MASTER'S THESIS 2020

Rendering Resolution Independent Fonts in Games and 3D Applications

Olle Alvin

ISSN 1650-2884
LU-CS-EX: 2020-14

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-14

Rendering Resolution Independent Fonts in Games and 3D Applications

Olle Alvin

Rendering Resolution Independent Fonts in Games and 3D Applications

Olle Alvin
tpi14oal@student.lu.se

April 7, 2020

Master's thesis work carried out at EA DICE, Stockholm.

Supervisors: Michael Doggett, michael.doggett@cs.lth.se
Göran Syberg Falguera, goeran.syberg@dice.se

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

This thesis describes how fonts can be rendered in 3D-applications. It provides a description of how to render glyphs using signed distance fields as well as the Slug-algorithm. The thesis provides an analysis of these methods and investigates how they can be combined. The combined method overcomes some of the artefacts produced by signed distance fields, while being faster than the Slug-algorithm in most cases. However it is not suited for complex glyphs or small font sizes.

Keywords: Graphics, Fonts, SDF, Slug

Acknowledgements

I would like to thank my supervisor at Lund University, Professor Michael Doggett for guidance and support during this project. I also want to thank Göran Syberg Falguera, my supervisor at EA DICE, for great feedback and discussions. I wish to thank Jason Chan for great discussions and advice on implementation and code. Finally I want to thank my family and friends for their support throughout the project.

Contents

Chapter 1
Introduction

Drawing text is a vital part of most computer software that needs to interact with the user. A video game is a very interactive application that constantly feeds the user new information to act on. As games and other similar 3D applications usually have a specific graphical style, there is a need to use custom fonts that fit the aesthetic of the application. It is common to use intricate user interfaces, where pieces of text are animated, scaled and changed frequently. The fonts need to look crisp on a variety of displays and resolutions and must be drawn within a few milliseconds. Usually games utilize powerful graphics hardware for drawing resolution independent text, as this is present in most gaming consoles and PCs. Some popular solutions are to cache pre-generated glyphs in a texture, or use signed distance fields but in the last couple of years other alternatives have surfaced. In this thesis we explore some of the techniques for drawing resolution independent text, as well as investigate different ways of combining the methods. We aim to create a combined solution that has not yet been tried, and determine its usefulness.

1.1 Graphics Hardware

The applications we are targeting in this thesis relies heavily on graphics hardware such as GPUs for drawing to the screen. A GPU (Graphics Processing Unit) is a processor which excels at converting graphical primitives to a pixel representations. A pixel contains a color, typically as an RGB value. The GPU writes the color values into a data buffer (or framebuffer) that contains every pixel on screen. This conversion is referred to as rasterization. In a high end graphics systems, GPUs are usually located on a graphics card. Modern graphics cards contain thousands of processors, which can be programmed to run small programs called shaders. There are a few different shader types, but most graphics application run at least a vertex shader and a pixel shader. The vertex shader, processes the individual vertices that make the primitives. Its main function is to apply coordinate transformations as well as manipulate any vertex attributes. The pixel shader determines the color for each pixel in the

framebuffer. It can be altered in a lot of ways, usually by applying textures or different light models.

1.2 Definition of a Font

A font usually refers to a collection of glyphs that share certain design features. Today, most digital fonts store a mathematical representation of the glyph outlines. Commonly used formats such as TrueType and OpenType represent glyph outlines as a set of Bézier curve segments and straight lines. TrueType uses quadratic Béziers [2], sometimes referred to as conics. We will use the term "font" more loosely, and refer to a font as a set of glyphs in the same font file. In TrueType and OpenType font formats the the outline is defined in a design space. The glyph outline is positioned relative to a grid known as the em-square, see Figure 1.1. The square has a side of 1 em.

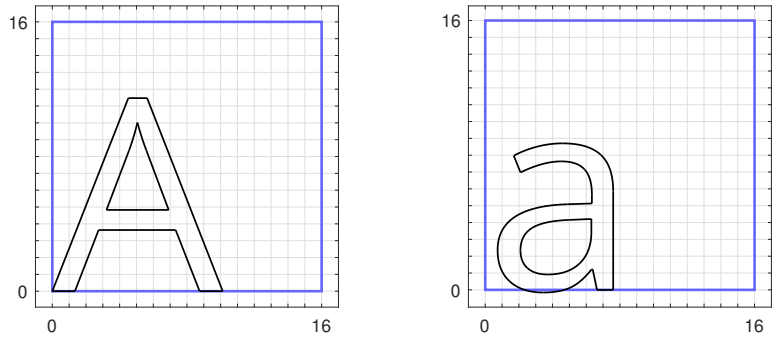

Figure 1.1: The outline of "A" and "a". The em-square is drawn in blue. In this example, an em is 16 pixels wide. This would correspond to a 12pt font on a 96 DPI screen.

An em is the size of the font. In a 12pt font an em is 12pt. In design units, an em is usually a fixed size. For TrueType fonts this is normally 2048 units or 1024 units. Note that the design of a glyph may extend outside the em-square. An example of a glyph outline with the Bézier control points drawn can be seen in Figure 1.2.

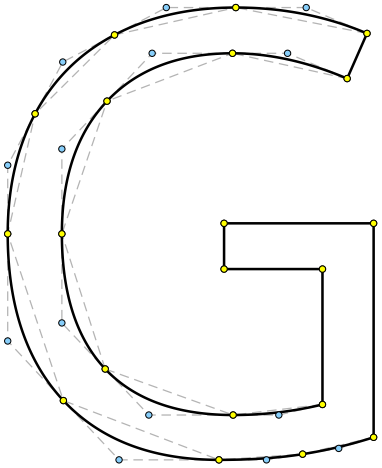

Figure 1.2: The outline of the letter G.

TrueType files hold a lot more information that help make text more readable such as hinting, kerning and ligatures. These are not considered in this thesis, as we will focus on the rasterization process.

1.3 Rasterizing a Glyph Outline

Any font rasterizer must convert outline of a font into a pixel representation. We can sample the outline at the pixel center to determine if the pixel is within the outline. How the pixel coordinates are mapped to the design space is determined by the size of the font and the DPI on the screen. A 23.6″ 1920×1080 screen has a DPI at about 96. A 12pt font at a 96 DPI would make the em-square 16 pixels wide or 16px, see Figure 1.1. We will denote font sizes by pixels/em or px in this thesis. A size of 16px means that most glyphs will only be sampled 16 or less times in the x- and y-direction. This usually means the glyph is largely under-sampled, causing aliasing artefacts. This is why many professional printers can print with over 1000 DPI. On a computer screen the problem needs to be handled in another way. To make the text less jagged usually the pixels are sampled at several locations and the alpha value of the pixel adjusted based on coverage. This produces a smoothing effect (anti-aliasing), which makes text more readable. An example of this can be seen in Figure 1.3.

Figure 1.3: A glyph rasterized at 24px with different subpixel sampling rate. From left to right 1, (2×2), (4×4), (8×8), (16×16), (32×32), (64×64) samples per pixel.

1.4 Previous Work

1.4.1 Pre-rasterized Glyphs

The most straight forward way to render text inside of a 3D game engine is to pre-rasterize the glyphs into a single image containing all the glyphs. This image is called a texture atlas. Each individual glyph is typically mapped onto two triangles making a quadrilateral (quad). This is very efficient and produces great quality when rendering static text. However, if the text is scaled to a larger size the magnification on modern GPUs often produces a blurry image. To maintain good text quality, all sizes used would need to be stored in the texture atlas or rasterized at runtime on the CPU, which is often too slow for computer games.

1.4.2 Signed Distance Fields

In 2007 Chris Green at Valve [3] introduced the technique of rendering glyphs using signed distance fields (SDF). An SDF is a grid that contains samples of the distance to the closest edge of a glyph. The sign of the distance is used to indicate if the point was sampled on the inside or the outside of a glyph. The samples are stored in a texture, which can be mapped to a quad. Using the bilinear interpolation on the GPU we can access an interpolated distance value in the pixel shader. This makes the method resolution independent, which eliminates the problem of having to store a texture for every size of a glyph. The interpolation produces some artefacts where sharp corners appear rounded at higher resolutions. It also requires higher resolution SDFs to represent more complicated shapes. A solution to the rounding problem suggested by Green was to use more several distance fields to represent different edges that make up the corner. This concept was further developed by Victor Chlumský [5] who showed that problem could be solved for an additional cost in performance. Both single-channel SDFs and multi-channel SDFs suffer from quality issues at small sizes since a single distance value cannot represent several curves intersecting a pixel. Due to the structure of an SDF some special effects can be implemented ”for free”. It allows for fast simple antialiasing, outlining, dropshadows and glow effects.

1.4.3 Loop Blinn Curve Rendering

A method for rendering vector graphics on the GPU including fonts appears in Loop and Blinn [8]. This method can render fonts exactly from the outline data by creating a triangle mesh from the outline control points. Each Bézier curve segment of a glyph outline is represented as a triangle. A simple calculation in the pixel shader can be used to determine if a point is on the inside of a glyph outline. The triangulation step is quite complicated and for complicated glyphs the triangle count could reach large numbers for each glyph. Also at small sizes when several of the outline curve segments intersect a pixel, the pixel shader will only use one of them to determine the color. This causes an incorrect representation. Anti-aliasing also requires additional triangles to be added to the outside of the glyph mesh or the use of super sampling.

1.4.4 Glyphy

Glyphy [10] is an SDF renderer that instead of sampling the SDF into a texture, computes the signed distance field on the GPU. The Bézier curves of a font outline are approximated using circular arcs. Then the pixel shader computes the distance to the closest arc. A Bézier segment could require several arcs to be approximated with a low error. This means that a lot of arcs need to be checked at each pixel.

1.4.5 Dobbie's Method

Will Dobbie [9] solved the issue of Loop Blinn's high triangle count by only using a quad for each glyph. His method stores the outlines of a font in a data texture. The pixel shader then determines coverage by casting rays in several directions and find intersections with the outline curves. In order to optimize performance, the outline representation is chopped up into a grid so that each pixel only consider the curves in the same grid cell. This causes problems at smaller sizes where a single pixel covers several grid cells. Entire cells get skipped over and the text starts to produce shimmering artefacts. Dobbie solves the problem by switching to pre-rasterised glyphs when text gets too small. The method also suffers some incorrectly drawn pixels due to numerical precision issues at larger font sizes.

1.4.6 Slug Algorithm

Eric Lengyel [4] introduced another method of rendering vector fonts onto quads. He uses an efficient way of computing the winding number of the glyph outline at any point in the pixel shader. By introducing a binary classification strategy for the Bézier curve segments numerical robustness is guarantied. This method produces a very good result at any resolution but good anti-aliasing is very costly. The method is similar to Dobbie's method in many ways but is not as prone to artefacts. Performance-wise it is many times slower than the texture based methods. This method has since Lengyel's paper been developed into a library called Slug, hence the name [7].

