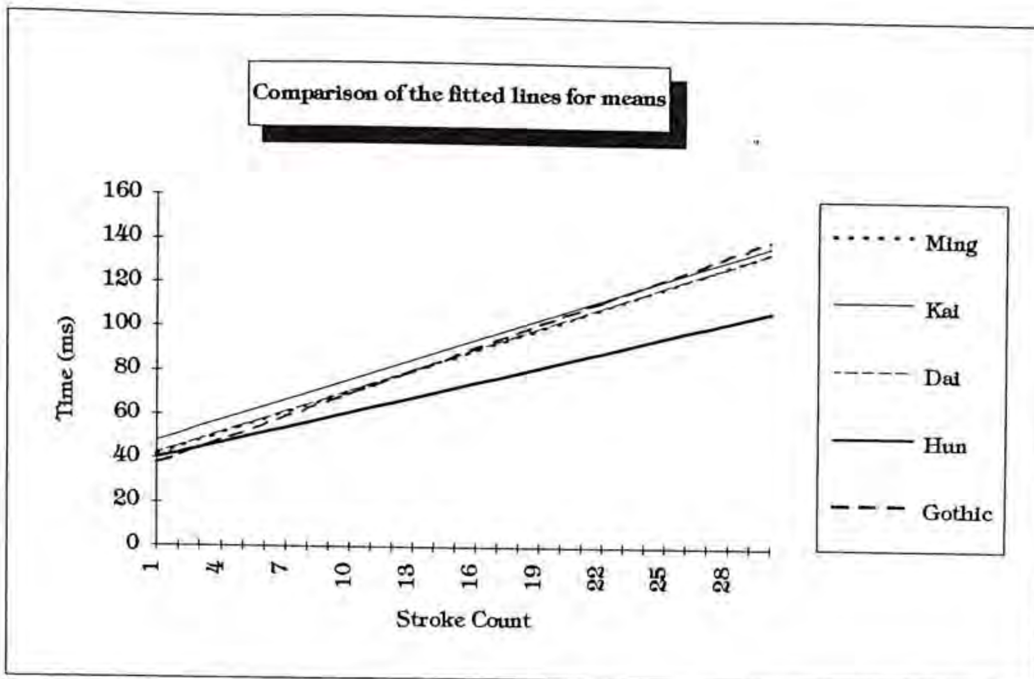
6.1.2 Effects of Style

Next we are going to inspect whether the font style has a significant effect on the rasterization time. Figure 6.2 is resulted from combining the plots in Figure 6.1: Figure 6.2a shows the combination of original lines; Figure 6.2b and 6.3c show the fitted lines for mean times and for all data respectively.

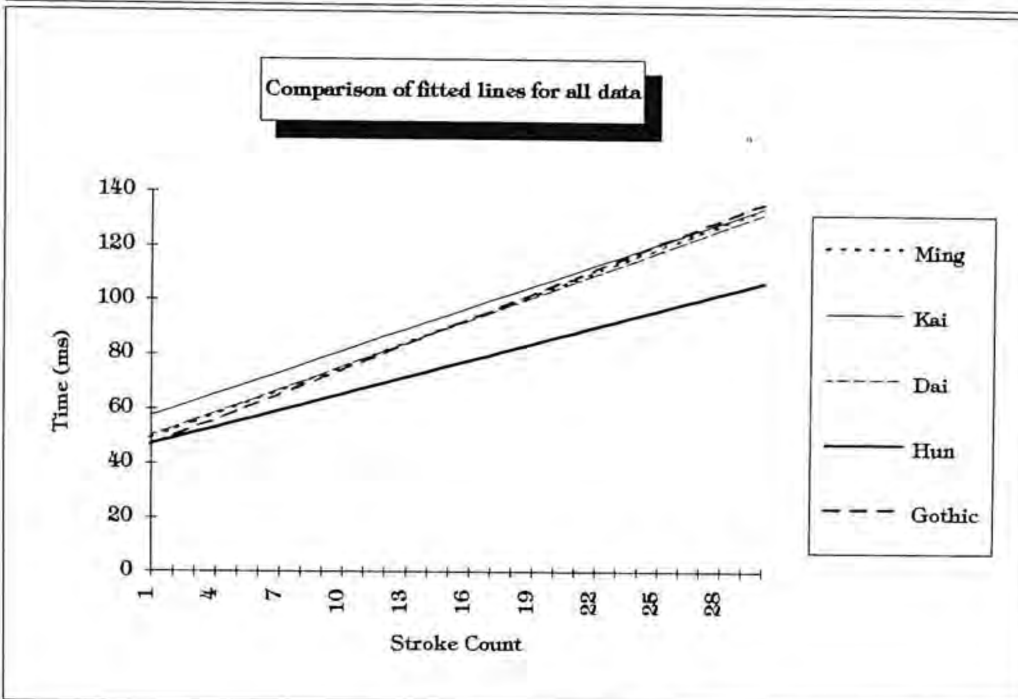
We can observe that, in Figure 6.2a, the curves except that for Hun style intersect with others at many points and, in Figure 6.2b and 6.2c, there is little difference among the fitted lines. The figures shown in Table 6.1 further indicate that the fitted lines are just slightly different from others with respect to the slope and vertical intercept. Therefore, we could arrive at the conclusion that, in general, the font style has insignificant effect on the rasterization time.



(a)



(b)



(c)

Figure 6.2: Plots show comparisons of execution times for the 5 typefaces

6.1.3 Investigation into the Observed Relationship

Now we are trying to investigate the rationale behind the above interesting observations. Figure 6.3 shows the relationship between number of control points and stroke count. We can see that, except for the characters in Hun style, the number of control points of the characters in the other 4 styles do not differ a lot. Even for a character in Hun style, provided that it has less than 25 strokes, its number of control points is not very far away from its counterparts. Figure 6.4 shows the plots of rasterization time against number of control points, from which we observe that the time for rasterization is linearly dependent on number of control points.

The number of control points would determine how many curve or line segments are in the outline, which would in turn affect the time for scan conversion. Also, more control points would imply larger net area of the outline and thus more time would be needed for filling. In other

words, the number of control points in an outline can approximately estimate the time for rasterization. This is supported by the experimental results shown in Figure 6.4 and 6.5.

