Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Vector Graphics Stylized Stroke Fonts

Examensarbete utfört i informationskodning
vid Tekniska högskolan i Linköping
av

Philip Jägenstedt

LITH-ISY-EX--08/4066--SE

Linköping 2008

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Vector Graphics Stylized Stroke Fonts

Examensarbete utfört i informationskodning
vid Tekniska högskolan i Linköping
av

Philip Jägenstedt

LITH-ISY-EX--08/4066--SE

Handledare: **Tim Johansson**
Opera Software ASA, Linköping

Examinator: **Ingemar Ragnemalm**
ISY, Linköpings universitet

Linköping, 4 June, 2008

Avdelning, Institution
Division, Department

Division of Information Coding
Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Datum
Date

2008-06-04

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English
☐ ______________

Rapporttyp
Report category

☐ Licentiatavhandling
☒ Examensarbete
☐ C-uppsats
☐ D-uppsats
☐ Övrig rapport
☐ ______________

ISBN
—

ISRN
LITH-ISY-EX--08/4066--SE

Serietitel och serienummer
Title of series, numbering

ISSN
—

URL för elektronisk version

http://www.icg.isy.liu.se/
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-12095

Titel
Title Vector Graphics Stylized Stroke Fonts

Författare
Author Philip Jägenstedt

Sammanfattning
Abstract

Stylized Stroke Fonts (SSF) are stroke fonts which allow variable stroke widths and arbitrary stroke ends. In this thesis project we implement SSF by extending concepts of the traditional vector graphics paradigm, giving what we call *Vector Graphics Stylized Stroke Fonts* (VGSSF). A stroking algorithm for the new stroke model is developed and implemented in the Opera web browser's internal vector graphics drawing toolkit. Both the HTML 5 Canvas JavaScript interface and SVG fonts are extended to support the new stroke model. An editor and renderer for the SVG font format is implemented inside the browser using the extended Canvas interface. Sample glyphs from Latin and Chinese typefaces are converted to the SVG font format to assess the suitability of the stroke representation. The results are excellent for Chinese Ming typeface glyphs, while there are some minor problems with Latin typeface glyphs. Approximations suggest that VGSSF gives a size reduction of the font definition file by at least 50%, with a potential reduction of around 85% for Chinese typefaces. The processing requirements increase by approximately 20–30% due to extra steps required to render each glyph.

Nyckelord
Keywords vector graphics, stylized stroke fonts

Abstract

Stylized Stroke Fonts (SSF) are stroke fonts which allow variable stroke widths and arbitrary stroke ends. In this thesis project we implement SSF by extending concepts of the traditional vector graphics paradigm, giving what we call *Vector Graphics Stylized Stroke Fonts* (VGSSF). A stroking algorithm for the new stroke model is developed and implemented in the Opera web browser's internal vector graphics drawing toolkit. Both the HTML 5 Canvas JavaScript interface and SVG fonts are extended to support the new stroke model. An editor and renderer for the SVG font format is implemented inside the browser using the extended Canvas interface. Sample glyphs from Latin and Chinese typefaces are converted to the SVG font format to assess the suitability of the stroke representation. The results are excellent for Chinese Ming typeface glyphs, while there are some minor problems with Latin typeface glyphs. Approximations suggest that VGSSF gives a size reduction of the font definition file by at least 50%, with a potential reduction of around 85% for Chinese typefaces. The processing requirements increase by approximately 20–30% due to extra steps required to render each glyph.

Contents

Chapter 1

Introduction

This thesis project implements and evaluates *Vector Graphics Stylized Stroke Fonts* (VGSSF) as a part of the the Opera web browser. Looking at desktop computing, it might seem that problems of font formats and font rendering technology have been solved since long. The currently most common format, TrueType, has been used since the early 1990's with high quality display both on-screen and in print. However, new font technology is continuously developed for niche markets and niche purposes. With the growth of the mobile web, reading large amount of text on low-memory, low-resolution devices is becoming more common. Mobile phones typically have very little memory, so using the same font formats as on desktop computers is not always possible. This is especially true of East Asian fonts with a huge number of characters – a typical Chinese TrueType font file can be 10–20 MB in size. These limitations have driven the development of stroke fonts, a type of font with a more compact representation especially well suited for East Asian typefaces. However, most stroke font formats are limited in their appearance and not so suitable for rich East Asian typefaces or Latin typefaces. In this thesis we look at type of stroke font which allows for a higher level of control over the glyph appearance.

New font formats are interesting not only for mobile devices, but potentially also for *web fonts*. The concept is over 10 years old, but it is getting renewed attention. In Cascading Style Sheets Level 2 (CSS 2) the `@font-face` declaration[23] allows web page authors to specify a download location for fonts which are not installed on the client system. This gives the page author greater control over page design and allows the display of text in rare scripts without requiring the user to install a new system font. However, since virtually no browser vendors supported it, it was dropped in CSS 2.1. In the emerging CSS 3 standard the `@font-face` declaration is once again included,[24] showing that the concept has not been forgotten. An important consideration is that web fonts – like all web standards – ought to use an open standard that can be implemented by any vendor free of charge. However, important parts of TrueType are encumbered by software patents and cannot be fully implemented. Thus, the font format developed in this thesis could possibly be used as a basis for web fonts in the future. As the web is device-independent, web fonts should be feasible to use also on low-memory devices such as mobile phones, the implications of which have already been mentioned.

Before the objectives of the thesis can be formulated, we will have a look at the history of printing and fonts. This is necessary to understand how we have come to where we are today and how digital fonts and typography are related to the technology that existed before the modern computer.

Chapter 2

From Woodblocks to Bézier: Printing and Fonts

Printing technology predates the computer by over a millennia. Woodblock printing of text was first used to print Buddhist scriptures in China as early as the 7th century.[22] The earliest known movable type system – also from China – was invented in the 11th century and used types made from baked clay. Wooden and metal movable type was developed during the 13th century in China and Korea.[18]

More than 200 years later, around 1450, Johannes Gutenberg independently invented movable type printing, which revolutionized printing in Europe and eventually across the world.[19] With movable type printing came uniform character sets with distinct type faces – fonts. The influence of movable type printing on digital typography and fonts is evident to this day.

2.1 Anatomy of Fonts

In order to reason about fonts we need to introduce a few core concepts. While these are very similar in traditional printing and in the digital era, the focus here is on modern use of the terms.

A *glyph* is a graphical representation of a particular symbol such as a character of the Latin alphabet or an Arabic numeral. The glyph is defined relative to an *em square*, which corresponds to the height of the metal type used to set text in movable type printing. The *advance* is the distance from the glyph origin to the origin of the following glyph. This is usually slightly longer than the visual width of the glyph. The *baseline* is used to vertically align adjacent glyphs so that they all appear on a single line.

A *typeface* is a collection of glyphs of a particular style. For example, Times New Roman is a Latin alphabet typeface common in printed material.

A *font* is a collection of related typefaces. For example, a Times New Roman computer font may include bold and italic typefaces in addition to the standard typeface.

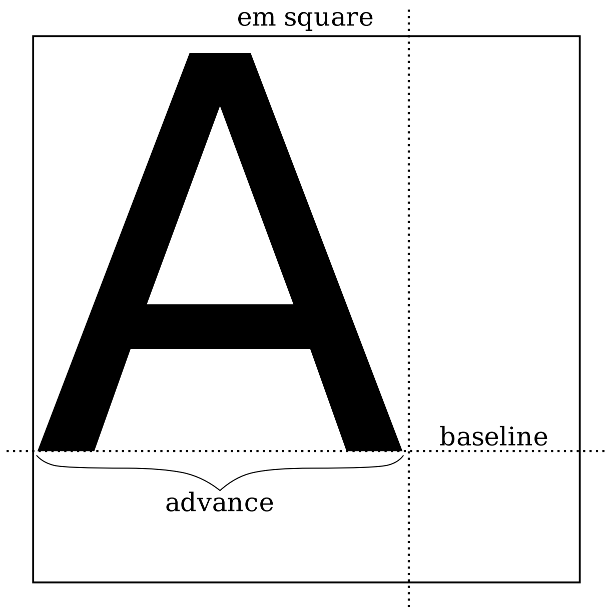


Figure 2.1. A glyph and its metrics

This description is very simplified and most writing systems require additional metrics. For example, *ascent* and *descent* gives the distance from the baseline to the top and bottom of the glyph respectively. Right-to-left scripts and vertical scripts also require additional metrics. Still, the simple description given above is enough for the purposes of this thesis.

2.2 Reference Typefaces

A few reference typefaces are used as examples and test cases in this thesis. The intention is to consider typefaces with a varying amount of complexity in line width and line ends. We believe that this selection of typefaces should give a good indication of the suitability of a new font format for real-world application.

2.2.1 Latin Typefaces

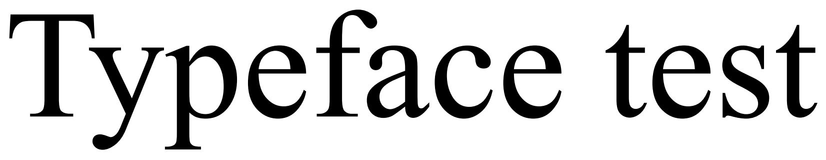


Figure 2.2. Serif typeface *Times New Roman*

Serif typefaces are common in print and are characterized by the triangular ornaments (serifs) at the ends of certain strokes. Vertical strokes are generally heavier (thicker) than horizontal strokes, while the width of curved strokes can vary significantly.

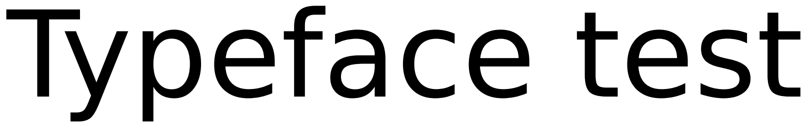


Figure 2.3. Sans serif typeface *Bitstream Vera Sans*

Sans serif means *without serif* and is a common typeface for on-screen display. It is relatively simple, as stroke width is fairly uniform and there are no complex ornaments at line ends.

2.2.2 Chinese Typefaces

中文字型範例

Figure 2.4. Ming typeface (明體) *Arphic PL Mingti2L* (文鼎PL細上海宋)

The Ming typeface (dating from the Song Dynasty, thus also known as Song) is common in Chinese print. Horizontal strokes are thinner than vertical strokes and have triangular ornaments on their right end. Compare with the Latin serif typeface.

中文字型範例

Figure 2.5. Heiti (黑體) *WenQuanYi Zen Hei* (文泉驛正黑)

Heiti is a Chinese Gothic typeface common in newspaper headlines, on billboards and similar. Stroke widths are uniform and there are no ornaments on stroke ends. Compare with the Latin sans serif typeface.

中文字型範例

Figure 2.6. Kaishu (楷書) *Arphic PL KaitiM* (文鼎PL中楷)

Kaishu (also known as regular script) is the most common Chinese calligraphy style. Writing it is considered an art form, where the proportions between and within strokes are crucial. It is arguably more complex than the Ming typeface as stroke widths are variable and decorations at stroke ends are non-uniform.

2.3 Digital Font Paradigms

In order to understand where we have come from and where we are going with this thesis, we take a short look at the history of digital fonts and how they are represented in different paradigms.

2.3.1 Stick Fonts

Early digital fonts in the 1960s were so called "stick" fonts used with pen-and-paper plotters.[29] Each glyph was defined by a number of straight lines segments approximating the characters appearance. Dr. Alan Hershey was a pioneer in digital fonts who wrote software to interpret text and font definitions from punch cards. By adding more lines, serifs and thick strokes could be achieved. Hershey went on to create many complex typefaces, including a Japanese font.[1] An example can be seen in figure 2.7.

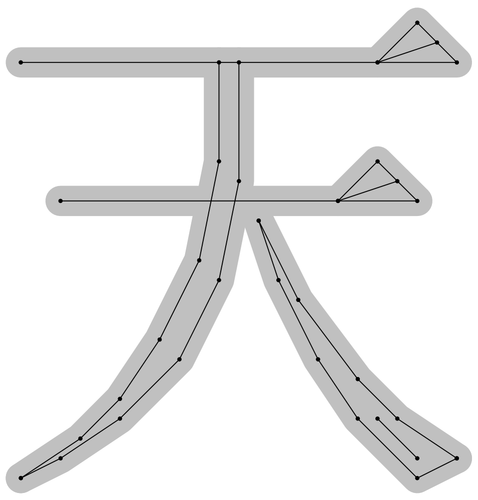


Figure 2.7. Stick font representation of Chinese character 天 (heaven). This is taken from Hershey's original Japanese font definition. The gray area illustrates what the character might look like if drawn with a plotter device on paper.

Stick fonts are in fact a type of vector font as they can be scaled to any size. Similar fonts were used in early vector-based video game systems, with the difference that they were drawn on a vector monitor (somewhat similar to an oscilloscope) instead of with a plotter. The most famous example is probably Atari's *Asteroids* from 1979.[17]

[1]Hershey's original font definitions are included with GNU Plotutils, see `http://www.gnu.org/software/plotutils/manual/html_node/plotutils_66.html`

2.3.2 Bitmap Fonts

Around 1970 Tektronix introduced their "storage tube" technology and the 4010 series of terminals.[29, 21] Unlike earlier monitors, they didn't require constant redrawing of the entire screen, thereby dramatically decreasing the cost of hardware needed to operate it. Fonts could be stored within the terminal as bit-by-bit maps of which pixels should be illuminated – thus the name *bitmap font*. This type of font became very widespread and is still in use today. An example of a bitmap glyph can be seen in figure 2.8.

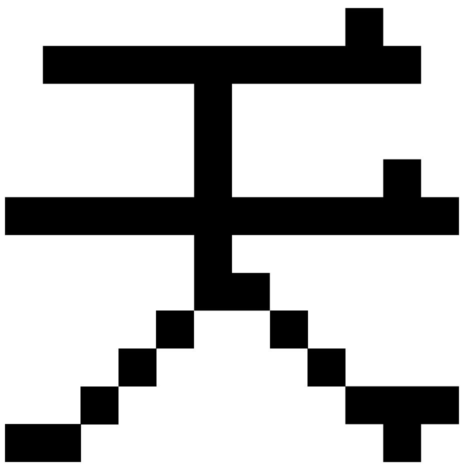


Figure 2.8. Pixel font representation of Chinese character 天

Bitmap fonts have some serious limitations. Every glyph of every font of every size must be represented as a raster image. Creating a complete font with an acceptable selection of sizes is time consuming and the resulting font definition still has some issues. Above all bitmap fonts do not scale well – reproducing them at too large a size gives a blocky appearance. This problem is inherent to raster graphics and makes bitmap fonts unsuitable for print.

2.3.3 Outline Fonts

At around 1977 a software company called ISSCO introduced "shaded fonts".[29] They were defined by an outline path made up of lines and arc segments which was "shaded" (filled). This was the first *outline font*.

John Warnock worked at Xerox PARC during 1978-1982 with a language called InterPress for use in Xerox printers. In 1982 he left together with Chuck Geschke to found Adobe and develop a similar language called PostScript. Adobe partnered with Apple to adapt PostScript to Apple's LaserWriter in 1985.[20] PostScript has since become the standard language for printers and has a strong standing to this day. In PostScript, paths are described as sequences of lines, arcs and cubic Bézier curves[15] which can be filled or stroked. However, PostScript fonts can only be represented as (filled) outlines, meaning that they are pure outline fonts.