1.5 Scope and Approach of the Thesis

These methods described in section 1.4 are only some of the different ways fonts can be rendered. These were chosen as they were the most commonly referenced in published works and by game developers consulted in the project. The amount of different approaches is to many to evaluate thoroughly in the time span of the project. Therefore it was narrowed down to three. The standard method of using pre-rasterized glyphs is the fastest and simplest solution. Therefore, it was considered the default method, which all other should be compared to. Then the obvious choice was to further analyze Signed Distance Fields as it seem to be widely used in the game industry and is both fast, simple to implement and works well. The newest research we encountered was done by Eric Lengyel, who provided the Slug algorithm. This is very different from any texture based method and also seemed much more flexible than both Loop Blinn curve rendering and Dobbie's method. Also, there is little work apart from Lengyel's documenting this method. To be able to understand the strength and weaknesses

of the different algorithms we needed to find ways of comparing them both in terms of performance and quality. Then finally, when understanding some parts of the field of GPU font rendering, the task was to see if there are any other unexplored approaches or optimizations that could be done. After trying various ideas the most promising was to attempt combine the two previously mentioned methods. This summarizes the thesis down to a few points:

- Implement three different font rendering techniques for comparison. These are:
  - Using pre-rasterized glyphs in a texture atlas.
  - Using signed distance field representations in a texture atlas.
  - Using the Slug algorithm.
- Create benchmarks that can be used to compare the fonts in both rendering performance and image quality. A few limitations have been made to enable this:
  - Text is rendered onto a 2-dimensional plane, parallel to the screen as opposed to being view from an angle.
  - Text is rendered in only one color, which represents the alpha value. (In the thesis, the color of the text is black as it is shown on a white background, meaning that black represents an alpha value of 1).
  - The alphabet used is the symbols of the ascii-table. There are plenty of different alphabets and languages, and testing them all is not feasible.
  - Three different fonts are used to test how different designs affect the quality and performance. They all use the TrueType font format.
- Explore various ways to combine the SDFs and Slug and compare using the same benchmarks.

The task was accomplished with a C++ implementation using OpenGL. OpenGL is a graphics API which provides much of the needed functionality for this project, such as loading primitives and texture data to the GPU. It also provides the shader language GLSL for shader code. OpenGL was mainly chosen as the author is familiar with it.

Chapter 2
Bézier Curves

As fonts are defined from Bézier curves we need to understand the definition of such a curve as well as some fundamental properties. Gerald Farins book [1] provides an excellent overview of Bézier curves and its properties. A Bézier curve is a parametric curve defined from a set of control points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n$. The curve is given by

$$C_n(t) = \sum_{i=0}^{n} \mathbf{p}_i B_{i,n}(t), \tag{2.1}$$

where $B_{i,n}(t)$ is the Bernstein polynomial

$$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{2.2}$$

and $t \in [0, 1]$. The first 3 orders of Bézier curves are

$$\begin{aligned}
C_1(t) &= \mathbf{p}_0(1-t) + \mathbf{p}_1 t, \\
C_2(t) &= \mathbf{p}_0(1-t)^2 + \mathbf{p}_1 2t(1-t) + \mathbf{p}_2 t^2, \\
C_3(t) &= \mathbf{p}_0(1-t)^3 + \mathbf{p}_1 3t^2(1-t) + \mathbf{p}_2 3t(1-t)^2 + \mathbf{p}_3 t^3.
\end{aligned}$$

There are some useful properties of Bézier curves which we will list as two theorems.

Theorem 1. *Bézier curves are invariant under affine transformation.*

Theorem 2. *Consider a Bézier curve $\boldsymbol{C}_n(t)$ as defined in (2.1). At the first and last control points $\boldsymbol{p}_0$ and $\boldsymbol{p}_n$ the direction of the tangent is given by the vectors $\boldsymbol{p}_1 - \boldsymbol{p}_0$ and $\boldsymbol{p}_n - \boldsymbol{p}_{n-1}$.*

We will make use of these properties when working with TrueType outlines in section 3.1 and 5.1.

Chapter 3

SDF

A signed distance field is essentially a texture that, instead of storing an image, holds distances as the pixel color values. Specifically it holds the distance to the edge of a shape. The sign of the distance is used to tell if the pixel is outside or inside the shape. We use a positive value for distances inside and a negative values for outside the shape. This is applicable to glyphs, since they are 2-dimensional shapes. For a specific point in font design space, we can sample the distance to the glyph outline. We choose a sample rate of, i.e. 64 samples / em. After collecting the samples, they can be stored in a texture format to be accessible on the GPU. Although OpenGL supports floating point textures, usually SDFs are represented by 8 bit integer textures to minimize memory usage (more on this conversion in section 3.3). Figure 3.1 contains an example of an SDF texture and a glyph rendered from it.

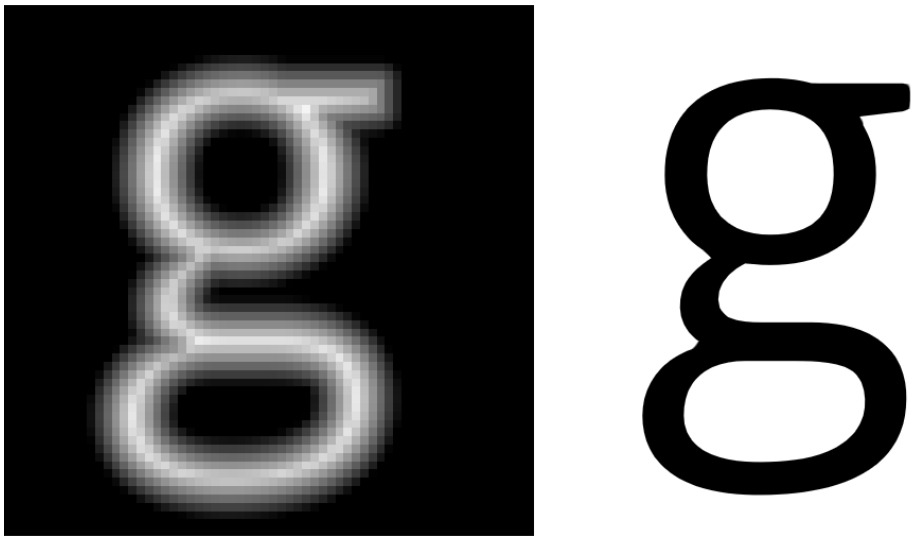

Figure 3.1: (Left) A 64×64 signed distance field of the letter g. (Right) A render of the letter g using the SDF on the left.

3.1 Generating Signed Distance Fields

There are different algorithms for computing an SDF from a glyph shape. Green [3] uses a brute force approach where the glyph is sampled at a very high resolution. Each sample is either ”inside” or ”outside” the glyph outline. When computing the distance for a point we can check the neighboring samples to find the closest one of the opposite type. This requires a lot of computations which makes it quite slow and using too low resolution can produce unwanted artefacts. Another way is to use the mathematical representation of the outline to compute the distance. The distance to the glyph could be obtained by computing the distance to the nearest outline segment. Chlumský presented an algorithm like this, in his thesis [5]. We will use a similar approach.

Consider the Bézier curve of 2nd order $\boldsymbol{C}(t)$. From theorem 1 we can conclude that a Bézier curve can be translated by simply translating the control points. Therefore we simplify by transforming the curve so that the sample point is at origin. The smallest distance to origin can be obtained by computing

$$d_{min} = \min_t \|C(t)\|, \quad t \in [0\ 1].$$

This is equvalent to minimizing $\boldsymbol{C}(t)^2$ which is done by finding roots of

$$\frac{d}{dt}C(t)^2 = 0$$

$$\Rightarrow \quad 2\mathbf{a}^2t^3 + 3\mathbf{ab}t^2 + (\mathbf{b}^2 + 2\mathbf{ac})t + \mathbf{bc} = 0.$$

where

$$\begin{aligned}\mathbf{a} &= \mathbf{p}_0 - 2\mathbf{p}_1 + \mathbf{p}_2\\ \mathbf{b} &= 2(\mathbf{p}_1 - \mathbf{p}_0)\\ \mathbf{c} &= \mathbf{p}_0.\end{aligned}$$

To solve this we need to find the roots of a 3rd degree polynomial. This can be done numerically or analytically by using Cardanos rule. The smallest distance is either one of the roots or the end points. We need to repeat this computation for every Beziér curve at every point. The sign on the distance is determined by whether the sample point is inside or outside the glyph. If the curves follow a winding order we can simply check if the point is on the right or left side of the curve. However, this would not work if a glyph has overlapping parts. Therefore we can determine if the point inside or outside with a winding number calculation. We will explain how this is done in section 4.1. With this method each sample point can be processed independently, and it can therefore be done with great efficiency on a GPU.

3.2 Multiple Channels

At larger sizes, low resolution SDFs make corners appear rounded. This effect comes from the bilinear texture interpolation, which is not really intended for distance values. A solution

for this is to use multiple color channels and store more SDFs. Such a solution was given by Victor Chlumský [5]. The plane around a corner is divided into 4 quadrants. The different quadrants are assigned to 3 color channels depending on the shape of the corner (convex or concave). The distance value is then computed from the median of the 3 color channels. Otherwise it is used the same way as a single channel SDF. We will use Victor Chlumský's msdfgen library [6] to generate msdf textures for comparison in the benchmarks.

3.3 Implementation

The SDF is stored as a texture in GPU memory. Typically it uses an 8 bit integer for each texel (a pixel in a texture). To ensure a good conversion to 8 bit, the user must choose the maximum distance to represent in the distance field. We refer to this as the "$spread$". Then the distance is mapped from $[-spread, spread]$ to $[0, 255]$. Let d_t be the distance value in texels. To convert to the color value d_c, d_t is first clamped between $-spread$ and $spread$. Then we compute

$$d_c = \frac{d_t + spread}{2 \cdot spread} \cdot 255,$$

and round to the nearest integer.

In the pixel shader we fetch the distance value from the texture. Texel values are given as a floating point value between 0 and 1. The value 0.5 represents a distance of 0 and means the pixel is on the outline exactly. To get smooth anti-aliasing on the edges we can choose a distance interval $[0.5 - \delta, 0.5 + \delta]$ and interpolate the alpha value of the final pixel color according to this interval. We can use the *smoothstep*-function provided by OpenGL but for this thesis we will instead use a linear step function as we found that it gives a slightly better approximation of the pixel coverage, see Figure 3.2. Figure 3.3 shows the edge of a glyph with the this function applied.

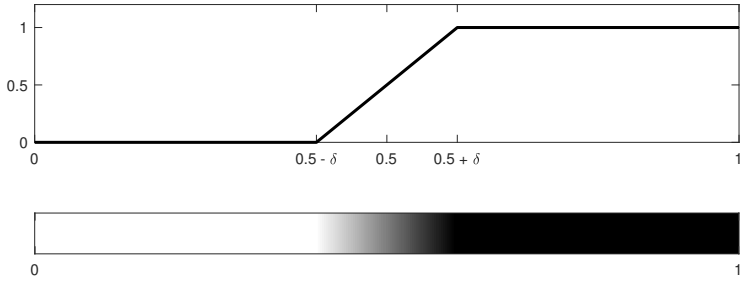

Figure 3.2: Plot of the step function used for setting the alpha value and the gradient. Here black is an alpha value of 1 (inside the glyph shape) and white is 0 (outside the glyph shape).

