

Font Rasterization for Mobile Devices

Magnus Andersson, D03
d03man@student.lth.se

Supervisor:
Tomas Akenine-Möller

Lunds Tekniska Högskola
June 27, 2008

Abstract

Writing memory efficient and fast text rendering for mobile devices through the use of vector graphics can prove quite a challenge. This is the premise for this report. Two widely different vector based rasterizer approaches, a triangle- and a plane sweep-rasterizer, were examined in terms of memory efficiency, speed and rendering quality. To enable these measurements, the rasterizers were implemented in C and compared to each other, as well as to the leading open-source rasterizer found in FreeType.

It was found that the triangle rasterizer always used less memory than FreeType for font sizes smaller than 60 pixels. The plane sweep rasterizer, in turn, was about 10-15% faster than FreeType, given the right parameters, when test run on a desktop computer. Depending on the demands, either of the rasterizers described in this report can be useful for mobile systems, although the triangle rasterizer will probably benefit from the use of new technology such as hardware acceleration on mobile devices in the future.

Contents

1	Introduction	5
2	Background	7
2.1	Brief Explanations	7
2.2	Numerical Considerations	8
2.2.1	Bézier Curves	8
2.2.2	Converting Higher Order into Quadratic Bézier Curves	10
2.2.3	Forward differencing	10
2.2.4	Recursive Subdivision	12
2.2.5	Interval Arithmetic	14
2.3	Key Concepts of Outline Fonts	15
2.4	Representing Glyphs	16
2.4.1	Outlines	16
2.5	Complex scripts	18
2.6	Outline font formats	19
2.7	Previous work	21
3	The Scanliner	25
3.1	Theory	25
3.2	Implementation	29
4	The Xorizer	35
4.1	Theory	35
4.2	Implementation	38
4.2.1	Trivial implementation	39

4.2.2	Plotting glyphs	41
4.2.3	Bit-counting algorithms	43
4.2.4	Sampling scheme	43
4.2.5	Optimizations	44
4.3	Anchor Point Placement	49
5	Improving Rendering for Small Font Sizes	51
5.1	Sub-pixel rendering	51
5.2	Hinting	53
5.2.1	Stem adjustment/Grid fitting	54
5.2.2	Drop out control	55
5.3	Font styles	55
5.3.1	Italic	56
5.3.2	Bold	56
5.3.3	Other styles	57
6	Results	59
6.1	Memory Usage	59
6.2	Output	60
6.3	Rendering speed	62
6.4	Discussion	65
6.4.1	Overdraw	65
6.4.2	Glyph Representation Issue	66
7	Conclusions	67
7.1	Future Research	67
7.2	Keep It Simple	68

Chapter 1

Introduction

Not long ago mobile phones had very poor screen resolution and could only display very few colours. There was no need for anything but bitmap fonts since text was only used in menus and in simple text messages, none of which require scalable text or fancy effects. But mobile devices quickly become faster and screen resolution and colour depth increases.

This opens for a lot of new possibilities such as web browsing and more advanced word processing on the phone. These features require some sort of solution where text can be freely scaled to any size or have attributes such as italic, bold, colour, and so on.

If bitmap fonts would be used for these purposes then a vast number of images for every single character would be needed. It's a much better idea to represent the glyphs with vector based graphics instead of bitmaps.

Such solutions can be found in many embedded devices today such as the Apple iPod and various high-end mobile phones for example. Many of them use a very well established open-source system called FreeType. However, that solution only handles rasterization and no layout issues such as line breaks, writing direction, text formatting and so on. FreeType can be used together with Pango, which is a, so called, *layout engine* under the LGPL license.

The result of this thesis is intended to be used within a new text rendering solution. Mainly the rasterizer step, which converts a vector representation of a glyph into a bitmap with desired attributes, will be investigated. Therefore, this report is not concerned with the issues of text layout, but concentrates on some new ideas for the rasterization, even if we will extend beyond this frame somewhat. Namely, we will also touch on some important improvements that increases readability for small print.

The primary goal of the thesis is to determine if these new ideas could be used in a particular font rendering solution software for complex scripts

– *v-rocs* [17]. The main purpose of this piece of software is to apply linguistic rules and correctly rendered strings for complex scripts such as Indic scripts, Arabic scripts, Thai, and so on. There are a number of such solutions on the market today, but it has been found that many of them contain errors, and are not used by the native speakers.

Many existing font formats hold their own linguistic instructions that are interpreted by some font rendering engines. In Microsoft Windows this rendering engine is called Uniscribe. Linux and other open source systems commonly use Pango.

There are limitations in the font formats and these rendering engines make erroneous assumptions about the linguistic rules. Thus, complex scripts are often not presented correctly linguistically (Uniscribe does a particularly poor job). *v-rocs* tries to remedy this by offering its own tailored solution for each complex language, moving most of the linguistic intelligence from the font files to code modules. This also means that several fonts can share the same logics, but any logic that is already within a font file is lost.

Because *v-rocs* uses a different approach a new outline font format has been tailored for its purposes. This format will be looked at briefly, because it changes the requirements of the rasterizer a little, compared to what a TrueType rasterizer would need to handle.

Until now *v-rocs* has only offered bitmap fonts, and the next natural step is to offer outline font support as well. A good rasterizer combined with the linguistic superiority of *v-rocs* should make it a very strong solution. Because it's intended to be used in mobile phones with sparse memory and limited processing power, there is a need for a rasterizer that satisfies both these demands.

Chapter 2

Background

Before we dive into the rasterization algorithms it's important to get some insight in some of the mathematics used and how outline font formats work. Also, a brief history of rasterizers can be found at the end of the chapter.

2.1 Brief Explanations

This report assumes that the reader has some experience in graphics programming. Some brief explanations of some concepts are presented here as a repetition of terms that are crucial for understanding the methods described in this report.

Unicode

A standard which allot numbers for glyphs in different world scripts. Not all glyphs are in the Unicode standard, but all that are have a unique number. More info at [14]

Edge function

Edge functions are given by the vertices of a triangle [16]. Represented as $ax + by + c$. Evaluating the expression for a point gives < 0 if the point is “outside” the edge, > 0 if it's “inside” and $= 0$ if it's directly on the edge. Which side is the “inside” and the “outside” depends on the winding of the vertices defining that edge.

Barycentric coordinates

Barycentric coordinates for triangles are the weights of the vertices for a point in the same plane as the triangle.

Texture coordinates

Texture coordinates are (usually) given at vertices and are often denoted by (u, v) . Using barycentric coordinates the values are interpolated over the triangle. The coordinates map to a 2d-image, and thus the triangle can be textured.

Sampling scheme

A sampling scheme is a pattern of sample points within a pixel. Sample points are evaluated against the edge functions of the triangle to determine if it's inside or outside that triangle.

Affine transformation

An affine transformation preserves co-linearity and proportion between points but does not preserve angles and lengths. Affine transformations include sheering and scaling. Euclidean transformations, such as translation and rotation, form a subset of affine transformations. We use the following translation matrix for affine transformations:

$$\begin{pmatrix} s_x & r_y & t_x \\ r_x & s_y & t_y \end{pmatrix} \quad (2.1)$$

2.2 Numerical Considerations

2.2.1 Bézier Curves

A *Bézier curve* is a smooth curve described by a polynomial with certain characteristics. The polynomial can be of any degree > 2 and for a given degree n it's given by:

$$B(t) = \sum_{i=0}^n \binom{n}{i} P_i (1-t)^{n-i} t^i, \quad t \in [0, 1]. \quad (2.2)$$

If we let t wander from 0 to 1 then the function $B(t)$ will assume all coordinate values along the curve. The curve appearance is given by the points P_i , where $i \in [0, n]$. Note that the number of points is one more than the degree of the curve.

The points P_0 and P_n are called *end points* and are the only points guaranteed to coincide with the curve. They do so for $t = 0.0$ and $t = 1.0$ respectively. The points in between the end points are *control points* since they control the behavior of the curve.

The simplest Bézier curve is a straight line, and is best described by two points. When varying t over $[0, 1]$, we get something that is the analogue of linear interpolation.

Introducing a third point causes some curvature to the line segment (unless all three points lie on the same line). This is a *quadratic Bézier curve* and from equation 2.2 we see that it can be described by the following function:

$$B(t) = (1 - t)^2 P_0 + 2t(1 - t)P_1 + t^2 P_2, \quad t \in [0, 1]. \quad (2.3)$$

P_0 and P_2 are control points and determine where the curve should begin and end. P_1 is the only control point of this curve and it decides the bending of the curve.

An easier way of visualizing the behavior of a quadratic Bézier curve can be found in figure 2.1.

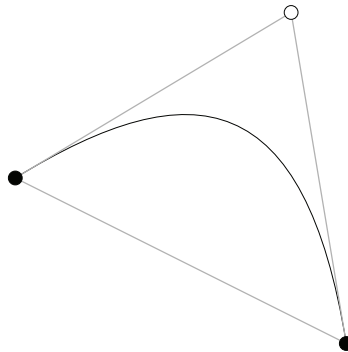


Figure 2.1: A quadratic Bézier curve. The two black dots are end points and the white one is the control point which determines the curvature.

The *bounding triangle* of a quadratic Bézier curve is the triangle defined by its three points. The curve can never go outside this bounding triangle, so if a point is outside the triangle, it can't be a part of the curve (for $t \in [0, 1]$).

We also define the *bounding box* to be the smallest axis-aligned rectangle to include all three points of the bounding triangle. The curve is of course always contained within this rectangle as well.

2.2.2 Converting Higher Order into Quadratic Bézier Curves

For reasons that become clear in section 2.6, we only need to be concerned with quadratic curves in the real time rendering process. Curves that are of higher order can be converted into quadratic sub-curves and saved as such in the font file.

The procedure for doing this is not entirely trivial and will only be described briefly. The following steps are generally used:

Approximate the higher order curve with a quadratic curve. Use some error measure to see how well the approximation fits the original curve. If the result is good enough then we're satisfied. If the approximation is too coarse, subdivide the curve into two sub-curves and repeat the process for the new curves. This is done recursively until the approximation is good enough.

The procedure with error measurement is described in the article [18].

2.2.3 Forward differencing

There are a number of ways to evaluate Bézier curves. Most of the resources found on the web seem to endorse the de Casteljau algorithm for this purpose. This algorithm will be presented in the next section. First we will take a look at another algorithm called *forward differencing*.

Kankaanpää has a good description of this algorithm on his page [6]. It tackles cubic Bézier curves, but we're only interested in evaluating quadratic curves, so we stray a little from his description.

Consider the Taylor series for the curve polynomial. Since we are dealing with quadratic curves, all terms above and including the third derivate of the curve polynomial will be 0. This leaves us:

$$B(t) = B(x) + B'(x)(t - x) + \frac{B''(x)(t - x)^2}{2!} \quad (2.4)$$

What we want is the next following point, which is given by $B(x + t)$. Inserting this into the Taylor series gives:

$$B(x + t) = B(x) + B'(x)t + \frac{B''(x)t^2}{2} \quad (2.5)$$

Notice that we now have all right hand side $B(x)$ functions and derivatives free of t 's. We can now calculate these for $x = 0$ (the start point), and doing so yields:

$$\begin{aligned} B(0) &= P_0 \\ B'(0) &= 2(P_1 - P_0) \end{aligned}$$

$$B''(0) = P_0 - 2P_1 + P_2$$

All of these functions are constant, but in the Taylor series $B'(0)$ is still multiplied with t , and $B''(0)$ with t^2 .

When we evaluate the curve we increase t with the same constant ($\in [0, 1]$) each iteration. This means that we can calculate the values with which we need to increase the different terms each iteration. We call this incrementation value $s = \frac{1.0}{steps}$. The *steps* value is the number of points we want to evaluate along the curve. This gives the following initial functions:

$$\begin{aligned} C &= P_0 \\ C' &= 2(P_1 - P_0)s \\ C'' &= \frac{s^2(P_0 - 2P_1 + P_2)}{2} \end{aligned}$$

To update the evaluation value $B(x + t)$ each iteration we then do the following:

$$C(x + n \cdot s) = C + C' + C''$$

Where n is the iteration number. We also need to update the other terms. Deriving the Taylor series in equation 2.5 gives:

$$B'(x + t) = B'(x) + B''(x)t \Rightarrow B'(x + t)t = B'(x)t + B''(x)t^2 \quad (2.6)$$

Note that the third order derivation will be 0. Also note that the last term is no longer divided by 2. We derive again and get:

$$B''(x + t) = B''(x) \quad (2.7)$$

This means that this term is constant, since t doesn't affect it. To sum up, from these equations we see that for each iteration we need to do the following work:

$$\begin{aligned} C &= C + C' + C'' \\ C' &= C' + 2C'' \end{aligned}$$

where s is the incrementation step introduced earlier. We need to multiply C'' by 2 since the last term in equation 2.6 is twice as big as the C'' we calculated.

By using this algorithm we only need multiplications to set up the C-variables. After that only additions are needed.

Important to note about this particular approach is that we have no control over how the points are distributed over the curve, only that they

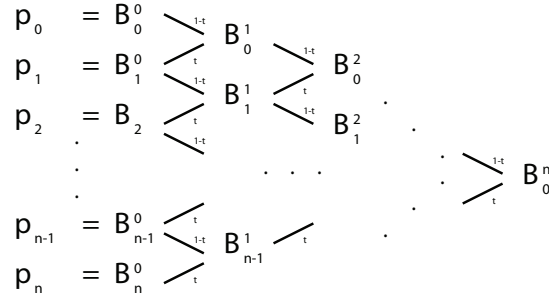


Figure 2.2: The de Casteljau pyramid. This scheme is used to get an overview of the weights to get a curve point for a certain t . n in this figure is the degree of the curve. For a quadratic Bézier curve, there will only be three levels in this pyramid (which builds horizontally).

have even spacing for the t value. This means that the algorithm is sensitive to velocity changes in the curve. This could be a good thing, since most often one would like more points where there is more “action” going on with the curve, but in reality the algorithm is simply just a little less flexible.