From this point of view, as the number of control points increases with the stroke count, the time for rasterization would increase with stroke count. We observe that the outline characters in the 5 typefaces have roughly equal number of control points, therefore, their times for rasterization would be more or less the same.

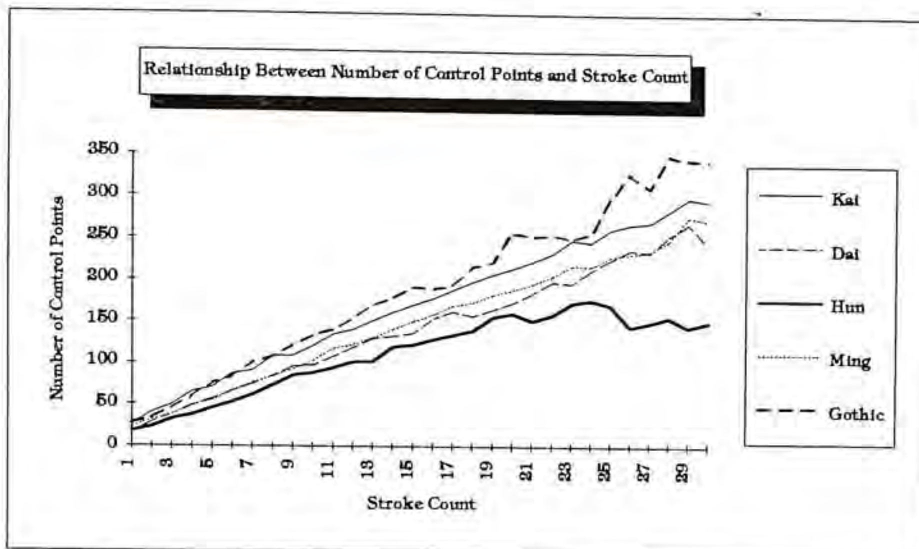
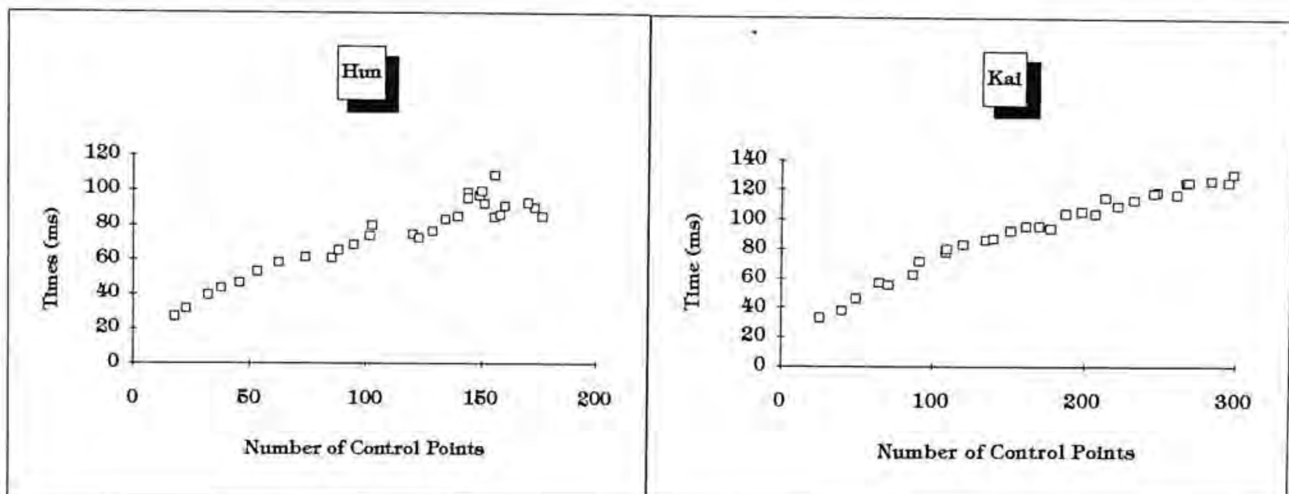
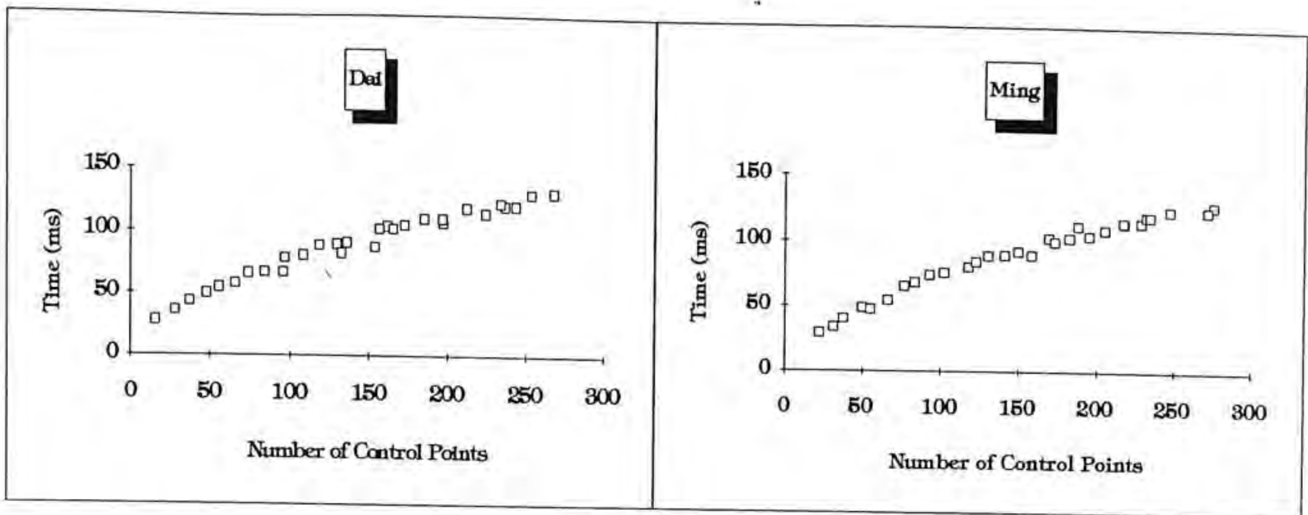


