7-1-2013

Front Rendering on the GPU

Jeffrey Bowles

Jeffrey Bowles

Candidate

Computer Science

Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Joe Kniss , Chairperson

George Luger

Ed Angel

Font Rendering on the GPU

By

Jeffrey R. Bowles
B.S., Computer Science, University of New Mexico, 1997

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Masters of Science
Computer Science

The University of New Mexico
Albuquerque, New Mexico

May 2013

DEDICATION

This work is dedicated to my wife, Amanda Quintana-Bowles, a talented artist with a passion for fonts and typography.

ACKNOWLEDGMENTS

I would like to thank Professor Ed Angel for fostering and manifesting a passion for computer graphics, starting with his introduction to C++ in 1993. I highly appreciate Professor George Luger for his encouragement and allowing me the flexibility to transform my studies in his class on an overview of languages into a graphics course. Lastly, I would like to express my thanks to my advisor, Professor Joe Kniss, for his tireless work and patience in introducing many advanced techniques in computer graphics and mathematics, and showing me that the impossible is achievable through daring, imagination and belief in oneself.

Font Rendering Using the GPU

By

Jeffrey R. Bowles
B.S., Computer Science, University of New Mexico, 1997
M.S., Computer Science, University of New Mexico, 2013

ABSTRACT

The glyphs described in a font are usually rendered using software that runs on the CPU. This can be done very quickly and is a well understood process. These techniques work well when the glyph is rendered such that each point on the rendered glyph maps one-to-one to pixels on the screen. When this one-to-one mapping is removed, visual quality can degrade dramatically.

This thesis describes a method to avoid this degradation by moving the rendering of glyphs from the CPU to the Graphics Processing Unit (GPU). This method extracts glyph metrics from Truetype font files using the FreeType library, processes all glyph data in the font file and places it into a form that can be used by GPU shader programs to draw the glyphs such that the visual quality does not degrade regardless of the applied transformations.

Contents

List of Figures

1 Introduction

During various graphics projects, I have frequently identified the need to add text to three-dimensional scenes. For example, I have had requests to integrate an interactive console. Alternately, I have had the occasion to label points in a graph that represent contributors to a source code repository. One can imagine all sorts of scenarios where high quality, realtime rendering of text is needed in a three-dimension scene, for instance, in computer games that are set within a city with signs, billboards and cloth banners. To ensure that users stay immersed in the environment, this text must look good from any camera position. Life is not pixellated.

When I started this project, I believed this to be a solved problem and spent time searching for an open source project that could be incorporated into my projects. I did not find any freely available software that met all of my criteria:

- **OpenGL**: the "Open Graphics Library" is a platform-independent software interface to graphics hardware [7].
- **Open 3.2 Core Profile**: a user selectable OpenGL state that does not allow deprecated functionality [7] and is the only available method to use OpenGL 3.2 specific features under OS X.
- **TrueType fonts**: an openly defined outline font standard developed by Apple Computer [1]. The open source project FreeType [4] can parse font files in this format and is available on all targeted platforms (OS X, Windows and Linux).
- High quality, non-pixellated rendering regardless of transforms applied to the text or the geometry onto which the text is mapped.

The first three criteria were chosen to maximize portability between platforms and to allow the use of the most advanced features of OpenGL on each of the targeted platforms. The fourth criterion was chosen to maximize visual quality.

Various freely available software packages were evaluated and it was found that none met the criteria, as described in Section 4. The time spent implementing these methods was well spent and served as an introduction to font rendering. For example, I learned that a font, which is alternately called a typeface, is composed of a set of glyphs and the data that describes how those glyphs interact. A glyph is a set of contours that describe a character and a contour is a set of curves and lines that form a closed loop. The interactions between glyphs is called kerning, which describes the amount of space to add or subtract between two glyphs that are placed next to each other.

After the search for existing software failed to turn up acceptable libraries, this thesis was started in an effort to show that a GPU-based implementation that renders high quality text regardless of how the glyphs are transformed or the geometry onto which they are mapped is possible.