The approach presented in the next section is recursive, and points could be evenly spaced, or according to velocity, depending on what one desires.

For some cases it's practical to use the Forward Differencing method, and in other cases the de Casteljau algorithm, or Recursive Subdivision, could be better, as we will see in the next section.

2.2.4 Recursive Subdivision

The de Casteljau algorithm uses linear interpolation for an arbitrary number of t 's, where $t \in [0, 1]$. It's common to use the scheme in figure 2.2 to get an overview over how different points weigh in to produce the final evaluation result for a certain t . For each entry a new value is interpolated, except for the first where the original end- and control points are. The new value is given by:

$$B_j^{i+1} = B_j^{i-1}(1-t) + B_{j+1}^{i-1}t \quad (2.8)$$

The final value of the pyramid is the point on the curve for the current t .

But to interpolate like this is quite costly, and instead we use recursive subdivision to get our curve points. We know that $t = 0.5$ for a quadratic Bézier curve gives:

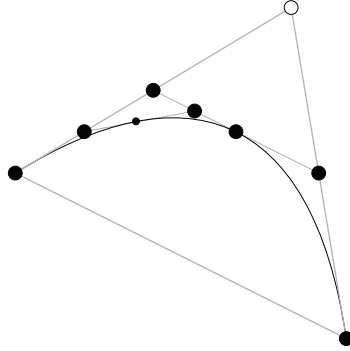


Figure 2.3: Subdivision using the recursive subdivision. Two recursive iterations of the recursive subdivision. In the first iteration the point on the top of the curve is found. The second iteration subdivides the left side of the curve and finds a new point on the curve that lie between the previously found point and the start point. The process can be repeated for each sub-curve until, for example, the desired amount of points are found, or until all sub-curves are smaller than a certain length.

$$\begin{aligned}
 t &= 0.5 \\
 (1 - 0.5)P_0 + 0.5P_1 &= Q \\
 (1 - 0.5)P_1 + 0.5P_2 &= R \\
 (1 - 0.5)Q + 0.5R &= S
 \end{aligned}$$

S is thus the value at the peak of the curve. Now that we have this point, we can create two sub-curves. One will be the curve defined by the bounding triangle P_0, Q, S and the other S, R, P_2 . We repeat the process for each sub-curve if we feel like it. The termination condition is usually when too many recursive calls have been made, or when a sub curve is small enough to be approximated by a line.

Two recursion steps of this algorithm is visualized in figure 2.3.

For floating point calculations this algorithm can become slow since it requires a multiplication by 0.5 for each recursive call. These calls can also slow things down a bit if the curve is large and/or there are a lot of points to evaluate. But if we use fixed point math the middle point can be calculated with one addition and one bit shift, which is often a great deal faster.

Both Recursive Subdivision and Forward Differencing are tried in the Scanliner algorithm. They have both been implemented using fixed point math. Read more about this under the Results chapter.

2.2.5 Interval Arithmetic

Let's denote an interval delimited by \underline{a} and \bar{a} as

$$\hat{a} = [\underline{a}, \bar{a}] \quad (2.9)$$

Furthermore, let x be a value such that $x \in \hat{a}$. We call x an interval variable.

If we were to add two of these interval variables together the result will be a value within the combined interval, as described by:

$$\hat{a} + \hat{b} = [\underline{a}, \bar{a}] + [\underline{b}, \bar{b}] = [\underline{a} + \underline{b}, \bar{a} + \bar{b}] \quad (2.10)$$

Addition and subtraction operations are inexpensive to implement in software. Multiplication, however, requires a lot more processing power and is defined as:

$$\hat{a} \cdot \hat{b} = [\min(\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b}), \max(\underline{a}\underline{b}, \underline{a}\bar{b}, \bar{a}\underline{b}, \bar{a}\bar{b})] \quad (2.11)$$

From this it can be determined !deduced that four multiplications and three comparisons are needed to determine the low and the high bound of the interval.

If we know that $\hat{a} = \hat{b}$ the interval would be:

$$\hat{a}^2 = [\min(\underline{a}^2, \underline{a}\bar{a}, \bar{a}^2), \max(\underline{a}^2, \underline{a}\bar{a}, \bar{a}^2)]$$

But since we're squaring we don't have to worry about negative values, and the $\underline{a}\bar{a}$ -term can never be neither greater than nor lower than the other two possible terms. This results in the following:

$$\hat{a}^2 = \begin{cases} [0, \max(\underline{a}^2, \bar{a}^2)] & 0 \in \hat{a}^2 \\ \min(\underline{a}^2, \bar{a}^2), \max(\underline{a}^2, \bar{a}^2) & 0 \notin \hat{a}^2 \end{cases} \quad (2.12)$$

For these squaring operations we need only two multiplications and two comparisons - a lot more inexpensive than the regular interval multiplication.

Interval arithmetic will become useful when evaluating Bézier curves in the Xorizer algorithm. There is, of course, a lot more to be said about this technique. For further information on the subject see [2].

2.3 Key Concepts of Outline Fonts

Glyphs

A *glyph* is most often a character such as a simple **a** or a σ . It can also be the two dots above **ä** or a blank character such as space.

Associated with a glyph is an outline, and some glyph metrics that are described below. The outline is what represents the shape of the glyph. It is composed of contours and they, in turn, are described by lines and curves and are infinitely scalable. This is described in more detail in section 2.4.1.

Composite glyphs

Let's assume we have the Latin characters **a** and **o** in our font, and that we want to add the characters **ä** and **ö**. It's obvious that these glyphs are variations of some common components - the bodies and the two dots.

One way of adding the two new glyphs to our font would simply be to draw two new glyphs for these characters. It's redundant to keep the same glyph information in several glyphs. A better alternative is to use a feature called *composite glyph*.

A composite glyph is a combination of glyphs that already has outline data in the font file. Each component has an affine transformation as well, so they can be positioned and scaled individually. For example, it's only necessary to keep the glyphs **a**, **o**, and **..** in the font, and then compose the two new glyphs with the existing pieces.

Glyph metrics

Not all glyphs have a body, and some glyphs may have special advancing requirements for the cursor and the beginning of the next glyph. This doesn't necessarily need to correspond to the actual width of the glyph. For these purposes a special width called *advance width* is introduced. This metric is useful for blank spaces and the like. The advance width is usually snapped to the pixel size of the current text rendering or glyphs after the first one would cause a sub-pixel offset.

The *offset* values and the *size* are more important metrics for the rasterizer. The size ensures that no excess memory needs to be allocated. Not all glyphs start at the origin of its glyph space, so the offset is useful to avoid rendering any white space before of the glyph body.

2.4 Representing Glyphs

2.4.1 Outlines

An *outline* is defined as a set of closed curves, or *contours*, in the plane. Each contour is made up of segments which are either straight lines or quadratic Bézier curves. The segments are delimited by points. The points are numbered in succession, starting at zero. The order in which the points are numbered correspond to the path that defines the shape of this contour. The first point is also the last and thus the contour becomes a closed curve.

The points can be either of the type *on-curve* or *off-curve*. On-curve points coincide with the actual curve, while off-curve points need not do so. Using only on-curve points results in an outline with only straight lines. Off-curve points are used to define smooth curvatures. Two end points of type on-curve, and an off-curve control point define the control triangle for a quadratic Bézier curve contour segment.

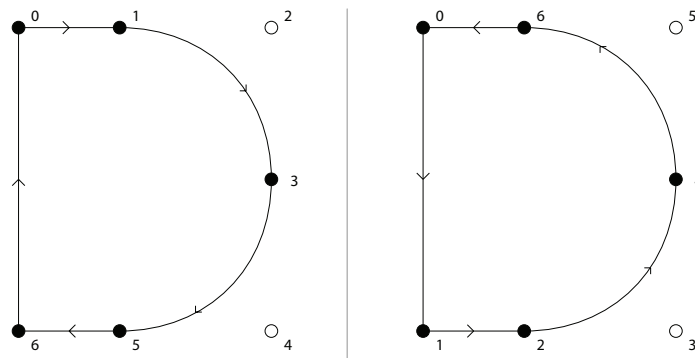


Figure 2.4: Contour winding. The left contour encloses the finite area within the figure. The right contour encloses the infinite area outside the figure, and needs to be closed by an outer contour.

At this point, as far as we know, the same path can be represented in two ways, each with different winding (see figure 2.4). We define different things depending on the enumeration order. An area enclosed by a contour we define as the area on the “right” hand side when facing the path direction. It is therefore important to use the correct winding when designing glyphs.

A contour with counter-clockwise winding encloses an infinite area and must be limited by an outer contour with the opposite winding. Think of the inner ring of the glyph O for instance - it needs to be “closed” by the outer ring in order to have a finite area.

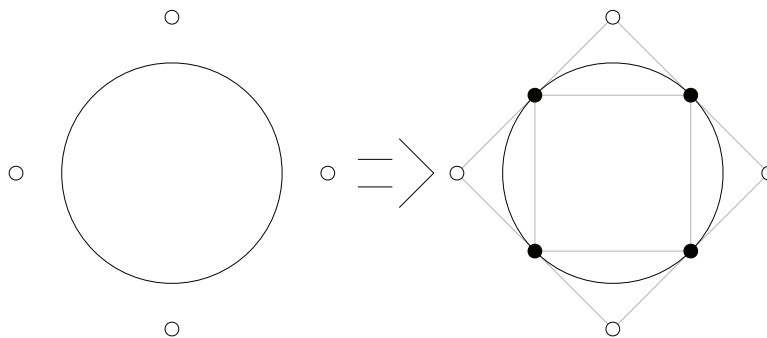


Figure 2.5: Circle approximation representation. An approximation of a circle can be described by four points. The four on-curve points can be calculated from the off-curve point coordinates, as they lie in the exact middle of the two surrounding off-curve points.

For one particular sequence of outline points, we don't need to store all the point coordinates. Instead it can be extracted from the adjacent points, thus we can save some storage space. This case occurs when an on-curve point lies in between two off-curve points. It is assumed that on-curve points lie in the exact middle of the two off-curve points. Therefore the on-curve coordinate information becomes redundant. This means, for example, that it is possible to define an entire contour with off-curve points (see figure 2.5).

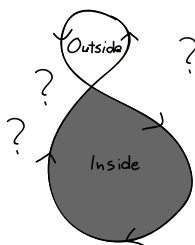


Figure 2.6: Ambiguous area definition. The inner and outer areas are as such because of the winding of the enclosing outline. But what is the outer, infinite area?

A contour should not cross its own path. Doing so will result in the contour both closing and opening some of the same areas at the same time, resulting in rather nasty ambiguities (figure 2.6). An area enclosed by contours can only be either inside or outside the outline. This should not be violated, even if some rasterizers may be able to handle these cases.

Contours may cross each others paths, there are no restrictions to this. An enclosing contour can never be "opened" by another contour, which

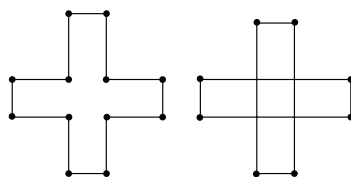


Figure 2.7: Plus signs defined differently. Plus sign defined in two different ways. The right plus will not be rendered correctly with an even-odd rasterizer, though such rasterizers are insensitive to winding.

makes it possible to define a plus sign in either of the ways described in figure 2.7.

A valid outline is composed of one or more contours, and has finite area. I.e. the contours make up an area entirely within the bounding box of the outline points of the glyph.

However, not all fonts follow these standards set by the TrueType format [10], and reverse the winding of some contours. In turn, some rasterizers instead follow the even-odd convention. This means that they only count the number of line crossings to determine whether a point is inside the shape or not. The result of this is that the winding no longer is an issue, but the right plus sign in figure 2.7 will not be rendered properly since the middle square will be outside the shape according to the even-odd rule.

The Xorizer presented in this report follows the even-odd rule, and the Scanliner doesn't.

2.5 Complex scripts

For the moment, v-rocs mainly targets Indic complex scripts. There are quite a lot of rules applied to these languages which differs quite a lot from the input characters, as opposed to Latin scripts where the rendered output string is more or less exactly the same as the input string.

Rules that need to be considered when writing Indic scripts include:

Mapping

Certain combinations of glyphs should be replaced by one or a few other glyphs. These combinations are called *compound characters*. Typical examples can be viewed in figure 2.8. Compound characters are very common in these scripts, and obviously the font needs to contain special glyphs for these cases.

Reordering

Some glyphs should sometimes be placed before certain other glyphs, even though they are both typed and pronounced after the other glyph(s). This concerns mainly some vowels that should be placed before consonants or the whole consonant part of a syllable. Examples of this can be found in figure 2.8.

Positioning

There are two different ways to reposition glyphs. One is called *kerning* and is analogous to the contraction of white space that is sometimes done in the glyph sequence **A V** -> **AV** for example. In Indic scripts it is used when there are certain parts of a preceding or succeeding glyph that should overlap the space of certain other glyphs.

The other kind of positioning is done using attachment points where a marker on a glyph is mapped to another marker on a target glyph. Through this, different glyphs can be combined with each other to produce different meanings. This is better explained visually in figures 2.8 and 2.9.