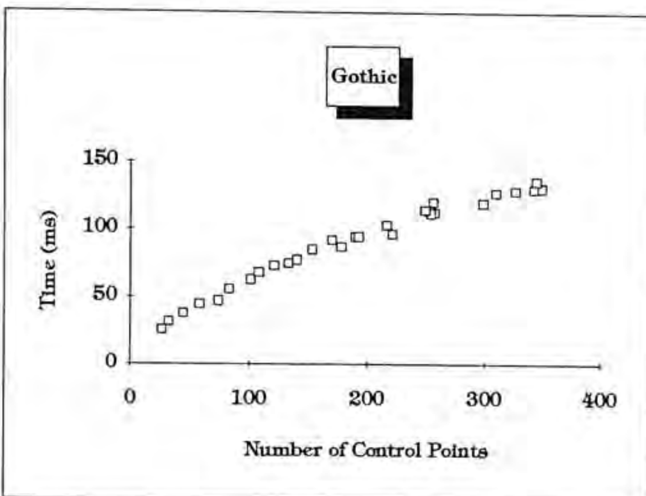
Figure 6.3: Plot showing the relationship between number of control points and stroke count



(a) (b)



(c) (d)



(e)

Figure 6.4: Relationships between execution times and number of control points

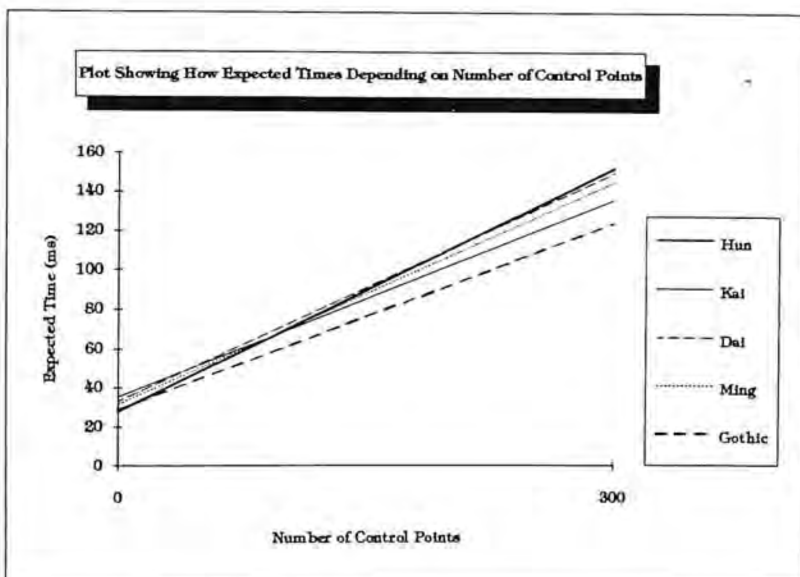


Figure 6.5: Fitted lines of times against number of control points

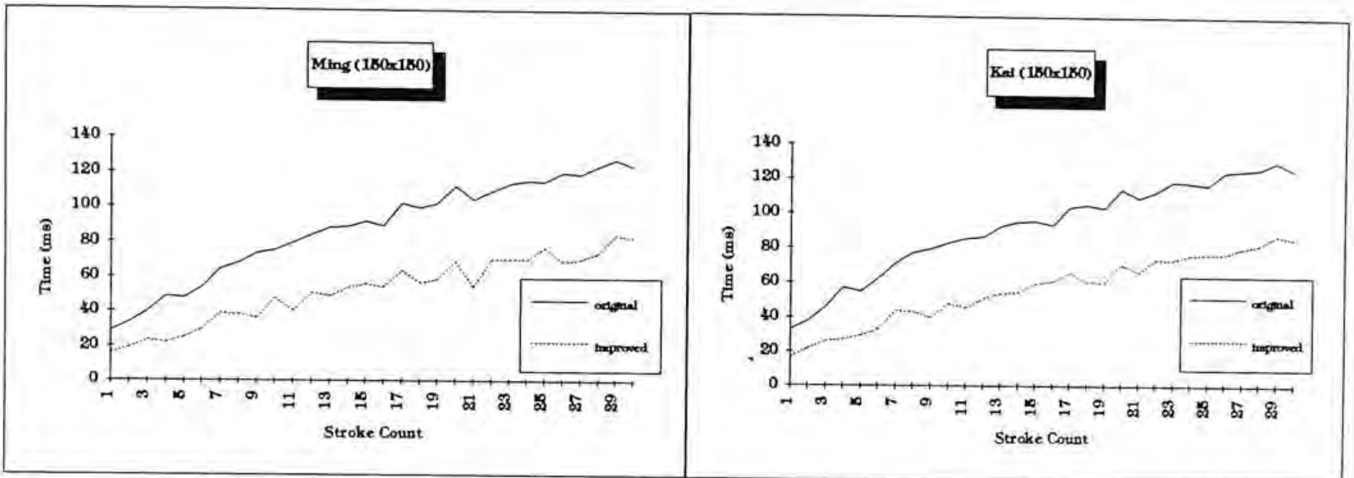
6.2 Improvement of Improved Rasterizer

In this section, we present the results of comparing the performance of two rasterizers: Rasterizer 1 implements the traditional rasterization algorithm and Rasterizer 2 implements the improved

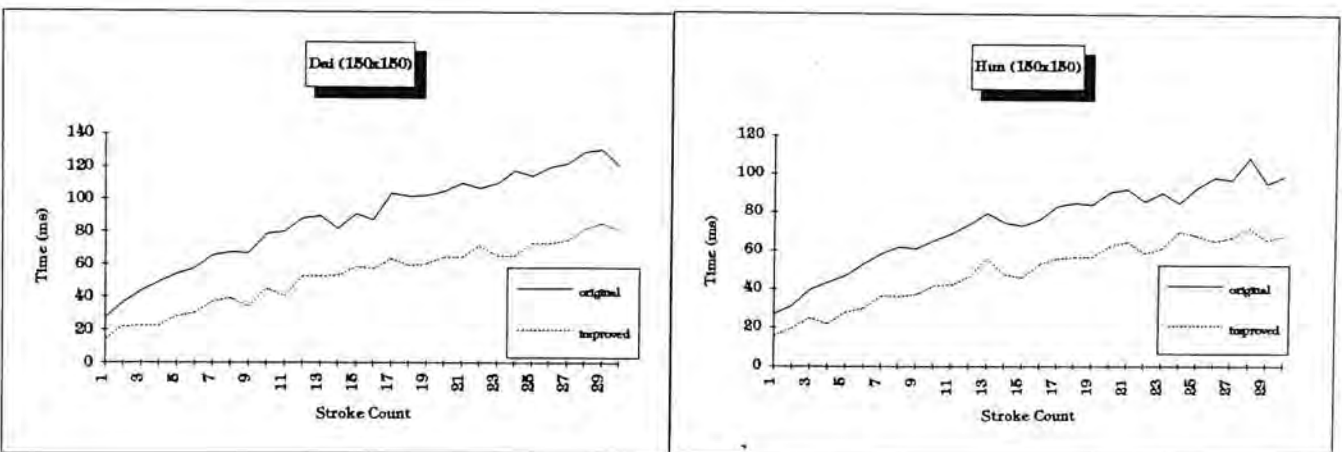
rasterization techniques. Our objective is to study the improvement of rasterizer 2 over rasterizer 1, where improvement is determined by the formula:

$$improvement = 1 - \frac{time_of_rasterizer_1}{time_of_rasterizer_2} \times 100\%$$

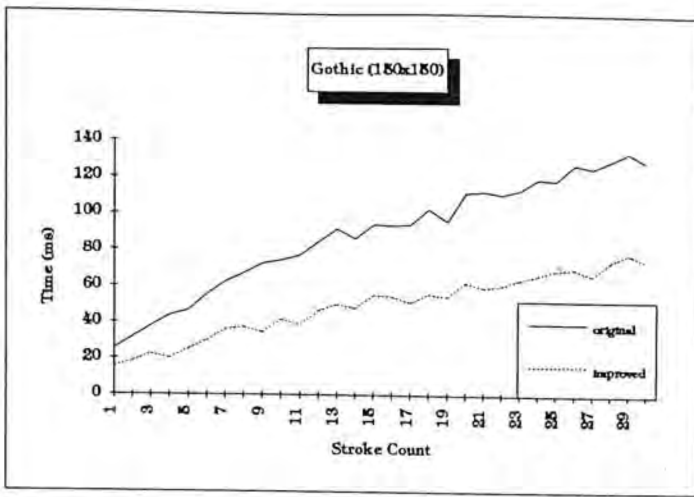
Figure 6.6 plots the execution time against the number of strokes of the characters for the 2 rasterizers for the 5 style and we observe that Rasterizer 2 is much faster than Rasterizer 1. Figure 6.7 plots the percentage of time improvement of rasterizer 2 over rasterizer 1 against the number of strokes. In all cases, we can observe that the improvements are over 30%. This implies that Rasterizer 2 can significantly improve over Rasterizer 1.



(a) (b)

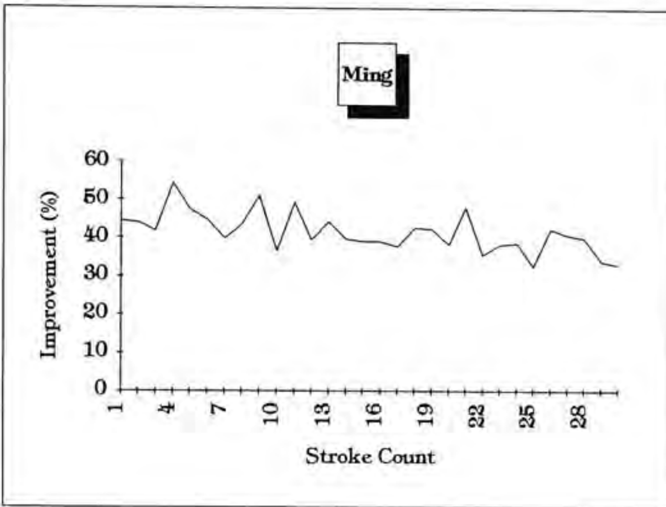


(c) (d)

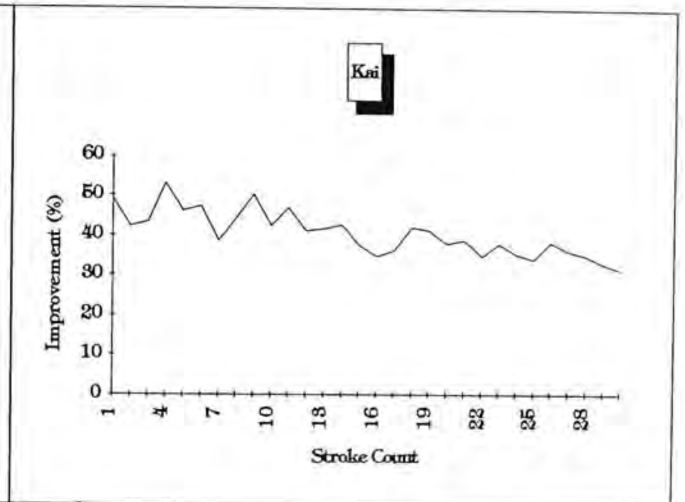


(e)

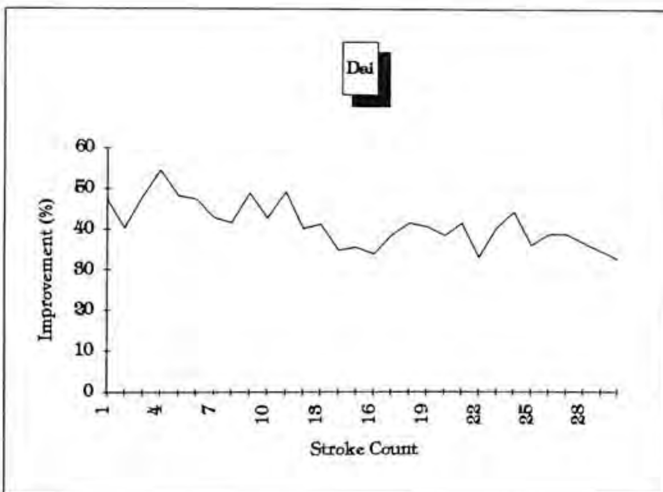
Figure 6.6: Series of graphs showing the relationship between execution time and number of character strokes for the two rasterizers.