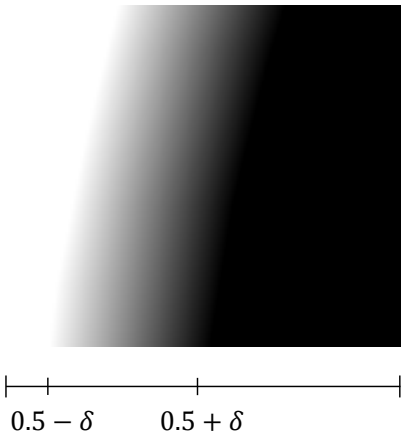

Figure 3.3: Image shows the alpha gradient on the edge of a glyph using a a linear step function.

The pixel shader processes pixels in groups of four. This allows us to access some values from neighboring pixels in the shader. GLSL gives us the functionality to get the derivatives of the texture coordinates in screen space, computed from the texture coordinates of the neighboring pixels. We can use the derivatives of the texture coordinates to keep the anti-aliasing border one pixel wide. In GLSL this would be provided by the *fwidth()* function.

The SDF images are combined into a single texture, referred to as a texture atlas. For testing purposes, a 1024×1024 atlas is used. For each glyph we create a quad (a rectangle made from two triangles), as in Figure 3.4. The quads are then positioned according to the glyph metrics specified in the font file and stored in a single vertex buffer on the GPU.

Figure 3.4: (Left) A set of quads with SDF textures. (Right) The quads on the left rendered with an SDF shader.

To be able to render more complex shapes such as decorative fonts we need to use a higher SDF resolution. In Figure 3.5 a glyph is rendered with five different SDFs. We use three fonts of varying complexity.

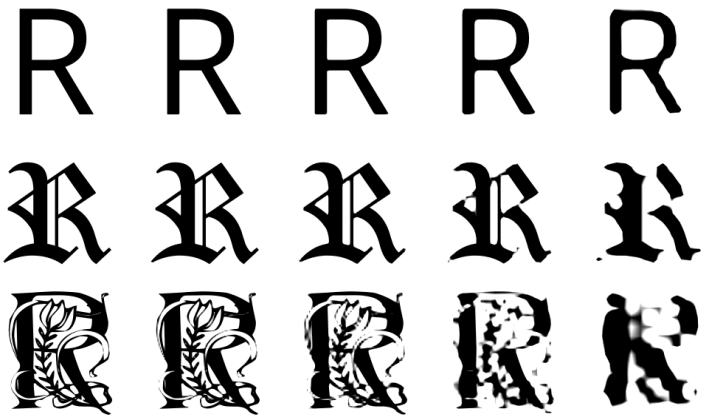

Figure 3.5: Renders using SDFs of different resolution. From left to right (256×256), (128×128), (64×64), (32×32), (16×16).

This shows some of the issues using SDFs. Glyphs with sharp corners appear rounded and thin features will suffer from visible artefacts. In Figure 3.5 we can see that the thinnest features of the "R" in row three are not preserved except with a 256×256 SDF resolution. For all three fonts the corners loose their sharpness for the lower SDF resolutions. This is more visible when the font is scaled up. With the SDF method of rendering fonts we make some sacrifices in quality for resolution independence.

Chapter 4
Slug Algorithm

The Slug algorithm does not use any pre-renderers or samples. Instead the glyphs are sampled directly from the actual outline in the pixel shader. The segments that make up the outline are stored in a texture for access in the shader. A glyph is rendered onto a quad sized as the glyphs bounding box. The texture coordinates at the four vertices are set to the coordinates of the corners of the glyphs bounding box, see Figure 4.1. That way the coordinates are interpolated at every pixel in the pixel shader.

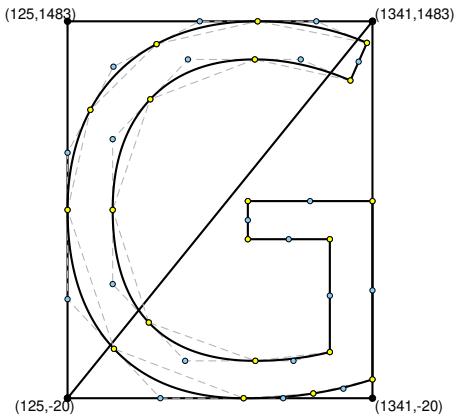

Figure 4.1: Quad with texture coordinates set to the glyphs bounding box.

Using a quad sized after the bounding box, it is possible to miss partially covered pixels that are just outside the edge of the box. Therefore the box must be slightly larger. We solve this by dynamically dilating the box by half a pixel width in the vertex shader. The texture coordinates are adjusted as well.

4.1 Computing the Winding Number

In general, a glyph shape can be sampled at any point by determining the winding number.

Definition: *For a closed curve in a 2-dimensional plane, the winding number for a point* $\boldsymbol{p} = (x, y)$ *is the number of times the curve loops clockwise around the point.*

A glyph outline is a set of closed contours. If it loops around a specific point, that point is inside the shape. One way to compute the winding number of a point $\mathbf{p}$ is to shoot a ray in any direction, originating from $\mathbf{p}$. Usually a TrueType outline follows a clockwise winding convention, meaning that the inside of a glyph is always to the "right" of the curve. Therefore whenever we intersect a curve from the right we add one to the winding number, and from the left, we subtract one from the winding number, see the example in Figure 4.2.

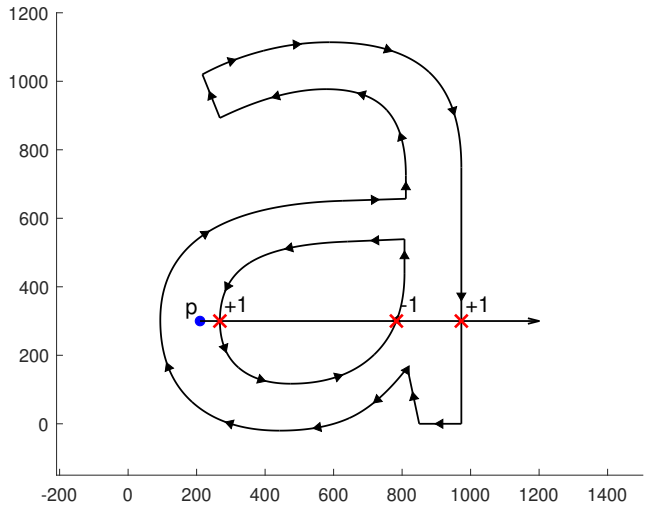

Figure 4.2: A ray is fired in positive x-direction. For every intersection the winding number is adjusted.

Then a non-zero winding number indicates that the point is inside the glyph shape, otherwise it is outside. Some fonts follow the opposite winding order, in that case the outline is simply reversed.

The procedure has two parts, compute the intersections between the ray and the outline and determine from what direction the ray hit. Lengyel[4] introduced a solution for this in his paper. Assume we shoot a ray in the positive x-direction. To find the intersections with the glyph outline, every curve needs to be evaluated. The curve is translated so that the pixel center is a origin, meaning that the intersections with the ray are the intersections with the x-axis. For a quadratic Bézier $C(t) = (C_x, C_y)$ the intersection points are given by the roots of C_y. We can write C_y as

$$C_y(t) = y_0(1-t)^2 + y_1 2t(1-t) + y_2 t^2$$
$$\Rightarrow C_y(t) = (y_0 - 2y_1 + y_2)t^2 - 2(y_0 - y_1)t + y_0.$$

By introducing

$$a_y = y_0 - 2y_1 + y_2$$
$$b_y = (y_0 - y_1)$$
$$c_y = y_0,$$

we get this simpler form

$$C_y(t) = a_y t^2 - 2b_y t + c_y.$$

We can write the solution to the equation as:

$$t_{1,2} = \frac{b_y \pm \sqrt{b_y - a_y c_y}}{a_y} \tag{4.1}$$

In the case that $a_y = 0$ the Bézier curve is a straight line. Then the intersection is instead given by

$$t_1 = \frac{c}{2b}.$$

The roots, t_1 and t_2 do not always contribute to the winding number. There is a way to classify them based solely on the y-coordinate of the control points of the curve segments. More specifically, we check if they are above or below the x-axis. This gives the eight possible combinations, which are listed in Table 4.1. These are referred to as equivalence classes. For each class we can determine which of the two roots t_1 and t_2 contribute to the winding number. In the table this is marked with a "1" for contribution. For a more detailed description of the classes, we refer to Lengyels paper [4]. The eight combinations for t_1 and t_2 listed in the two rightmost columns of Table 4.1 can be stored in a 16 bit look up table $T = 0010111001110100$. Given y_0, y_1 and y_2 we can look up which roots contribute by computing

$$((y_0 > 0)\,?\,2 : 0) + ((y_1 > 0)\,?\,4 : 0) + ((y_2 > 0)\,?\,8 : 0),$$

and right shifting T by that value. The two least significant bits give the corresponding bits in the table. This way we know what roots we need to consider. If they do not contribute to the winding number they are simply ignored.

Class	$y_2 > 0$	$y_1 > 0$	$y_0 > 0$	t_2	t_1
A	0	0	0	0	0
B	0	0	1	0	1
C	0	1	0	1	1
D	0	1	1	0	1
E	1	0	0	1	0
F	1	0	1	1	1
G	1	1	0	1	0
H	1	1	1	0	0

Table 4.1

If $C_x(t_1) > 0$ and the segment belongs to either **B**, **C**, **D** or **F** 1 is added to the winding number. If $C_x(t_2) > 0$ and the segment belongs to **C**, **E**, **F**, or **G** 1 is subtracted from the winding number.

There is a special case in class **C** and **F** where both roots are marked with 1, but none of them are real. This is avoided by clamping $b_y - a_y c_y$ to $[0, \infty)$. In case of unreal roots, we then get $t_1 = t_2$ and their winding number contributions cancel each other out.

To get anti-aliasing with this algorithm, pixel coverage is approximated using the intersection points of the rays. The intersection of a horizontal ray is given by $(C_x(t), 0)$. We can approximate the coverage with distance from the pixel center to the intersection point along the ray direction. If the intersection point is to the left or the right side of the pixelbox, the coverage is either 0 or 1. Otherwise it is given as

$$0.5 + f(C_x(t)),$$

where f is a function that converts design units to pixels, visualized in Figure 4.3. These coverage values are accumulated for every Bézier curve that intersects the ray.

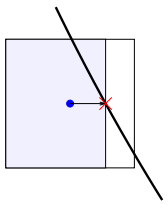

Figure 4.3: Image showing how coverage is approximated from ray-casting.

Rays are fired in both positive x- and positive y- direction to acquire two coverage values.