2 Overview

In Section 3, there is a brief overview of existing software packages used to render text in OpenGL and why those packages do not meet the criteria listed in the introduction. Section 4 discusses work performed to utilize existing CPU-based methods for rendering text and using the results with OpenGL. Section 5 discusses some background that is needed to understand the method used to

render glyphs on the GPU. Section 6 discusses the methods used to preprocess glyph data on the CPU, how that data is formatted for use on the GPU and the shaders used to render the glyphs. The final sections show the rendered glyphs, discuss future work and the conclusion.

3 Existing Software Packages

Many packages exist that render glyphs in a three dimensional environment, but none met the requirements. Only Open Source packages were evaluated. One of the requirements is that a font rendering system needed to work in an OpenGL 3.2 core context. It was found that most packages required functionality that was deprecated in OpenGL 3.2 and only worked when used with an OpenGL compatibility context.

3.1 GLUT

GLUT has font rendering functionality, but the rendering method is only available for OpenGL 2.1 compatible contexts due to its use of the deprecated function call glRasterPos. It also does not render text onto arbitrary geometry, nor does it use Truetype fonts.

3.2 FTGL

FTGL is a very well written font handling library that uses Truetype fonts, but it is dependent on the deprecated gluTess family of function calls, which are not available when using the OpenGL 3.2 core profile. FTGL can be used to place text onto an arbitrary surface, but requires generating a texture map of the text on the CPU. This works well when the text is billboarded, e.g., the geometry is flat and perpendicular to the view, and when the exact size of the text is known in screen space.

3.3 VSOFont

VSOFont uses its own font format and requires the user of the library to create their own fonts. It also relies on OpenGL 2.1 fixed-functionality that was deprecated in OpenGL 3.2.

4 Rendering Text Using the CPU

The research into rendering fonts in a three dimensional scene starting during a project that needed to label a large number of point sprites. A point sprite is a graphics primitive that renders a single point as a square that always faces the camera. In my previous projects, FTGL was used for this purpose, and it worked well. For the reasons listed in the introduction, this project required OpenGL 3.2 specific functionality. This made FTGL an unsuitable solution for this project.

Because the project used a small subset of FTGL's functionality, it was determined that FreeType could be used for rendering text. This was expected to be an easy task. The expectation was that FreeType would render a string of text into a bitmap that would be applied as a texture to a quad.

FreeType is not as simple to use as one might hope. It does not have the capability of rendering a string of text. It can only render a glyph at a time, which can be copied into a texture map. While there is some trickiness with getting FreeType to render glyphs, it is more tedious than difficult to get the process correct. The process has two passes. The first pass determines the size of the texture

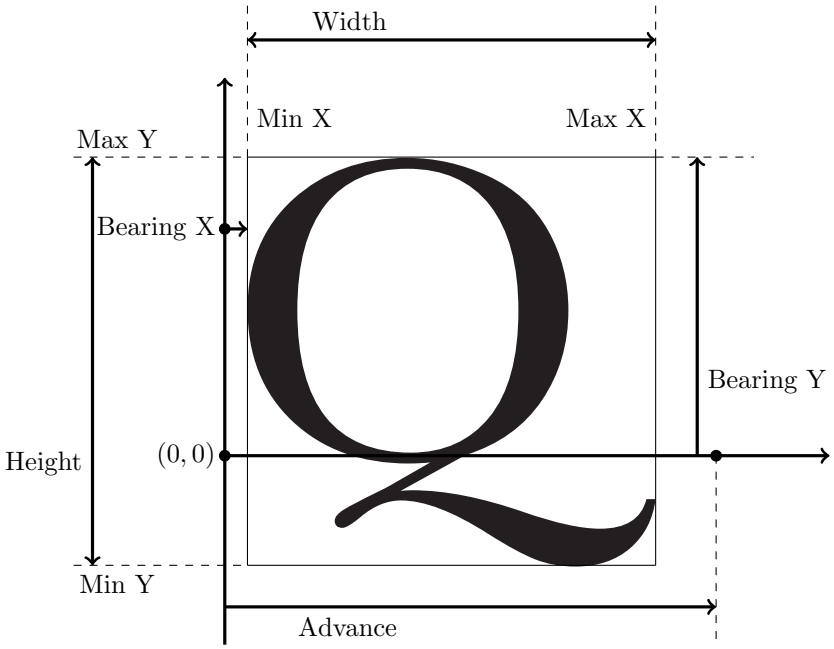


Figure 1: Glyph metrics

map based on the string to be rendered and the second pass renders the string into a bitmap. That bitmap is then uploaded to the GPU and applied as a texture to the surface that will display the label.