One of the key properties of outline fonts is that they are scalable and device-independent – the same font can be used for on-screen display and when printing. An example glyph can be seen in figure 2.9.

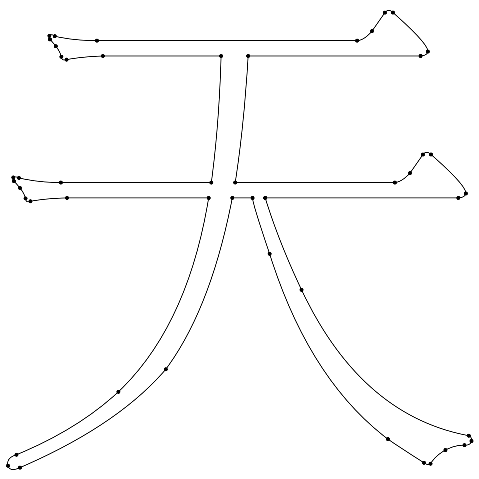


Figure 2.9. Outline font representation of Chinese character 天. Only on-line control points of Bézier curves are shown.

TrueType[3] is a later date outline font format which at the path level differs in that it uses the slightly simpler quadratic Bézier curve instead of cubic Bézier. It also differs in how rendering at small sizes – glyph hinting – is handled. This is of critical importance on limited-resolution devices and is discussed further in section 10.3.

2.3.4 Stroke Fonts

A somewhat different vector based font format (or font description language) is Donald Knuth's Metafont from 1984.[8] As with PostScript, paths are constructed from cubic Bézier curves. However, instead of filling an outline, glyphs are drawn by following the path with a pen. The path defines the centerline of the stroke, while the thickness and appearance of strokes are determined by the pen size and shape. Metafont was never widely adopted, but is interesting because it is in fact a type of *stroke font* (even though it is never called such).

Stroke-based fonts define glyphs by their stroke centerline paths instead of their outlines. It could be argued that stroke fonts provide a more natural representation of glyphs than outline fonts do, as each stroke of the pen (or similar writing tool) is explicit. Furthermore, as only the centerline of a stroke need be described the number of points required to describe a glyph can theoretically be reduced by more than 50%. However, line width and possibly other data is needed to actually stroke the path, which offsets some of these savings.

Recent developments in stroke fonts use a model more similar to the traditional vector graphics stroking model. Monotype Imaging and Bitstream have independently developed proprietary stroke font formats for use with their respective fonts engines.[13, 4] In both cases strokes are constant in width and the appearance of the line caps is limited. An example glyph using this font model can be seen in figure 2.10.

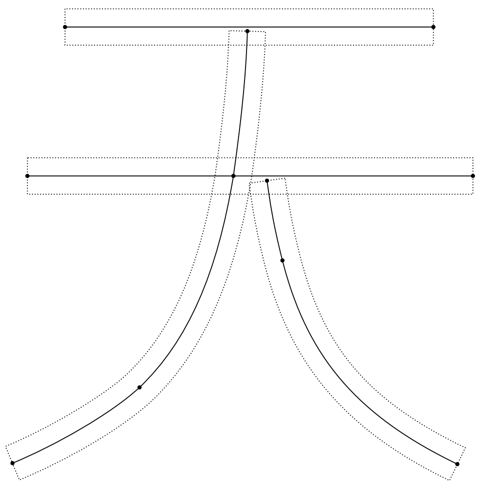


Figure 2.10. Stroke font representation of Chinese character 天. Only on-line control points of Bézier curves are shown.

The compact representation makes the font file smaller and the savings are more noticeable the larger the character set is. In fact, both Monotype Imaging and Bitstream specifically target East Asian fonts for use on memory-limited devices such as mobile phones. However, due to the constraints of the representation, stroke fonts are suitable mainly for typefaces such as Latin sans serif and Chinese heiti, as these have uniform stroke widths and no serifs or similar ornaments. At small display sizes this is acceptable, but the fonts lack the typographical detail to scale well beyond a certain point.

2.3.5 Stylized Stroke Fonts

Jakubiak et al. suggest improvements to the stroke-based model in what they call *stylized stroke fonts* (SSF).[7] In particular, two improvements make it possible to create richer and more detailed glyphs.

- Stroke widths are no longer constant, but allowed to vary along the path according to a *stroke profile.*
- The ends of a stroke are not uniform, but rather defined by a custom *stroke end.*

They demonstrate that by combining the same stroke path with different stroke profiles and stroke ends a variety of typefaces can be achieved. An example glyph in the SSF model can be seen in figure 2.11.

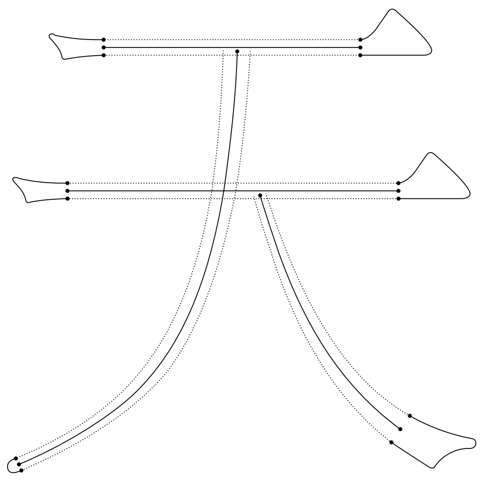


Figure 2.11. SSF representation of Chinese character 天

Few implementation details of SSF have been disclosed. Rendering is done using *adaptively sampled distance fields*,[11] a patented technology which has little to nothing in common with the traditional vector graphics paradigm. Still, the core concepts of SSFs are not bound to any specific glyph representation or rendering technology. This thesis work will start from SSF and see how it can be adapted to the traditional vector graphics paradigm.

Chapter 3

Objectives

The objective of this thesis work is to adapt the SSF model to a form which is suitable for implementation and use with modern web browser technology. By building on mature and well understood concepts and technologies, existing library support and developer know-how can be leveraged. While our implementation environment is the Opera browser, the the results should be applicable in a more general context as well. The intent is not to implement a production quality stroke font system, as that would require a robust, compact font format specification and font engine support, as well as at least one complete typeface. Rather, existing web technologies are reused or extended to create a prototype implementation of the core concepts, whereby the feasibility of this type of stroke font can be evaluated.

More specifically, we define *Vector Graphics Stylized Stroke Fonts* (VGSSF) as a variant of SSF using a path representation and stroking algorithm closer to those of PostScript, SVG and other vector graphics technologies. To support VGSSF in Opera, the following will be required:

- Changes to the underlying vector graphics rendering library to support the new graphics primitives required by the font model.
- Bindings to expose said new graphics primitives in the browser.
- A stroke font format specification.
- A renderer for said font format.

Additionally, a simple VGSSF editor will be developed. Although this is not necessary for supporting VGSSF in the browser, it can say something about the nature of stroke font representation itself.

Beyond simply implementing VGSSF, its feasibility as a format for widespread use will also be evaluated. A few questions are of special interest:

- How does rendering time and memory footprint compare to that of equivalent outline fonts?
- Is it possible to create typefaces which match or surpass the quality of outline typefaces?

- Is the font model general enough? To get some idea about this, both Latin and East Asian typefaces will be converted to the VGSSF format for comparison.
- How is editing affected by the stroke representation? What are the benefits and problems?

Apart from these questions, there will inevitably be other issues that come up during the course of implementation. While the main objective is a functional prototype, we will also try to identify what is additionally needed for a production quality VGSSF system.

Chapter 4

Traditional Stroke Model

This chapter describes the traditional path stroking model common to most vector graphics systems. Standard computer graphics textbooks such as Hearn and Baker's *Computer Graphics* include similar descriptions[6, Ch 4-5 Line Attributes] and common vector graphics technologies such as PostScript[2, Ch 4 Graphics] and SVG[27, Ch 11 Painting: Filling, Stroking and Marker Symbols] implement this very model. Furthermore, we consider a possible rendering algorithm, which will be adapted to support an extended stroke model in the following chapter.

In very general terms, to stroke a path is to draw that path with a certain width, somewhat similar to following it with a pen of a certain thickness. The width and appearance of the stroke at certain points along the path is controlled by the following parameters.

line width is a single width which applies to the whole path. In other words, only constant-width strokes are possible.

line cap is the shape at path ends. The shape can be one of three predefined line caps, shown in figure 4.1.

line join is the shape at corners of paths. The shape can be one of three predefined line joins, shown in figure 4.2. For miter joins, an extra **miter limit** parameter is used to set a limit on the length of a miter join relative to the line width. Miter joins which exceed that length are converted to bevel joins. This is necessary as the length of the miter joins approaches infinity when the angle of the join approaches zero.

What should be noted about this model is that it does not allow for a very high level of control of the strokes appearance. Should a content author want, for example, a line cap with the shape of an arrowhead, it would be necessary to instead create an outline path, as there is no predefined arrowhead line cap. SVG allows *marker symbols* by which arbitrary shapes can be rendered at start-, mid- and end-vertices of a path, which goes some way to provide a solution. This is discussed further in the following chapter.

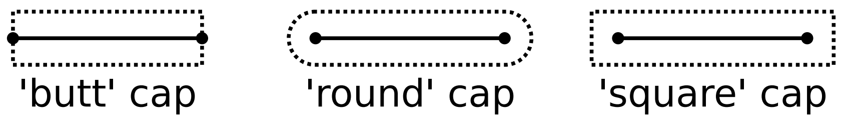


Figure 4.1. Predefined line caps in the traditional stroke model
Adapted from example at `http://www.w3.org/TR/SVG/painting.html`

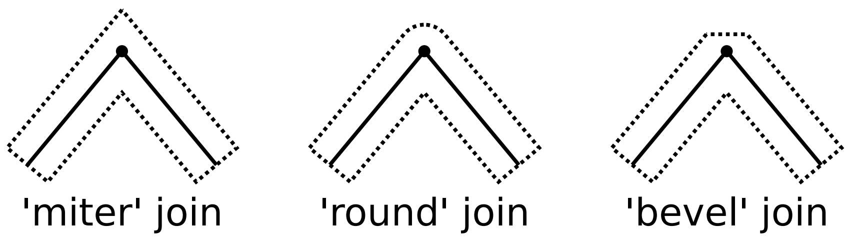


Figure 4.2. Predefined line joins in the traditional stroke model
Adapted from example at `http://www.w3.org/TR/SVG/painting.html`

4.1 Stroking Algorithm

An algorithm to stroke a path using these parameters can be fairly simple. The basic principle is to convert the stroke path into an outline path which is then filled, as illustrated in figure 4.3.

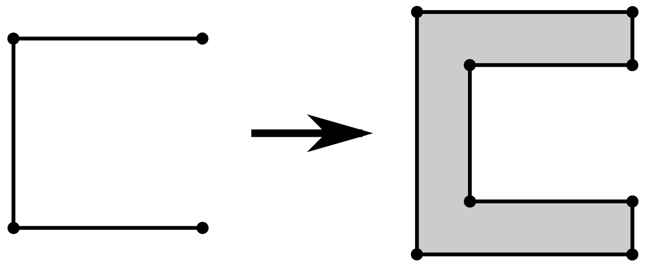


Figure 4.3. Stroking of path by conversion to outline path followed by filling

Calculating a mathematically exact outline path requires finding the parallel curves[16] (usually known as offset curves in computer graphics) at a given distance for each path component. While this is simple for lines and arcs, the case for Bézier curves is far more complicated.

A Bézier curve can be written as $C(t) = (f(t), g(t)), t \in [0, 1]$, where $f(t)$ and $g(t)$ are Bernstein polynomials of order 2 or 3 for cubic and quadratic Bézier curves respectively. The offset curve at a distance k from the base curve is defined by

$$x = f \pm \frac{kg'}{\sqrt{f'^2 + g'^2}}, y = g \mp \frac{kf'}{\sqrt{f'^2 + g'^2}}$$

The denominator $\sqrt{f'^2 + g'^2}$ can generally not be written as a polynomial, which implies that the offset curve is generally not defined by a rational function. A fair amount of research has been done on rational offset approximation of Bézier curves,[5] but such methods will not be worthwhile in a stroking algorithm. When filling the outline path it must first be converted to a polygon, so generating an offset polygon directly allows for a faster stroking algorithm.

Given an arbitrary path composed of lines, arcs and Bézier curves, it is necessary to first convert it into a path with only straight lines (a polyline). Algorithms to approximate arcs and Bézier curves are well known[1] and not of key concern in this thesis. Using such an approximation, the path can be seen as a polyline – a list of coordinates (x, y). Given a line width w, the stroke outline is the polygon which follows the polyline at offset $w/2$ on both sides and follows the shapes of chosen line cap and line join at path ends and corners respectively.

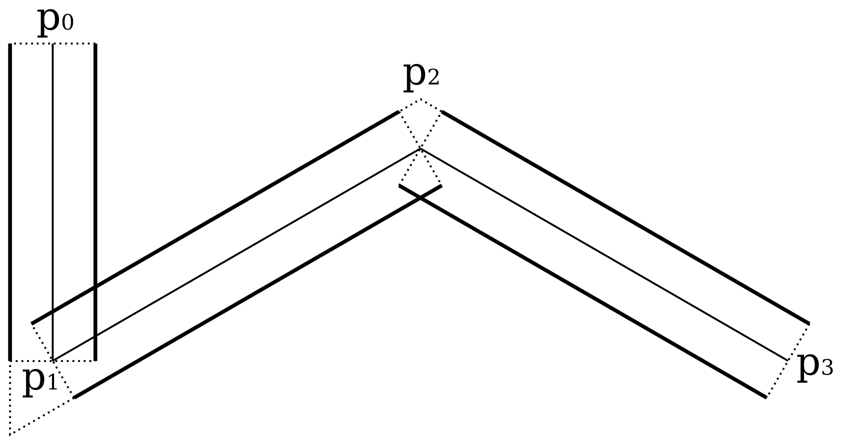


Figure 4.4. Offset line segments of a polyline

Figure 4.4 illustrates the basics of an algorithm for finding the outline polygon of a polyline. Each line segment is offset a distance $w/2$ orthogonally in both directions. Each pair of adjacent offset line segments is connected to form the outline polygon according to the below cases.

- If the line segments are on the inner side of a corner, connect them via the intersection point. The inner side is the side with an angle $< 180°$.
- If the line segments are on the outer side of a corner, connect them as defined by the line join. The outer side is the side with an angle $> 180°$. Round line joins are approximated.

[1]For example de Casteljau's algorithm for approximating Bézier curves

- If the line segments are parallel, they need not be connected as they are equivalent to a single line segment.
- If the line segments are at the beginning or end of the path, connect them as defined by the line cap. Round line caps are approximated.

The order in which the polyline is traversed and the two sides of the stroke outline polygon is computed depends largely on the data structures used for the polyline and outline polygon. No special order has been assumed here.

When stroking a path in this way, the following generally is true:

- Adjacent line segments on the inner side of a corner intersect.
- Adjacent line segments on the outer side of a corner do not intersect, but the extrapolation of the line segments intersect at a point which forms a corner between the line segments.
- Adjacent parallel line segments are co-linear, i.e. they both lie on a single straight line.

The only exception is when the stroke width is so large that two line segments on the inner side do not intersect, as illustrated in figure 4.5. In this type of situation, the best solution is often to connect the line segments with a simple bevel join (i.e. the line segments are connected with a straight line).