4.2 Implementation

Data Textures and Banding

To make the structure of the outline data simpler, every line segment is converted to a Bézier segment by inserting a third point in between the two points defining the line. The control points of the Bézier curves are stored in a 16 bit floating point RGBA texture. The first control points xy-coordinates are stored in the R- and G- channel, the second in the B- and A-channel. The last point is stored in the next texels R- and G- channels, see Figure 4.4. For most segments the last point of a curve is the first point of the next. Therefore the data can be shared between the two segments. The texture's width is set to 4096, and the number of rows chosen so that all curve segments can be stored. If a row is not entirely filled, the remaining values are set to 0. There is also a special case when a there is only one or two available texels at the end of a row, not enough to fit the three control points. In this case the color values are set to 0 and the next segment is stored at the beginning of the next row.

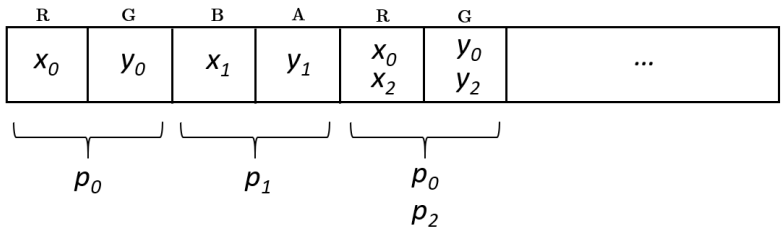

Figure 4.4: Structure of data texture. Each Bézier control point is stored as in 2 color channels. If the last control point of a segment is the same as the first point in the next segment, the data is shared.

To increase performance the bounding box of a glyph is divided into bands of equal width. Lengyel uses a varying amount of bands depending on the glyph with a maximum of 16. Using more bands increase performance but uses more memory. To keep things simple we use a fixed amount of 16 horizontal and 16 vertical bands. A list of all segments with control points within the band is stored in a texture. In the pixel shader we access a list of Bézier curves within the current band. This reduces the number of segments that need to be evaluated at each pixel. As an example, in Figure 4.5, for a pixel located in the bottom band, only four Bézier segments need to be evaluated.

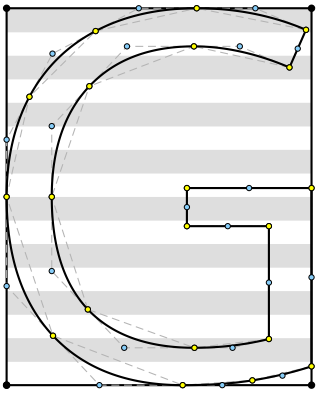

Figure 4.5: Image showing an example of horizontal banding.

The vertical and horizontal bands are stored in a 16 bit unsigned integer RG texture, see Figure 4.6. This texture contain a number of headers, one for each band. The headers store the number of bands in the texture as well as the offset to the list of curves intersecting the band. This information is contained in a single texel. The list contain the texel-coordinates for the curve segments in the data texture. For the horizontal bands the curves are sorted on the x-coordinate of the left most control point in ascending order. The vertical bands are stored in the same way except the lists are sorted on the y-coordinate of the lowest control point in the Bézier segments.

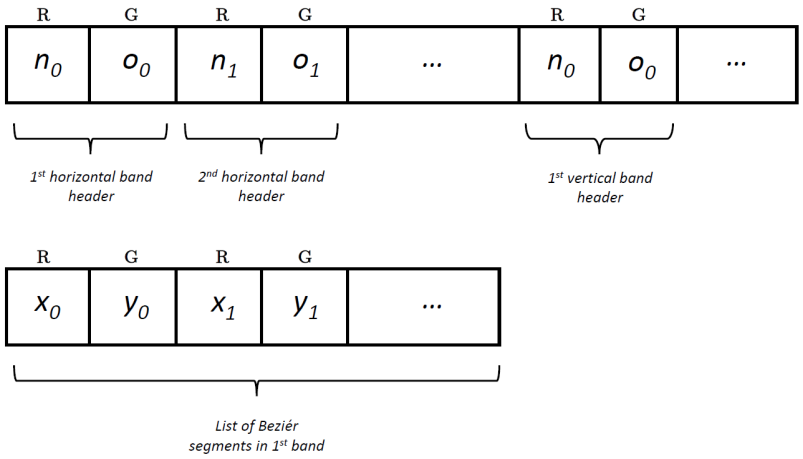

Figure 4.6: Structure of band texture. First comes a set of band headers, each consisting of one texel. Then follows the list of Bézier segments within each band.

The curve- and band-data for every glyph in a text are stored in the same textures. The offset in the band data texture is stored in the vertex buffer as a vertex attribute. The sorted lists of curves in each band allows for an early out condition in the pixel shader. If a curve is more than 0.5 pixels to the right of the pixel center, there are no other curves that will contribute to the coverage value an we can stop the loop. No more curves need evaluation. To identify which curve list to fetch the band must be determined. This is done by multiplying the texture coordinates with a scale-factor. This scale-factor is different for each glyph and is therefore stored as a vertex attribute. The basic shader is summarized in Listing 4.1:

```
1  procedure:
2      Determine which band contains the pixel.
3      Fetch the horizontal and vertical curve lists from the
4      band texture.
5      for each curve in horizontal list do
6          Compute intersections with horizontal ray and
7          accumulate coverages.
8      for each curve in vertical list do
9          Compute intersections with vertical ray and
10         accumulate coverages.
11     Average the 2 coverage values.
```

Listing 4.1: The basic Slug pixel shader.

4.2.1 Anti-Aliasing

For our implementation of anti-aliasing with the Slug algorithm we will try three different approaches. Averaging the the two coverages and the use of super sampling are the implementations suggested in Lengyel's paper. In addition we also try to weight the averages based on the derivatives of the Bézier curves.

Average Coverages

The pixel coverage is approximated in both the x- and y- direction. For 2D anti-aliasing we can average the coverages.

$$cov = \frac{1}{2} \cdot (cov_x + cov_y) \tag{4.2}$$

Super Sampling

For better anti-aliasing we can cast more rays. We cast three rays in each direction. One ray is fired from the pixel center, the other two are offset with a quarter pixel. For the horizontal rays we offset the rays in the y-direction, and compute the intersections for each one. Then we average the coverage over the samples. The same strategy is used for the vertical rays. This produces smoother edges. It does not require any additional texture look ups but the extra computations still come with cost in performance. Since the band is computed from the pixel center, the top or bottom part of the pixel can be in another band. This can make the curve list invalid for one of the offset rays. Therefore, the bands must be expanded in order for the curve lists to remain valid. This increases the number of curves evaluated at each pixel and causes more overlap of the bands. The expansion must be half a pixel width of the smallest font size we wish to draw.

Weighted Average

To simply average the coverages give decent anti-aliasing, but edges that are close to vertical or horizontal will still be slightly jagged. A way to improve upon this is to weigh the coverages differently. We can use the actual slope of the closest edge to find suitable weights. The derivate of a quadratic Bézier is

$$\frac{d}{dt}C(t) = (2t - 2)\mathbf{p}_0 + (1 - 2t)\mathbf{p}_1 + 2t\mathbf{p}_2. \tag{4.3}$$

When casting the rays in the pixel shader we simply use the closest one an compute the derivative at the intersection with the ray. The weights and coverage can then be computed as

$$(w_1, w_2) = \left(\left| \frac{\frac{d}{dt}C_y(t_0)}{\frac{d}{dt}C_x(t_0) + \frac{d}{dt}C_y(t_0)} \right|, \quad \left| \frac{\frac{d}{dt}C_x(t_0)}{\frac{d}{dt}C_x(t_0) + \frac{d}{dt}C_y(t_0)} \right| \right) \tag{4.4}$$

$$cov = cov_x \cdot w_1 + cov_y \cdot w_2 \tag{4.5}$$

This requires us to keep track of the closest curve as well as an extra texture look up and computation of the derivative. This will add a bit of computation time as we shall see in the results.

For the Slug algorithm, how we do anti-aliasing has a large impact on both performance and quality. Therefore, we will analyze these three strategies to determine how large the impact is and which one to use.

Chapter 5
Combining Methods

Signed distance fields and the Slug algorithm are very different in how they are used to render text. Slug offers great quality and is truly resolution independent. It requires more computation than texture based methods and is therefore much slower. Signed distance fields offers fast scalable text with great anti-aliasing but struggles to recreate corners in a satisfying way. To utilize the performance of SDF, we can render only the areas around the corners with Slug. Specifically we use a 64 samples per em SDF, and Slug with weighted average anti-aliasing, (denoted Slug+WA in result tables in chapter 7). We developed three different solutions for this.

1. Stencil buffer
2. Tiling
3. Combined shading

In our first approach (see section 5.2) we try separate the different part of the glyphs into separate primitives, so that they can rendered separately. To avoid overlap we utilize the stencil buffer, an integer data buffer present on most graphics hardware. In the second approach (see section 5.3) we do the same except that we omit the stencil buffer and divide the glyphs into tiles instead. We contruct two sets of tiles that are rendered with separate shader. In the third and final approach (see section 5.4) we separate different parts of the glyphs using a second SDF, and determine what algorithm to use within the pixel shader.

5.1 Identifying Corners

We define a corner as a point where two Bézier segments meet and the angle between the tangents is smaller than some threshold θ_{max}. Fortunately, finding the tangent of a Bézier curve of degree 2 at $\mathbf{p}_0$ or $\mathbf{p}_2$ is simple. As theorem 2 states, it is the lines defined by $(\mathbf{p}_0, \mathbf{p}_1)$

and $(\mathbf{p}_1, \mathbf{p}_2)$ respectively. So given two 2nd order Bézier curves $\boldsymbol{C}$ and and $\boldsymbol{C}'$ that share a control point $\mathbf{p}_2 = \mathbf{p}'_0$, the shared point is a corner if

$$\frac{\mathbf{p}_2 - \mathbf{p}_1}{\|\mathbf{p}_2 - \mathbf{p}_1\|} \cdot \frac{\mathbf{p}'_0 - \mathbf{p}'_1}{\|\mathbf{p}'_0 - \mathbf{p}'_1\|} > \cos(\theta_{max}). \tag{5.1}$$

The choice of θ_{max} depends on what we wish to classify as a corner. This parameter must be chosen by the user. It is not necessary to handle obtuse angles that are close to 180^o as the artefacts from the SDF are hardly noticeable in this case. This parameter is very much up to the user. For our tests we assume that corners with and angle larger than 170^o don't need any special handling. Therefore we choose θ_{max} to be 170^o for our tests.

5.2 Stencil Buffer

A simple way to combine these methods is to utilize the stencil buffer. As described in section 3.3, a signed distance field for each glyph is mapped onto a quad which is placed according to the text layout. These are pushed to a vertex buffer on the GPU, V_1. This set of primitives is rendered using the SDF method described in section 3.3. To be able to render the corner parts of the glyphs with Slug, we generate a second set of primitives. The outline is searched for corners using equation 5.1. For each corner we create a isosceles triangle primitive with the same angle and orientation as the corner. This second set of primitives is pushed to a second vertex buffer V_2. If the corners are close, some triangles will overlap. We do not want to draw the same thing twice so we combine any overlapping polygons into a polygon that covers them both. We compute the convex hull for the points of the 2 triangles. The convex hull makes a new polygon that we simply triangulate before pushing it to the vertex buffer. The convex hull of a set of points in a plane is computed using Jarvis algorithm [15].