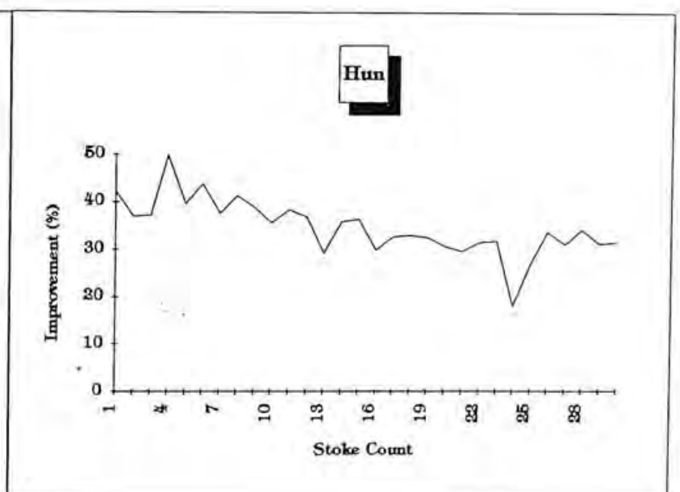
(a)



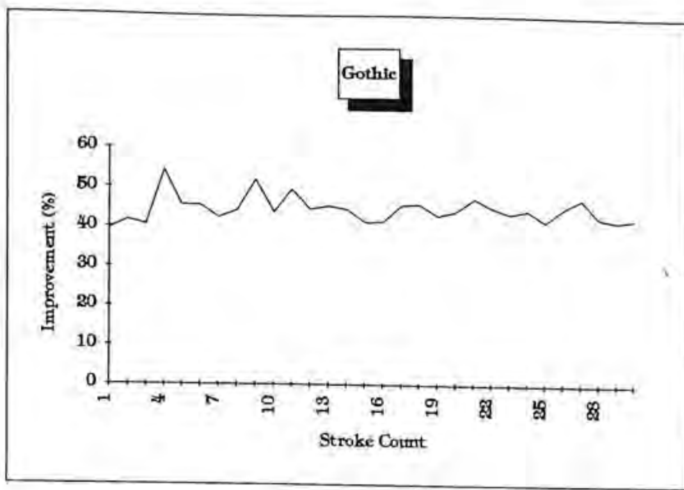
(b)



(c)



(d)



(e)

Figure 6.7: graphs showing the relationship of % of time reduction and number of strokes for the 5 typefaces and pixel sizes of the output bitmaps.

Other than the experimental results presented above, we have also performed some experiments for generating bitmaps of various sizes, such as bitmap characters of size 50x50, 100x100 and 200x200. From our results, the larger the pixel size of the bitmap is, the better the improvement would be. For instance, Figure 6.8 shows how the rasterization time varies with size of the bitmap character 籲 for the original rasterization process with and without hintings, the fast rasterization process with and without hintings. It shows that the fast algorithm requires relatively less time for generating larger bitmap characters. This fact is in accord with our expectation because, when the area to be filled is larger, the filling algorithm would be more efficient with fully utilizing byte filling.

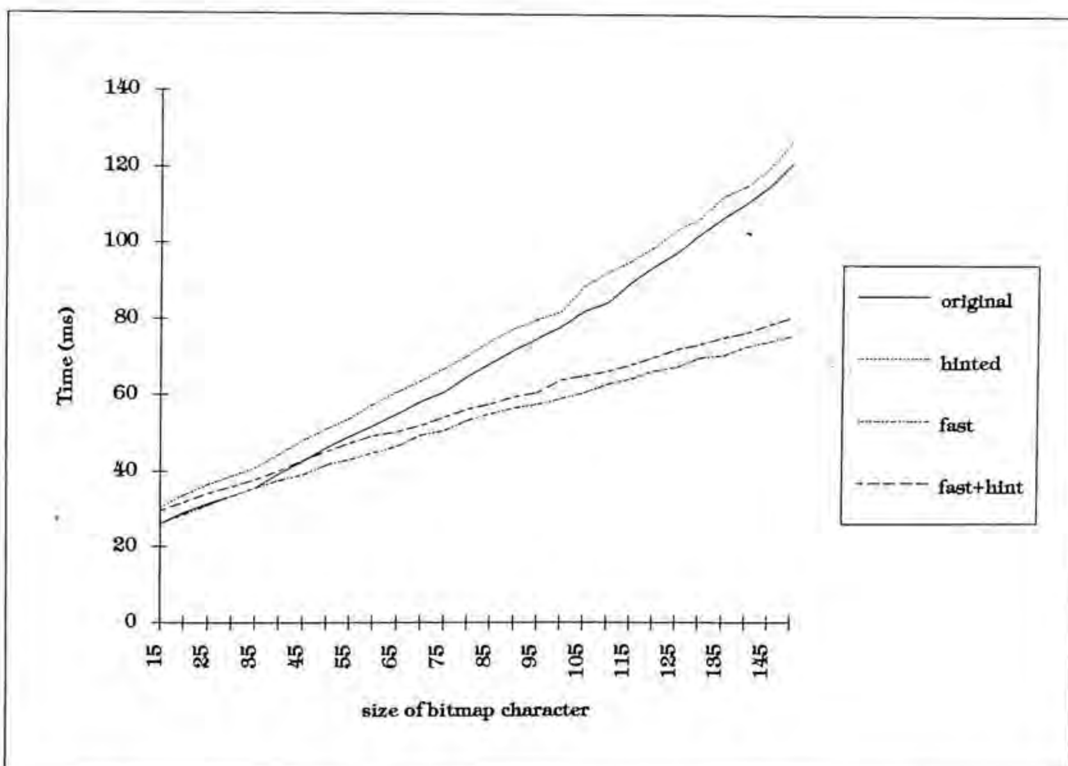


Figure 6.8: Graph shows how the rasterization times varies with size of bitmap character 籲 generated by different rasterization processes.

6.3 Gain and Cost of Inserting Hints into Font File

This section summarizes the results obtained from the experiment for comparing the font file with hints with the one without hints described in section 5.3 in Chapter 5.

6.3.1 Cost

The first row of Table 6.3 shows the sizes of the two font files. The second row shows the time for rasterizing 100 randomly selected characters of size 32x32 from the font file without hints and that from the font file with hints. We can see that the increases in file size and execution time for generating 32x32 bitmap of the 100 characters are reasonable, that is the cost is acceptable.

	without hints	with hints	% increase
file size	40862 bytes	50297 bytes	+23%
execution time for 32x32 bitmap	403.5 ms	422.1 ms	+5%

Table 6.3: Comparison of file size for 100 randomly selected characters.