Before a FreeType based text layout and rendering algorithm can be presented, it is important to understand glyph metrics.

4.1 Glyph Metrics

To understand how text layout is performed, it is important to understand the various metrics used when describing a glyph. The standard unit of measurement is the *point*. The size of a glyph is defined by its point size. A point is 1/72 of an inch, thus making the glyphs in a 72 point font approximately 1 inch tall.

FreeType uses 1/64th of pixel as a measurement in some cases. It is convenient to divide by 64 using bit shift operators, which is very fast as opposed to using floating point arithmetic to divide by 72. The FreeType documentation does not specify which units are being used, leaving it to the user of the library to empirically determine the units being used in any particular function call or data structure member. Table 1 and figure 1 show how each of the metrics are used as part of the description of a glyph's positioning.

Table 1: Glyph Metrics

Metric	Description	Units
Min Y	Minimum Y position of a point in the glyph	pixels
Max Y	Maximum Y position of a point in the glyph	pixels
Min X	Minimum X position of a point in the glyph	pixels
Max X	Maximum X position of a point in the glyph	pixels
Bearing X	Distance from origin to min X	pixels
Bearing Y	Distance from origin to max Y	pixels
Advance	The amount to advance to the next glyph	1/64th pixel
Kerning	The distance to subtract between two glyphs	1/64th pixel

Definitions of important glyph metrics. These are shown visually in figure 1.

4.2 Building a Texture Map with FreeType

The process of creating an OpenGL-compatible texture map that contains a horizontally laid out string is text is presented using pseudo-code.

```
// Initialize free type library
initialize_freetype_library();
load_font_face();
set_point_size_and_dpi();

...

max_height = 0;

// First pass - determine width and height of destination bitmap
for(glyph : text_string) {
   height = glyph.get_height();
   width = glyph.get_width();
   min_y_position = glyph.get_minimum_y_position();
   advance = glyph.get_advance();

   if(first_iteration) {
      kerning = 0;
   } else {
      kerning = kerning between this glyph and the previous glyph;
   }

   glyph_max_y = glyph_min_y + height;
   glyph_max_x = prev_x_pos + advance - kerning;

   width += glyph_max_x;
   if(height > max_height) {
```

```
      max_height = height;
   }
}

// Allocation destination bitmap
destination_bitmap = new Bitmap(glyph_max_x, max_height);

// Second pass - render text
for(glyph : text_string) {
   // Using the previously calculated positions when determining
   // the width and height of the bitmap
   set_x_position();
   set_y_position();
   render_glyph();
}

...
// Call glTexImage2D or glTexSubImage2D
upload_texture_to_gpu();

// Display text in scene
display_quad_textured_with_bitmap();
```

This algorithm produces good results when it is known exactly how the textured quads will map into screen space and the one-to-one mapping of texels to pixels is preserved. It also works well when there is not a large amount of text in the scene. However, this violates visual quality and low memory usage constraints. In some instances, all of the texture maps for each of the labels required 3GB of memory. Figure 2 shows how visual quality degrades when the one-to-one mapping of texels to screen pixels begins to break down as the transformation of the quads puts the images farther away from the viewer. In this case, the texture map was very detailed and looked good when viewed up close and became foggy as the images moved away from the viewer.

Because it was not known how the textured quad would map into screen space, the texture maps need to be very high resolution in order to combat aliasing problems. There were times where up to 3GB of memory would be used to hold high quality textures for the hundreds of labels that were being placed in some scenes. The problem of high memory usage was the first problem that needed to be addressed. While the aliasing and fogginess are undesirable, the text is still readable, as seen in figure 2.