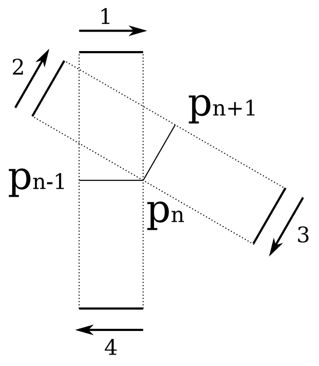


Figure 4.5. Path stroked at large width

The algorithm outlined above is a simplified version of what might be used in an actual vector graphics drawing toolkit, but still complete enough to give correct results in all but a few cases. It provides the basis for a modified stroking algorithm which handles the changes to the stroke model made in the next chapter.

The first generation of stroke fonts use a stroke model very similar to the one described in this chapter. With respect to fonts, its main limitations are the constant stroke width and the inability to represent serifs and other typographical ornaments at stroke ends.

Chapter 5

Extended Stroke Model

The traditional stroke model described in the previous chapter is limited as a basis for stroke fonts, as only typefaces such as Latin sans serif or Chinese heiti can be represented. In this chapter we suggest changes to the traditional stroke model and stroking algorithm to support richer, more complex stroking. It draws from the core concepts of SSF[7], but is implemented in a pure vector graphics environment. This will be the basis of VGSSF.

Starting from the traditional stroke model, two things are generalized:

- Line width is variable. For now it is not important *how* the line width is represented, only that there is a method of getting the width at an arbitrary point on the path.
- Line caps can be defined to be of arbitrary shape, individually for the beginning and end of the path. Again, exact representation is deferred until later.

Note that arbitrary line joins are not supported, but the possibility is discussed in section 10.2.

5.1 Stroking Algorithm

In the traditional stroke model a path is converted to a polyline from which an outline polygon is calculated. With the changes in the extended stroke model, the basic data structure in the stroking algorithm is not a simple polyline (list of (x, y) coordinates), but rather a list of (x, y, w) coordinate-width triplets. This single change is central to subsequent changes in the stroking algorithm. As the width in each point is independent, offset line segments are defined by the start and end points of the original line segment individually offset orthogonally. Thus, the offset line segment need not be parallel to the original path. Figure 5.1 illustrates a simple case where the width is decreasing monotonously along the path and the resulting offset line segments.

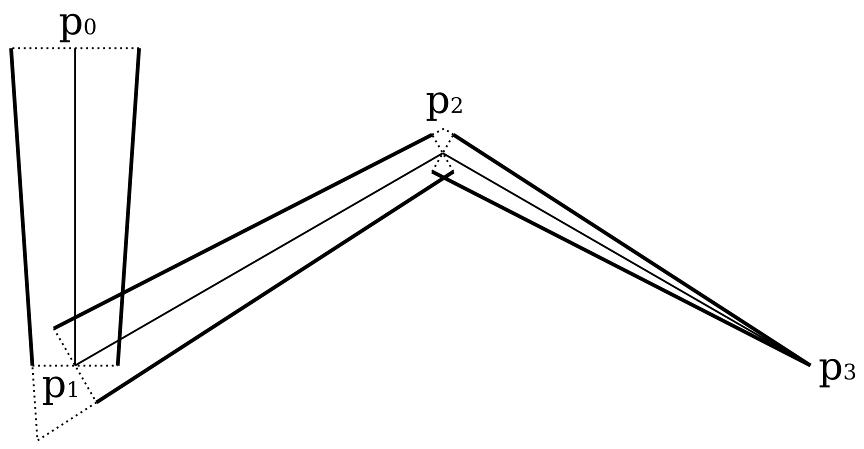


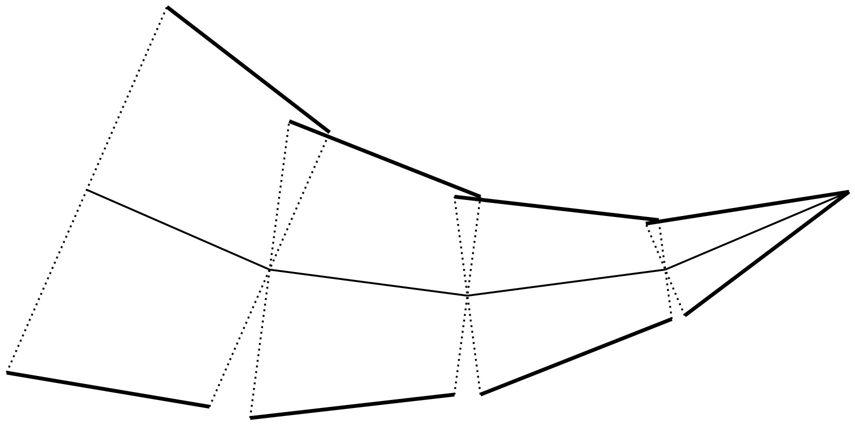
Figure 5.1. Well-behaved cases of offset line segments in a variable width polyline

Looking at such simple cases, it is tempting to think that the stroking algorithm of traditional stroke model can be used almost unchanged. In this naive implementation, adjacent line segments which intersect are joined at the intersection, while non-intersecting line segments are joined with a bevel join. While this does work in simple cases, it is far from satisfactory. Figure 5.2 illustrates a visible artifact in curved paths with increasing or decreasing line width that may occur.

On the concave side of the path (inner corner) adjacent line segments do not intersect, giving sawtooth shaped artifacts. On the convex side (outer corner) adjacent path segments are aligned such that the width of the stroke polygon is not monotonically decreasing, but rather constant in the joining line segment. This effect is noticeable even at moderate line widths. Beyond not being (mathematically or aesthetically) correct, this naive adaptation will use more points than necessary for the stroke outline polygon, resulting in longer computation times for any subsequent operations such as transformation or filling.

This example shows that a stroking algorithm which continues to make the same assumptions as before will fail to consider the relevant cases and produce unacceptable results in certain cases. As offset line segments are no longer necessary parallel to the centerline path, none of the assumptions made in the previous chapter for constant width stroking are any longer true.

- Adjacent line segments on the inner side of a corner need not intersect. See figures 5.3(a), 5.4(a) and 5.5(a).
- Adjacent line segments on the outer side of a corner still do not intersect. However, *if* they intersect in extrapolation, they may intersection at a point which does not form a corner between the line segments. See figures 5.3(b), 5.4(b) and 5.5(b).
- Adjacent parallel line segments need not be co-linear. See figure 5.4.


(a) Detail of stroking artifacts

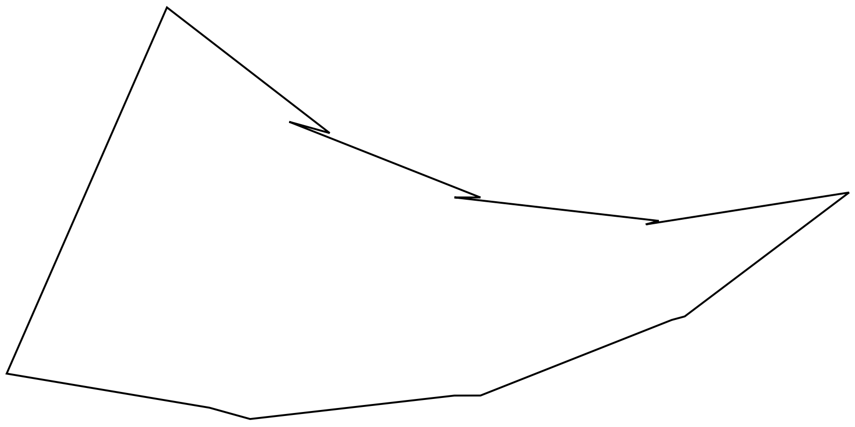

(b) Outline stroke polygon

Figure 5.2. Naive adaptation of the constant width stroking algorithm.

Figures 5.3–5.5 show that most types of intersections can occur on both the inner and outer side of a corner. None of the old assumptions are true, and it is clear that the case analysis can no longer be based on inner/outer sides. Instead, case analysis must be based on the type of intersection.

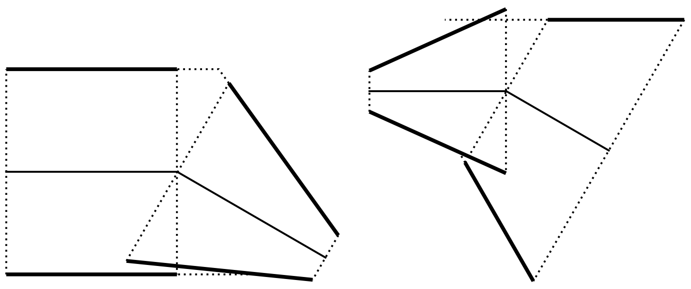

(a) inner side (b) outer side

Figure 5.3. Extrapolated intersection on one of the line segments

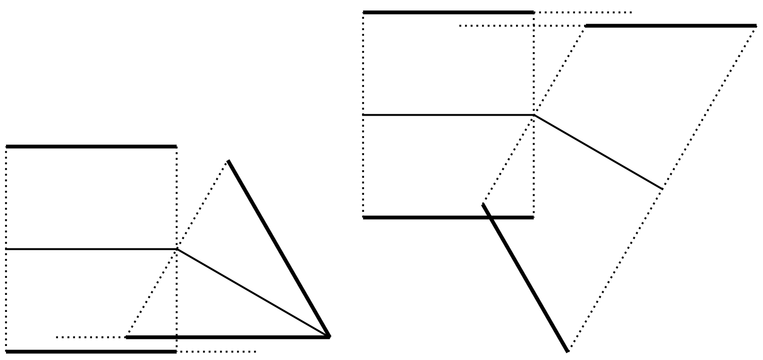

(a) inner side (b) outer side

Figure 5.4. Parallel line segments without intersection

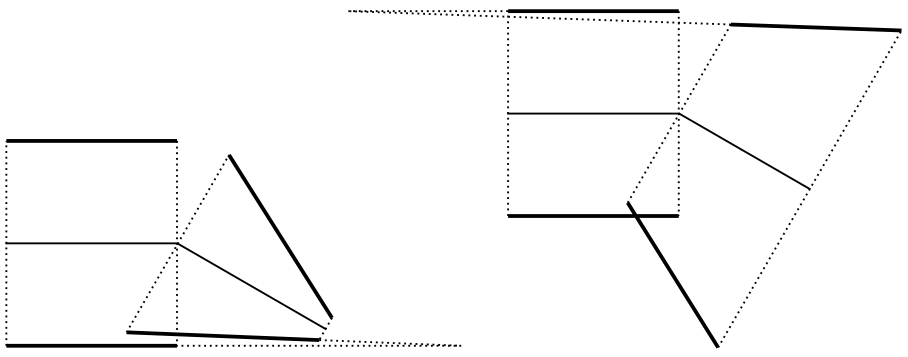

(a) inner side (b) outer side

Figure 5.5. Almost parallel line segments without useful intersection

Let L_n denote an offset line segment with with start point S_n and end point E_n. Let L_i and L_{i+1} denote two adjacent offset line segments, and $I_{i,i+1}$ denote the (extrapolated) intersection between them.

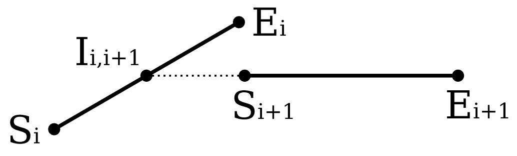


Figure 5.6. Two adjacent line segments and their (extrapolated) intersection

Consider a single offset line segment L between points S and E and an intersection I with an adjacent line segment. The intersection can be classified as being either before, on or after line segment L, as illustrated in figure 5.7.

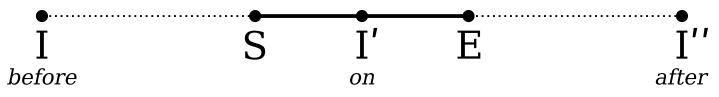


Figure 5.7. The three intersection cases: before, on and after

Now consider the vectors $\overline{SE}$, $\overline{SI}$ and $\overline{EI}$. These vectors are parallel (linearly dependent) and the sign of their dot products can be used to determine the type of the intersection. The following logical predicates on a line L are defined as shorthand notations. An intersection I is implied.

- before(L) := $\overline{SE} \cdot \overline{SI} < 0$
- on(L) := $\overline{SE} \cdot \overline{SI} \geq 0 \geq \overline{SE} \cdot \overline{EI}$
- after(L) := $\overline{SE} \cdot \overline{EI} > 0$

These predicates are the tools with which we can analyze the types of intersections. While this is not the only possible way to reason about the offset line segments and their intersections, we assert that it is effective in describing the relevant cases of the stroking algorithm.

5.1.1 Case Analysis

The core of the stroking algorithm is based on the following case analysis. Just as in the traditional stroke model, the algorithm builds an outline by adding zero or more points for each pair or adjacent offset line segments. The algorithm outlined below differs in that the case analysis is based on classification of the intersections rather than a distinction between inner/outer corner sides.

Case 1

$$\text{on}(L_i) \wedge \text{on}(L_{i+1})$$

A well-behaved intersection that lies on both line sections. This corresponds to the case of intersection on the inner corner side of the path in the traditional model. Connect the line segments via the intersection point as shown in figure 5.8.

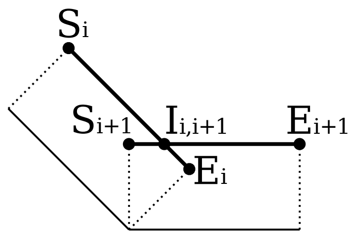


Figure 5.8. Simple on-line intersection

Case 2

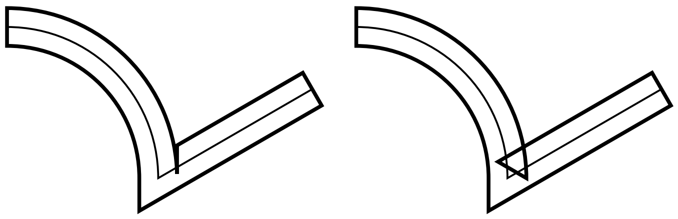
$$(\text{on}(L_i) \wedge \text{before}(L_{i+1})) \vee (\text{after}(L_i) \wedge \text{on}(L_{i+1}))$$

A well-behaved intersection that lies on one line segment and is extrapolated from the other as per figure 5.3. Connect the line segments via the intersection point. If this case is not handled properly, artifacts similar to those in figure 5.2 will occur.

Case 3

$$(\text{on}(L_i) \wedge \text{after}(L_{i+1})) \vee (\text{before}(L_i) \wedge \text{on}(L_{i+1}))$$

Given that the intersection *is* on one of the line segments $(\text{on}(L_i) \vee \text{on}(L_{i+1}))$, this case is the complement of case 1 and 2.



(a) Joined by intersection point (b) Joined by E_i and S_{i+1}

Figure 5.9. Potential stroking artifacts due to polygon decomposition

In figure 5.9, the arc has been converted into many short lines, meaning that the two line segments on the inner side of the corner will give rise to the situation

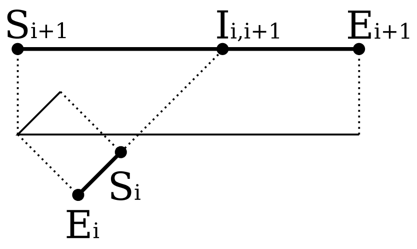


Figure 5.10. before(L_i) ∧ on(L_{i+1}). Reversed order gives on(L_i) ∧ after(L_{i+1}).

shown in figure 5.10. Joining them by their intersection results in an visible artifact in the form of a missing wedge in the stroke. Using both points gives an outline polygon which may seem incorrect at first sight, but if it is filled using the non-zero winding rule[1] the result will be correct.