For positioning there are sure to be overlaps in glyph shapes. Though it might not seem like it at first, this could be somewhat problematic. How this should be handled is discussed in section 4.2.2.

2.6 Outline font formats

As mentioned in the Introduction, there are some special circumstances when selecting which font format to use.

TrueType and OpenType are the most common font formats used today. TrueType is quite an old format, and it contains a lot of information that is simply not needed for our purposes. Instead it would seem like a good idea to write a new format which only contains information relevant to v-rocs and things that could aid the rasterizer.

The OpenType outline point data comes in two flavors - TrueType or CFF. The former only supports quadratic Bézier curves, whereas the latter can have cubic Bézier curves as well. The TrueType flavoured variant is by far the most commonly used. It is also much simpler to process in real-time. Therefore only quadratic Bézier curves are considered, but a way of dividing higher order curves into second order curves is presented in section 2.2.2. Because a custom font format is used for this thesis, any such curve splitting could be done offline and would of course not affect real-time performance nor appearance in any way.

ऑस्ट्रेलियाई

1. ऑ स ्ट र े ल ि य ा ई
2. ऑ स् ट ् े ल ि य ा ई
3. ऑ स्ट्र े ल ि य ा ई
4. ऑ स्ट्रे लि या ई
5. ऑस्ट्रेलियाई

Figure 2.8: A Hindi word written with the Indic Devanagari script. Note that the dotted circles are not part of the glyphs. They only indicate that something to combine with is “missing” where the circles are. 1. The input Unicode string that composes the word. 2. Through mapping rules the 5th and 6th glyphs are combined to a new below-base accent. The 3rd glyph is positioned below the second glyph through positioning rules. The second glyph always positions below-base accents below its right stem, and the positioned glyph has an attachment point above its shape, as can be seen in figure 2.9. 3. The 2nd, 3rd and 4th glyphs are combined. Note that the stem is removed from the 2nd glyph. This happens when two consonants are connected. Again we’ve used mapping and positioning rules. 4. An above base glyph has been positioned above the cluster formed in the previous step. Also, the 4th and 5th glyphs from the previous step has been reordered, and then kerned in order to form the 3rd cluster in the current step. 5. All clusters, or syllables, have now been formed, and the word has been rendered using a variety of mapping-, reordering- and positioning rules.

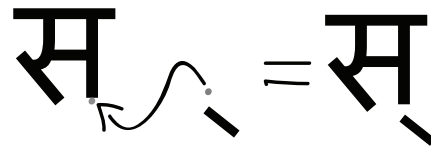


Figure 2.9: Positioning of a below-base dependent vowel. The base consonant (left) has an anchor point. The vowel sign (right) has an attachment point. These two glyphs are aligned by moving the attachment point to the anchor point, and the vowel with it, to produce a combined syllable.

More or less, the only difference between OpenType and TrueType, other than the glyph outline flavouring, is that the OpenType format has introduced a lot of new tables with linguistic rules. These aren't interesting to us in any way, neither linguistically nor for the rasterization process, so we won't even bother with these tables. In conclusion, the OpenType format offers no added value to us.

Other than the mentioned issues, TrueType is a sound format, and by far the most widely used, so it seems like a good idea to base our format on it. And since most OpenType fonts use TrueType "flavouring", these can be used as well.

We introduce our own format which we call VOFF - v-rocs Outline Font Format. Here we have some freedom to stir a little in the data tables to suit our needs better. The format is basically a new branch from TrueType, but the fact that we have our own format means that we don't strictly have to follow the TrueType standard when it isn't optimal. Some of these cases are mentioned in the report. The cubic-to-quadratic offline conversion is one such case.

2.7 Previous work

Over the years there have been many rasterizers for outline fonts and vector graphics in general. They have been needed for almost as long as computers have had the capability to display graphics. The target platforms have also varied a lot, so rasterizers tend to be rewritten to suit new demands and limitations.

Basically, there are two main approaches to produce a bitmap from an outline, be it a glyph or any vectorial shape. One is the *triangle rasterizer* and the other is the *scanline rasterizer*.

The most common rasterizer type is the scanline rasterizer (also known as plane sweep rasterizer). Since this method is so commonly used, the general solution will be described, and certain characteristics for some specific rasterizers will be mentioned.

The idea here is to scale the shape to the right size in screen space and then fill one pixel row, or scanline, at a time. Usually intervals of covered pixels are identified, and then the intervals, or *spans*, are plotted onto the screen. One particular of the FreeType rasterizers (it has many) does exactly this. It's simply called *ftgrays*, and this is the fastest and most exact of the FreeType rasterizers.

It decomposes the glyph into horizontal spans using the winding of the outline points. It divides the outline into two sets of curves, ascending

and descending. The bitmap is created by sweeping and filling the spans. Due to the winding of the curve outline, crossing an ascending curve implies that filling should start, and crossing a descending curve means filling should stop.

If the even-odd rule is used, there will not even be any use for differentiating between ascending and descending lines. If the intersections are sorted primarily on scanline, and secondarily on x-value, every odd crossing enters the shape (puts the pen down for drawing) and every even crossing leaves it (lifts the pen). This FreeType rasterizer does not follow this rule though.

For Bézier curve segments, subdivision using the de Casteljau algorithm is done recursively until the subdivided segments are no taller than a single scanline.

Another possible solution would be to use forward differencing. For quadratic Bézier curves it requires only a few multiplications for the setup, and then the curve can be divided into line segments using only additions. There is a problem with numerical stability, however, since the rounding error is magnified for each iteration of the algorithm.

The real issue with these kinds of algorithms is the pixel coverage problem. How do you fast and efficiently determine the colour of the semi-covered pixels, the ones that intersect the outline?

Monochrome (1 bpp) rasterizers of course get away with simpler pixel coverage checks since they only need to determine if a pixel is intersected by the shape. This can be done more or less conservatively (figure 2.10). The convention, though, is to use the pixel centre as the sampling point. Monochrome renderers have a hard time coping with small shapes since details and narrow structures are easily missed. Therefore, it's quite common to use certain methods to grid-fit and reshape the outline before rendering. Such methods can be important in font rendering, and are discussed in section 5.2.

Today FreeType is without any doubt the most used rendering solution by many different products (outside of the Windows environment of course). It includes a couple of different rasterizers and they are all variants of the scanline approach.

Another rasterizer on the market is the D-Type Font Engine, which claims to have the fastest rasterizer in the world, and supports both the even-odd rule and the winding rule. Little information on this product is revealed though, and it is even unclear if it uses this scanline approach, or some sort of triangle rasterization.

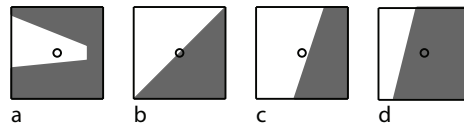


Figure 2.10: Difficult cases for monochrome rasterizers. It's not always easy to determine if a pixel should be lit or not. In a) most of the pixel is covered, yet if the center is sampled this pixel won't be coloured. In b) we land exactly on an edge. Exactly half the pixel is covered, so how should this be handled? In c) and d) the outcome is a bit easier to predict, yet if we don't colour c), is there perhaps a chance that the neighbouring pixel to the right will not be coloured either, and will we then end up with a gap in our glyph?

Triangle rasterizers are usually most useful in hardware accelerated rendering, where triangles are very quickly processed in hardware.

A triangle rasterizer is not unlike the scanline rasterizer. One can even argue that the triangle rasterizer is a scanline rasterizer with the limitation that it can only handle triangles and not arbitrary polygons. There is truth to this; the triangle is (most often) traversed and drawn scanline by scanline. However, knowing that we are dealing with only triangles opens certain opportunities but sets some restrictions, as will be come apparent in this report.

It's not that common to use triangle rasterizers in font rendering solutions. Mostly these rasterizers are used for games and simulations where a lot of information is stored and interpolated for the triangles.

Text has no texture so when glyphs are drawn using triangles we only need flat shading, so texture coordinates should not be needed. However, in certain algorithms they can be used to speed up rendering. Loop and Blinn [11] suggest rendering glyphs on the GPU. Their method require that the glyph is tessellated into triangles first, and then each of these is drawn using hardware acceleration. For Bézier segments texture coordinates are used instead of edge function tests, and this method is also used by the Xorizer. In these cases we can benefit from using the texture coordinates.

Chapter 3

The Scanliner

When memory consumption is not vital but correctness is essential, one would benefit from using an algorithm that computes the exact pixel coverage. Thus it gives the most accurate results. This algorithm was written as a comparison to the Xorizer.

The Scanliner uses some tried and true ideas from different sources. These will be pointed out throughout this chapter. The differences will be discussed briefly.

Also interesting to note is that, as opposed to sample based algorithms, such as the Xorizer, this algorithm theoretically computes the exact pixel coverage of the outline.

3.1 Theory

All of the outline segments are processed in no particular order. The idea behind this algorithm is to store intensity information for the pixels that intersect the outline. The intensity information will be the same no matter what order the segments are visited in. This saves us the trouble of having to do any sorting of segments, which is sometimes needed in plane sweep algorithms [1] (The libart library does this [3]). This sort of algorithm, where intersection information for pixels is stored, is called an *edge list* algorithm.

For each line (segment line or approximated from a curve) each pixel it intersects is processed. Two values are computed per pixel – *coverage* and *trail*.

The Scanliner algorithm doesn't follow the even-odd rule. It strictly uses the winding (ascending/descending) of the segments to calculate pixel coverage. An ascending line encloses the area that lies to the right of it. Therefore, we define the coverage of a pixel as described in figure 3.1. If

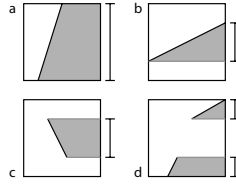


Figure 3.1: Pixel coverage calculation. There are two values calculated. The coverage value is the grey area, and is defined as the area enclosed by the line and the right edge of the pixel, as the figure describes. The trail value is defined as the sum of the height of the covered area within a pixel. The concept can be viewed to the right of each pixel. The coverage and the trail of descending lines are defined the same way as the ascending lines, only with the difference that they are negative.

the line is ascending, the value is added to the pixel's coverage, and if it's descending, it's subtracted.

The second value, trail, is the length of the line projected onto the right pixel edge. This is also described in figure 3.1. If no lines intersect the following pixel(s), then the trail value multiplied with the pixel width (which is 1) is the area covered for this/these pixel(s). For ascending curves we add to, and for descending, we subtract from the trail value. While the coverage value is local to the individual pixels, the trail values are accumulated.

First, let's consider the line $0 < slope < 1$. We traverse each pixel the line intersects and compute the pixel coverage and trail. The line traversal algorithm is a modified version of the classic *Bresenham Line Algorithm* [13] and is described briefly below. If we don't consider the end points of the line we have two different cases described in figure 3.2.

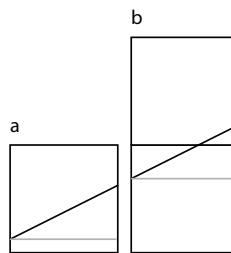


Figure 3.2: Different line-pixel intersections. The lines have a slope of $0 < k < 1$. In case a) the line only covers one single pixel vertically, for this x interval (which is one pixel wide). In case b) two pixels are intersected. The grey lines help visualize how the area will be calculated for these pixels.

```

function line_algorithm
{
    Compute the lengths in the x and y directions of the line (dx and dy)
    Compute the slopes dx/dy and dy/dx

    Compute coverage and trail for the start point, and push results to list

    for each pixel from x_start to x_end {

        The next intersections are calculated using the slopes

        if the y value of next integer x-intersection is larger than the
            current y value {

            The line spans two pixels vertically for this x (case b)
            Compute the coverage and trail for the first pixel and push to list
            increase y by 1
            Compute the coverage and trail for the second pixel and push to list

        } else {

            The line only covers one pixel (case a)
            Compute the coverage and trail and push to list

        }

        increase x by 1
    }

    Compute coverage and trail for the end point, and push results to list
}

```

We need to take some special care of the pixels where the start and end points are located. All possible cases for the $0 < slope < 1$ sloping line are covered in figure 3.3. Since we calculate the points where the line intersects the edges of the pixel we can compute the information we need, as illustrated in figure 3.4. We wish to know the coverage and trail for the two pixels.

In the figure we have a line which ends in the point p . It intersects the pixel edges at a and b .

The origin is at the lower left corner. The upper trail value t_0 then becomes the fractional part of the p_y . The lower trail t_1 becomes 1 - fractional part of b_y . Now on to the coverage. The upper pixel will have coverage $c_0 = t_0 \cdot \text{frac}(\frac{a_x + p_x}{2})$, where $\text{frac}(x)$ is the fractional part of x . And for the lower pixel we get $c_1 = t_1 \cdot \text{frac}(\frac{b_x + a_x}{2})$.

The special cases for vertical lines and single pixel coverage are not included here, but these are trivial once you get the idea of the rest of the algorithm.

Especially consider the case in figure 3.6. From this example we can see that we don't need to process horizontal lines at all. They are handled by the trail values.

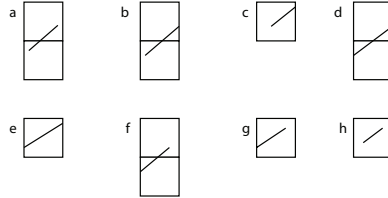


Figure 3.3: Pixel-line intersections. All possible line coverage scenarios for lines intersecting pixel, where $0 < slope < 1$.