A large amount of work was performed in an effort to make the rendering of glyphs on the CPU work better and provide better results. For example, in an effort to reduce memory usage, a *glyph texture map* was created that contains the set of all of the glyphs in a font. This glyph texture map is then uploaded to the GPU. To use this map, a single quad is drawn for each glyph in a string. The CPU lays out the quads along the y-axis with the correct spacing between the quads to render nicely kerned text. Each quad is given texture coordinates that index into the glyph texture map so that the correct glyph is displayed on each quad. Further work was done to move the layout onto the GPU by passing the string of characters to be rendered in a uniform variable to a geometry shader. The geometry shader then created the set of quads to be rendered in the scene. This moved

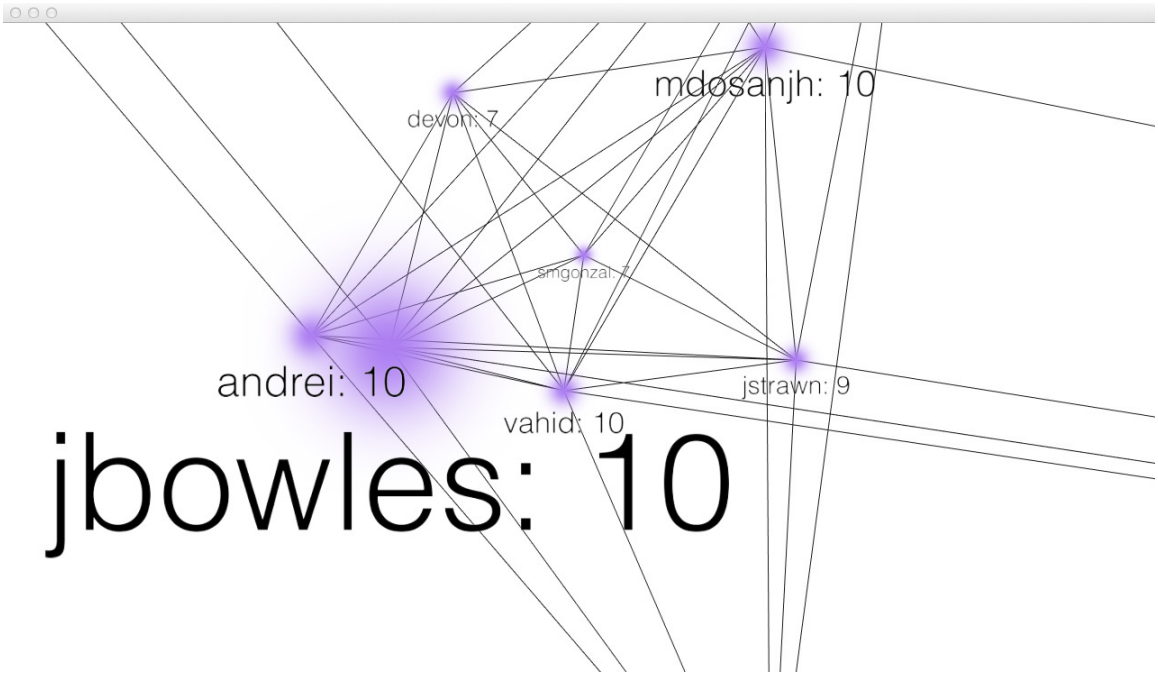


Figure 2: Font rendering using texture maps for each string

more of the computation onto the GPU, but did not solve the aliasing and fogginess problems. It also did not allow for an easy method of placing text onto arbitrary geometry with relative ease and with high quality results.

5 Background

5.1 Bézier Curves

Bézier curves may be more accurately called curves that were used and made popular by Pierre Bézier, who used these curves in the design of automobile bodies. It was Paul de Casteljau that developed *de Casteljau's* algorithm that allows these curves to be evaluated in a numerically stable manner.

Bernstein Polynomial:

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

A Bézier curve of order n is the sum of $(0, n)$ Bernstein polynomials where each term of the sum scales the ith control point. Control points are vectors and are written as $\mathbf{p_i}$

$$C(t) = \sum_{i=0}^{n} B_i^n(t)\mathbf{p_i}$$

t ranges from 0 to 1. Notice that $C(0) = p_0$ and $C(1) = p_n$.