V_2 is rendered first using the slug algorithm. Each pixel covered is marked with a 1 in the stencil buffer. For V_1 we perform a stencil test. All pixels marked with a 1 in the stencil buffer are discarded. This way, we avoid overdraw in these areas. We wish to keep the areas rendered with slug as small as possible, since the pixel shader is much slower than the SDF pixel shader due to all the computations needed. The bilinear sampling of the SDF texture is what is causing the rounding effect, see the 32×32 SDF renders in Figure 3.5. The GPU does the interpolation from the four texels closest to the sampling point. To avoid this, the triangle we create for the corner must have a side corresponding to at least 1 texel in the SDF. However, sharp angled corners make for very thin features of the glyph. If the resolution of the SDF is too low, this area is under-sampled, causing more artefacts, it is therefore a good idea to extend the slug area a bit more to improve the sharper corners. We let the triangles have the side of 2 texels. Also, each triangle is uniformly scaled slightly so that the edges extend a little bit beyond the glyph outline to allow for anti-aliasing. An example of vertex buffers generated this way is shown in Figure 5.1.

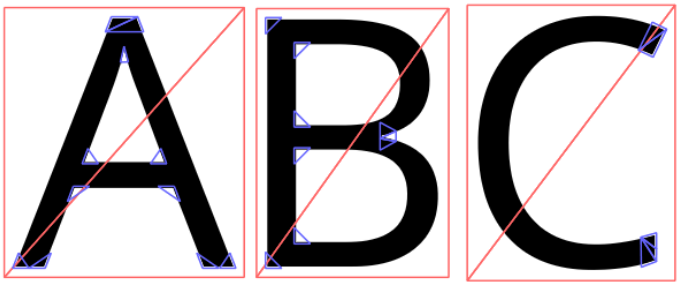

Figure 5.1: A rendering of glyphs using the algorithm above. The primitives are drawn in red and blue. Blue: rendered with the slug-shader, Red: rendered with the SDF shader.

5.3 Tiling

Using the stencil buffer is an easy and efficient solution, but we assume that glyphs do not overlap each other. Since all glyphs in a text use the same stencil buffer, it is possible they can draw over one another. A solution that avoids this problem can be implemented by dividing the glyph into tiles. Each individual tile can be rendered with either Slug or SDF and since they do not overlap, there is no need to use the stencil buffer. This requires more primitives than the stencil solution since each tile will be rendered as a quad. To minimize the triangle count, adjacent tiles that are rendered with the same shader can be merged into larger tiles. We use the largest rectangle search algorithm from Sanjiv Kumar's web article[13]. We represent the grid with a binary matrix M. Each element is marked with 1 for SDF, or 0 for Slug or vice versa. Then we merge the tiles by first finding the largest rectangle in the tile grid, create a new tile to replace these, then flip the corresponding bits in M, repeat. We list the steps in Listing 5.1.

```
1 procedure:
2     while M contains ones
3         Find the largest rectangle consisting of ones,
4         R, in M.
5         Create tile from R.
6         In M, set all elements in R to 0.
```

Listing 5.1: Algorithm for merching tiles.

Unlike the stencil solution we can not align the tiles with the corners, so we choose a shader for the tile based on distance to the closest corner. We can choose to render a tile with Slug if the distance to the closest corner is less than two texels. However, this will result in a lot of small tiles around the corners. Preferably we want the tiles to make up rectangles so they can be merged into larger tiles. So we instead classify a tile as a corner tile if it intersects a four texel wide square centered around a corner point. With a solution like this we run the risk of producing very large vertex buffer for the tiles. Also if tiles are to small they might not contribute at all to the final output. We therefore limit the minimum tile size to two pixels in width (according to the intended font size). This means that small fonts will have fewer tiles per glyph. An example of tiles produced by this algorithm is shown in Figure 5.2.

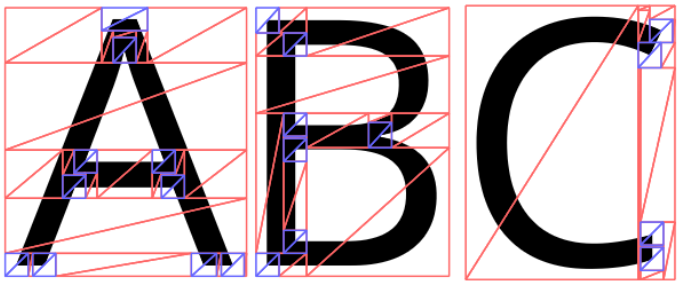

Figure 5.2: A rendering of glyphs using the algorithm above. The primitives are drawn in red and blue. Blue: rendered with the slug-shader, Red: rendered with the SDF shader.

5.4 Combined Shading

Using more polygons to separate the glyph into different areas will create large vertex buffers which will slow down the vertex shader. Both Slug and SDF only require one single quad per glyph. We can accomplish this for this hybrid approach as well by putting some extra work in the pixel shader. We utilize a second signed distance field, which stored distance to the corners of the glyph outline, see Figure 5.3.

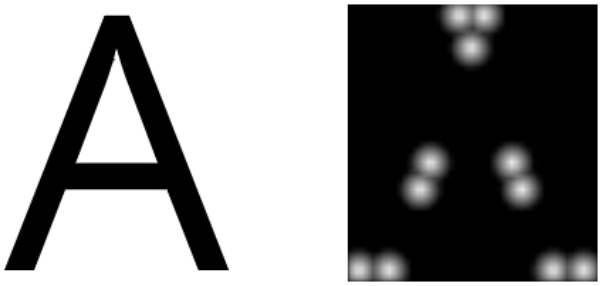

Figure 5.3: A signed distance field with distances to the corners of the letter A.

By sampling the distance field in the pixel shader we can determine what algorithm to use based on the distance. We use a 2-channel texture to store both the corner distance field and the outline distance field in the R- and G-channel respectively. Then the shader follows the format in listing 5.2.

```
1     t = sampleTexture()
2     if (t.r < 0.5)
3         renderWithSDF()
4         return
5     renderWithSlug()
```

Listing 5.2: The combined shader.

In Figure 5.4 we show how the different areas are separated.

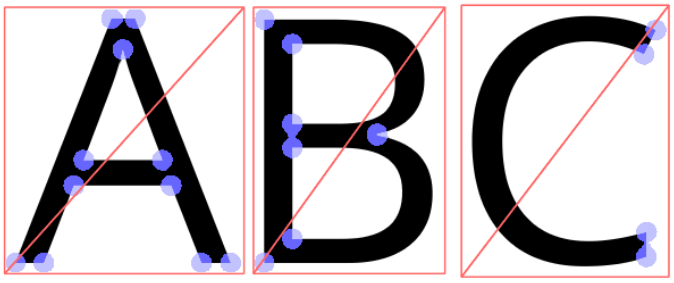

Figure 5.4: A rendering of glyphs using the algorithm above. The primitives are drawn in red. The area rendered with slug is marked in blue.

Chapter 6
Benchmarks

We use two benchmarks to compare the methods. We have a performance benchmark, which we use to compare rendering times. We also have a image quality benchmark that we use to compare quality of the renders produced by each method.

6.1 Experimental Setup

To test these font rendering methods we use a small OpenGL application written in C++. All tests run on a Windows PC with a xeon e5-1650 processor and a GTX 1070 graphics card. The benchmarks are designed to compare the font rendering techniques to each other in a couple of different scenarios. The quality benchmark requires a reference render for comparison. For this we have written a simple glyph rasterizer which samples the glyph 4096 times at each pixel in a 64×64 grid. This provides a very high quality render.

We will test three different fonts of varying complexity, see section 6.2. For the SDFs/MSDFs we will use two resolutions, 32 samples/em and 64 samples/em. For the last test font we will instead use 64 samples/em and 128 samples/em, as this font is more complex and requires higher resolutions.

6.2 Test Fonts

To investigate these methods we will render text using three different fonts of varying complexity, see Figure 6.1. The purpose of this is to evaluate how well Slug and SDF preserves small details. Also, we wish to test how the methods compare in rendering time for fonts with different complexity. More complex font designs need more curves to define the glyph outlines which directly affects the performance of the Slug algorithm and therefore also the combined methods.

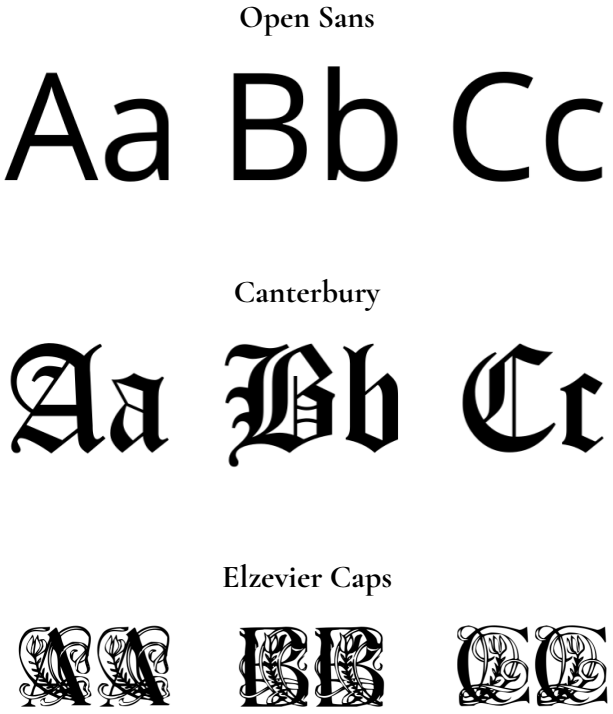

Figure 6.1: The three fonts used in the benchmarks.

6.3 Performance

To compare performance of the rendering methods we measure average render times for large amounts of text. Using our application we create a window with a resolution of 2000×1000. The same string of text is used for every benchmark. We pick a substring large enough to fill the entire window. Note that due to the design differences of the fonts, a fixed space of text might contain a different amount of glyphs. We will state the amount of glyphs rendered in the result tables. We render five different sizes of text: 12px, 24px, 48px, 96px and 192px. For the all methods but the combined methods in section 5.2 and 5.3(stencil and tiling), all triangles needed to display the entire text is pushed to a single vertex buffer and drawn using a single draw call to OpenGL. The other two methods use two separate vertex buffers and is therefore drawn with 2 draw calls. The transform of the text is updated every frame and the textures needed for each type of text bound. We render several frames of the same text and measure the GPU rendering time of each frame. The rendering times is measured with OpenGL by using Querys [11]. This way we can isolate the measurements to the GPU and provide an approximation of the added rendering time if these methods were to be used in a game engine. The difference in computation time on the CPU between the methods we consider is negligible. The benchmark measures the average rendering time over 10000 frames. To reduce the impact of other processes running on the same machine, we run each benchmark three times and average the result.