Case 4

$$\text{after}(L_i) \wedge \text{before}(L_{i+1})$$

A well-behaved extrapolated intersection which forms a corner between the two line segments. This corresponds to the case of outer side joins in the traditional model, with one important difference. In the traditional model the corner formed will always be an isosceles triangle, i.e. $|\overline{E_i I_{i,i+1}}| = |\overline{S_{i+1} I_{i,i+1}}|$. In the extended model, there is no such guarantee, but the same line joins are to be applied. Bevel and miter joins are unchanged, but extra care must be taken with round joins, as illustrated by figure 5.11.

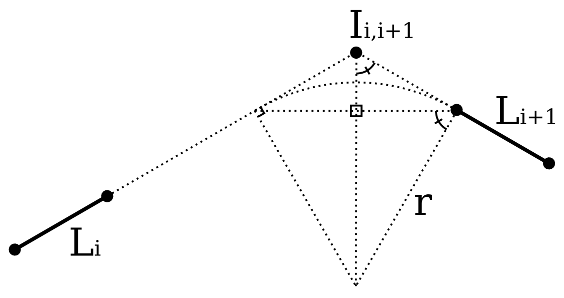


Figure 5.11. Possible round join for variable width stroking

Noting that the top and bottom triangles in the figure are *similar* (have corresponding angles and sides), deducing the radius from the known points is trivial. The arc can then be drawn with the same routines as in the algorithm for the

[1]See `http://www.w3.org/TR/SVG/painting.html#FillProperties` for an explanation of different fill rules.

traditional model. Finally, the remainder of the longer line segment is connected with a straight line.

Case 5

$$(\text{before}(L_i) \wedge \text{before}(L_{i+1})) \vee (\text{after}(L_i) \wedge \text{after}(L_{i+1}))$$

The intersection is either before both line segment or after both line segments, as in figure 5.5. This case can occur with line segments which are almost parallel and the intersection is not useful for our purposes. Again, failing to properly handle this case will result in artifacts similar to those in figure 5.2.

It is generally desirable for the outline the polygon to follow the line segments, but unless the degenerate intersection point is used, the outline polygon can follow at most one of the line segments. In other words, some compromise must be made to get a visually satisfactory result which still follows the line segments fairly closely.

One trivial method is to use the midpoint between the two points E_i and S_{i+1}. However, this method gives visually unsatisfactory results, as seen in figure 5.14.

Note instead that the two line segments can be considered "inner" and "outer" in relation to each other and the original path.[2] Specifically, the inner line segment is that which in extrapolation intersects the area between the original path and the other line segment (which would consequently be the outer line segment).

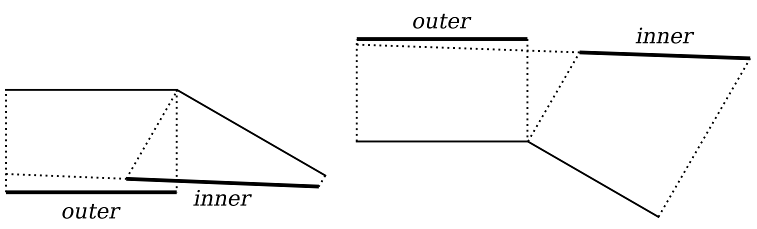


Figure 5.12. Inner and outer line segments

This test is straightforward to implemented by taking the intersection of the first line segment and the line segment orthogonal to the second (dotted line in the figure). If the intersection point lies inside the second line segment then the first line segment is inner. Otherwise, the opposite must be true.

Using this test the algorithm can be designed to always follow the inner line segment or always follow the outer line segment. To *follow* in this context means to use only the start/end point on that line segment, ignoring the other line segment. The results can be seen in figure 5.15 and 5.16.

Both methods prove to be somewhat lacking. Using the inner line segment results in artifacts on the convex (outside corner) side of the stroke, and vice versa. The ideal solution would be to always use the inner line segment on inner corner side and the outer line segment on the outer corner side. This may seem

[2]This is not directly related to the inner and outer sides of corners previously discussed.

overly complex, but there is a very simple test which can be used. Compare the width gradient of the line segments in figure 5.12 and identify the line segment with the minimum gradient away from the common point; on the inner corner side it is the inner line segment and on the outer corner side it is the outer line segment. In other words, a simple comparison of the width gradients is enough to select the best line segment. This is the method that was finally used in our implementation. The result can be seen in figure 5.17.

Finally, it should be noted that the width of the stroke is slightly different depending on which method is used. If precise stroke widths are deemed important then further research on this case is advised.

Case 6

$$\text{before}(L_i) \wedge \text{after}(L_{i+1})$$

Compare with case 4. The extrapolated intersection forms a “reverse” corner. This can happen on the inner corner side in the traditional model, as already seen in figure 4.5. In the extended model it can additionally occur on the outer corner side, as seen in figure 5.13.

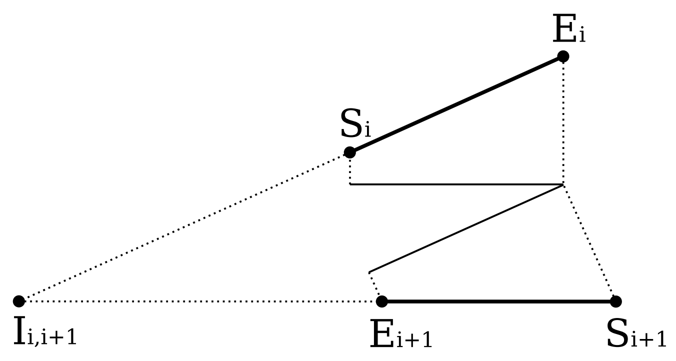


Figure 5.13. Example of reverse corner intersection

For the situation to arise, the width of the path must decrease fairly dramatically in both directions away from the common point. Suffice to say, this is a corner case (no pun intended) of doubtful importance as it is unlikely to occur in any natural path. For our purposes it is acceptable to simply use a bevel join.

Case 7

Case 1–6 cover all cases where an intersection exists. However, if an intersection in fact does not exist, the line segments must be parallel. Parallel and co-linear line segments occur also in the traditional model and can be treated as if they were a single line segment. If the line segments are parallel but not co-linear, the situation is very similar to case 5 and can in fact be handled in the same way. This similarity can be seen in figures 5.5 and 5.4.

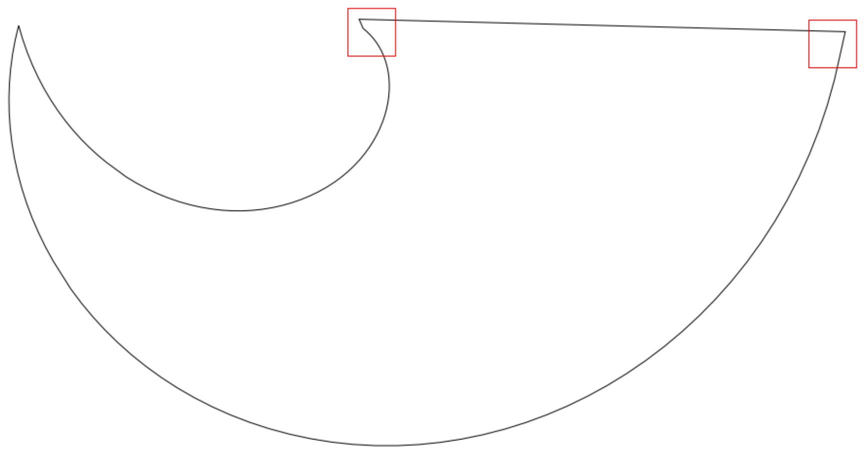


Figure 5.14. Case 5: Join at midpoint of E_i and S_{i+1}

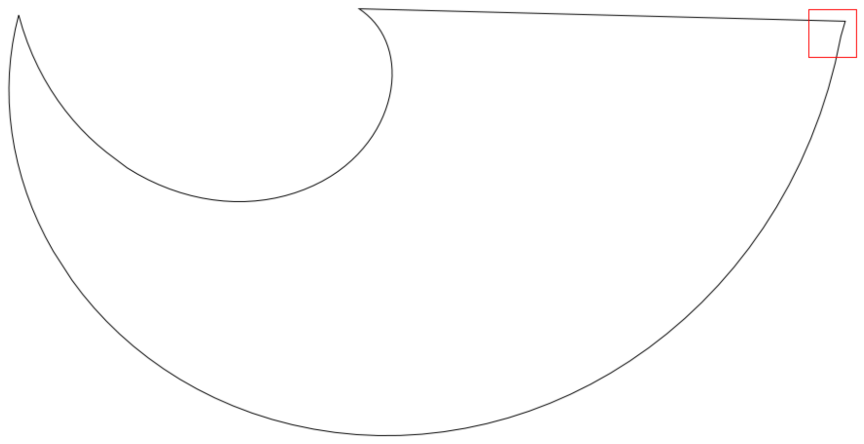


Figure 5.15. Case 5: Follow inner line segment

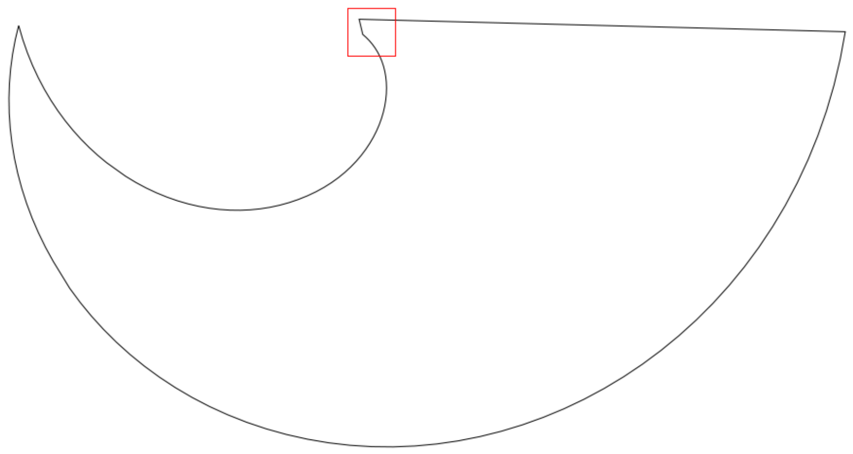


Figure 5.16. Case 5: Follow outer line segment

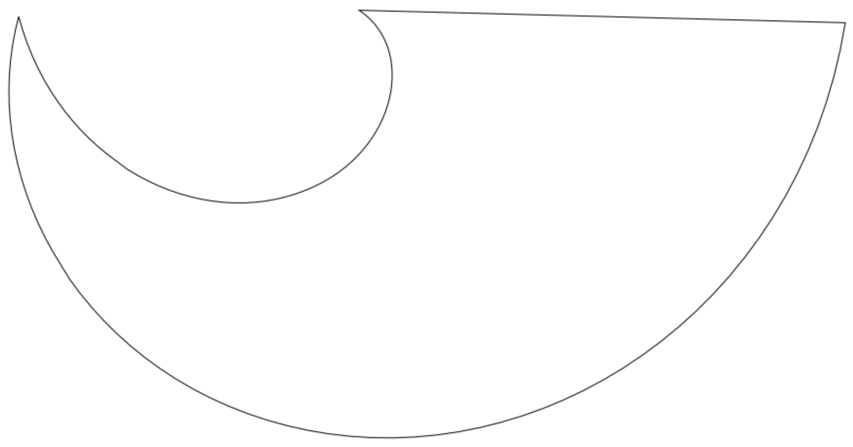


Figure 5.17. Case 5: Follow inner line segment on inner corner side and outer line segment on outer corner side

Logically Exhaustive

Case 1–3 cover all cases where the intersection is *on* either line segment, while case 4–6 cover all cases where the intersection is either *before* or *after* both line segments. Together with case 7 (where no intersection exists) this makes the case analysis logically exhaustive. A propositional logic proof of this is given in appendix A. Exhaustiveness in itself does not guarantee that the case analysis is fine-grained enough, so it is still possible that there are undiscovered cases which are not handled correctly. However, the algorithm has been tested thoroughly during the course of implementation and we believe it correctly handles all the relevant cases.

5.1.2 Line Caps

Line caps connect the two sides of the outline polygon at the start or end of the stroked path and can be seen as a path which connects 2 points. Seen in this way, the predefined round line cap is a 180° arc. Note that the direction of the line cap path must match the direction in which the outline polygon is constructed in the stroking algorithm. Otherwise, it would create a round hole instead of a round end on the stroke.

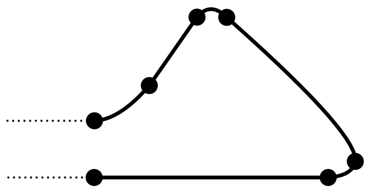


Figure 5.18. User-defined line cap

User-defined line caps are defined as open paths and rendering them is fairly simple. As the line cap can be defined in any (Cartesian) coordinate system, it must be transformed to fit with the stroke end.

1. Translate the line cap path so that the first point is in origo.
2. Rotate the line cap path so that the angle between the first and last point equals the angle between the two points on the stroke end.
3. Scale the line cap path so that the distance between the first and last point equals the width of the stroke end.
4. Translate the line cap path so that the first and last points align with the two points on the stroke end.

When the line cap is correctly aligned, it is converted to a polyline and merged with the stroke outline polygon. This marks the big difference between SVG marker symbols and user-defined line caps. An SVG marker symbol is independent (in geometry, color, etc) and overlayed on top of the stroked path. Due to the exact fitting needed, small numerical errors in size or position may become visible. As user-defined line caps are an integral part of the stroking algorithm, this problem does not occur.

5.2 Summary

Using the stroke model and algorithm described in this chapter, the traditional model has been extended to support both variable line widths and user-defined line caps. Support in Opera's internal vector graphics drawing toolkit was implemented using the stroking algorithm described in this chapter. This is the first and most important building block for supporting stylized stroke fonts.

Chapter 6

Extended JavaScript Canvas

In the previous chapter, extended stroking support was added to the internal graphics toolkit of the Opera browser. However, in order to make use of this added functionality it must also be exposed to the browser client.

HTML 5 includes a `canvas` element,[26] which is a simple raster canvas with a JavaScript interface. It uses the traditional stroke model and is implemented using Opera's internal vector graphics toolkit. In this chapter we extend the Canvas interface to support the extended stroke model.

The following example shows simple path stroking using the standard HTML 5 Canvas JavaScript interface.

```
<!DOCTYPE HTML>
<html>
<head>
<title>HTML 5 Canvas DOM</title>
<script type="text/javascript">
//<![CDATA[
document.addEventListener("load", paint, false);
function paint() {
  canvas = document.getElementById("canvas");
  canvas.setAttribute("width", 400);
  canvas.setAttribute("height", 300);
  var ctx = canvas.getContext("2d");

  ctx.beginPath();
  ctx.moveTo(100, 100);
  ctx.bezierCurveTo(100, 200, 200, 100, 200, 200);
  ctx.lineTo(300, 100);
  ctx.lineWidth = 25;
  ctx.lineJoin = "round";
  ctx.lineCap = "round";
  ctx.stroke();
}
//]]>
</script>
```

```
</head>
<body>
<canvas id="canvas" style="border: 1px solid black">
</canvas>
</body>
</html>
```

The `getContext("2d")` call returns the standard `CanvasRenderingContext2D` rendering context. The rendered result can be seen in figure 6.1.