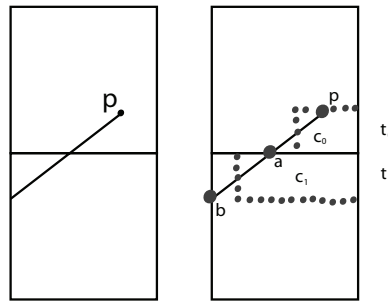


Figure 3.4: Pixel coverage and trail calculations.

The line algorithm is the same for lines with $1 < slope < \inf$. We only need to “mirror” it along the line $x = y$ to see that we can use the same approach, only with x and y changing places with each other. The area coverage calculations will be slightly different, though, since only the area to the immediate right of the curve is handled by each line. We will not go in to detail on these cases though, the one described will suffice.

If the current segment is a Bézier curve, then we can use either forward differencing or recursive subdivision, presented in sections 2.2.3 and 2.2.4, to break the curve into line segments which approximate it. Since we use a finite number of lines for this approximation, the pixels intersected by the curve will not get the exact pixel coverage, unless a great number of lines is used. Most often, though, it is quite enough to use only a few lines to approximate the curve, and more so if we are dealing with small fonts, where curves are generally quite small also. Figure 3.5 shows the use of different number of lines approximated by the forward differencing algorithm.

Now we have all the information about the outline that we need. All we have to do is to use this information to produce a bitmap of the glyph.

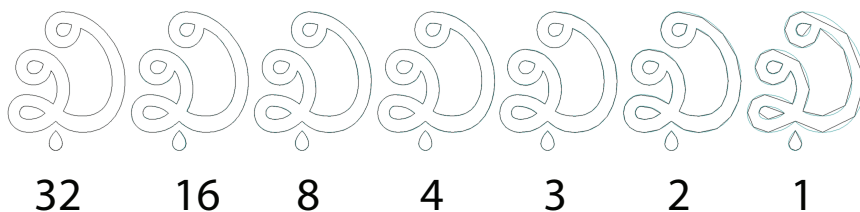


Figure 3.5: Bézier curve approximation using forward differencing. Different number of points have been used to approximate a glyph from the Telugu script. Each curve along the outline is approximated with the given number of points between their end points. Imagine that this glyph is only going to be about 10 pixels tall or so. Two or three approximation points are probably sufficient to get good results.

We process each scanline by itself and we begin plotting the from left to right. Trail, t , is set to 0. We continue to draw with 0 intensity until we hit an intersection. The pixel is given an intensity i , and the trail is updated with the trail value from the current intersection t_c :

$$i = t + c_x \quad t = t + t_c$$

where c_x is the coverage for the current pixel. In other words, this means that the trail is the inherited area coverage from the previous intersections, and the coverages are the pixel local changes to this. If a new intersection is found on the next pixel, we use the above formulas. Otherwise we just keep drawing with the current t value. Once the end of the scanline is reached the t value will be 0.

3.2 Implementation

The implementation of this algorithm is quite straight forward. We iterate over the points and call the line processing function for each line, as described above. The Bézier curve processing also follows the described procedure, where the curve is approximated by lines.

For maximum speed, fixed point math should be used. The font files delivers point information in 26.6 form, but this precision is probably not enough for slope calculations, and for Bézier curve evaluation. For these purposes 20.12 fixed point was used. Both Bézier curve evaluation methods were tested. The two alternatives for approximating the curve are described below. First we look at forward differencing:

```
function render_bezier
{
```

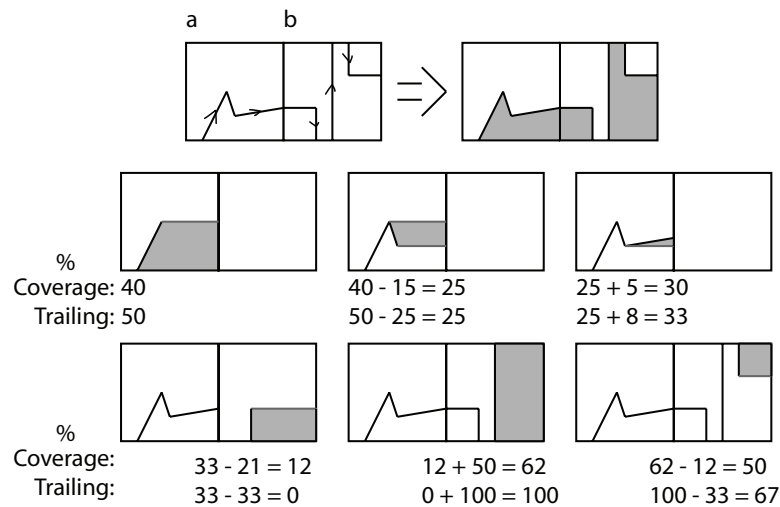


Figure 3.6: Two neighbouring pixels processed by the Scanliner. We process one line at a time. Initially, the coverage and the trail are both set to 0%. The first line is ascending and covers about 40% of the pixel. It takes up half the height of the pixel, and therefore the trail is 50%. The next line is descending, and we therefore subtract it's coverage and trail values, which are -15% and -25% respectively. Next up is an ascending line. No more lines covers this pixel. The coverage value is then complete and won't change any more. The trail value is inherited by the next pixel (b). The horizontal line is skipped since it will have 0 area and doesn't take up any height (has no trail). We follow the same principles as before for the right pixel. Ultimately we have visited all lines and coverage and trail have been calculated for both pixels.

```

curve_point_n = number of points we want
                 to approximate with
t = 1 / curve_point_n

// Multiplications are only needed in the setup phase of this algorithm.

Set up f, fd, fdd_2, fdd

for curve_ponts_n - 1 { // Note: -1
    // uses only additions in the loop!

    q = f
    f = f + fd + fdd_2
    fd = fd + fdd

    call render_line
}

```

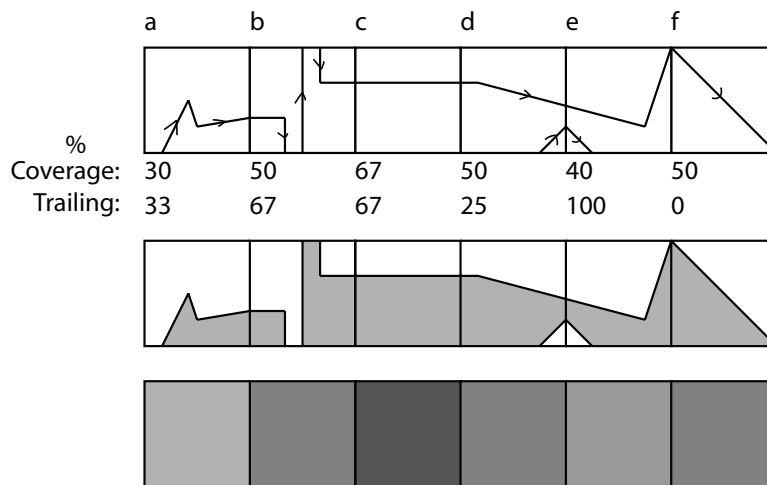


Figure 3.7: A long sequence of pixels rasterized by the Scanliner. This shows the a whole scanline being rasterized. Notice that the trail doesn't change for the horizontal line (pixel c). The coverage of each pixel is dependent on the trail that is accumulated before it. The bottom row shows the resulting pixel intensities as given by the coverage and trail values.

```
// last point
call render_line
}
```

Forward differencing can lead to some nasty rounding errors, especially for fixed point math. It is wise to use an excessive amount of decimal bits for these calculations. It is also wise to skip the last iteration where the last point should coincide with the end point of the Bézier curve, and use the end point of the curve instead of the approximated value. This ensures that the line is closed, and no rounding errors risk causing odd rendering artifacts. Also, for fixed point math the division can be substituted by a bit-shift if *curve_point_n* is a power of 2.

Now we look at the recursive subdivision in pseudo code:

```
function render_bezier
{
  if ending conditions are not met {
    subdivide(offset, offset)
  } else {
    call render_line
  }
}

function subdivide
{
```



```

Calculate mid point of the lines p0 -> p1 and p1 -> p2
Use this new line and calculate its mid point

if we're not at the maximum number of recursions and ending
    condition for the left sub-curve not met {
    call subdivide for the left sub-curve

} else {
    call render_line
}

if we're not at the maximum number of recursions and ending
    condition for the right sub-curve not met {
    call subdivide for the right sub-curve

} else {
    call render_line
}
}

```

The ending condition is subjective. A good idea is to continue until each sub-curve is a pixel or so in size before calling the render line function for it. If too narrow conditions are used the algorithm can become slow due to recursion calls, but this depends on the target platform. Also note that for fixed point math, all divisions can be replaced with with a bit-shift operator.

The ending condition used for this reference implementation for the recursive subdivision was if the curve spans less than two pixels in the x- and y-directions, the curve is small and/or has too little curvature for us to proceed. For forward differencing 4 interpolation points were used.

The pixel information, coverage and trail, is stored in one linked list per scanline. When new values are inserted in a list they are placed in the right position, so that the linked lists remain sorted.

The pseudo code below describes the algorithm:

```

function rasterize
{
    for each contour {
        for each set of 3 points {

            if point 0 and point 1 are on-curve points {
                call render_line

            } else { // we have a bezier curve

```

```

        if on-curve point missing {
            interpolate new on-curve point
        }

        call render_bezier // forward diff. or rec.subdiv.
    }
}

call plotting_algorithm
}

function render_line(p0, p1)
{
    is_asc = p0.y < p1.y // true if line is ascending

    if line is contained within a pixel {
        call line_algorithm_single_pixel(p0, p1, is_asc)

    } else if line is vertical {
        call line_algorithm_vertical(p0, p1, is_asc)

    } else if line is horizontal
        // ignored!

    } else {
        switch (slope of line) {
            call line_algorithm(p0, p1, is_asc) // one of four variants

            // the line algorithms differ somewhat depending on
            // the slope of the line. See the Scanliner Theory
            // section for more details.

        }
    }
}

```

Some inspiration for this algorithm was fetched from the (very sparse) libart documentation [3]. The libart algorithm stores one single intensity delta value for each changing pixel, where as the Scanliner stores local intensity, and an intensity that will be inherited by succeeding pixels. Instead of sorting the edges before rasterization, though, an edge list based approach was used instead. The edge list algorithm specifics can be read about in [1].

In the Results section both images and performance for bitmaps produced through the use of recursive subdivision as well as forward differencing are presented.

Chapter 4

The Xorizer

Keeping in mind that embedded devices often have limited resources and weak processing power, it would of course be desirable to have an algorithm for rendering glyphs which is both memory efficient and fast to execute. In this chapter we explore an algorithm that uses small amounts of memory and see if it could be suitable for these limited systems. A very brief concept of the algorithm was suggested by [12], but is evaluated here in much more detail.

This algorithm has similarities to classical triangle rasterizer approaches, in that it utilizes triangles as building blocks to construct the image representation of the glyph. However, instead of breaking down the outline area into triangles through tessellation, this algorithm uses a slightly different method, which ultimately results in that only a small, limited number of points need to exist in memory at any time throughout the rasterization. Yet this advantage does not cancel out any of the other benefits of the tessellation methods, which potentially makes this algorithm efficient.

4.1 Theory

The *Jordan Curve Theorem* states that if we have a non-self-intersecting closed curve we also have an inside and an outside. Now let's assume that we have a line from a point p to infinity. From the theorem it follows that we can determine if the point p is inside or outside by counting the number of times the line crosses a curve. An even number of times means it's outside, and an odd number of crossings means it's inside.

We use some sample points for each pixel to determine intensity. Any given sample point can only be either inside or outside the outline. Grey pixels that occur through anti-aliasing is achieved by using more than one

sample per pixel. The pixel's intensity is selected between 0 and 1 according to the percentage of samples in the pixel that are inside the glyph.

The state of a sample point can obviously be represented with one bit; 0 for outside and 1 for inside.

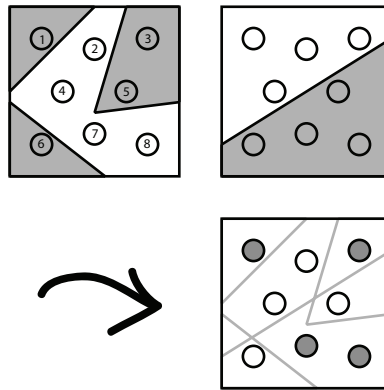


Figure 4.1: Sample pattern change. The samples are inverted if they are inside a triangle. The two top pixels' sample patterns XOR'd results in the bottom pixel's sample pattern.

The general idea of this algorithm is to kind of “carve out” the shape by inverting areas created by a fixed point and one contour at a time.

An arbitrary anchor point a is selected. With this anchor point we create a triangle with a line segment on the contour we wish to evaluate. We only concern ourself with straight line segments for now.

The three edge functions of this new triangle are calculated. Sample points inside the bounding box of the triangle are evaluated, and if they fall inside the triangle, they are inverted (NOT'd). The line segment will never be needed again, so it can be disposed of at once. Only the two segment points plus the anchor point need to be in memory. For Bézier segments we need three points instead of two, but still, this is not a lot. This is the reason why this algorithm is so memory efficient.

Let's enumerate each unique sample point position within a pixel with a number between 0 and n . Remember, a sample point can have either state 0 or 1. We use a data type that holds n or more bits. Each sample point state then corresponds to a bit position in the data type. Each sample has a corresponding state bit in an array of a certain data type which have as many bits as there are samples in a pixel. This is useful because when we invert our samples, we can do so n samples at a time by using the XOR-operation (this is why we call it the **Xorizer**).

For instance, consider the following case; We have a pixel with 8 samples. It has the sample state 00110101 from previous triangle traversals.