6.4 Image Quality

We consider a finished render good if it accurately depicts the pixel coverage of the glyph outline. Therefore we compare our renders to a reference and compute the Root Mean Square Error of the pixels. We only consider one color channel. Each pixel has a gray-level between 0 and 255. Let $\mathbf{x}_i = (x_0, x_1, \ldots, x_N)$ be the pixel values of the render we wish to test and $\mathbf{x}'_i = (x'_0, x'_1, \ldots, x'_N)$ be pixel values in the reference render. The RMSE is computed according to:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(x_i - x'_i)^2}. \tag{6.1}$$

We compute these for the letters and symbols in the ascii table and then average the result over all the glyphs. For the Elzevier caps font, we use only the capital letters as our test set, as there are no lower case letters in the font. The grid of pixels used in equation (6.1) is chosen so that it covers all glyphs in the set. We test five different sizes of text: 12px, 24px, 48px, 96px, 192px. We do this to compare how well fonts scale using the different methods. This is important since using SDFs result in round corners at large sizes (see Figure 3.5), so we would expect it to perform worse than the Slug algorithm for large fonts.

Chapter 7
Results

7.1 Performance Benchmark

The result of the performance benchmark, described in section 6.3 can be seen in Table 7.1, 7.2, and 7.3. SDF, MSDF, Slug and the combinations are represented. For slug we include the different anti-aliasing strategies, as the differ in performance, see section 4.2.1. We denote these **Slug** for simply averaging the horizontal and vertical rays, (**Slug + SS**) for using super sampling and **Slug + WA** for using weights based on Bézier derivatives, to combine coverages. For comparison, a benchmark of text rendered with a pre-rasterized buffer is also included. The combined approach always uses an SDF sampled a 64/em and slug with weighted average anti-aliasing (**Slug+WA**).

Table 7.1: Average GPU-time in milliseconds using the Open Sans font.

Method	192px (92 glyphs)	96px (379 glyphs)	48px (1508 glyphs)	24px (6243 glyphs)	12px (25046 glyphs)
Pre-rasterized glyphs	0.0131	0.0121	0.0162	0.0251	0.0436
SDF	0.0168	0.0189	0.0205	0.0272	0.0438
MSDF	0.0184	0.0181	0.0205	0.0406	0.0980
Slug	0.156	0.193	0.256	0.370	0.583
Slug + SS	0.231	0.294	0.392	0.589	0.962
Slug + WA	0.170	0.213	0.286	0.414	0.652
Stencil	0.127	0.100	0.0829	0.132	0.191
Tiling	0.121	0.0755	0.124	0.218	0.426
Combined shading	0.123	0.148	0.188	0.273	0.362

Table 7.2: Average GPU-time in milliseconds using the Canterbury font.

Method	192px (111 glyphs)	96px (451 glyphs)	48px (1804 glyphs)	24px (7450 glyphs)	12px (29857 glyphs)
Pre-rasterized glyphs	0.0126	0.0130	0.0173	0.0260	0.0468
SDF	0.0184	0.0194	0.0216	0.0293	0.0479
MSDF	0.0188	0.0187	0.0224	0.0410	0.0972
Slug	0.202	0.267	0.365	0.527	0.814
Slug + SS	0.334	0.451	0.631	0.943	1.50
Slug + WA	0.220	0.292	0.401	0.582	0.896
Stencil	0.133	0.140	0.178	0.329	0.431
Tiling	0.142	0.233	0.381	0.659	0.920
Combined shading	0.209	0.277	0.375	0.552	0.708

Table 7.3: Average GPU-time in milliseconds using the Elzevier Caps font.

Method	192px (57 glyphs)	96px (235 glyphs)	48px (946 glyphs)	24px (3934 glyphs)	12px (15733 glyphs)
Pre-rasterized glyphs	0.0187	0.0197	0.0232	0.0299	0.0428
SDF	0.0234	0.0237	0.0266	0.0325	0.0464
MSDF	0.0248	0.0231	0.0279	0.0405	0.110
Slug	1.35	1.71	2.22	3.21	5.14
Slug + SS	2.84	3.68	4.95	7.18	11.3
Slug + WA	1.45	1.84	2.46	3.61	5.67
Stencil	1.71	2.27	3.04	4.61	6.93
Tiling	1.52	2.25	3.167	4.19	5.65
Combined shading	1.38	1.84	2.47	3.48	5.18

7.2 Quality Benchmark

The result of the Quality benchmark in section 6.4 can be seen in Table 7.4, 7.5 and 7.6. The result is represented with the mean of the RMSE error of the glyph set as well as the variance in parenthesis.

Table 7.4: RMSE error of a set of glyphs from the Open Sans font. Result format is *mean* (*variance*). Lower is better.

Method	12px	24px	48px	96px	192px
SDF (64 × 64)	8.04 (5.32)	2.29 (0.806)	1.37 (0.276)	1.95 (0.47)	2.92 (0.94)
SDF (32 × 32)	5.86 (4.20)	3.13 (1.40)	3.73 (2.04)	5.60 (3.71)	7.25 (5.32)
MSDF (64 × 64)	13.1 (12.6)	2.81 (1.45)	1.71 (0.354)	1.15 (0.142)	1.10 (0.273)
MSDF (32 × 32)	6.47 (6.05)	3.26 (1.85)	2.15 (0.617)	2.05 (0.990)	2.48 (1.99)
Slug	12.0 (14.3)	9.19 (9.04)	6.71 (4.06)	5.08 (2.92)	3.36 (1.02)
Slug + SS	5.48 (5.85)	4.49 (2.61)	2.58 (1.51)	2.16 (1.02)	1.55 (0.56)
Slug + WA	6.26 (4.08)	3.72 (1.90)	2.53 (0.874)	1.72 (0.437)	1.15 (0.289)
Stencil	7.57 (4.44)	2.35 (0.801)	1.51 (0.250)	1.10 (0.145)	0.885 (0.169)
Tiling	7.02 (4.12)	2.63 (0.985)	1.67 (0.320)	1.12 (0.154)	0.864 (0.143)
Combined shading	7.74 (4.33)	2.36 (0.739)	1.56 (0.278)	1.11 (0.168)	0.894 (0.224)

Table 7.5: RMSE error of a set of glyphs from the Canterbury font. Result format is *mean* (*variance*). Lower is better.

Method	12px	24px	48px	96px	192px
SDF (64 × 64)	9.62 (13.54)	3.81 (2.65)	2.87 (2.50)	3.53 (2.91)	4.56 (3.47)
SDF (32 × 32)	7.13 (9.14)	5.65 (8.09)	8.61 (23.4)	12.4 (49.7)	15.4 (72.6)
MSDF (64 × 64)	14.6 (29.2)	6.06 (20.5)	4.41 (23.8)	3.71 (28.3)	3.50 (29.8)
MSDF (32 × 32)	9.35 (22.3)	7.08 (27.1)	7.55 (35.5)	9.33 (58.2)	11.2 (75.8)
Slug	13.5 (24.1)	10.0 (13.8)	6.86 (8.14)	4.95 (3.38)	3.67 (1.58)
Slug + SS	5.51 (7.94)	4.46 (5.05)	3.01 (2.28)	2.21 (1.26)	1.57 (0.653)
Slug + WA	7.85 (9.76)	5.32 (3.89)	3.34 (1.83)	2.34 (0.779)	1.64 (0.411)
Stencil	8.92 (12.8)	4.17 (2.74)	2.86 (2.31)	2.32 (1.82)	1.72 (0.811)
Tiling	8.14 (7.98)	4.84 (3.37)	2.20 (1.80)	2.21 (1.28)	1.61 (0.691)
Combined shading	9.04 (12.9)	4.11 (2.48)	2.84 (1.93)	2.21 (1.59)	1.69 (0.95)

Table 7.6: RMSE error of a set of glyphs from the Elzevier Caps font. Result format is *mean* (*variance*). Lower is better.

Method	12px	24px	48px	96px	192px
SDF (128 × 128)	36.7 (24.0)	20.6 (6.22)	12.5 (3.03)	9.32 (1.48)	9.87 (1.42)
SDF (64 × 64)	25.8 (12.8)	17.8 (3.91)	15.96 (3.97)	23.6 (9.30)	32.3 (20.1)
MSDF (128 × 128)	42.4 (76.2)	41.3 (179)	44.5 (423)	46.0 (521)	47.5 (553)
MSDF (64 × 64)	40.6 (173)	44.5 (367)	48.7 (528)	53.4 (516)	58.2 (487)
Slug	25.5 (16.9)	21.6 (7.74)	16.76 (3.70)	11.4 (1.31)	8.23 (0.507)
Slug + SS	12.3 (4.31)	10.8 (2.84)	8.24 (1.31)	5.66 (0.555)	4.19 (0.446)
Slug + WA	17.1 (7.94)	12.93 (2.80)	9.23 (0.852)	5.84 (0.259)	4.13 (0.268)
Stencil	18.4 (11.7)	13.1 (2.86)	9.54 (1.24)	6.75 (2.27)	5.88 (6.59)
Tiling	17.2 (7.82)	13.1 (2.92)	9.67 (1.13)	10.2 (3.31)	14.2 (8.69)
Combined shading	22.0 (12.5)	14.7 (3.60)	11.3 (1.73)	13.6 (5.09)	17.8 (12.0)

7.3 Slug Anti-Aliasing

We test the different anti-aliasing strategies for Slug, as described in section 4.2.1. A text rendered using the different strategies can be seen in figure 7.1. The top text is rendered by averaging coverages. This is the fastest according to our performance benchmark in section 7.1. The middle text uses super sampling to achieve better anti-aliasing. The bottom text, uses a weighted average of the coverage values which is also better than just taking the average. From the image it is clear that the two latter ones achieve smoother edges. This is especially visible on the horizontal and vertical parts of the glyphs. On the leftmost part of the "C", "a", "c" and "d" the top text is noticeably more jagged. Between the two lower texts it is hard to tell which is better with the naked eye.