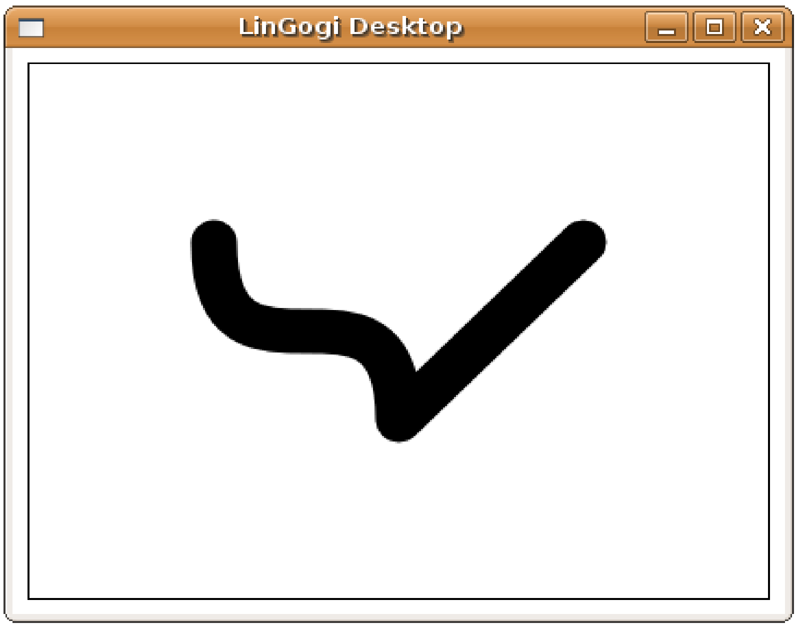


Figure 6.1. Path stroking with standard HTML 5 Canvas

We define a new `CanvasRenderingContext2DExt` rendering context which is extended to support the new functionality in the underlying vector graphics toolkit.

- New `createCustomLineWidth` method returns a `CanvasCustomLineWidth` object.
- The `lineWidth` attribute can be set to a `CanvasCustomLineWidth` object.
- New `createCustomLineCap` method returns a `CanvasCustomLineCap` object.
- The `lineCap` attribute can additionally be set to ‘`custom`’.
- New `lineCapStart` and `lineCapEnd` attributes of type `CanvasCustomLineCap` will be used if `lineCap` is ‘`custom`’.

This rendering context is returned when `"opera-2d-ext"` is passed to `getContext()`.

6.1 Line Width

A new `CanvasCustomLineWidth` object is used to define the stroke width along a path. A width function is constructed by repeated calls to `addWidthStop(offset, width)`, where `offset` is between 0 and 1 and `width` is non-negative. This is very to similar to `addColorStop(offset, color)` on `CanvasGradient`.

```
var lineWidth = ctx.createCustomLineWidth();
lineWidth.addWidthStop(0,30);
lineWidth.addWidthStop(1/3,25);
lineWidth.addWidthStop(2/3,10);
lineWidth.addWidthStop(1,5);
ctx.lineWidth = lineWidth;
```

The width between the defined offset points is linearly interpolated, as shown in figure 6.2. While there are certainly other – possibly better – ways to define the line width function, this simple definition is a good starting point. Other possible solutions are discussed in section 10.1.

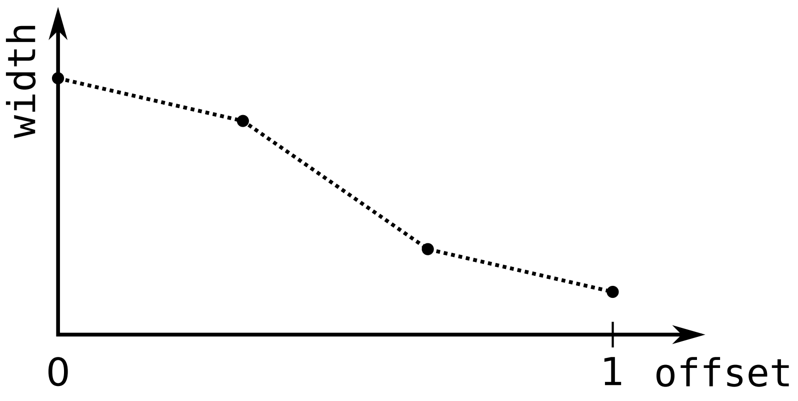


Figure 6.2. Example stroke width function

6.2 Line Caps

A new `CanvasCustomLineCap` is used to encapsulate a single, open path which defines the shape of the line cap. To create that path, it is very natural to use the same path operations as in the standard 2d context: `moveTo`, `lineTo`, `quadraticCurveTo`, `bezierCurveTo` and `arcTo`. The first and last point must be separate and the direction of the path is important, as already described.

```
var lineCap = ctx.createCustomLineCap();
lineCap.moveTo(1,0);
lineCap.lineTo(0,5);
lineCap.moveTo(-1,0);
```

```
ctx.lineCap = "custom";
ctx.lineCapStart = lineCap;
ctx.lineCapEnd = lineCap;
```

The above code adds a triangular line cap on both ends of the stroke.

6.3 Example

This is a complete example using both variable line width and custom line caps. Only the JavaScript code is included. The rendered result can be seen in figure 6.3.

```
canvas = document.getElementById("canvas");
canvas.setAttribute("width", 400);
canvas.setAttribute("height", 300);
var ctx = canvas.getContext("opera-2d-ext");

var narrowingWidth = ctx.createCustomLineWidth();
narrowingWidth.addWidthStop(0,40);
/* optionally more stops here */
narrowingWidth.addWidthStop(1,20);

var hammerCap = ctx.createCustomLineCap();
hammerCap.moveTo(1, 0);
hammerCap.lineTo(2, 0);
hammerCap.lineTo(2, 2);
hammerCap.lineTo(-2, 2);
hammerCap.lineTo(-2, 0);
hammerCap.lineTo(-1, 0);

var diamondCap = ctx.createCustomLineCap();
diamondCap.moveTo(1, 0);
diamondCap.lineTo(3, 2);
diamondCap.lineTo(0, 5);
diamondCap.lineTo(-3, 2);
diamondCap.lineTo(-1, 0);

ctx.beginPath();
ctx.moveTo(100, 100);
ctx.bezierCurveTo(100, 200, 200, 100, 200, 200);
ctx.lineTo(300, 100);
ctx.lineWidth = narrowingWidth;
ctx.lineJoin = "round";
ctx.lineCap = "custom";
ctx.lineCapStart = hammerCap;
ctx.lineCapEnd = diamondCap;
ctx.stroke();
```

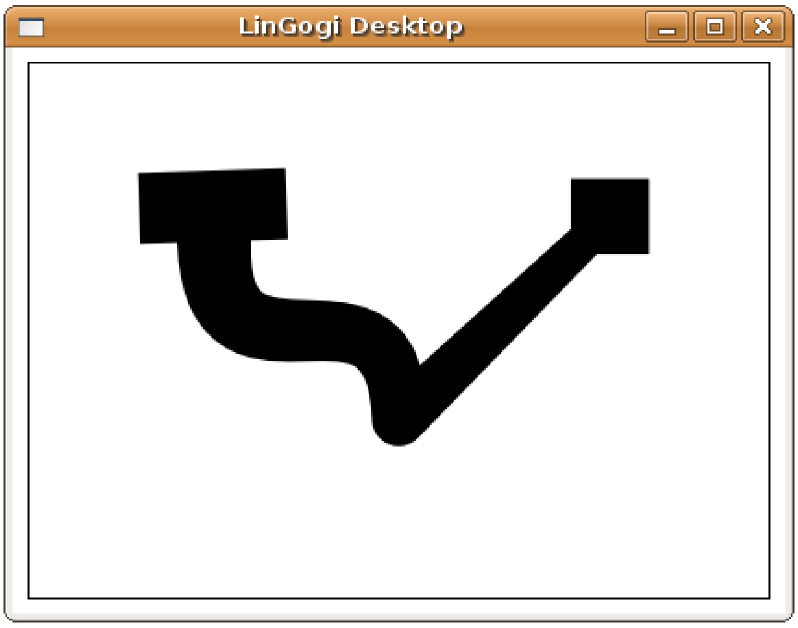


Figure 6.3. Path stroking with extended HTML 5 Canvas

The possible applications of an extended Canvas interface are many. The most trivial is to define a custom line cap such as an arrowhead to simplify drawing certain geometry. Further, animating the line width could give interesting effects with possible applications in games or visualization. While the extended Canvas interface is useful in its own right, in this thesis project it is mainly used to test the underlying vector graphics drawing toolkit and finally to render VGSSFs.

Chapter 7

Extended SVG Fonts

With the basic graphics toolkit support in place, a format with which to define stylized stroke fonts is needed. Not surprisingly, SVG fonts[27, Ch 20 Fonts] already fill most of our needs. All that is required are a few extensions for path stroking similar to those made in HTML 5 Canvas. In order to not "pollute" SVG and for other technical reasons, extensions are done in a separate XML namespace with prefix `op` and name `http://opera.com/`, but there is *no* formal schema or semantics implied. Only these changes are described in this chapter, for a the complete description of the font format refer to the SVG specification.

Line widths are defined similarly to SVG's native `linearGradient`:

```
<op:lineWidth id="narrowingWidth">
  <stop offset="0" stop-width="40"/>
  <!-- optionally more stops here -->
  <stop offset="1" stop-width="20"/>
</op:lineWidth>
```

Line caps are simpler and can be defined as a single SVG path:

```
<op:lineCap id="hammerCap"
            d="M 1 0 L 2 0 L 2 2 L -2 2 L -2 0 L -1 0"/>
<op:lineCap id="diamondCap"
            d="M 1 0 L 3 2 L 0 5 L -3 2 L -1 0"/>
```

Applying these to an SVG path is equally straightforward:

```
<path d="M 100 100 C 100 200 200 100 200 200 L 300 100"
      op:stroke-width="url(#narrowingWidth)"
      op:stroke-linecap-start="url(#hammerCap)"
      op:stroke-linecap-end="url(#diamondCap)"/>
```

Since SVG fonts allow defining glyphs using arbitrary SVG content[27, Ch 20.4 The glyph element], the above additions are all that are needed to support VGSSF in SVG. An SVG font with a single glyph could look as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<svg xmlns="http://www.w3.org/2000/svg"
     width="100%" height="100%"
     xmlns:op="http://opera.com/">
<defs>
<font horiz-adv-x="512" >
<font-face font-family="AR PL SungtiL GB"
           units-per-em="1024"
           ascent="900"
           descent="-123"
           alphabetic="0" />
<glyph unicode="&#x6728" horiz-adv-x="1024">
  <path d="M 188 590 H 803" stroke-width="31"
        op:stroke-linecap-start="url(#heng-start)"
        op:stroke-linecap-end="url(#heng-stop)" />
  <path d="M 512 719 V 152" stroke-width="48"
        op:stroke-linecap-start="url(#shu-start)"
        op:stroke-linecap-end="url(#shu-stop)" />
  <path d="M 463 590 Q 398 327 57 40"
        op:stroke-width="url(#pie)"
        op:stroke-linecap-end="url(#round)" />
  <path d="M 551 589 Q 675 283 845 145"
        op:stroke-width="url(#na)"
        op:stroke-linecap-end="url(#na-stop)" />
</glyph>
<op:lineWidth id="pie">
  <stop offset="0" stop-width="46" />
  <stop offset="1" stop-width="24" />
</op:lineWidth>
<op:lineWidth id="na">
  <stop offset="0" stop-width="18" />
  <stop offset="1" stop-width="58" />
</op:lineWidth>
<op:lineCap id="heng-start"
  d="M 187 607 H 175 Q 129 607 91 616 Q 84 620 80 617
     Q 76 613 84 606 Q 99 593 104 575
     Q 105 566 114 569 Q 145 575 187 576" />
<op:lineCap id="heng-stop"
  d="M 804 576 H 931 Q 944 577 946 585 Q 947 600 876 663
     Q 867 672 860 663 L 834 626 Q 818 608 804 607" />
<op:lineCap id="shu-start"
  d="M 536 718 Q 536 779 543 798 Q 547 808 563 816
     Q 579 825 565 832 Q 547 842 493 859
     Q 481 863 483 846 Q 487 798 488 718" />
<op:lineCap id="shu-stop"
  d="M 488 151 Q 487 60 481 -73 Q 480 -94 493 -84
     Q 521 -66 535 -57 Q 541 -53 540 -45
     Q 536 34 536 151" />
<op:lineCap id="round"
```