6.3.2 Gain

Figure 6.9a shows the 100 characters generated from file without hints and Figure 6.9b shows the corresponding characters produced from file with hints. Obviously, the hinted characters look much better than those generated without hintings. Figure 6.10 shows two characters generated from Microsoft Windows and our algorithm respectively for different sizes. We can observe that our control of the spaces among strokes is better.

戶刊央旦札禾吏字旭汗即局沛洹災牢甬巡刮協
帚昆炊肫肯芰咦奕幽毒迴面倥倘卿悔捆挽疲盎
蚊問排眾羞莩訢陷喂單喬圍幀愕愎港焜痢詐募
廈搔滂溫煙綏虜叜鈸雉馴慣斡榜褚領戮敵潔蔓
操濂澧諺踱骼鴕簌韓擷瞿儳櫥櫝瀟瘺蹺僟躋驃

(a)

戶刊央旦札禾吏字旭汗即局沛洹災牢甬巡刮協
帚昆炊肫肯芰咦奕幽毒迴面倥倘卿悔捆挽疲盎
蚊問排眾羞莩訢陷喂單喬圍幀愕愎港焜痢詐募
廈搔滂溫煙綏虜叜鈸雉馴慣斡榜褚領戮敵潔蔓
操濂澧諺踱骼鴕簌韓擷瞿儳櫥櫝瀟瘺蹺僟躋驃

(b)

Figure 6.9: (a) 100 randomly selected bitmap characters (32x32) are generated without hintings; (b) the corresponding characters are generated with hintings.

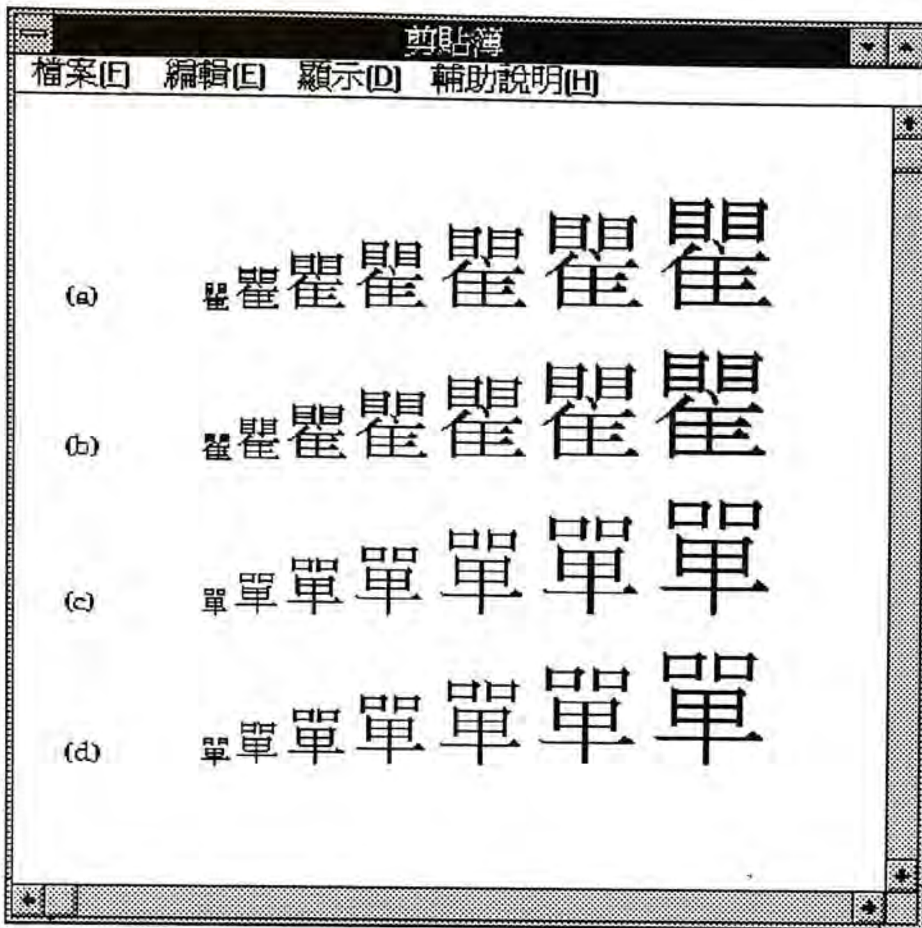


Figure 6.10: Comparison of bitmaps generated from Microsoft Chinese Windows 3.1 (a, c) and from our hinted font file (b, d).

6.4 Conclusions

In conclusion, we have suggested some techniques which make improvement on the scan conversion and contour filling steps of the rasterization process for Chinese outline fonts. Our experimental results show that these techniques can speed up the rasterization process. Also, we demonstrate an auto-hinting approach for Chinese font with relatively straight horizontal strokes, such as Ming style and Gothic style. With hints added into the font file as part of font data, the rasterizer can create much more beautiful bitmap characters, not costing too much in terms of space and time. In addition, based on our experimental results, we observe an interesting relationship: the time taken by the rasterization process is linearly dependent on the stroke count of the character to be generated but quite independent of in what style the character is.

6.5 Future Work

Storing font data in bitmap format, we have to keep one font data copy for each of the various styles in many distinct sizes. Storing font data in outline font format, we have to keep one font data copy for each of the various styles. This trend may imply that there exist a format in which only one copy of font file would be required for generating fonts in different sizes in different styles.

By studying the skeleton of a character, the Chinese can write a character in different styles with adequately amending the skeleton and directing the movement of the brush-tip of the writing

brush. Similarly, we can store Chinese characters with their skeletons and fonts in distinct styles can be produced by appropriately altering the skeleton and some parameters which control the appearance of the stroke applied to each stroke. Figure 6.11 shows the idea of transforming a character skeleton into fonts in Gothic and Ming styles.