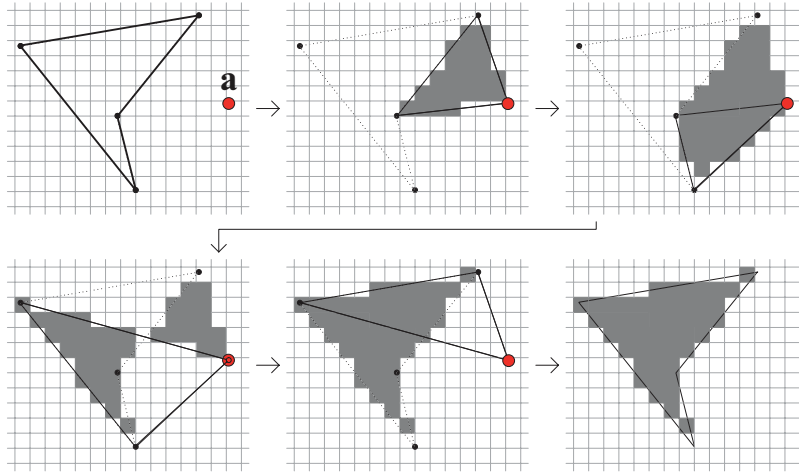


Figure 4.2: The Xorizer in action. The algorithm uses the anchor point a and one edge at a time to create a triangle. The contents of the triangles are inverted, one by one. The resulting image has the correct samples coloured. In this example there is only one sample at the center of each pixel, but the idea is the same no matter how many samples are used.

From the next triangle we get the new sample state pattern 11110000. This means that bits 5 - 8 were inside the triangle and these slots should be inverted. Using the XOR operation with the old sample state and the one from the current triangle we get the new sample state 11000101 for this pixel. This case can be viewed in figure 4.1.

Figure 4.2 shows the Xorizer in action as it rasterizes a shape with four edges using 1 sample-per-pixel.

The anchor point must not be moved while evaluating samples against a contour. If the whole contour is processed, the result will always be the same, regardless of anchor point placement. When selecting this anchor point, one should try to minimize the overdraw (discussed in section 6.4.1). This selection could of course be offline. Anchor point selection is discussed in section 4.3.

Bézier curve segments require some extra work, since they are a bit more difficult to evaluate than simple lines. A curve requires one triangle pass, as described above, and then one curve evaluation pass. We use the end points of the curve as input to the triangle processor, which means that the triangle evaluated will be defined by the base of the curve's bounding triangle and the anchor point. Once this is done, it's time to evaluate the area enclosed by the line between the end points and the curve. The shape we are inverting is shown in figure 4.3.

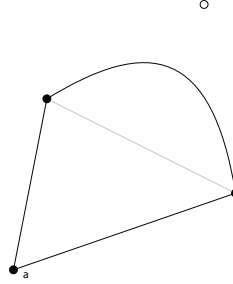


Figure 4.3: Bézier evaluation shape. When a curve segment is found, we need to process both the curve and the triangle created by the base of the bounding triangle and the anchor point.

An interesting technique for Bézier curve evaluation was suggested by Loop and Blinn in the article [11]. Suppose that we have a set of sample points and a quadratic Bézier curve. Suppose we want to know on which side of the curve each of the sample points is. Sample points that are within the bounding triangle, and are “below” the curve are considered to be a part of the Bézier segment.

The idea, then, is that we set the texture coordinates (u,v) to $p_{t0} = (0, 0)$, $p_{t1} = (0.5, 0)$ and $p_{t2} = (1, 1)$, where p_{tn} is the texture coordinate at vertex n ($n = 1$ for the control point). This is visualized in figure 4.4. If we let the bounding triangle defined by these three points be the end points and control point for a Bézier curve, the polynomial for the curve for these particular points coordinates will be very simple:

$$u^2 - v < 0 \quad (4.1)$$

This means that instead of having to evaluate some nasty Bézier polynomial we can use the texture coordinates to determine where we are in texture space, and from that we know on which side of the curve we are.

The other portions of the Loop and Blinn article are not suitable for embedded devices, and their article mainly targets GPUs. The idea involves tessellation through Delanuay triangulation. Also, their article doesn’t have any particular emphasis on memory usage.

4.2 Implementation

In the Xorizer only a small number of points are processed at a time, as opposed to a tessellation approach. The drawback is that some sample points may (and most definitely will) be visited more than once.

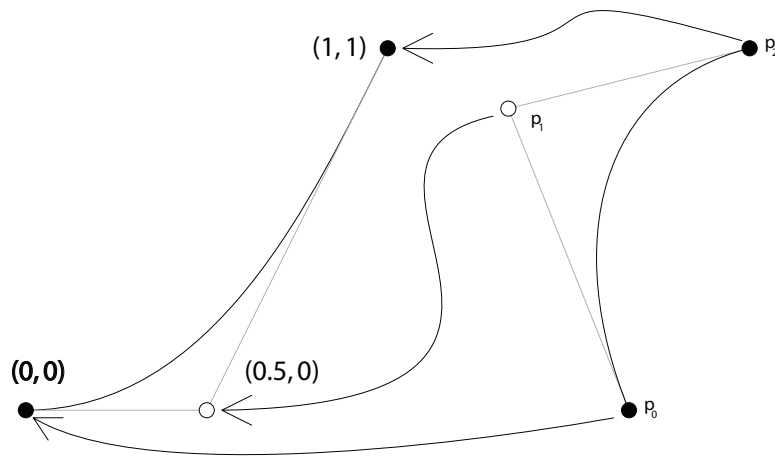


Figure 4.4: Texture coordinate mapping for Bézier evaluation. The points of the bounding triangle are given texture coordinates according to this figure. Using these coordinates, u and v , when we traverse the triangle we can more easily check on which side of the curve we are on by using equation 4.1.

4.2.1 Trivial implementation

The most trivial implementation of the Xorizer algorithm is quite easy to construct, and doesn't differ much from a classic triangle rasterizer used in many graphics applications throughout the latter decades. A simple way to describe the algorithm is through the pseudo code below. Note that this reference implementation uses 8 samples per pixel since that corresponds nicely to the number of bits in a char.

```
function rasterize
{
  for each contour {
    for each set of 3 points {
      if point 0 and point 1 are on-curve points {
        call render_triangle

      } else { // we have a bezier curve
        if on-curve point missing {
          interpolate new on-curve point

        }

        call render_triangle
        call render_bezier
      }
    }
  }
}
```



```

    // each pixel now contains its sample pattern

    count the number of bits in each pixel
}

function render_triangle
{
    calculate the triangle size
    calculate the three edge functions
    calculate the bounding box
    set up tie breaker rules
    initiate iteration variables

    for each row in the bounding box {
        for each pixel (k) on the current row {
            xor_me = 0

            for each sample (i) in the pixel {
                if sample i is inside triangle {
                    in xor_me set bit i to 1
                }
            }

            Xor with xor_me for the current pixel
            update iteration variables
        }
        update iteration variables
    }
}

function render_bezier
{
    calculate the triangle size
    calculate the three edge functions
    calculate the bounding box
    set up tie breakers
    initiate iteration variables
    initiate barycentric coordinates for first pixel

    for each row in the bounding box {
        for each pixel (k) on the current row {
            xor_me = 0

            for each sample (i) in the pixel {
                if sample i is inside triangle {
                    calculate tex.coords. from bary.coords.

                    if tex.coords.  $u \cdot u - v < 0$  {
                        // if we are inside the curve in tex.space.
                        in xor_me set bit i to 1
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}

Xor with xor_me for the current pixel
update iteration variables // including bary.coords.
}
update iteration variables // including bary.coords.
}
}

```

To further decrease rasterization time consumption, several different ways of optimizing the algorithm were implemented and evaluated. These optimizations are described in section 4.2.5.

4.2.2 Plotting glyphs

The Bengali and Devanagari scripts have headstrokes, a horizontal line that runs above the glyph bodies. This line should be continuous and not have gaps between glyphs. Most fonts ensure that there is no gap by using some extra headstroke length. This means that the headstroke actually goes outside the advance width of the glyph, and thereby overlaps the successive glyph's bounding box, and thus the headstrokes are being overlapped.

The fact that glyphs can overlap results in special requirements when plotting the rasterized bitmaps to the canvas area. Most of the time the headstroke doesn't align with the pixel grid, and therefore there are some grey, anti-aliased pixels at the end of the headstroke. If we simply plot over whatever is on the canvas with our glyph then these grey pixels will create something resembling a gap in the headstroke. This is so because any black pixels already there on the canvas will be overwritten. This phenomenon can be viewed in figure 4.5.

Since glyphs are produced in separate bitmaps we can't know *which* areas of a pixel that are covered, only *how much* of it. Therefore, when two bitmaps overlap it is a problem knowing the correct way of blending pixels.

A decent way to solve this for headstroke based scripts is to use the maximum value of the two pixels from the two bitmaps, such as in figure 4.5. The result is a good compromise for these complex scripts even though it is not entirely correct. Consider the case in figure 4.6. The pixel coverage is actually 75 percent of the pixel, yet only 50 percent intensity will be applied to the pixel by using this maximum value-method. Instead consider the case in figure 4.5 again. This is a much more common case in headstroke based scripts, and the result will also be correct since fonts use extended headstroke length.

न त
न त → न त
नत नत

Figure 4.5: Overlap plotting problem. Two Devanagari glyphs are rasterized. If the second glyph overwrites the first one the anti-aliased pixels in between will create a gap between them. If instead the maximum value of each pixel is used, they are nicely drawn as a unity, as they should.

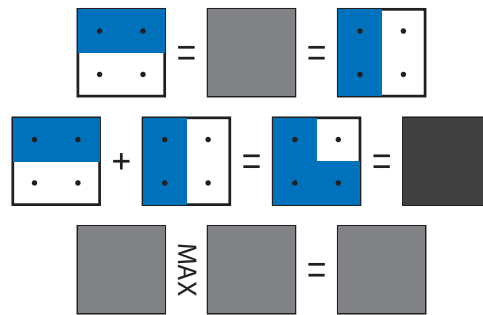


Figure 4.6: Plotting problems. The two topmost pixels are 50% covered each. If they were to be drawn on top of each other, the covered area would be 75%. There is no way, however, to know *which* parts of the pixel that is covered once the rasterizer has produced the bitmap and thrown away the outline information. Using the maximum value of the two pixels will only give 50% coverage, so in this case, this is a potentially poor solution.

4.2.3 Bit-counting algorithms

There are a number of ways to count the number of bits for a certain data type. Basically, they can be divided into two categories.

Lookup table

The first method is to use the value of the variable to be bit-counted as the slot in a lookup table. The slot holds the number of bits in its slot value. For instance, the number 209 is 11010001 in binary (in the Xorizer this is equivalent to sample points 1, 5, 7 and 8 being inside the glyph shape). So the slot 208 (the 209th slot) holds the value 4.

This is the fastest way of counting bits, but it requires some memory to store the lookup table. The table needs to hold 2^n values to cover all the variants of bit patterns, i.e. the size doubles for each added bit. For the reference implementation the table size is 256 since 8 sample points per pixel is used. This seemed a reasonable trade-off considering the 256 bytes it entailed spending.

Counter

Counting bits without using a pre-computed table means that the number of bits is somehow calculated from the bit pattern. The most trivial approach is looping over the bits of the variable and simply counting the number of 1's found.

However, there are more sophisticated solutions. Gurmeet Singh Manku has compiled a list of different bit-counting algorithms, and how well they perform. His findings can be read at [5].

4.2.4 Sampling scheme

Different rasterization algorithms handle pixel coverage differently. Ideally, a pixel's shade should reflect the exact pixel coverage percentage of the outline. There is only one *correct* result for any given number of bits per pixel. How well a rasterized glyph agrees with this correct result is dependent on the sampling scheme.

It's impossible to consider every thinkable outline shape, but there are some rules of thumb for selecting a better scheme. This is especially important for very small font sizes, where poor correctness can hinder readability.

Some effort was made to find a suitable sampling scheme for the sample points. The human perception is sensitive to jaggedness for lines that are supposed to be straight or nearly straight. Therefore no two sample points

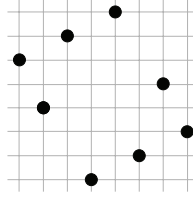


Figure 4.7: The 8 rooks sampling scheme.

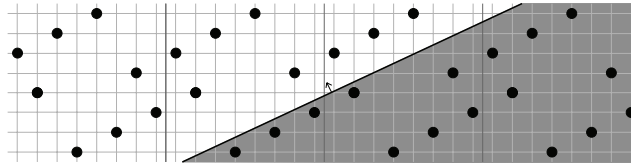


Figure 4.8: 8 rooks sampling problem. The outline overlaps some pixels with the 8 rooks sampling scheme. Since the edge has the same angle as two three-point constellations we can only get the intensities: $0/8$, $3/8$, $4/8$, $5/8$ and $8/8$.

should lie on the same vertical or horizontal line. We should maximize the number of samples in these directions, and thus we also maximize the number of greys for these cases.

The scheme 8 rooks 4.7 was considered, but was later dropped. Even though it doesn't have samples sharing vertical or horizontal lines, it does have up to three samples on a single shared line. This means that for certain angles, such as illustrated in figure 4.8, there are only five shades of grey. For every angle to have the maximum amount of greys no more than two samples should lie on the same line.

It's difficult to avoid these cases. The sample scheme which ended up in the implementation also have three sample points more or less lined up. The sample pattern used is kind of a combination of two RGSS sample schemes, and can be viewed in figure 4.9.

The selection of sampling scheme is highly subjective, and doesn't really affect the rasterization algorithm itself in any way.