```
  d="M 1000 0 C 1000 276 776 500 500 500
     C 224 500 0 276 0 0" />
<op:lineCap id="na-stop"
  d="M 827 124 Q 860 93 895 65 Q 902 58 906 63
     Q 912 79 927 90 Q 946 102 963 100
     Q 976 99 979 106 Q 980 113 975 115
     Q 923 126 865 168" />
</font>
</defs>
</svg>
```

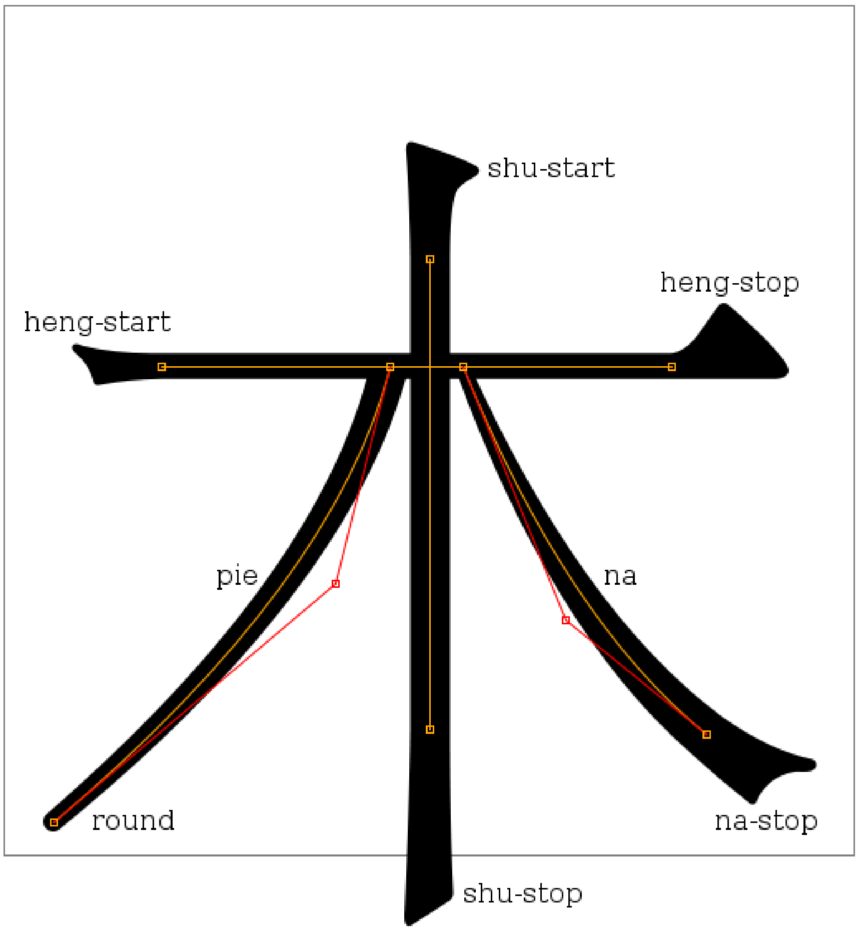


Figure 7.1. The Chinese character 木 (wood) in Ming typeface

Some knowledge of SVGs path data format[27, Ch 8 Paths] is required to be able to interpret the above example. Note especially that SVG fonts are defined to have an inverted y-axis relative to generic SVG, for compatibility with the conventions in most other font formats. Figure 7.1 is labeled to show which part of the SVG code corresponds to which part of the rendered glyph.

One important aspect of the font format is that line caps and line widths once defined can be used any number of times in any number of glyphs. This way, the amount of extra data required for this representation would in relative terms be small, as the number of unique ornamental structures in most typefaces is limited. In the same way, commonly occurring strokes would share the same line width definition.

7.1 A JavaScript Renderer

Unlike the extended stroking algorithm and the extended Canvas interface, the extensions to SVG fonts were actually not implemented *in SVG*. The reasons for this are mainly practical – SVG is very complex and making the needed changes would likely be very time consuming. As we only need a tiny subset of SVG, it was deemed easier to implement a JavaScript renderer which interprets the SVG font file and draws glyphs using the JavaScript Canvas interface. Reading and writing SVG font files is dealt with in the following chapter, assume for now that we have access to all the SVG glyph data.

As seen in the above example, VGSSF is defined in SVG by using one `path` elements per stroke of the glyph. To stroke each path, the SVG path data must first be translated to JavaScript canvas calls. The mapping between SVG and Canvas is simple for most commands.

Path Data	Canvas JavaScript
`M x y`	`moveTo(x, y)`
`Z`	`closePath()`
`L x y`	`lineTo(x, y)`
`H x`	`lineTo(x, y_prev)`
`V y`	`lineTo(x_prev, y)`
`C x1 y1 x2 y2 x y`	`bezierCurveTo(x1, y1, x2, y2, x, y)`
`Q x1 y1 x y`	`quadraticCurveTo(x1, y1, x, y)`

This is not a complete mapping of all SVG path data commands, as only those mappings which were deemed necessary for this thesis work were implemented.

VGSSF further requires extended line caps and line joins. Parsing an `op:lineWidth` element into a `CanvasCustomLineWidth` object is trivial as the semantics of the two are exactly the same. Parsing an `op:lineCap` element into a `CanvasCustomLineCap` object is again a question of mapping SVG path data to JavaScript Canvas calls, with the only difference that the calls are made to a special `CanvasCustomLineCap` object.

With these very simple means a JavaScript renderer for VGSSF has been implemented. (Note however that some mundane details have been omitted here.) While it is very far from a complete SVG font renderer, it is sufficient as an evaluation tool for VGSSF.

Chapter 8

Font Editor

Individual example glyphs can be created by manually editing an SVG font file, but this process is time-consuming and error-prone. In order to create a larger number of glyphs a graphical font editor is required. As rendering support was developed in the Opera browser, implementing the font editor on top of the browser is natural. This is unconventional, but the kind of user interface that can be achieved with modern browser technology is sufficient for the simple editor we need.

Manipulating the font file is most conveniently done using the Document Object Model (DOM).[25] Using the DOM Level 3 Load and Save interface the file can be loaded as a DOM object. After some editing, the DOM object can be serialized back to SVG.[1]

While the document structure can be effectively edited via DOM, the SVG path data can not. Instead, it is parsed into a list of commands and their arguments. All editing is done on this internal data structure, which is converted back to SVG path data before the font file is serialized. For simplicity, this internal structure is also used when rendering the glyph.

The main view of the editor is drawn with JavaScript Canvas by building on the glyph renderer described in the previous chapter. In addition to drawing the glyph itself, all control points are also shown. When the user clicks the canvas, the path command lists are search for a vertex at that point. If one is found, the appropriate action will be taken based on the editing mode. These actions have been implemented:

- Selecting and deselecting points.
- Moving selected points.
- Deleting selected points.
- Subdividing selected path segments. If two neighboring vertices are selected, a new vertex is inserted between them.

[1]As the browser cannot write to the local filesystem the SVG code is simply shown to the user in a new window.

Note that there is no way to edit line widths or line caps or add new paths. While these are serious limitations, it is acceptable for our purposes, as the font editor was used mostly as a tool for recreating existing glyphs. Figure 8.1 shows an example of what the editor looks like in action.

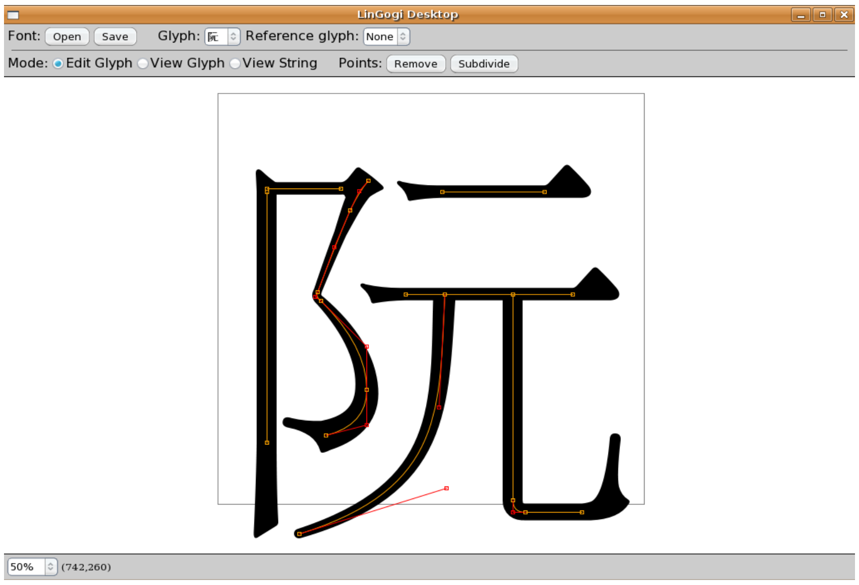

LinGogi Desktop
Font: Open Save Glyph: 阮 Reference glyph: None
Mode: Edit Glyph View Glyph View String Points: Remove Subdivide
50%
(742,260)

Figure 8.1. Editing Chinese character 阮 (a family name)

Beyond editing and viewing single glyphs, the editor also has primitive support for rendering strings. An example is shown in figure 8.2. This is done as a proof-of-concept using only the horizontal advance distance for each glyph. SVG fonts support kerning pairs, but these are simply ignored. Still, it shows that the extended SVG fonts is a practical and realistic format for VGSSF.

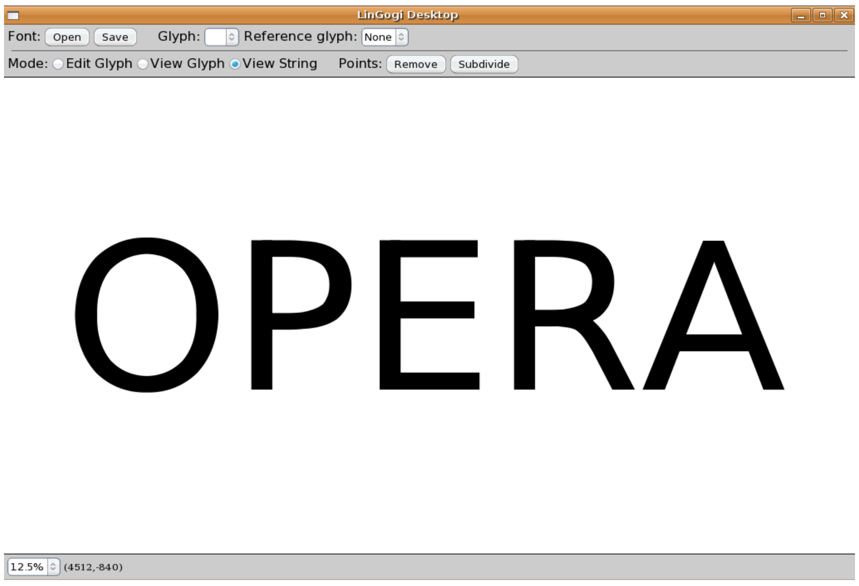


Figure 8.2. Rendering the string “OPERA”

Chapter 9

Evaluation

This chapter evaluates several aspects of VGSSF. First and foremost, the suitability of VGSSF to our reference typefaces is discussed. In doing this, we will see what the limitations of the stroke representation are. Further, the impact on resource usage is evaluated, both in terms of storage and processing requirements. Finally, we consider how editing is affected by the stroke representation.

9.1 Glyphs

Using the font editor, glyphs from Arphic Technologies' *PL Mingti2L Big5* and Bitstream's *Vera Sans* TrueType fonts were converted into stroke representation. The result can be seen side-by-side with the original outline glyphs in figures 9.1 and 9.2 respectively.

Judging by the selection of Chinese characters that were converted, VGSSF is very suited for the Chinese Ming typeface. Chinese heiti typeface can be made with existing stroke font formats (or standard SVG fonts) and could certainly be made with VGSSF. The Chinese calligraphy style Kaishu has less ornamental decorations but more complex line widths. Representing line width as linearly interpolated offset-width pairs is likely not the best option for this typeface, as the number of pairs required for a smooth result is fairly large.

Although result for Chinese fonts are encouraging, it should be noted that there is some discrepancy between the concept of a "stroke" in stroke fonts and the strokes of Chinese characters. Figure 9.3 shows how it may be necessary to break down a "complex" stroke into several shorter strokes. This is done because the ornament in the top right corner (essentially a line join) must be represented as a line cap and a second glyph stroke fitted to finish what is usually considered a single stroke.

The results for Latin glyphs are more mixed. One problem can be seen in the capital letter "A", as illustrated in figure 9.4. While the two stems of an "A" are slanted, it is expected that the base and top of the glyph should be horizontally aligned. In order to achieve this, triangular "wedge" line caps which match the angle of the stem are used. While visually acceptable results using this technique

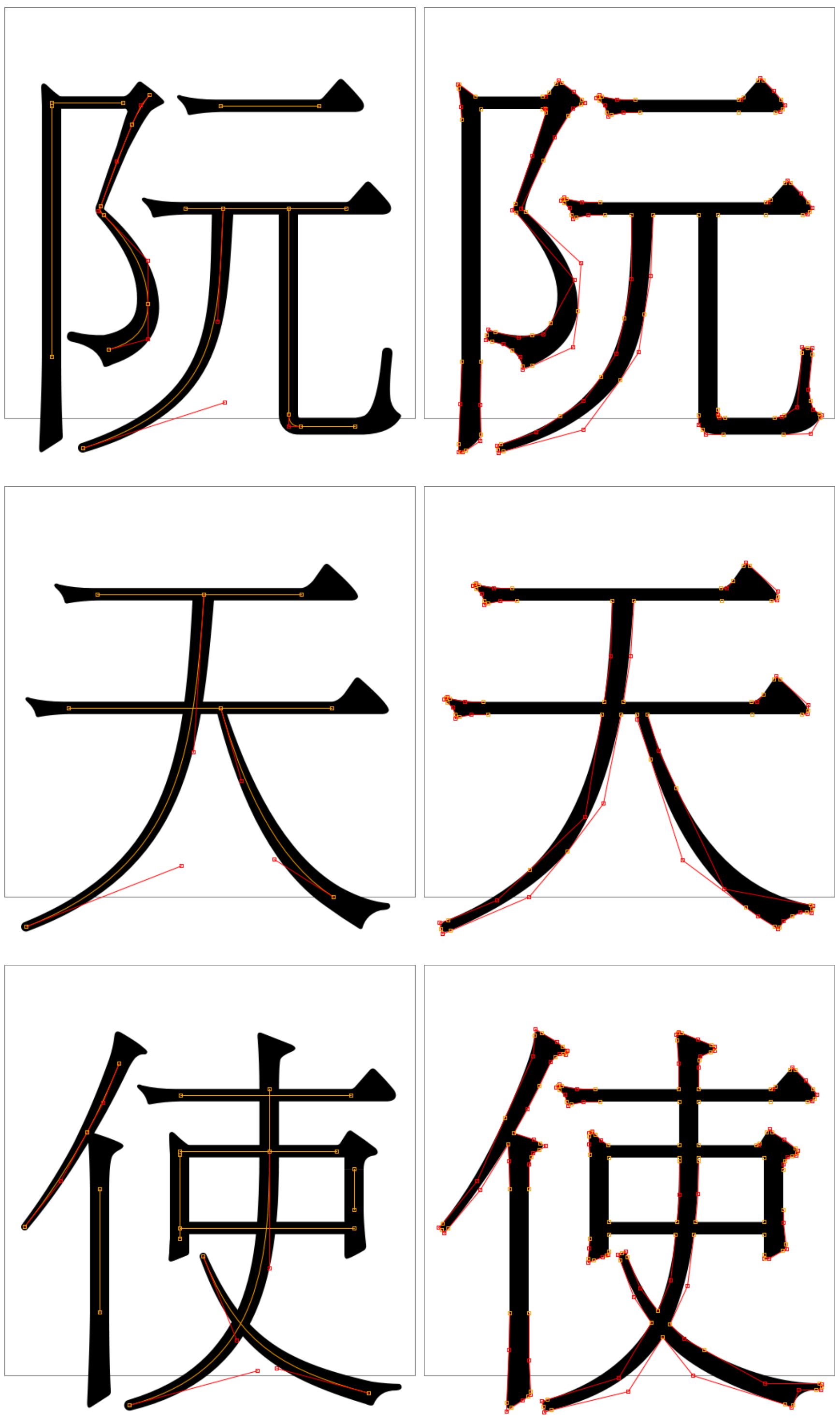


Figure 9.1. Glyphs from Arphic Technologies' *PL Mingti2L Big5* Chinese Ming typeface

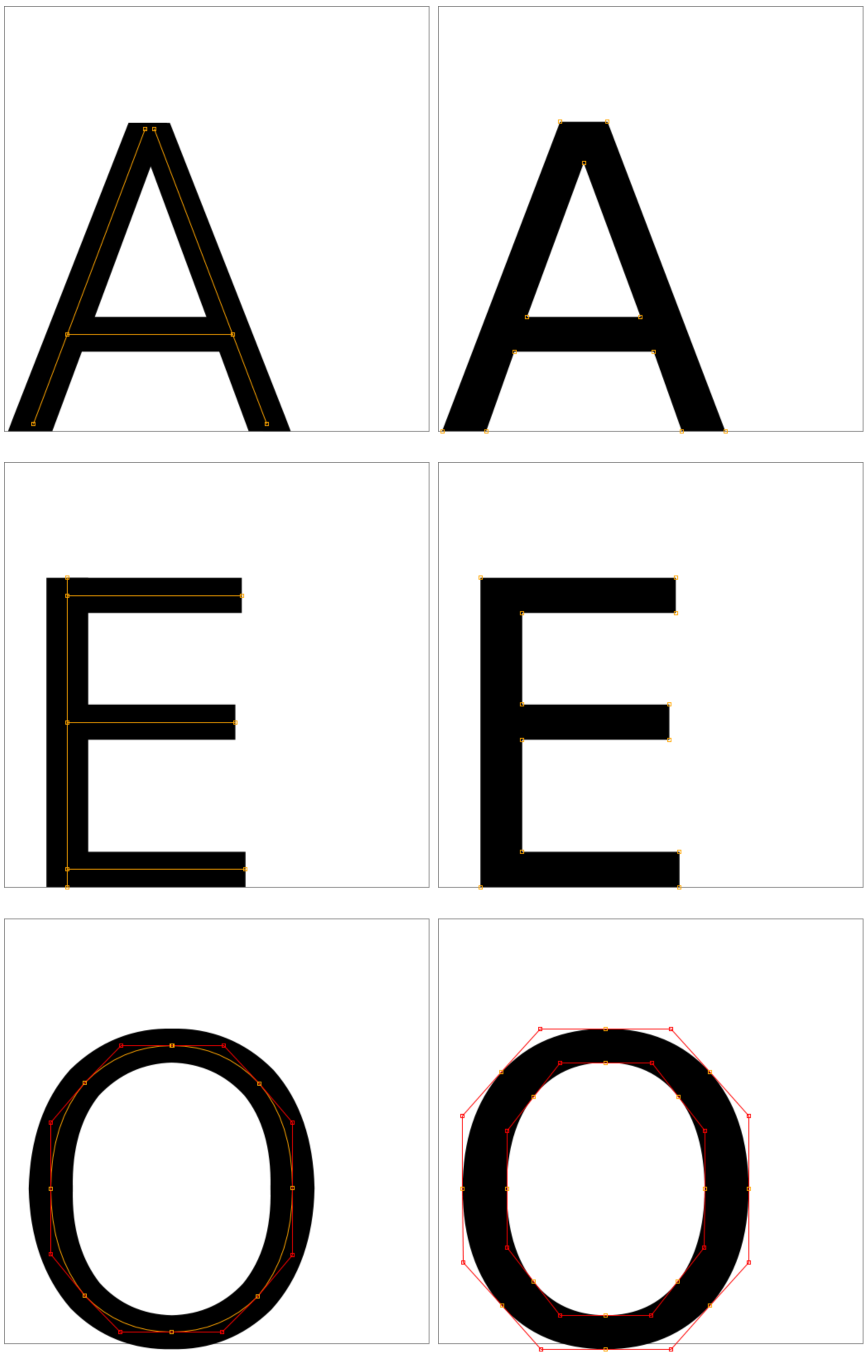
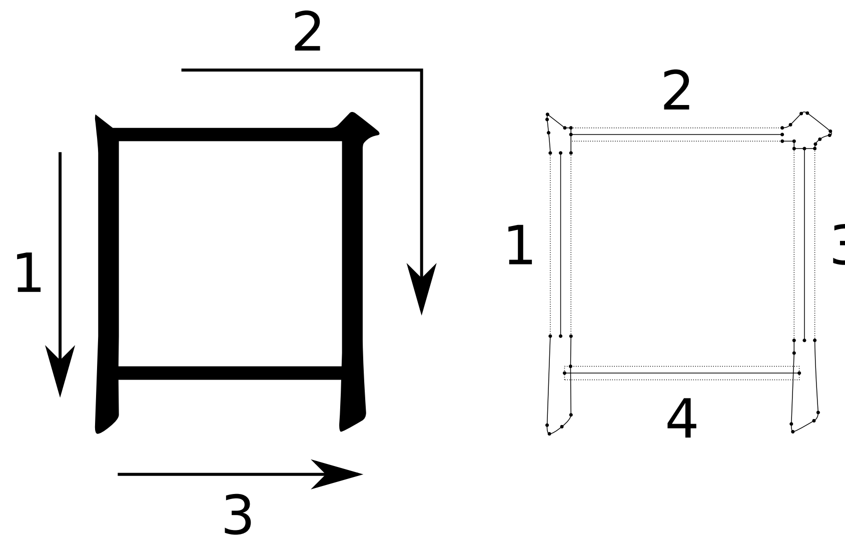


Figure 9.2. Glyphs from Bitstream's *Vera Sans* Latin sans serif typeface



(a) Natural stroke order (b) Glyph stroke decomposition

Figure 9.3. Strokes of the Chinese character 口 (mouth)

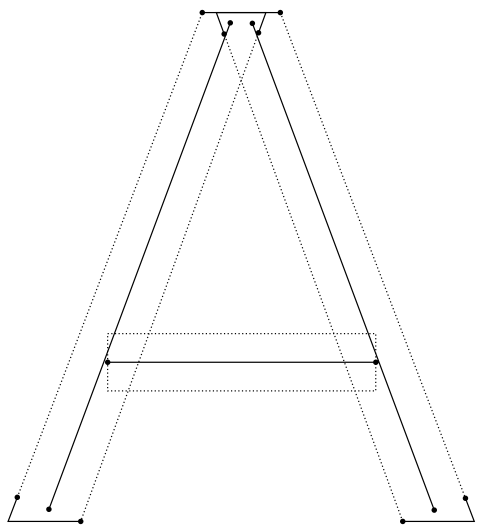


Figure 9.4. Stroked glyph of capital letter "A"

are possible, an *exact* match will only be possible for certain angles and widths. In a serif font where the serifs are represented as line caps this problem would likely be even more accentuated.

A related problem is the fitting of the two stems at the top of the “A”. Since there is no line join with the wanted shape, the only option is to use two strokes with “wedge” line caps which are carefully aligned. If the stroke width were slightly changed (for example to create a bold typeface) the lines would have to be realigned.

While there are cases in Chinese and Latin fonts that must be solved with more or less clever “hacks”, there are still other glyphs that arguably *cannot* be represented with strokes as they were originally not drawn with a pen or similar writing tool. Figure 9.5 shows some such glyphs.

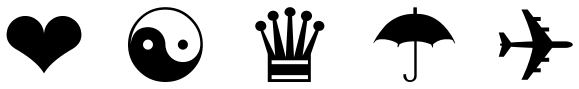


Figure 9.5. Glyphs not suited for stroke representation

The conclusion must be that if a generic font format (one suited for all needs) is wanted, then a pure stroke font format is insufficient. The extended SVG fonts used here already allow the use of any SVG content, which effectively makes it a outline-stroke hybrid format. If deemed necessary, it would be possible to represent some glyphs as outlines, some as strokes and yet others as a mixture of outlines and strokes.

9.2 Resource Usage

9.2.1 Storage Requirements

The stroke representation reduces the number of points needed to represent a glyph and ultimately the size of the font file. Monotype Imaging claim storage savings of 90% or more on a simplified Chinese font.[1] Bitstream claims storage savings of at least 90% on a traditional Chinese font.[2] Note that a part of this saving is not due to the stroke representation, but rather due to the reuse of character components.[3] However, such reuse is possible in both TrueType[4] and SVG[5] and is not due to the stroke-based representation.

Table 9.1 and 9.2 show the approximate storage requirements (measured in number of floats or integers) for equivalent glyphs using outline and stroke repre-

[1]GB2312 character set; 7,663 glyphs; Outline font ∼2.7 MB; Stroke font <250 KB [13]

[2]Big5 character set; 13,711 glyphs; Outline font 6—12 MB; Stroke font 250–650 KB [4]

[3]This includes the character radicals by which Chinese dictionaries are organized, but also other commonly occurring graphic elements.

[4]See the `glyf` table specification. Referred to as *compound glyphs* in Apple’s specification[3] and *composite glyphs* in Microsoft’s.

[5]See the `use` element.