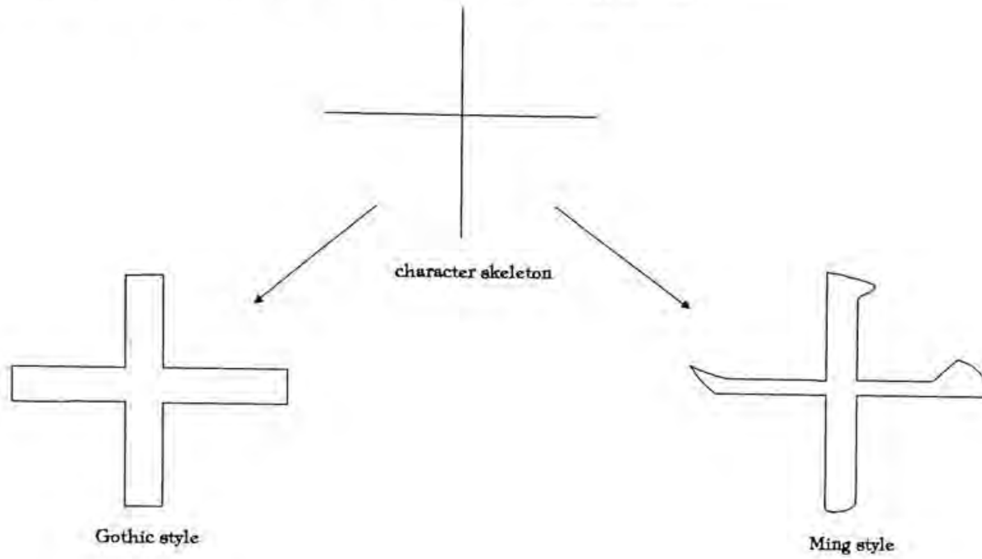


Figure 6.11: Generation of Chinese fonts from character skeleton.

Some problems of implementing this approach includes: how can we convert the skeletons into strokes? How should we adjust the parameters of the model and the character skeleton so that we can generate strokes in exactly the same shape of strokes in a common style (like Ming style, Dai style and Kai style)?

Appendix

Our simple random sample of size 100 is selected using the following random digits table. Since the population size is 5401, we read 4 adjacent digits from the table to get a number each time, skipping any number which is greater than 5401, until 100 *distinct* numbers have been drawn.

To choose which line to start, we draw a random number between 0 and 19 by a calculator and we obtain 7. Therefore, beginning with line 7, the randomly selected numbers are 1778, 3000, 1510, 8068 (skipped), 3091, ...

ONE THOUSAND RANDOM DIGITS

	00–04	05–09	10–14	15–19	20–24	25–29	30–34	35–39	40–44	45–49
00	54463	22662	65905	70639	79365	67382	29085	69831	47058	08186
01	15389	85205	18850	39226	42249	90669	96325	23248	60933	26927
02	85941	40756	82414	02015	13858	78030	16269	65978	01385	15345
03	61149	69440	11286	88218	58925	03638	52862	62733	33451	77455
04	05219	81619	10651	67079	92511	59888	84502	72095	83463	75577
05	41417	98326	87719	92294	46614	50948	64886	20002	97365	30976
06	28357	94070	20652	35774	16249	75019	21145	05217	47286	76305
07	17783	00015	10806	83091	91530	36466	39981	62481	49177	75779
08	40950	84820	29881	85966	62800	70326	84740	62660	77379	90279
09	82995	64157	66164	41180	10089	41757	78258	96488	88629	37231
10	96754	17676	55659	44105	47361	34833	86679	23930	53249	27083
11	34357	88040	53364	71726	45690	66334	60332	22554	90600	71113
12	06318	37403	49927	57715	50423	67372	63116	48888	21505	80182
13	62111	52820	07243	79931	89292	84767	85693	73947	22278	11551
14	47534	09243	67879	00544	23410	12740	02540	54440	32949	13491
15	98614	75993	84460	62846	59844	14922	48730	73443	48167	34770
16	24856	03648	44898	09351	98795	18644	39765	71058	90368	44104
17	96887	12479	80621	66223	86085	78285	02432	53342	42846	94771
18	90801	21472	42815	77408	37390	76766	52615	32141	30268	18106
19	55165	77312	83666	36028	28420	70219	81369	41943	47366	41067

References:

[Abe, Yamamoto & Ohno 91] Hiroshi Abe, Yoshimichi Yamamoto, Yoshio Ohno; *"High Quality Gray-scale Kanji Font Generation Using Automatic Stroke Displacement"*; In *Raster Imaging and Digital Typography II*; Cambridge University Press, 1991.

[Ackland 81] B. D. Ackland and N. H. Weste; *"The Edge Flag Algorithm - A Fill Method for Raster Scan Displays"*; In *IEEE Trans. on Computers*, vol. 30, no. 1, January 1981.

[Adobe 90] Adobe Systems Inc.; *PostScript Language Reference Manual, 2nd Edition*; Addison-Wesley, 1990.

[Andler 90] Sten F. Andler; *"Automatic Generation of Gridfitting Hints for Rasterization of Outline Fonts or Graphics"*; In *EP90*, 1990.

[Betrisey & Hersch 89] Claude Betrisey and Roger D. Hersch; *"Flexible application of outline grid constraints"*; In *Raster Imaging and Digital Typography*, Cambridge University Press, 1989.

[Betrisey & Hersch 91a] Claude Betrisey and Roger D. Hersch; *"Model-based Matching and Hinting of Fonts"*; In *Computer Graphics*, vol. 25, no. 4, July 1991.

[Betrisey & Hersch 91b] Claude Betrisey and Roger D. Hersch; *"Advanced Grid Constraints: Performances and Limitations"*; In *Raster Imaging and Digital Typography II*, Cambridge University Press, 1991.

[Bhat & Johson 77] Gouri K. Bhattacharyya & Richard A. Johnson; *Statistical Concepts and Method*; Wiley, 1977.

[Cao & Suen 87] X. Cao and C. Y. Suen; *"A New Phonetic and Ideographic Coding Technique for Chinese Information Processing"*; In *Computer Processing of Chinese & Oriental Languages*, vol. 3, no. 2, December 1987.

[Cheang 90] Cheang Sio Man; *"Chinese Windows System with Distributed Fonts"*; *Thesis (M. Phil.)*, the Chinese University of Hong Kong, 1990.

[Chen, Li & Chang 88] Keh-Jiann Chen, Kuo-Chun Li and Yeong-Long Chang; *"A System for On-Line Recognition of Chinese Characters"*; In *Computer Processing of Chinese & Oriental Languages, An International Journal of the Chinese Language Computer Society*, vol. 3, no. 3 & 4, March 1988.