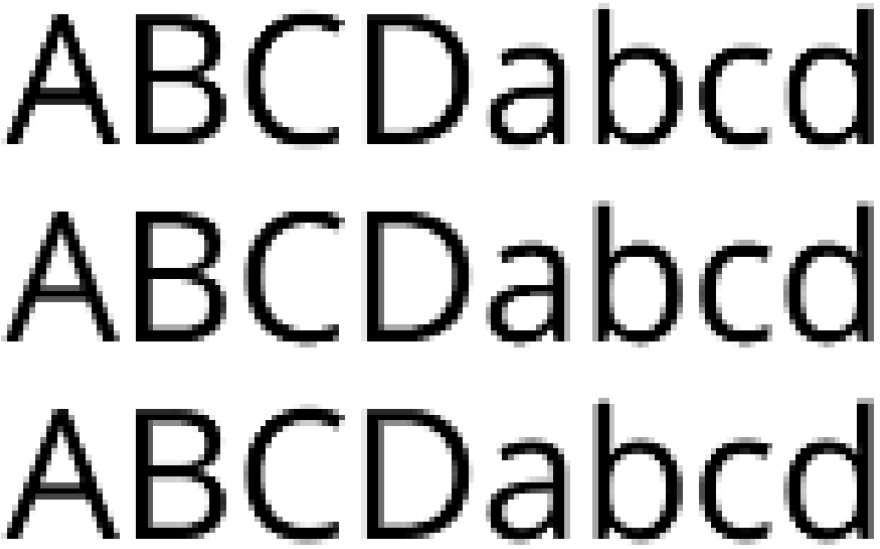

Figure 7.1: Text rendered using the Slug algorithm. (Top) **Slug**, (Middle) **Slug+SS**, (Bottom) **Slug+WA**

Chapter 8
Discussion

From our tests we can get a clear idea of what we must sacrifice in performance and quality to have a resolution independent solution based around these methods.

8.1 Rendering Performance

To compare the rendering performance we will discuss the relative performance to using pre-rasterized glyphs. This is shown in Table 8.1, 8.2 and 8.3.

Table 8.1: Relative rendering performance using the Open Sans font.

Method	192px (92 glyphs)	96px (379 glyphs)	48px (1508 glyphs)	24px (6243 glyphs)	12px (25046 glyphs)
Pre-rasterized glyphs	1	1	1	1	1
SDF	1.28	1.57	1.27	1.09	1.00
MSDF	1.40	1.50	1.26	1.62	2.25
Slug	11.9	16.0	15.8	14.7	13.4
Slug + SS	17.6	24.3	24.3	23.5	22.1
Slug + WA	13.0	17.6	17.7	16.5	15.0
Stencil	9.70	8.26	5.12	5.25	4.38
Tiling	9.20	6.26	7.65	8.68	9.78
Combined shading	9.40	12.2	11.6	10.9	8.31

Table 8.2: Relative rendering performance using the Canterbury font.

Method	192px (92 glyphs)	96px (379 glyphs)	48px (1508 glyphs)	24px (6243 glyphs)	12px (25046 glyphs)
Pre-rasterized glyphs	1	1	1	1	1
SDF	1.46	1.49	1.25	1.13	1.02
MSDF	1.49	1.44	1.30	1.58	2.08
Slug	16.0	20.5	21.2	20.3	17.4
Slug + SS	26.4	34.7	36.6	36.3	32.1
Slug + WA	17.4	22.5	23.3	22.4	19.1
Stencil	10.5	10.8	10.3	12.7	9.21
Tiling	11.2	18.0	22.1	25.4	19.7
Combined shading	16.5	21.3	21.7	21.3	15.1

Table 8.3: Relative rendering performance using the Elzevier Caps font.

Method	192px (92 glyphs)	96px (379 glyphs)	48px (1508 glyphs)	24px (6243 glyphs)	12px (25046 glyphs)
Pre-rasterized glyphs	1	1	1	1	1
SDF	1.25	1.20	1.15	1.09	1.09
MSDF	1.32	1.17	1.20	1.35	2.58
Slug	72.0	86.9	95.6	107	120
Slug + SS	152	186	213	240	264
Slug + WA	77.3	93.4	106	121	132
Stencil	91.1	115	131	154	162
Tiling	80.9	114	137	140	132
Combined shading	73.5	93.2	107	116	121

8.1.1 SDF/MSDF

SDF is almost as fast as a pre-rasterized glyph cache, see Figure 8.1, 8.2 and 8.3, row one and two. This is not very surprising as the only difference is a few small computations for anti-aliasing in the pixel shader. This does not differ much between the fonts. The slight differences are most probably due to the different amount of glyphs rendered, and the glyph size relative the em square. For MSDFs, we can see that for the worst case with all three

fonts, the rendering performance is about twice as slow as SDF. When there are less than 2000 glyphs on screen there is basically no difference in performance.

8.1.2 Slug

As expected, the slug algorithm requires a lot more computations and therefore makes for longer rendering times. For the Open Sans font, see Table 8.1, row four, it is about 12-16 times slower than pre-rasterized glyphs. With super sampling anti-aliasing, it is about 17-24 times slower. Using a weighted average is cheaper but is still around 13-18 times as slow. For both the Canterbury and Elzevier fonts, we can see that the Slug algorithm slows down considerably. Particularly rendering a full window of text with 12px Elzevier font, takes 5.14 milliseconds, see Table 7.3. This is 120 times slower than keeping a glyph cache, see table 8.3, and even worse with super sampling. This is the largest problem with the Slug algorithm. Performance is very dependant on glyph complexity. This makes the method unfit for rendering complex fonts or vector art in real time graphics applications. My implementation of the algorithm is probably not as fast as in the slug-library. Lengyel reported in a presentation [12], rendering a 4k display filled with text in 0.7ms. We also did not implement all optimizations Lengyel used in his paper. These were, splitting the bands in two and ray casting in both positive and negative x/y-direction and using more complex polygons to minimize empty area in the glyph bounding box. These improve performance for large fonts, see Lengyel[4], but hurts performance for smaller fonts, which is why these should be used in those circumstances.

8.1.3 Combination of Slug and SDF

The two different techniques are very different in both how they work and what kind of result they deliver. In general it is clear that not all fonts can benefit from a combined method like this. First we look at the Open sans font, see table 8.1. The fastest method was the utilizing the stencil buffer (row seven), as described in section 5.2. It is around 4-10 times slower then using glyph cache, depending on the size and amount of glyphs. The tile-based version (row eight) performs worse, when there is a lot of small text. This is not very surprising as we put a limit to tile size. When the minimum tile size is two pixels, a 12px font will use only a few tiles per glyph. If there is a corner in all the tiles, the whole glyph is rendered with Slug.

We expected the combined shader approach to be similar in performance to the stencil approach but it is about 90% slower than the stencil version, Table 8.1, row seven and nine, column five. This is most likely due to the extra branching in the pixel shader as well as slighly larger areas rendered with Slug.

Looking at the Canterbury font, Table 7.2, it is a similar result. The difference is that the tile-based method is slower than just using Slug, at least in the worst case (compare row six and eight). This font has so many corners that the whole glyph is simply rendered with Slug. Since it requires more primitives than Slug it makes sense that it is slightly slower.

For the last font, Elzevier, in table 7.3, both stencil and tiling perform worse than just using Slug. These glyphs have so many corners that trying to separate them in this manor is

very ineffective, at least for small font sizes. For the stencil approach the algorithm usually result in a large polygon covering the entire glyph, as an effect of the merging. This creates many small triangles when this polygon is tessellated. This is not optimal for the pixel shader. Most Graphics hardware use a 2×2 grouping for the pixel shader. The four pixels are processed simultaneously using SIMD. This means that if only one pixel in the group belong to a triangle, the rest is simply discarded. Also using more primitives per glyphs slows down the vertex shader. Therefore it is important to keep the triangle count to a minimum. From Table 7.3 row six to nine we can see that the combined shader is the only approach that provide a significant performance increase. This is probably because it more flexible in the separation of the different areas. Also this approach does not put any extra load on the vertex shader, as each glyph is drawn onto a single quad.

8.2 Image Quality

8.2.1 SDF/MSDF

For the Open Sans font SDF achieves a high quality render, see table 7.4. The error is higher for the largest and smallest size. This is about what we expected since for large fonts, we loose a lot of detail around corners. For small fonts the loss of quality is due to the problem with GPU texture minification. Sampling a texture when the a pixel covers multiple texels is an old problem for Graphics. This is usually resolved by using mipmaps. As can be seen from table 7.4, a lower resolution SDF makes for better results at small sizes. Using different resolution SDFs as mipmap levels could make the method slightly more scalable at the cost of memory. It is evident by this test that MSDFs fare much better at larger sizes. This is consistent with the expectation. The corners are preserved much better. For the Canterbury font the error is larger for SDF. The font has many thin features and corners that are not accurately reproduced. This is evident by looking at the example renders in figure 3.5. Even worse still is the Elzevier font, see table 7.6 which requires a lot higher SDF sample rate to produce good quality. For these fonts, we got quite severe artefacts using the msdfgen library. It might be that these fonts are not fully compatible with the library. Due to this, we don't want to draw conclusions about the quality performance of MSDFs based on my tests alone. The method has some issues where to thin features can produce holes in the glyph shape. Neither method is particularly good when fonts are to thin. We found that if a glyph is thinner than two texels, we will most of the time not be able to compute an accurate distance value from the samples.

8.2.2 Slug

The Slug algorithm can generally reproduce the shape of any glyph. The main downside is the anti-aliasing. Just using an average of two rays gives a slightly jagged appearance, see Figure 7.1. Using super sampling or weighting the averages both improve the quality. This is the case with all the fonts, see Table 7.4, 7.5, 7.6. I did notice however that weighting the averages can sometimes result in missed pixels. This is due to the weights being either $(0, 1)$ or $(1, 0)$ as a result of the algorithm. This is visible in Figure 7.1 on the top of the "b" in the bottom text. Since this artefact usually only affect single pixels, this is mostly a problem at very small

font sizes. For the Elzevier font, the Slug algorithm produced the best quality. If we use a high sampling rate on the SDF we might be able to get similar results but at that point it just becomes unpractical. We would need to store a 256×256 SDF to draw i.e. a 96px glyph.

8.2.3 Combination of Slug and SDF

As expected the RMSE the three versions are similar. For the Open Sans and Canterbury fonts, the difference is very small. For Open Sans the quality is better than both slug and SDF separately, for some sizes. This was a bit unexpected but easily explained. We can see in table 7.4 that the 64×64 SDF produces the highest quality at sizes 24 and 48. This is exactly the case where SDFs work very well, a simple font and a font size that is not to far from the SDF size. The distance values approximate coverages very well, better than all versions of Slug. At the larger sizes the errors are mostly caused by the rounding of the corners. As we eliminate the errors using Slug for the corners we get great quality for all glyphs. The same effect is present in the Canterbury font, table 7.5. When rendering very detailed font like Elzevier the the corners are so dense that the whole glyph is rendered with the slug method. The tile-based, and combined shader solution both allow for a very fine separation of the two shaders at large font sizes. The resolution of the SDF has a larger impact for these methods. This is the biggest problem with this combination. An SDF still struggles with thin features. For a complicated font, we get many of the artefacts that come with an low resolution SDF, like in Figure 3.5, bottom row, and the performance cost of Slug.

8.2.4 Caveats

There are some flaws to this way of testing quality. The main problem is that the RMSE does not necessarily measure how we perceive the fonts. Some artefacts are more prominent than others, even if they do not affect the RMSE as much. It is hard to design tests that measure this. Also it is not very exact when comparing different sizes to each other. This needs to be taken into consideration when reviewing these results.