4.2.5 Optimizations

A number of optimization ideas were tested to speed up rendering. Some were more successful than others. The tested algorithms are listed in this section.

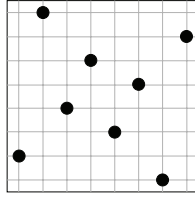


Figure 4.9: Combined RGSS sample scheme. The sampling scheme used in the reference implementation of the Xorizer algorithm.

Tiled traversal

Visiting every single sample within the bounding box of a triangle requires a lot of work. It would be better if whole chunks of samples could be evaluated at once. The idea of tiled traversal is just this - make a conservative sampling to determine whether all samples contained within the rectangle are completely inside or completely outside the shape. The rectangle, or *tile*, is $n \times m$ pixels, and which n and m to choose depends a lot on the font size. For small font sizes 1x1 pixels is probably the best option, and this is what was used in the reference implementation (other sizes were tried but with worse results).

To know whether a tile is contained within or is outside a triangle you need to know that all four corners of the tile is inside or outside respectively. Only two of the corners need to be evaluated against each of the edge functions, though. Looking at the signs of the a and b values of the current triangle edge function we know which direction the edge normal has. The direction can be in either of the four quadrants

This actually means that only the tile corner in the same direction as the normal direction needs to be evaluated against the edge function to know whether the entire tile is inside the edge. This is because while inside of the edge this corner will always be the closest to it, and if we move the edge in its normal direction this corner of the tile will be the first to cross.

The opposite corner of the tile will, in turn, be the most conservative sample point to evaluate whether the entire tile is outside the edge. We use the same logic as above, only we use the negative normal direction.

For tiles inside the shape we XOR with a value with all 1's. In other words, for our 8 sample-per-pixel, 1x1 tile implementation we XOR the pixel value with 11111111 which will invert the value and thus "visit" all pixels at once.

If both of these tests fail and we're neither completely outside nor completely inside of the triangle, the tile must be super sampled. This scenario

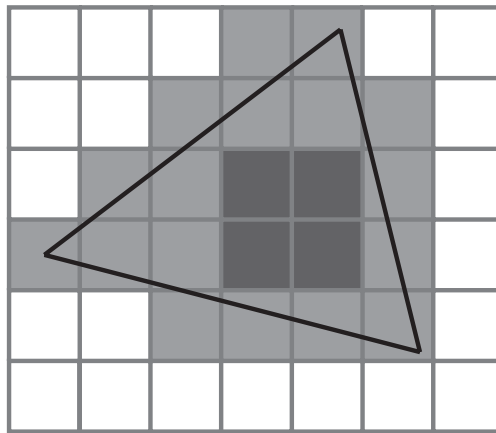


Figure 4.10: Tiled traversal. White squares are completely outside the triangle, dark grey are completely inside. Light grey must be evaluated in greater detail. In this case there is a lot of super sampling. Perhaps using smaller squares would be beneficial if all triangles are this small.

occurs around the edges of the triangles, as can be viewed in figure 4.10. All in all, this a good optimization which saves a lot of computations, especially for large triangles.

Quad evaluation

For each triangle, each sample is currently always evaluated against all three edge functions. Let's look at the triangle constellation in figure 4.12. Both triangles are traversed with the same bounding box. The shared edge is evaluated twice, once for each triangle. This seems a bit unnecessary, since the shared edge is obviously inside the area that is to be covered.

This optimization is only useful if the two triangles compose a convex quad. For triangle constellations such as in figure 4.11, only the left one can be processed using this method. The right example has to be broken into two separate triangles. If the quad isn't convex, it no longer holds that a point is inside the shape if the edge functions give positive values for the point. Therefore it must first be determined if the two triangles form a convex quad, or if the triangles should be processed each by themselves.

The easiest way to do this is to calculate the three edge functions for one of the triangles. Then the fourth point is evaluated with the three edge functions. The point must lie inside the two opposing edges, but outside the shared edge in order to be convex. The different cases are illustrated in figure 4.11.

The optimization can be extended further to incorporate shapes with

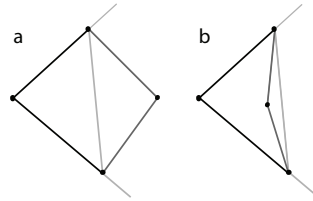


Figure 4.11: A convex and a concave quad. The left quad (a) can be evaluated using the optimized method, but the right one (b) cannot, since the edge test will fail for some areas that are actually inside the quad.

more than four points. The same criterion still hold, the shapes would have to be convex. The shared edges are not sampled in detail, but each sample that actually is evaluated must do so against the number of edges in the shape, and this will soon become a quite costly process.

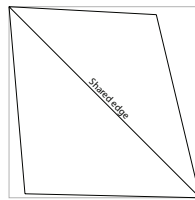


Figure 4.12: Convex quad. Two triangles composing a convex quad. The edge dividing the quad is shared by the triangles.

Ranges

Previously each sample in the bounding box of each triangle was evaluated against all three edge functions of the triangle. But in some sub-intervals of the bounding box, one or more edge functions need not be evaluated. It is already certain that samples within this sub-interval are always on the same side of the edge. This is shown in figure 4.13.

The trade-off of this optimization is that some portions of the triangle setup needs to be recalculated for each range. This includes several multiplications, though once set up, like before, only additions are needed when iterating over the current range. In total, up to seven ranges may need to be evaluated. For small triangles the setup workload for the ranges may outweigh the benefit of the reduced edge function evaluations. Also, since the samples are not traversed in order, there may potentially be somewhat worse caching behavior.

Because of the triangle setup workload, the performance gain of this optimization is lost for small triangles.

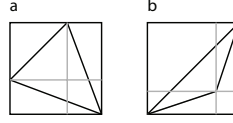


Figure 4.13: Ranges calculations. Two triangles with their bounding boxes split into ranges. In a) the lower right range has to be edge tested for two edges, and the other three only for one edge. In b) the lower right range doesn't need to be traversed at all. The upper left range is edge tested once, and the remaining two ranges are tested against two edges. We must also do a more thorough edge evaluation along the pixels that contain they grey lines, where the edge functions meet.

The greatest downside of this optimization is however that the many special cases can result in quite messy code and larger code size. If this optimization were to be used in combination with the quad evaluation optimization, the code would become a lot more complex and a lot larger. For convex quads there would have to be up to seventeen individual ranges to set up and traverse. Would the quad evaluation optimization be more generalized to handle any convex shape, the range-optimization would have to be redesigned entirely to be useful. Perhaps using a separate buffer which holds sample information would be a good idea, though this will use more memory.

For these reasons the reference implementation only utilizes the range optimization method for triangles.

Interval arithmetic for Bézier curves

The two optimizations above have not been useful for Bézier segments. This method, on the other hand, is only used for these particular segments. It's desirable to find a way to speed up the evaluation of equation 4.1. It's now time to use the concepts introduced in section 2.2.5.

We used the same idea for Tiled Traversal. The tiles we know for sure will be entirely above or entirely beneath the curve can be processed as a whole, instead of being super sampled. For this purpose we first fetch the texture coordinates at the corners of the tile and use them as our interval. The interval \hat{u} is calculated through $\hat{u} = [\min(u_0, u_1, u_2, u_3), \max(u_0, u_1, u_2, u_3)]$, where $u_i, i \in [0, 1, 2, 3]$ are the u components of the texture coordinates in each of the corners. We get \hat{v} the same way.

Now, if we rewrite equation 4.1 using interval arithmetic we get:

$$\hat{u}^2 - \hat{v} = \hat{u}^2 + [-\underline{v}, -\overline{v}] = [\underline{u}^2 - \underline{v}, \overline{u}^2 - \overline{v}] = \hat{r} \quad (4.2)$$

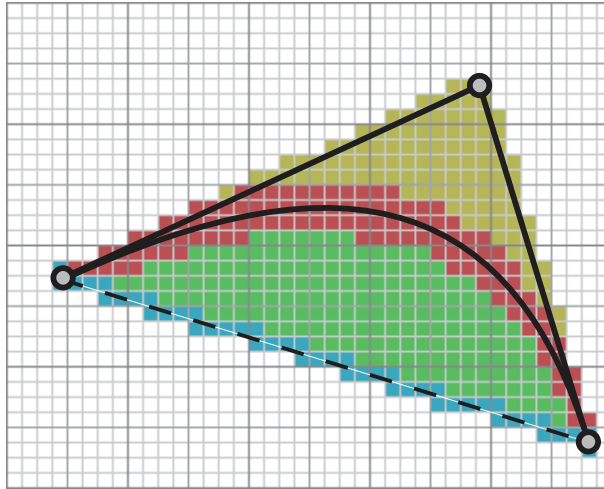


Figure 4.14: Bézier interval arithmetic. A Bézier segment evaluated using interval arithmetic. The yellow tiles are entirely outside the curve and the green tiles are entirely beneath. Red tiles need to be evaluated in greater detail, since they could not be omitted by the conservative interval test. Blue tiles also need to be evaluated against the dotted triangle edge.

Using this equation we can determine if the tile is entirely beneath the curve if $\bar{r} < 0$, or entirely outside if $\underline{r} > 0$.

In figure 4.14 we see how the interval arithmetic can help reduce super sampling.

4.3 Anchor Point Placement

The placement of the anchor point is important in the Xorizer algorithm. Different selections will cause different amounts of overdraw, i.e. the total amount of sample visits. When testing different anchor point placements it quickly becomes apparent that the average of the points of an outline gives the best (or very close to) results.

It has been found that different anchor point placements within the bounding box of a glyph roughly can give up to 70% decrease in rendering speed. If the anchor point is placed outside the bounding box there is no limit to how bad the rasterizer will perform. To help the rasterizer with some offline calculations, the average of all points can be computed before hand and stored within the font.

Chapter 5

Improving Rendering for Small Font Sizes

5.1 Sub-pixel rendering

Microsoft has patented a technology called ClearType in which they are able to produce “real” sub-pixel precision.

Realizing that the pixel is not the actual display atom, there is a way to represent glyph shapes more accurately. A single pixel is really a combination of three colour components - red, green and blue. Rendering shapes with sub-pixel precision means taking advantage of how these sub-pixel components are lined up. Basically, each sub-pixel is thought of as an individual pixel, and thus we are granted three times the accuracy in which any direction the sub-pixels are lined up.

For example, LCD-displays are very common on hand-held devices today. The sub-pixel components are lined up in the manner seen in figure 5.1.

Since the sub-pixels sizes don’t have the ratio 1:1 like whole pixels, but instead has the ratio 1:3, the glyph is scaled down by the same ratio. To compensate for this we re-scale the original outline three times in the sub-pixel lineup direction, in this case horizontally. By doing so we avoid the

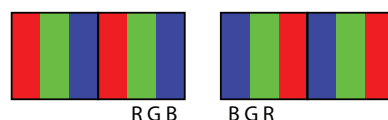
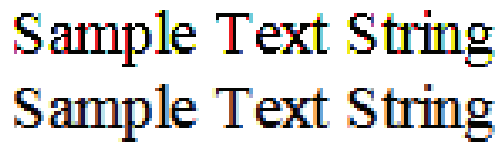


Figure 5.1: LCD colour component order. These are the most common colour component arrangements for LCD displays. They can also be vertical. Widely different patterns also exist [15].



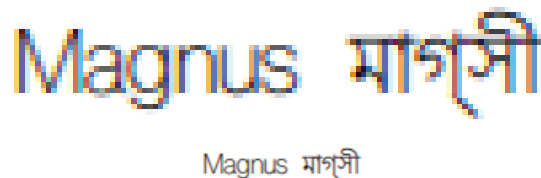
Sample Text String
Sample Text String

Figure 5.2: Unfiltered and filtered sub-pixel rendering. The top most text string is rendered without being filtered. It’s quite colourful. The bottom string is filtered. Each sub-pixel has spread its intensity evenly over itself and its two neighbouring sub-pixels. (Picture from [7])

final glyph image from being “squished”.

A problem with this technique is that the sub-pixels are obviously not shades of grey, like regular, non-sub-pixel renderings. This results in very brightly coloured edges, like in figure 5.2, and this is not acceptable. A common solution to this problem is to filter the colour. This is done by distributing the intensity over several pixels, as described in [7]. The result is much more aesthetically pleasing, as figure 5.2 shows.

Even though ClearType-like solutions where individual R-, G- and B-stripes act as individual pixels is a patent that is hard to avoid, it seems that different groupings of sub-pixel is possible to avoid patent issues. Kim Øyhus presents an algorithm on his page [8] where this is used. Figure 5.3 is drawn using this technique.



Magnus माग्जी
Magnus माग्जी

Figure 5.3: SubLCD sub-pixel rendering. A Unicode string rendered using two sub-pixels per pixel, green and purple (red + blue). The string is filtered by distributing half of the intensity for each sub-pixel to the two neighbouring sub-pixels equally.

The idea is that instead of dividing each pixel into three sub pixels, they are divided into two, one green and one purple. The motivation is that the green sub-pixel is more luminous than the other two. In fact, it is even more so than the blue and red sub-pixels combined (according to Øyhus).

This means that sub-pixel rendering can be used, but we scale the glyphs to twice their size along the line up direction, instead of to three times their size. The rest of the theory for this variation of the method is analogous.

font, but the results can be well worth it, especially for complex fonts which are intended to be used in very small font sizes.

The standard font Verdana is an excellent example of a well hinted font. It is a heavily hinted font, and is very good looking down to very small sizes even when using only a single bit per pixel.