[Chou & Tsai 91] Sheng-Lin Chou and Wen-Hsiang Tsai; *"Recognizing Handwritten Chinese Characters by Stroke-Segment Matching Using an Iteration Scheme"*; In *Character & Handwriting Recognition*, World Scientific Publishing Co. Pte. Ltd, 1991.

[Falhander 89] Olvo Fahlander; *"A spline contour method with efficient filling"*; In *Raster Imaging and Digital Typography*, Cambridge University Press, 1989.

[Foley & van Dam 90] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes; *Computer Graphics, Principles and Practice, 2nd Edition*; Addison Wesley, 1990.

[Gonczarowski 89] Jakob Gonczarowski; *"Fast Generation of Unfilled and Filled Outline Characters"*; In *Raster Imaging and Digital Typography*, Cambridge University Press, 1989.

[Gonczarowski 91] Jakob Gonczarowski; *"A Fast Approach to Auto-tracing (with Parameteric Cubics)"*; In *Raster Imaging and Digital Typography II*, Cambridge University Press, 1991.

[Hersch 87] Roger D. Hersch; *"Character Generation Under Grid Constraints"*; In *Computer Graphics*, vol. 21, no. 4, July 1987.

[Hersch 88] Roger D. Hersch; *"Outline Phase Control for Character Rasterization"*; In *Eurographics '88*, 1988.

[Hersch 89] Roger D. Hersch; *"Introduction to font rasterzation"*; In *Raster Imaging and Digital Typography*, Cambridge University Press, 1989.

[Holzgang 92] David A. Holzgang; *Understanding PostScript, 3rd Edition*; Sybex Inc., 1992.

[Hsu & Cheng 85] Wen-Hsing Hsu and Fang-Hsuan Cheng; *"Recognition of Handwritten Chinese Characters by Structural Analysis of Strokes"*; In *Computer Processing of Chinese & Oriental Languages*, vol. 2, no. 2, October 1985.

[Karow 89] Peter Karow; *"Automatic hinting for intelligent font scaling"*; In *Raster Imaging and Digital Typography*, Cambridge University Press, 1989.

[Knuth 86] Donald E. Knuth; *The METAFONT Book*, Addison-Wesley, 1986.

[Kohen 89] Eliyezer Kohen; *"A Simple And Efficient Way to Design Middle Resolution Fonts"*; In *Raster Imaging and Digital Typography*, Cambridge University Press, 1989.

[Lee 92] 李明清; *中文系統徹底研究輸入法與秀字*; 旗標出版社, 1992.

[Liao & Huang 91] Chia-Wei Liao and Jun S. Huang; *"Font Generation by Beta-Spline Curve"*; In *Computer & Graphics*, Vol. 15, No. 4, 1991.

[Liu 87] 廖明德; *倚天中文系統*; 倚天資訊有限公司, 1987.

[Lus 90] K. T. Lua; *"Analysis of Chinese Character Stroke Sequences"*; In *Computer Processing of Chinese & Oriental Languages*, vol. 4, no. 4, March 1990.

[Maag 89] Bruno Maag; *"Shape investigations with bitmapped characters"*; In *Raster Imaging and Digital Typography*, Cambridge University Press, 1989.

[Moon & Cheang 91] Y. S. Moon and S. M. Cheang; *"Deficiencies of PostScript in Displaying/Printing Chinese Fonts"*; In *Communications of COLIPS*, vol. 1, no. 1, 1991.

[Moon & Hui 89] Y. S. Moon and W. K. Hui; *"High Quality Chinese Fonts Generation for Desktop Publishing - A Computer Vision Approach"*; In *Pattern Recognition Letters* 9, 1989.

[Moon & Shin 90] Y. S. Moon and T. Y. Shin; *"Chinese Fonts and their Digitization"*; In *EP90*, 1990.

[Morishita, Ooura & Ishii 88] Tetsuji Morishita, Masahiko Ooura and Yasuo Ishii; *"A Kanji Recognition Method Which Detects Writing Errors"*; In *Computer Processing of Chinese & Oriental Languages, An International Journal of the Chinese Language Computer Society*, vol. 3, no. 3 & 4, March 1988.

[New Image 92] *中國文字造形設計*; 新形象出版事業有限公司, 1992.

[Ou & Ohno 89] Chialing Ou and Yoshio Ohno; *"Font Generation Algorithms for Kanji Characters"*; In *Raster Imaging and Digital Typography*, Cambridge University Press, 1989.

[Pavlidis 79] Theo Pavlidis; *"Filling Algorithms for Raster Graphics"*; *Computer Graphics Image Proc.* 10, 1979.

[Pavlidis 81] Theo Pavlidis; *"Contour Filling in Raster Graphics"*; In *Computer Graphics*, vol. 15, no. 3, 1981.

[Pavlidis 85] Theo Pavlidis; *"Scan Conversion of Regions Bounded by Parabolic Splines"*; In *IEEE Computer Graphics and Applications*, vol 5, no. 6, June 1985.

[Rogers 85] David F. Rogers; *Procedure Elements for Computer Graphics*; McGraw-Hill Book Company, 1985.

[Rosenberg 91] Charles Rosenberg; *"A Low Complexity Method for Compressing Kanji Font Bitmaps"*; In *Raster Imaging and Digital Typography II*, Cambridge University Press, 1991.

[Rubinstein 88] Richard Rubinstein; *Digital Typography - An Introduction to Type and Comparison for Computer System Design*; Addison-Wesley, 1988.

[Seybold 92] *"Outline Font Hints and Rasterization: A Technology Primer"*; In *The Seybold Report on Desktop Publishing*, vol. 6, no. 7, 1992.

[TrueType 90] T*rueType Font Files*; Microsoft Corporation, 1992.

[Warnock 80] John E. Warnock; *"The display of characters using gray level sample arrays"*; In *Computer Graphics*, vol. 14, no. 3, July 1980.

[Yang 86] Jiben Yang; *"A Psychological View on the Standardization of the Structural Elements of Chinese Characters in Information Encoding"*; In *Computer Processing of Chinese & Oriental Languages*, vol. 2, no. 1, May 1986.

CUHK Libraries
000249446