8.3 Rendering at Small Font Sizes

We have tested to render sizes as small as 12px. Rendering small text is a fundamental problem in text rendering. Even if we use a perfect algorithm for the rasterization of the glyphs, the text can still appear blurry or difficult to read. Usually font rasterizers use different techniques to improve the result. The collective term for this is hinting. Certain features of the fonts may be scaled or modified to improve sampling at low resolution. Also the glyphs are aligned with the pixel grid so that features are less likely to be missed. It would of course be possible to use slug or SDF with hinted outlines but the font would then have an incorrect design when scaled up to higher resolution. An example of hinting using Freetype [14], can be seen in Figure 8.1.

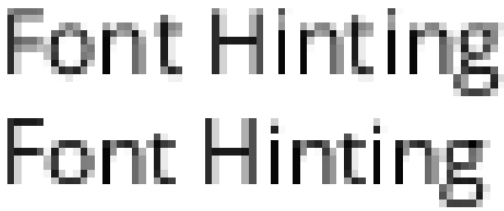

Figure 8.1: An example of hinted text from using Freetype. (Top) Unhinted text, (Bottom) Hinted text.

Some font rasterizers also utilize sub-pixel anti-aliasing. By using the sub-pixels in an LCD display, fonts can be rendered at higher resolution along one axis. This can make fonts more smooth but could also produce color artefacts. This technique requires knowledge of the sub-pixel layout, which can vary between different screens. Game applications may run in many different platforms and screens, like TVs, PC monitors or mobile displays which makes sub-pixel rendering unpractical.

To achieve the best results at small sizes is impossible without making some alterations to the outlines and layout. For this case it makes sense to use a pre-rasterized glyph cache since small glyphs do not require a lot of memory and it is so much faster than the other methods.

8.4 Rendering in World Space

This thesis mainly covers texts in a 2D-plane parallel to the screen. This is the most common for text rendering applications. Most games have some sort of user interface where text is rendered in this manner but it is not uncommon to render text in a 3D-world. In that scenario text can be view at an angle, with different pixel density at different areas of the glyphs. Using a pre-rendered glyph cache can result in blurry text and insufficient anti-aliasing at some angles. Both SDFs and Slug can use screen space derivatives for anti-aliasing which makes them more resolution independent. An example of text rendered at an angle is shown in Figure 8.2.

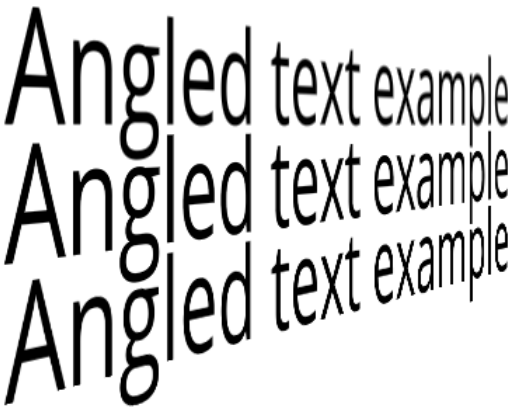

Figure 8.2: Example rendering with a text in viewed from an angle. (Top) **SDF**, (Middle) **Slug + WA**, (Bottom) **pre-rasterized glyphs**

Chapter 9
Conclusion

9.1 Summary

In this thesis, we have implemented and analyzed several methods for rendering fonts. We have shown that fonts rendered using signed distance fields will result in low quality for sharp corners or thin features. Through our tests it is also clear that the Slug algorithm is more flexible, and preserves the font better when scaled. This method of rendering fonts also proved to require a lot more shader computations, and as a result is many times slower than SDF. We have demonstrated three different solutions for combining these methods to improve upon some of the drawbacks of each. In most cases the best method was to use a separate set of primitives for the corners and utilize the stencil buffer for separation of the glyph. All three solutions managed to provide high quality fonts with faster rendering times than Slug and better scaling than SDF for the Open Sans font. However when font complexity increase, the separation of the glyphs makes these methods slower than the Slug algorithm. In this case there no benefit to use a combination.

The corner rounding problem, as we mentioned earlier, can also be countered by using more SDFs. The rendering time for this method is much faster then our combined approach. If the corners are not too close together (more than 2 texels), this method provides better quality over SDF, and due to the better render time, is a better solution than ours. If corners are very close together or the angle of a corner is very small, the limited resolution of the SDFs can be insufficient to recreate the corner, while Slug will work for these cases as well. This brings us to believe that this method might be very good for a serif font, something we unfortunately did not have time to test.

We have successfully managed to analyze some methods of font rendering. We have also succeeded in creating a new way to use them in combination for rendering high quality text. Our combined method is an interesting concept and managed to reduce some of the prob-

lems we encounter with SDFs and Slug. In its current state however, it will not replace the current state of the art methods of rendering fonts.

9.2 Future Work

As this thesis shows, it is possible to combine SDF and Slug to render high quality text. This is a working solution for improving the image quality of SDF font rendering or for optimizing the Slug algorithm. However it still suffers from the inability to render thin glyph features. We opted for using Slug for the corner parts of the glyphs, as these were the most problematic areas for SDF. The method can be made more general, and thin stems or small details that do not have corners can also be rendered with the Slug shader. The challenge lies in finding these areas, and separating them. Using a second SDF, as in our combined shading approach, removes the need for extra primitives, and allows for more complex shapes. Therefore this method should be the most suitable for such a generalization.

Bibliography

[1] Farin, G. *Curves and Surfaces for CAGD a Practical Guide*, 5th ed, Morgan Kaufmann, 2001, Accessed on: Mar. 30, 2020 [Online]. Available at: https://www-dawsonera-com.ludwig.lub.lu.se/abstract/9780080503547

[2] *TrueType Reference Manual*. Apple Inc. Accessed on: Mar. 30 2020. [Online]. Available: https://developer.apple.com/fonts/TrueType-Reference-Manual/RM01/Chap1.html

[3] C. Green, "Improved Alpha-Tested Magnfication for Vector Textures and Special Effects", *ACM SIGGRAPH 2007 courses*, pp. 9-19, ACM, New York, USA, 2007. Accessed on: Feb. 19 2020. [Online]. Available: https://steamcdn-a.akamaihd.net/apps/valve/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf

[4] E. Lengyel, "GPU-Centered Font Rendering Directly from Glyph Outlines", *Journal of Computer Graphics Techniques(JCGT)*, vol. 6, no. 2, pp. 31-47, 2017. Accessed on: Feb. 19 2020. [Online]. Available: http://jcgt.org/published/0006/02/02/

[5] V. Chlumský, "Shape Decomposition for Multi-channel Distance Fields" M.S. thesis, Czech Technical University in Prague, Faculty of Information Technology, Prague, 2015. [Online]. Available: https://dspace.cvut.cz/bitstream/handle/10467/62770/F8-DP-2015-Chlumsky-Viktor-thesis.pdf.

[6] V. Chlumský, "Multi-channel signed distance field generator", Accessed on: Oct. 10 2019. [Online]. Available: https://github.com/Chlumsky/msdfgen

[7] "Slug Dynamic GPU Font Rendering and Advanced Text Layout", Accessed on: Jan. 31, 2020. [Online]. Available: https://sluglibrary.com/

[8] C. Loop, J. Blinn, "Resolution Independent Curve Rendering using Programmable Graphics Hardware", *ACM Transactions in Graphics*, vol. 24, pp. 1000-1009, ACM, July 2005. Accessed on: Feb. 19, 2020. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/p1000-loop.pdf

[9] W. Dobbie, "GPU text rendering with vector textures" Jan 2, 2016. Accessed on: Feb. 19, 2020. [Online]. Available: https://wdobbie.com/post/gpu-text-rendering-with-vector-textures/

[10] B. Esfahbod, "GLyphy is a signed-distance-field (SDF) text renderer using OpenGL ES2 shading language", Accessed on: Oct. 14, 2019. [Online]. Available: https://github.com/behdad/glyphy

[11] Khronos Group, "Query Object", Accessed on: Jan. 28, 2020. [Online]. Available: https://www.khronos.org/opengl/wiki/Query_Object

[12] E. Lengyel, "GPU-Centered Font Rendering Directly from Glyph Outlines", Presentation slides, I3D, Montréal, 2018. Accessed on: Feb. 19, 2020. [Online]. Available: http://terathon.com/i3d2018_lengyel.pdf

[13] S. Kumar, "Maximum size rectangle binary sub-matrix with all 1s" Accessed on: Feb. 6 2020. [Online] Available: https://www.geeksforgeeks.org/maximum-size-rectangle-binary-sub-matrix-1s/

[14] "The Freetype Project" Accessed on: Feb. 3, 2020. [Online]. Available: https://www.freetype.org/

[15] R. A. Jarvis, "On the identification of the convex hull of a finite set of points in the plane", *Information Processing Letters*, vol. 2, no. 1, pp. 18 - 21, March 1973. [Online]. Available doi: 10.1016/0020-0190(73)90020-3

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2020-03-12

EXAMENSARBETE Rendering Resolution Independent Fonts in Games and 3D Applications
STUDENT Olle Alvin
HANDLEDARE Michael Doggett (LTH), Göran Syberg Falguera (EA DICE)
EXAMINATOR Flavius Gruian (LTH)

Sharpening The Corners of Video Game Fonts

POPULAR SCIENTIFIC SUMMARY **Olle Alvin**

Player names, ammunition count and high scores are all common pieces of information displayed as text to a player in a video game. To draw text on a screen is something most of us take for granted. For a high performance application such as a AAA game, it becomes a difficult task. In my thesis I explore a few solutions for this and attempt to combine them to create something better.

A pretty common solution for rendering text, that works for any display is to use a Signed Distance Field (SDF). This method utilizes distance information to draw vector art using graphics hardware. It works very well in that regard but it comes with a problem. When trying to draw large characters or draw text on a high resolution screen, the font looses some of its defining traits. The sharp corners you would see on an "A" or a an "N" will be rounded of causing the text to look soft.

In my thesis I explain another method of rendering fonts (The Slug Algorithm), which can produce images much more true to the wanted result. It uses the mathematical definition of the character and draws it exactly as the designer intended it. However, this is at least 10 times slower than drawing from signed distance fields. This is not an option for a high performance game application. The extra quality over SDF is not always worth the performance cost. Then again, looking at the SDF rendering in the above figure, not all parts of the "A" look distorted. It is only the sharp corners that do not look the way we want them too. So we can use the more expensive method only in the small areas where the SDF fails. Thereby we can keep performance cost to a minimum but still get a good rendering.

This is good in theory, but in practice separating the corner areas in a good way is challenging. The separation itself comes with a performance cost. Therefore the actual benefit is not as high as one could imagine. In the best case I found that it we gain about 70% in performance over the expensive Slug algorithm but it is still 3 times slower than using the SDF only.

The worst case is when we deal with more decorative fonts with very intricate designs. The letters might be decorated with tiny flowers, or made to look like hand drawn scribbles. To draw fonts like this is extremely slow with this method. In conclusion, the combination of these methods can be used to render high quality fonts fairly fast but it is not efficient for very intricate fonts.