Not many alternative font renderers actually use the hints from the font file though – using certain hinting instructions for glyphs is patented by Apple. Instead systems like FreeType and the like use something called auto-hinting. It means trying to analyze a glyph in real-time before rendering it, and making modifications accordingly. It can produce very good results, although sometimes not quite as good as an instructed font would give.

Auto-hinting actually does a number of different things to improve the glyph shape. Which auto-hinting features are useful vary with the font, and the cost of the different features vary greatly too. Auto-hinting is not an exact science, and to decide which features to apply to which fonts is rather subjective. Also, it can become painfully slow, and for anti-aliased fonts for embedded systems it is often not worth the cost.

There are however some features that can be beneficial. The most important ones are as follows.

5.2.1 Stem adjustment/Grid fitting

The middle, vertical line of the letter T is called a *stem*. Many Latin letters have stems, and these must be clearly visible for maximum readability. They are quite important features for us when we identify letters.

Generally for Latin fonts, consecutive points that lie on a straight vertical line, and have a corresponding vertical line somewhere within its vicinity are probably parts of a stem. Stems that fall in between two pixel centres will come out greyish instead of black, and it could somewhat disrupt the readability. Therefore this auto-hinting feature is useful for Latin fonts, and it is the author's opinion that this is the only feature worth bothering implementing for anti-aliased fonts. The difference between a hinted and an unhinted stem can be viewed in figure 5.6.

The way the glyph is modified through this feature is that the stem is snapped to the closest pixel edge, so that its centrum coincides with the pixel centrum.

The same process can of course be used for horizontal stems as well, though FreeType, for instance, currently does not. This is much more useful for certain complex scripts such as Devanagari (Hindi, Marathi) and



Figure 5.6: Stem adjustment. An untouched **m** and a stem adjusted **m**. Note that for a monochrome rasterizer, these could use some dropout control for the arcs as well, as they are completely missed by the pixel centers and therefore cause gaps.

Bengali. Both of these scripts have something called a headstroke, which is a horizontal line that flows above the text and connects the glyph bodies. It's as important to the native reader as the stems are for Latin speakers.

Since we use our own font format, VOFF, stems can be automatically flagged offline and saved with the font, so there is no need to identify these in real time. This makes this feature much more attractive, as the performance increases. It is unclear, however, if this is a breach of the Apple hinting instruction patent...

5.2.2 Drop out control

For monochrome renderings it's quite common for small font sizes, and even for narrow glyphs, that small parts of glyphs fall in between two pixel centers, and is therefore missed entirely. Such a case can be viewed in figure 5.7. Small gaps of white pixels within glyphs can be the result of this, which is annoying, to say the least. For anti-aliased fonts this is not really a problem though, because the sample points are so closely spaced, and it is highly unlikely that any significant part of any glyph would be missed by all of them, and if so, they're too small to be visible anyway.

On top of that, this feature can be quite costly to implement, both performance wise and in time consumption.

5.3 Font styles

Many fonts can have attributes applied to them, such as bold, italic, underlining, and so on. For bold and italic it is common that the font contains special glyphs for these cases. The reason is that mostly the hinting instructions will not be the same for the corresponding character when italic, bold and normal formatting is applied. Also, having a new set of glyphs give the font artist better control over the appearance of the glyphs when the different attributes are applied.

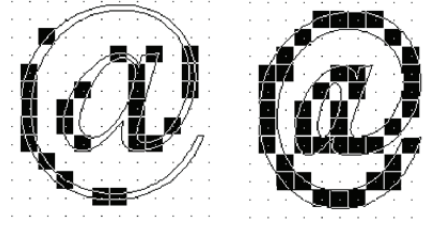


Figure 5.7: Drop-outs. The left @ has severe drop-out problems. The right @ has had its drop out problems fixed, as well as being corrected with hinting instructions. (Picture from [10])

With new glyphs come larger font files and more work for the artist to complete the font. It can be desirable to auto-generate the bold and italic styles somehow. For bold and italic we use different approaches to achieve this, but both modify the shape of the outline. These modifications should be applied to the point set before scaling to screen space.

5.3.1 Italic

By using an affine transformation we can shear the glyph. This will make the text lean with an arbitrary factor that the user can decide. We use the following transformation matrix on the point set composing the outline:

$$\begin{pmatrix} 1 & 0 & 0 \\ r_x & 1 & 0 \end{pmatrix} \quad (5.1)$$

where r_x is the shearing factor.

This matrix should also be applied to the glyph metrics size and offset in order to keep a correct bounding box of the glyph. The advance width should remain the same, as the base of the glyph (that is on the x-axis) is not modified, and we still want the next glyph to begin at the same position as without the shearing. Some special care must be taken, though, if a non-italic glyph follows an italic glyph, so that the italic glyph doesn't overlap the non-italic. This is a concern of the layout engine though.

5.3.2 Bold

For this font style we use a different method. In the offline step of the font conversion process to the VOFF format we can calculate the normal of each point. We let a value in the interval $[0, 255]$ correspond to a direction $[0, 2\pi]$. The normal information is probably easiest to calculate from the normals of the edge functions from the two adjacent line segments. The normal of a point is the mean of those two adjacent edge normals.

Right before it is time to rescale the outline point set to screen coordinates we let each point wander an arbitrary distance along its normal. Thus, we get an expanded, or bold, glyph (see figure 5.9).

This process is not without its drawbacks, though. Two problems can (and will) occur.

- The font file size will increase with one byte per point as we need to store the normals. This could be avoided by computing normals on start-up or in real time, although this could be time consuming. The file size is still smaller than what a new set of glyphs would result in.
- Two points could cross each others paths and create loops, such as in figure 5.8. For even-odd rule renderers loops could create problems with small gaps along the outline. Different scanline renderers can handle this differently depending on their implementations, but the Scanliner algorithm will not begin drawing until the right edge of the loop is reached, which is not the intention. The Xorizer will invert the area enclosed by the loops one time too many, and render them white. These loop problems are hard to avoid.

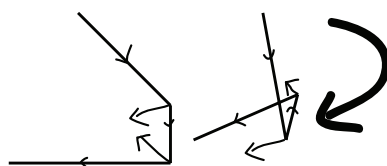


Figure 5.8: Bold loop problem. Loop created by moving outline points in their respective normal direction.

5.3.3 Other styles

Underlining, colouring and so on, are quite simple cases that we will not bother with here. We are not constricted to using only shearing when applying font styles. We could just as easily rotate, scale or reflect the glyphs, for example. The font size is actually also an affine transformation, as it is a scaling operation.

Note that if we want to apply both bold and italic, or any other affine transform, to our glyphs, we must apply the bold operation first, otherwise the normal information will be incorrect for the transformed outline.

Some different font attributes examples can be viewed in figure 5.9.

কিমেন্টী
কিমেন্টী
কিমেন্টী
কিমেন্টী
কিমেন্টী

Figure 5.9: Font attributes. The following font attributes are used (listed in order): normal, bold, italic, outline, bold/italic/colour.

Chapter 6

Results

6.1 Memory Usage

In this section, we analyze the memory requirements for FreeType and for the Xorizer. For mobile phones, it is important to keep the memory requirements as low as possible due to the limited resources of the device. Although none of the methods really require any grave amounts of memory, it's desirable to have as small a footprint as possible.

Both FreeType and the Xorizer need some general information about the glyph about to be rendered, such as the number of contours, the number of points per contour, contour end points and so on. However, FreeType also needs about 3400 bytes of allocated memory to store information used throughout the rasterization process.

The FreeType rasterizer documentation actually states that “A 4KByte pool is enough for nearly all renditions” [4], but it was found that 3400 bytes was enough for the renderings of the test font. Furthermore, all the outline points of the glyph need to be in memory. Each point is represented by two four-byte values, and an additional one-byte flag indicating whether it is an on- or an off-curve point.

In summary, the following amount is needed for FreeType:

$$3400 + 9n \text{ bytes} \tag{6.1}$$

where n is the number of points in the glyph.

FreeType can draw directly on the target bitmap and therefore does not need an intermediate working bitmap. For the Xorizer, we need a temporary frame buffer as large as the glyph's bounding box. In addition, we

need the source point and n points, where n can be as low as three depending on the implementation. The total amount of memory needed for the implementation of the Xorizer algorithm with nine shades of grey (i.e., eight samples per pixel) is thus:

$$wh + 9n + 8 \text{ bytes} \quad (6.2)$$

where w and h are the width and height of the temporary frame buffer. If the lookup table (see section 4.2.3) is used for speedup, an additional 256 bytes are spent, and thus we get:

$$wh + 9n + 8 + 256 \text{ bytes} \quad (6.3)$$

The test font used in the examples have an average count of 35.8 points. This corresponds to an average usage of $3400 + 9 \cdot 35.8 \approx 3722$ bytes for the FreeType rasterizer. Hence, for all font sizes less than roughly 60 pixels (assuming square glyphs), and 58 pixels when using the lookup table, the Xorizer uses less memory. However, it is often desirable for caching purposes that the glyph rasterizer renders its glyphs to a separate bitmap. And even more so if the font contains a lot of glyphs (complex scripts, Chinese, etc.).

In this event, the wh component should be added to the FreeType memory equation. For the Xorizer algorithm, the working bitmap can be used both for the xor-counters and later as the resulting glyph bitmap, and so no extra memory needs to be allocated since these aren't used at the same time. As a result, the Xorizer algorithm always uses less memory in these cases.

6.2 Output

The different rasterizers produce different results. The aim is, of course, to have a rendering which is as close to the correct image as possible. As we've seen the methods to achieve this differs greatly. Using too coarse approximations will give renderings that are too different from the expected result, but using too exact methods may fast become expensive.

In this section we examine a few images of glyph renderings produced by the Scanliner, the Xorizer and FreeType. In each of the comparisons the top row is the FreeType rendition. FreeType claims that it produces *exact* results, so let's buy that assumption. The second row consist of glyphs

rendered by one of our algorithms. The third row is the absolute difference between the two, and if a forth row is present, it is merely an enhancement of the third row for clarification.

Figure 6.1 shows glyphs rendered by the FreeType and the Xorizer. The difference is mainly a consequence of fewer grey levels in the Xorizer and sample point placement. It is also likely that some of the differences around curved edges are due to fixed point precision.

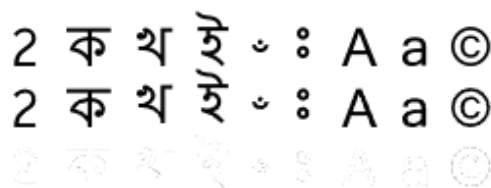


Figure 6.1: FreeType vs. Xorizer output.

In figure 6.2 FreeType is compared to the Scanliner with some parameters, namely a subdivision cutoff of 1 pixel for the initial recursion, and 2 pixels otherwise. This means that subdivision will stop when a sub-curve is no longer than the specified pixel length. The reference implementation has 64 levels of grey. The difference is significantly less than that of the Xorizer vs. FreeType, and is mainly present around curved segments. Again, approximation of curves and fixed point precision is the most likely cause. However, it has been found that the difference doesn't decrease especially much if the cutoff value for subdivisions are lowered even further.

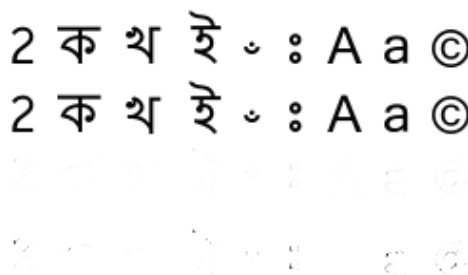


Figure 6.2: FreeType vs. Scanliner output (recursive subdivision A). Fine cutoff. Row four shows the difference enhanced for clarity.

Figure 6.3 also compares the Scanliner to FreeType. This time the subdivision cutoff is set to 2.5 pixels. The differences are still quite small, and the renderings still look very nice and clear. If the cutoff is raised, or the font size is increased, then it becomes more and more apparent that the curves

are actually composed of line segments.

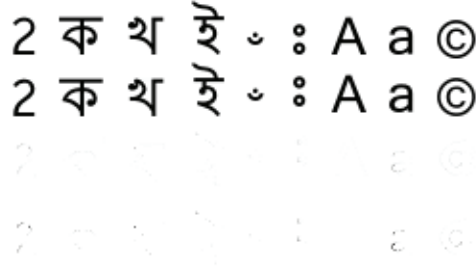


Figure 6.3: FreeType vs. Scanliner output (recursive subdivision B). Coarse cutoff.

Now we examine the results produces by forward differencing. In figure 6.4 we see that using two approximation points doesn't give as good results as subdivision. Using four approximation points, as in figure 6.5, is better, but as we will see in the next section this is not the fastest Scanliner version.

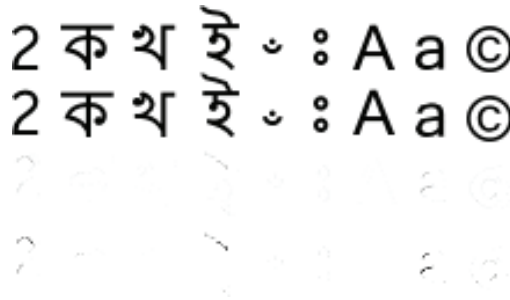


Figure 6.4: FreeType vs. Scanliner output (forward differencing A). 2 approximation points used.

6.3 Rendering speed

The last very important factor to examine is the rendering speed of the algorithms. Again, FreeType has been used as reference. The testing was performed on a computer with 2.0 GHz and 1 GB RAM. The compiler was optimizing for maximum speed for all tests. The test font used was Vrinda.ttf which contains 255 Latin and Bengali glyphs. All glyphs were drawn 1000 times and the time it took was measured (in other words, lower value is better). This was done for font sizes between and including 10-40 pixels in increments of 5, which is mapped to the x-axis of the figures in this section. The y-axis is the milliseconds it took to cycle through all 1000 iterations.