Glyph	Outline	Stroke Path	Line Width	Line Caps
体	330	38		
使	378	56		
天	178	24		
木	190	20		
阮	332	56		
Total	1408	194	65	470

Table 9.1. Relative storage requirement for Chinese Ming typeface

Glyph	Outline	Stroke Path	Line Width	Line Caps
O	68	34		
P	52	26		
E	26	16		
R	74	38		
A	26	12		
Total	246	126	36	18

Table 9.2. Relative storage requirement for Latin sans serif typeface

sentation. As line widths and line caps are partially shared between glyphs only totals are shown.

The 5 Chinese Ming typeface glyphs were approximately 50% smaller in the stroke representation. This is not particularly impressive, but as the number of unique line widths and caps is fairly small, the storage requirement will be determined largely by the stroke paths as the number of glyphs grow, i.e. it should be closer to 14% of the size of equivalent outline glyphs.

A large majority of Chinese characters in modern use are composed from smaller components, which in turn can be used to form many other characters. If component reuse were employed to take advantage of this, the storage requirements of the stroke font would decrease even further.

The 5 Latin sans serif glyphs were only about 27% smaller in the stroke representation. Counting only the stroke paths, we find a best case reduction of approximately 50%. However, given the relatively small number of characters in total it is too optimistic to think that a complete sans serif typeface could be shrunk radically simply by using a stroke representation. For a serif typeface the potential savings are greater as the complexity of the serifs could be reduced to a small number of serif line caps.

Note that the above numbers are by no means exact, as storage requirements for the file format structure have not been taken into account.

9.2.2 Processing Requirements

	256 px		1 px		dummy	
Glyph	Outline	Stroke	Outline	Stroke	Outline	Stroke
体	7099	8142	2151	2836	842	1239
使	7339	9897	2544	3868	1024	1713
天	3941	4445	1193	1601	487	697
木	5011	5030	1263	1560	514	649
阮	6870	7469	2151	2959	852	1308
Total	30260	34983	9666	12824	3719	5606

Table 9.3. Processing time (μs) for Chinese Ming typeface

	256 px		1 px		dummy	
Glyph	Outline	Stroke	Outline	Stroke	Outline	Stroke
O	3309	3729	611	749	179	285
P	1946	2173	501	693	167	278
E	1142	1232	301	258	112	99
R	2669	3075	675	1103	230	453
A	1791	2177	281	753	120	385
Total	10857	12386	2369	3556	808	1500

Table 9.4. Processing time (μs) for Latin sans serif typeface

When rendering outline glyphs, we convert an outline *path* to an outline *polygon* by simple approximation. For stroke glyphs we instead convert each stroke to an outline polygon using a complex stroking algorithm. Therefore, we would generally expect that rendering stroke glyphs is slower, as there are extra conversion steps before filling. Table 9.3 and 9.4 show the processing times for rendering equivalent glyphs using outline and stroke representation. In order to effectively compare the two, the relevant differences must be isolated. By rendering the glyphs only 1 pixels high, the time spent filling polygons should be minimized. Further, the amount of JavaScript processing done for outline and stroke glyphs is different. To account for this, a "dummy" version of the renderer was created which removes all calls to the Canvas interface. The measured time should account roughly for the JavaScript overhead.

An approximation of the processing requirement for outline and stroke representation is given by subtracting the JavaScript overhead from the processing time for a 1 pixel glyph. Using this metric, we see that the processing time of Chinese stroke glyphs is approximately 21% longer than that of equivalent outline glyphs. Latin stroke glyphs are approximately 32% more demanding than the equivalent outline glyphs. Whether this increased processing time is acceptable or not is difficult to judge without comparing it to the other requirements of the system.

As the implementation and timing is done in JavaScript there are large uncertainties in the results. If stroke font support were more tightly integrated with the browser, the results would likely not be the same.

9.3 Editing

Even though the editor implemented in this project is very primitive, some advantages of the stroke representation are apparent. As it is very close to the natural way of writing, it has potential as an intuitive design tool. For example, if a more sophisticated editor could be used with a drawing tablet input device, strokes could be input simply by writing them. Advanced drawing tablets are pressure sensitive, so the width of the stroke could be correlated to the pressure. Further editing of the strokes is also simplified, as only the stroke center path need be manipulated.

9.4 Summary

On the whole, results are very promising for representing Chinese typefaces in VGSSF. Compared with outline representation, VGSSF is both more compact and more natural for Chinese characters, while the visual results are almost identical and the processing time is not dramatically increased.

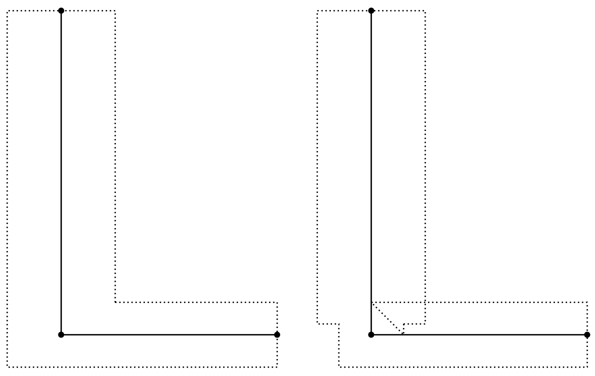
The results for the Latin typefaces is more troublesome. Size savings are not as big and there are subtle problems when fitting strokes together. Although the increase in processing time is not dramatic, it is still greater than for the Chinese typeface.

Chapter 10

Future Prospects

In the chapter we look at some possible future improvements to VGSSF (some of them critical) and possible uses of the technology it is based on.

10.1 Line Widths



(a) Desired result (b) Actual result

Figure 10.1. Problem of disassociated path and width

VGSSF as implemented in this project define line widths as linearly interpolated offset-width pairs, which is a sub-optimal for describing smooth line width variations. One could use some form of spline interpolation to solve this, but there is a much more subtle problem lurking beneath. If we want the line width to change exactly at a corner as in figure 10.1(a), we would need to express that point as an offset along the path, scaled to a value between 0 and 1. Except in very simple cases numerical imprecision would cause the width to change slightly before or slightly after the corner, giving the effect shown in the figure 10.1(b).

The core problem is that path and width are very weakly linked via the path offset, so merging them into the same data structure could solve the problem. Expressed as SVG path data, the above example might be written as `"M 0 30 10 L 0 0 10 L 0 0 6 L 20 0 6"`, where the third argument to each command is the line width. This could be expanded to include Bézier curves, thereby also solving the initial problem of smooth line widths.

10.2 Line Joins

In the previous chapter we saw that certain strokes in both Chinese and Latin fonts are joined by a shape which cannot be achieved with the standard line joins. One solution is to break such strokes into several shorter strokes with line caps carefully fitted together. A drawback is that changing the stroke width will cause the strokes to almost literally "fall apart".

The traditional stroke model has already been generalized to allow arbitrary line widths and line caps, is it then also possible to include arbitrary line joins? Line joins could be defined as open paths which are fitted between the two corner points much line caps already are. However, just a quick look at figure 9.3 reveals why this is not enough – the irregular structure is located between two unconnected paths and not merely in a corner of a path.

Another option would be something similar to SVG line markers, which allows placing arbitrary SVG content at start-, mid- and end-vertices of a path. These can be made to depend on the line width, solving the problem of strokes "falling apart" when line width is changed. A drawback is that line markers are separate paths which would not be merged into the outline polygon in the same way as line caps are. Finally, SVG line markers are set for *all* mid-vertices, whereas for stroke fonts it would be desirable to specify joins for each vertex individually.

10.3 Font Hinting

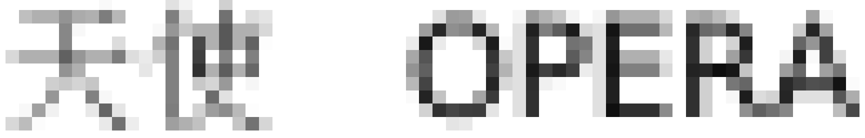


Figure 10.2. Strings rendered at 10 px without hinting

Since the target environment of the stroke fonts in this project is a web browser, it stands to reason that they will be used mainly to render text in the size range 8–16 pixels. Figure 10.2 clearly shows the problems with unhinted fonts – blurry and uneven glyphs with low legibility, caused by poor matching between the glyph coordinates and the target raster. Techniques to deal with this problem are called *hinting*, but are unfortunately very complex.

PostScript uses *declarative hints* to specify size and position constraints, glyph stems and other information about the glyphs.[1, Ch 5.1 Declarative Hints] A PostScript interpreter modifies the glyph based on these hints, but the exact result depends on the algorithms of the interpreter.

TrueType takes a more explicit approach [3, Ch 3 Instructing Fonts] by using *font programs* which modify points on the glyph outline depending on the font size. These programs can be generated automatically (auto-hinting), but for high-quality fonts they may also be written by hand. The rendered result should be exactly same result on any TrueType-compatible system[1].

Monotype Imaging have developed a hinting technology called SmartHint[12], which presumably works with their stroke-based fonts. It is especially tailored for Japanese, Chinese and Korean fonts in that it can adaptively remove strokes from characters depending on glyph size. However, very little about the implementation is revealed.

As evident, there are many options to consider when implementing hinting for a font format. One advantage of the stroke representation is that glyph stems are explicit. There is therefore no need be for explicit stem hints as in PostScript. It is however unlikely that fully acceptable results could be achieved using *only* automatic run-time hinting. Another important consideration is that if outline and stroke glyphs are allowed in the same format, then the complexity of the hinting system will probably increase as two paradigms must co-exist. The available options must be researched further, as hinting is necessary if VGSSF is to be useful on any low-resolution device, including desktop computers.

10.4 File Format

While (extended) SVG fonts has been excellent for testing new concepts, it is not a very compact file format. If small font files are required, a binary representation is critical. The greatest potential size saving would come from storing repeated glyph components only once, something which both Bitstream and Monotype Imaging stress as an important feature of their respective formats.

10.5 A Note of Caution

When asked in 1996 whether or not PostScript is a good enough replacement for Metafont, Donald Knuth was self-critical with regards to Metafont's complexity:[9]

> "Asking an artist to become enough of a mathematician to understand how to write a font with 60 parameters is too much. Computer scientists understand parameters, the rest of the world doesn't."