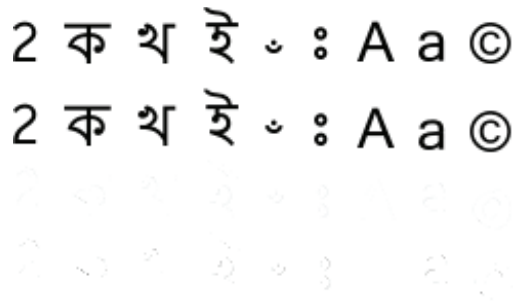


Figure 6.5: FreeType vs. Scanliner output (forward differencing B). 4 approximation points used.

The graph in figure 6.6 shows rendering speeds for different subdivision cutoffs as well as different amounts of approximation points used for forward differencing.

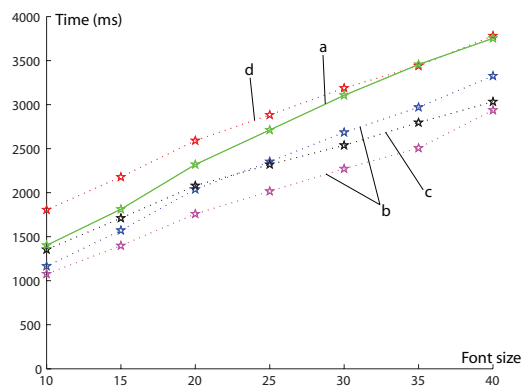


Figure 6.6: FreeType vs. Scanliner speed comparison. The green line (a) is FreeType with 16k memory. The blue and purple lines (b) are the Scanliner with fine and coarse subdivision cutoffs respectively. The black line (c) is the Scanliner with forward differencing with two approximation points, and the red (d) is with 4 approximation points.

In figure 6.7 the Xorizer is benchmarked with FreeType in its most memory efficient version. The fastest FreeType version (“unlimited” memory) is compared to the Scanliner in figure 6.6.

The graph in figure 6.8 shows the collected measurements for all rasterizers.

Compared to the Xorizer algorithm, FreeType is around 2-5 times faster depending on font size. This could probably be improved for the Xorizer using cache tuning and more clever ways of optimization.

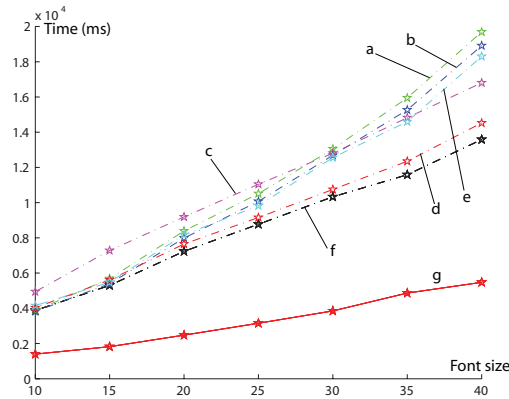


Figure 6.7: FreeType vs. Xorizer speed comparison. The green line (a) is the trivial implementation of the Xorizer. The blue (b) line is for interval arithmetic, the purple (c) for ranges, the dotted red (d) is for tiled traversal and the cyan (e) is for quad evaluation. The black line (f) is the Xorizer with all optimizations. The red solid line (g) is FreeType with 3400 bytes of memory.

The algorithms were also ported to Symbian UiQ and tested on a Sony-Ericsson P1i, in font sizes 10, 20, 30 and 40 pixels. There were major problems with running FreeType with 3.4 kB of memory for any font size other than 40 pixels. The test was performed with the scanline algorithm vs. FreeType with 16kB memory, and the Xorizer vs. FreeType with 3.4kB memory. The results are presented as time relative to FreeType. The results of first test were as follows:

Font size	Time consumption
10 px	94.2%
20 px	91.9%
30 px	88.7%
40 px	90.2%

We see that the scanline algorithm is about 10 percent faster than FreeType in this test. The Xorizer, however, compared to the only font size FreeType would render (40px) needed 383% of the time FreeType needed to produce the same results.

Clearly, if speed is the most important factor, this implementation of the Xorizer algorithm is inferior. However, in the Conclusions chapter ideas on how to improve the Xorizer using OpenGL ES is presented as a suggestion for future research.

The Scanliner, however, is a good choice for both correctness and speed.

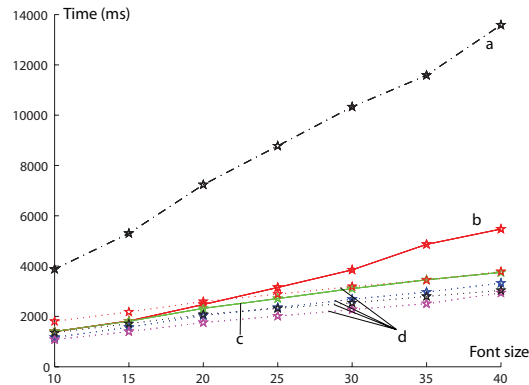


Figure 6.8: Speed comparison overview. The black line (a) is the Xorizer and the red line (b) is FreeType with 3400 bytes of memory. The green line (c) is FreeType with 16k memory. The dotted lines (d) are the Scanliner with fine and coarse subdivision cutoffs found in figure 6.6.

6.4 Discussion

6.4.1 Overdraw

There is an obvious problem with overdraw in the Xorizer algorithm - it is based on the idea that the same samples are inverted several times before they are set to their final state. This idea is both to the algorithm's advantage and disadvantage. The algorithm may become slow because of complex outlines, with many triangles to evaluate, and possibly high overdraw. However, it still just needs a few points in memory at a time, and neither time nor effort needs to be put into tessellating the shape before the rasterization itself begins.

One way of determining overdraw for a cross section is to draw a line from a point that is outside of the shape to the anchor point. Use an overdraw counter which starts at 0. We begin wander towards the anchor point. When an edge is crossed the counter is increased by one. Continue until the anchor point is reached.

Now the line has a number of intervals with increasing overdraw values. These correspond to the number of times samples that coincide with this line will be visited throughout the rasterization process.¹

Figure 6.9 shows a shape and its overdraw levels.

¹Compare Jordan's theorem.

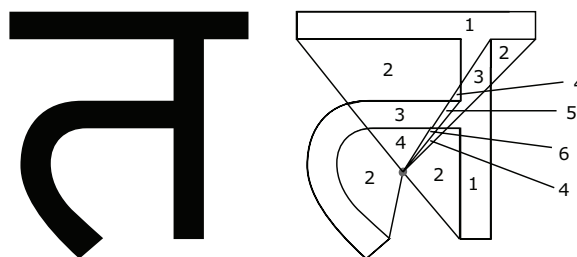


Figure 6.9: Overdraw values. The grey dot is the anchor point. By counting the number of times we cross a line we can see how many times each area has to be visited before it is inverted to it's right state. In this example there is a small area that will be visited as many as 6 times.

6.4.2 Glyph Representation Issue

Because the Xorizer follows the even-odd rule overlapping shapes will not be rendered as intended. This introduces some restrictions to the glyph outline information. Contours should not overlap, as the right plus sign in figure 2.7. The middle square will remain white since the area within it will be inverted twice.

This is not a major problem since most font editing tools have functionality to automatically merge contours. Never the less, it is an added requirement that is not part of the TrueType font standard. But for this rasterizer's sake, it is a requirement for our new font format VOFF.

Chapter 7

Conclusions

7.1 Future Research

It is obvious that the Xorizer algorithm suffers from quite slow rendering speed when compared to FreeType. And it is questionable if the memory gain will be worth the loss of speed performance on mobile devices in the very near future.

There are still a lot of low end mobile phones on the market, mainly in the developing parts of the world, and these may still have very little memory to work with. But the mobile phone market moves at an incredible speed, and soon the memory will not be an issue any longer.

One might think that this marks the end of the Xorizer algorithm idea, immediately after its birth. However, the future could hold the very opposite.

FreeType is a scanline rasterizer and this approach is not suitable for hardware accelerated rendering. The Xorizer is a triangle rasterizer and could therefore greatly benefit from the migration of graphic hardware acceleration to mobile devices.

The newly completed standard OpenGL ES (developed and supported by over a hundred companies) is a very interesting platform upon which the Xorizer could be implemented. The latest WTK-releases for all the big mobile phone companies include the OpenGL ES JSR, and there are already quite a few phones on the market with support for it. It's also used in devices such as the Sony PlayStation 3. It's beyond any shadow of a doubt that OpenGL ES will be the most (only?) used graphics API for mobile devices very soon.

The Xorizer could be implemented using the invert operation in the stencil buffer, or perhaps in the accumulation buffer. The next natural step

for the Xorizer algorithm is therefore to determine its potential for OpenGL ES, but this is left for future research.

As for the Scanliner, there are still things to optimize. The implementation benchmarked here has at least two minor aspects that could probably be improved. The first one is that it uses recursive subdivision, which could be replaced by iterative subdivision. This is probably a much better alternative on some platforms where function calls can be expensive. The other aspect worthy of optimization is the list handling. When a new entry is inserted into the list, a simple linear search is used to find the right position for it. Also, it is probably possible to rewrite the coverage handling to use only one single delta value, as in FreeType and libart.

7.2 Keep It Simple

Many optimization methods were tried under the duration of this thesis. Not many of them were altogether successful. The added overhead is often very complex and the cluttered code is probably in most cases worse than a speed up of one or two percent. It's often much more difficult to get the optimizations to run, and the maintenance can become really horrendous. The Range-optimization, for instance, doubled the code size many times over, and was really a nightmare to debug. In the end it wasn't really worth the effort, at least not in its current implementation form. The best that can be hoped for in this case is that someone else might be able to use the idea in a more useful way, or refrain from spending time trying to implement it.

Overall it seems that the Xorizer is best with fewer optimizations. Even though the triangle setup is rather slow, not much can be done about it. The pixel traversal, however, could probably benefit from combining the tiled traversal with other traversal schemes, although one must be careful not to thrash the cache by doing so.

No less does the Scanliner support the conclusion to keep it simple. The algorithm is not at all cluttered with a lot of complex and "clever" optimizations. The ideas behind it are rather simple, and the implementation just relies on known concepts like fixed point math, cheap curve approximations, line traversal order invariance, and so on. Obviously this was a rather good strategy.

By now, I think we have established why there are so few triangle font rasterizers on the market, but with the introduction of GPU's and hardware graphics acceleration in embedded devices there is still most likely a market for such algorithms.

Bibliography

- [1] David F. Rogers,
Procedural Elements for Computer Graphics.
McGraw-Hill Book Co., Singapore,
1985,
ISBN: 0-07-053534-5
- [2] Moore, Raymon E.,
Interval Analysis.
Prentice-Hall,
1966
- [3] Mathieu Lacage and Raph Levien,
The libart Library.
<http://www.gnome.org/mathieu/libart/libart.html>,
2001
- [4] David Turner, Robert Wilhelm and Werner Lemberg,
The FreeType Project.
<http://freetype.sourceforge.net/index2.html>,
1996-2007
- [5] Gurmeet Singh Manku,
Fast Bit Counting Routines.
<http://infolab.stanford.edu/manku/bitcount/bitcount.html>,
2002
- [6] Hannu Kankaanpää,
Calculating Bezier curves fast via forward differencing.
<http://www.niksula.cs.hut.fi/hkankaan/Homepages/bezierfast.html>,
2001
- [7] Steve Gibson,
Sub-pixel Font Rendering Technology.
<http://www.grc.com/cttech.htm>,

Gibson Research Corporation,
2003-2007

- [8] Kim Øyhus,
SubLCD.
<http://www.oyhus.no/SubLCD.html>,
2006
- [9] Roger D. Hersch,
Font Rasterization: the State of the Art.
<http://diwww.epfl.ch/w3lsp/publications/typography/frsa.pdf>,
Visual and Technical Aspects of Type, Cambridge University Press,
1993
- [10] *Microsoft Typography - Features of TrueType and OpenType*.
<http://www.microsoft.com/typography/SpecificationsOverview.msp>,
Microsoft,
2004
- [11] Charles Loop and Jim Blinn
Resolution Independent Curve Rendering using Programmable Graphics Hardware.
<http://research.microsoft.com/cloop/LoopBlinn05.pdf>,
Microsoft,
2005
- [12] Yoshiyuki Kokojima, Kaoru Sugita, Takahiro Saito and Takashi Take-
moto
Resolution Independent Rendering of Deformable Vector Objects using Graphics Hardware.
ToshibaCorp.
- [13] Bresenham, J.E.
Algorithm for Computer Control of a Digital Plotter.
<http://www.research.ibm.com/journal/sj/041/ibmsjIVRIC.pdf>,
IBM Systems Journal, Vol. 4, No. 1,
1965
- [14] *What is Unicode?*.
<http://www.unicode.org/standard/WhatIsUnicode.html>,
Unicode, Inc.
1992-2008
- [15] *Subpixel rendering*.
http://en.wikipedia.org/wiki/Subpixel_rendering,
2008

- [16] Juan Pineda
A parallel algorithm for polygon rasterization.
Computer Graphics vol. 22, issue 4
1988
- [17] *v-rocs - Visual Rendering of Complex Scripts.*
www.mobilelabs.se/vrocs.php,
Mobile Labs Sweden AB,
2008
- [18] Jens Alfke *Converting Bezier Curves to Quadratic Splines.*
steve.hollasch.net/cgindex/curves/cbez-quadspline.html,
1994