Many of the improvements suggested here – perhaps even some of the concepts already implemented – are fairly complex and it may not be reasonable to expect

[1]Due to patent issues some implementations cannot be fully compliant, e.g. FreeType. See `http://www.freetype.org/patents.html` for more information.

at font designer to understand or care about them. Knuth's words should be taken as a serious warning against making a font format too complex. Outline-stroke hybrid representation, arbitrary line joins and yet another hinting system are some of the things that could push a font format too far – into inevitable obscurity.

10.6 SVG Development

The stroke model and algorithm developed in this thesis has applications also outside the domain of fonts. The extensions made to both Canvas and SVG are general and can be used for any purpose. The draft requirements for SVG 2.0 state that "SVG should support definable stroke styles. Possible examples of defined styles are wave strokes, strokes with multiple lines and the brushes that are supported by many illustration packages."[28] Variable line width could be one part of these definable stroke styles. There have been minor discussion about how variable line width could be realized,[14, 10] but to our knowledge this thesis is the first systematic analysis of the problem and likely the first implementation (within the domain of vector graphics). The lessons learned in the process could be valuable in the future development of SVG.

Chapter 11

Conclusion

The extended stroke model and the stroking algorithm developed in this thesis have proven to be a sound foundation on which to build stroke fonts. The potential size reduction is encouraging, while the increased processing requirement are likely acceptable for most applications. Like existing stroke font technologies, VGSSF is especially well suited for East Asian typefaces, exemplified by the Chinese Ming typeface in this thesis. As the Chinese heiti typeface is possible to create using existing stroke font formats, it would also be possible using VGSSF. While designing Latin sans serif typefaces using VGSSF is possible, there are some problems due to the exact fitting of strokes and their caps necessary. These problems would likely be even more accentuated in a serif typeface, where serifs must be perfectly aligned horizontally or vertically.

This thesis has focused on the feasibility of VGSSF and has intentionally excluded many components which are necessary for real-world use, including a compact file format and font engine support. Equally critical for low-resolution devices is a hinting system which can guarantee text legibility even at small sizes. This is the most complex of the missing components, so future development of VGSSF ought to start by investigating possible approaches for this.

Bibliography

[1] Adobe Systems Incorporated. *Adobe Type 1 Font Format*, 1993. http://partners.adobe.com/public/developer/en/font/T1_SPEC.PDF.

[2] Adobe Systems Incorporated. *PostScript Language Reference, Third Edition*, 1999. http://partners.adobe.com/public/developer/en/ps/PLRM.pdf.

[3] Apple Computer, Inc. *TrueType Reference Manual*, 1996. http://developer.apple.com/textfonts/TTRefMan/index.html.

[4] Bitstream. Stroke-Based Fonts. http://www.bitstream.com/font_rendering/pdfs/strokedbasedfonts_whitepaper.pdf.

[5] CHENG Min and WANG Guo-jin. Rational offset approximation of rational Bézier curves. *Journal of Zhejiang University SCIENCE A*, 7(9):1561–1565, 2006. http://www.zju.edu.cn/jzus/2006/A0609/A060914.pdf.

[6] Donald Hearn and M. Pauline Baker. *Computer graphics with OpenGL - 3rd edition*. Pearson Prentice Hall, 2004.

[7] Elena J. Jakubiak, Ronald N. Perry, and Sarah F. Frisken. An Improved Representation for Stroke-based Fonts, 2006. http://www.merl.com/people/perry/SIG06_SSF_FinalSketch.pdf.

[8] Donald E. Knuth. *The METAFONTbook*. American Mathematical Society and Addison-Wesley Publishing Company, 1986.

[9] Donald E. Knuth. CSTUG, Charles University, Prague, March 1996 — Questions and answers with Prof. Donald E. Knuth. *TUGboat*, 17(4):355–367, 1996. http://www.tug.org/TUGboat/Articles/tb17-4/tb53knuc.pdf.

[10] Kevin Lindsey. Variable Width Stroke. http://www.kevlindev.com/geometry/2D/strokes/index.htm.

[11] Mitsubishi Electric Research Laboratories. Adaptively Sampled Distance Fields (ADFs). http://www.merl.com/projects/adfs/.

[12] Monotype Imaging. SmartHint Technology for East Asian Fonts. http://monotypeimaging.com/Markets/Smarthint.aspx.

[13] Monotype Imaging. Stroke Fonts. http://monotypeimaging.com/ProductsServices/stroke_fonts.aspx.

[14] Douglas Alan Schepers. Diff'rent Strokes. http://schepers.cc/differentstrokes.html.

[15] Eric W. Weisstein. Bézier Curve. http://mathworld.wolfram.com/BezierCurve.html.

[16] Eric W. Weisstein. Parallel Curves. http://mathworld.wolfram.com/ParallelCurves.html.

[17] Wikipedia. Asteroids. http://en.wikipedia.org/wiki/Asteroids_(computer_game) (accessed 2008-06-06).

[18] Wikipedia. History of typography in East Asia. http://en.wikipedia.org/wiki/History_of_typography_in_East_Asia (accessed 2008-05-16).

[19] Wikipedia. Movable type. http://en.wikipedia.org/wiki/Movable_type (accessed 2008-05-16).

[20] Wikipedia. PostScript. http://en.wikipedia.org/wiki/PostScript (accessed 2008-05-22).

[21] Wikipedia. Tektronix 4014. http://en.wikipedia.org/wiki/Tektronix_4014 (accessed 2008-05-22).

[22] Wikipedia. Woodblock printing. http://en.wikipedia.org/wiki/Woodblock_printing (accessed 2008-05-16).

[23] The World Wide Web Consortium. *Cascading Style Sheets, level 2 (CSS2) Specification*. http://www.w3.org/TR/REC-CSS2/fonts.html#font-descriptions.

[24] The World Wide Web Consortium. *CSS3 module: Web Fonts*. http://www.w3.org/TR/2002/WD-css3-webfonts-20020802/#font-descriptions.

[25] The World Wide Web Consortium. *Document Object Model*. http://www.w3.org/DOM/.

[26] The World Wide Web Consortium. *HTML 5*. http://www.w3.org/TR/html5/.

[27] The World Wide Web Consortium. *Scalable Vector Graphics (SVG) 1.1 Specification*. http://www.w3.org/TR/2003/REC-SVG11-20030114/.

[28] The World Wide Web Consortium. *SVG 1.1/1.2/2.0 Requirements*. http://www.w3.org/TR/2002/WD-SVG2Reqs-20020422/#req-stroke.

[29] Tom Wright. History and Technology of Computer Fonts. *IEEE Annals of the History of Computing*, 20(2):30–34, 1998.

Appendix A

Proof That Case Analysis Is Logically Exhaustive

This appendix proves that the case analysis in section 5.1 is logically exhaustive.

From the definition of the predicates before(L), on(L), and after(L) it follows that if there is an intersection, exactly one of the three predicates will be true, i.e. they are mutually exclusive. This can be stated in propositional logic as

$$\begin{aligned}&(\text{before}(L) \wedge \neg\text{on}(L) \wedge \neg\text{after}(L))\\ \vee&(\neg\text{before}(L) \wedge \text{on}(L) \wedge \neg\text{after}(L))\\ \vee&(\neg\text{before}(L) \wedge \neg\text{on}(L) \wedge \text{after}(L))\end{aligned}$$

This property is important in the following proofs. Especially note that

$$\begin{aligned}\text{on}(L) &\equiv \neg\text{before}(L) \wedge \neg\text{after}(L)\\ \neg\text{on}(L) &\equiv \text{before}(L) \vee \text{after}(L)\end{aligned}$$

A few important propositional calculus laws for reference:

Law of excluded middle	$\alpha \vee \neg\alpha \equiv \top$
Distributive laws	$\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$
	$\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$
De Morgan's laws	$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
	$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
Identity laws	$(\alpha \wedge \top) \equiv \alpha$
	$(\alpha \vee \bot) \equiv \alpha$

We first show that case 1–3 cover all cases where at there is an (extrapolated) intersection on either line segment.

Proof

$$
\begin{array}{lcl}
\textbf{(case 1)} & & (\mathrm{on}(L_i) \wedge \mathrm{on}(L_{i+1})) \vee \\
\textbf{(case 2)} & & (\mathrm{on}(L_i) \wedge \mathrm{before}(L_{i+1})) \vee (\mathrm{after}(L_i) \wedge \mathrm{on}(L_{i+1})) \vee \\
\textbf{(case 3)} & & (\mathrm{on}(L_i) \wedge \mathrm{after}(L_{i+1})) \vee (\mathrm{before}(L_i) \wedge \mathrm{on}(L_{i+1})) \\
& \equiv & (\mathrm{on}(L_i) \wedge \mathrm{on}(L_{i+1})) \vee \\
& & (\mathrm{on}(L_i) \wedge \mathrm{before}(L_{i+1})) \vee (\mathrm{on}(L_i) \wedge \mathrm{after}(L_{i+1})) \vee \\
& & (\mathrm{after}(L_i) \wedge \mathrm{on}(L_{i+1})) \vee (\mathrm{before}(L_i) \wedge \mathrm{on}(L_{i+1})) \\
& \equiv & [\text{distributive law}] \\
& \equiv & (\mathrm{on}(L_i) \wedge \mathrm{on}(L_{i+1})) \vee \\
& & (\mathrm{on}(L_i) \wedge (\mathrm{before}(L_{i+1}) \vee \mathrm{after}(L_{i+1}))) \vee \\
& & (\mathrm{on}(L_{i+1}) \wedge (\mathrm{before}(L_i) \vee \mathrm{after}(L_i))) \\
& \equiv & [\text{mutually exclusive}] \\
& \equiv & (\mathrm{on}(L_i) \wedge \mathrm{on}(L_{i+1})) \vee \\
& & (\mathrm{on}(L_i) \wedge \neg\mathrm{on}(L_{i+1})) \vee \\
& & (\mathrm{on}(L_{i+1}) \wedge \neg\mathrm{on}(L_i)) \\
& \equiv & [\text{distributive law}] \\
& \equiv & (\mathrm{on}(L_i) \wedge (\mathrm{on}(L_{i+1}) \vee \neg\mathrm{on}(L_{i+1})) \vee (\mathrm{on}(L_{i+1}) \wedge \neg\mathrm{on}(L_i)) \\
& \equiv & [\text{law of excluded middle; identity law}] \\
& \equiv & \mathrm{on}(L_i) \vee (\mathrm{on}(L_{i+1}) \wedge \neg\mathrm{on}(L_i)) \\
& \equiv & [\text{distributive law}] \\
& \equiv & (\mathrm{on}(L_i) \vee \mathrm{on}(L_{i+1})) \wedge (\mathrm{on}(L_i) \vee \neg\mathrm{on}(L_i)) \\
& \equiv & [\text{law of excluded middle; identity law}] \\
& \equiv & \mathrm{on}(L_i) \vee \mathrm{on}(L_{i+1})
\end{array}
$$

□

We further show that case 4–6 include all cases where at there is *not* an (extrapolated) intersection on either line segment.
Proof

$$\begin{array}{lll}
\textbf{(case 4)} & & (\text{after}(L_i) \wedge \text{before}(L_{i+1})) \vee \\
\textbf{(case 5)} & & (\text{before}(L_i) \wedge \text{before}(L_{i+1})) \vee (\text{after}(L_i) \wedge \text{after}(L_{i+1})) \vee \\
\textbf{(case 6)} & & (\text{before}(L_i) \wedge \text{after}(L_{i+1})) \\
& \equiv & (\text{before}(L_i) \wedge \text{before}(L_{i+1})) \vee \\
& & (\text{before}(L_i) \wedge \text{after}(L_{i+1})) \vee \\
& & (\text{after}(L_i) \wedge \text{before}(L_{i+1})) \vee \\
& & (\text{after}(L_i) \wedge \text{after}(L_{i+1})) \\
& \equiv & [\text{distributive law}] \\
& \equiv & (\text{before}(L_i) \wedge (\text{before}(L_{i+1}) \vee \text{after}(L_{i+1}))) \vee \\
& & (\text{after}(L_i) \wedge (\text{before}(L_{i+1}) \vee \text{after}(L_{i+1}))) \\
& \equiv & [\text{mutually exclusive}] \\
& \equiv & (\text{before}(L_i) \wedge \neg\text{on}(L_{i+1})) \vee \\
& & (\text{after}(L_i) \wedge \neg\text{on}(L_{i+1})) \\
& \equiv & [\text{distributive law}] \\
& \equiv & \neg\text{on}(L_{i+1}) \wedge (\text{before}(L_i) \vee \text{after}(L_i)) \\
& \equiv & [\text{mutually exclusive}] \\
& \equiv & (\neg\text{on}(L_{i+1}) \wedge \neg\text{on}(L_i)) \\
& \equiv & [\text{De Morgan's law}] \\
& \equiv & \neg(\text{on}(L_i) \vee \text{on}(L_{i+1}))
\end{array}$$

□

As cases 1–3 and 4–6 are each other exact complements, they cover all cases where an intersection exists. Given that case 7 (no intersection) is also included, the case analysis is logically exhaustive.