A second order, or quadratic, Bézier curve function has the form:

$$C(t) = \binom{2}{0} t^0 (1-t)^2 \mathbf{p_0} + \binom{2}{1} t^1 (1-t) \mathbf{p_1} + \binom{2}{2} t^2 (1-t)^0 \mathbf{p_2}$$

$$C(t) = (1-t)^2 \mathbf{p_0} + 2t(1-t)\mathbf{p_1} + t^2 \mathbf{p_2}$$

This can be rewritten in terms of a matrix-vector multiplication:

$$C(t) = \mathbf{P} \begin{bmatrix} (1-t)^2 \\ 2t(1-t) \\ t^2 \end{bmatrix}$$

Where $\mathbf{P}$ is a matrix whose column vectors are the control points:

$$P = \begin{bmatrix} p_{0_x} & p_{1_x} & p_{2_x} \\ p_{0_y} & p_{1_y} & p_{2_y} \\ p_{0_w} & p_{1_w} & p_{2_w} \end{bmatrix}$$

The vector portion of this equation can also be written as a matrix-vector multiplication operation where the vector is the power basis Rewrite the left hand term in the dot product as a matrix multiplied against the power basis, $\mathbf{b} = [1, t, t^2]$. This is desirable because it creates an invertible operation and allows for a change of basis.

Let $M(t)$ be the left side of the dot product:

$$M(t) = \left[(1-t)^2, 2t(1-t), t^2\right]$$

Make each component of the vector the same by introducing terms that are multiplied by zero:

$$M(t) = \left[1 - 2t + t^2, 0 + 2t - 2t^2, 0 + 0t + t^2\right]$$

How do we figure out what the operation looks like that produces this vector? What does this matrix look like? Because we are searching for an operation that produces a 3x1 vector, we only have find three dot products that produce the elements of $M(t)$:

$$M(t)_0 = [1, -2, 1] \cdot [1, t, t^2] = 1 - 2t + t^2$$

$$M(t)_1 = [0, 2, -2] \cdot [1, t, t^2] = 0 + 2t - 2t^2$$

$$M(t)_2 = [0, 0, 1] \cdot [1, t, t^2] = 0 + 0t + t^2$$

Resulting in the coefficient matrix Q:

$$\mathbf{Q} = \begin{bmatrix} 1 & -2 & 1 \\ 0 & 2 & -2 \\ 0 & 0 & 1 \end{bmatrix}$$

The function, $C(t)$ can be written as:

$$C(t) = \mathbf{PQb}$$

Solve for $\mathbf{b}$ and determine if the point $C(t) = [x_t, y_t, w_t]$ is on the curve:

$$\mathbf{b} = \mathbf{Q}^{-1}\mathbf{P}^{-1}C(t)$$

The point $[x_t, y_t, w_t]$ is on the curve if $\mathbf{b}$ has the form $\left[1, t, t^2\right]$. The point is under the curve if ${x_t - y_t}^2 \geq 0$ and $y_t \in (0, 1)$. This is the key to filling the area under the curve in a fragment shader.

5.2 Winding Numbers

Winding numbers are used to distinguish between the inside and the outside of a glyph. The information about each glyph in a font can be reduced to:

- A glyph outline describes a set of contours.
- A contour is made up of a set of segments.
- A segments can be a line or a section of a curve
- A contour always starts and ends at the same point.

With this information, it is up to the programmer to determine if a point contributes to a glyph. Winding numbers are used to determine a point's contribution. If a point's winding number is non-zero, then that point contributes to the glyph. The method used to calculate winding numbers is described in the TrueType Reference Manual [1]. To briefly summarize, a ray is created that starts at the point to be tested and is cast off to infinity. In practice, rays point from left to right or top to bottom. In this implementation, rays are cast from left to right. The initial winding number is zero.

When a ray crosses a curve, it's winding number increases if the transition is on *on* transition, or the winding number decreases if it is an *off* transition. And on transition occurs when the ray crosses the contour from right to left. An off transition occurs when the ray crosses the contour from left to right.

We use the incenters of triangles as ray starting points.

5.3 Finding the Centers of the Triangles

The algorithm presented in this thesis needs to find a point on the inside of a triangle to cast a ray from left to right to determine whether or not a triangle is a contributing triangle or a non-contributing triangle. The *center* of the triangle is chosen as this point. There are many different types of centers and for this algorithm, and the incenter was arbitrarily chosen.

The *incenter* is the center of a circle that is tangent to the three line segments that form a triangle. This circle is called the incircle. The center of the incircle can be found by bisecting the angle at each corner of the triangle and casting rays towards the inside of the triangle. The point where they intersect is the center of the incircle.

To bisect two vectors, Take two vectors, $\mathbf{A_B}$ and $\mathbf{A_C}$, and then add them together. The resulting vector will bisect the two original vectors. When this process is applied to vectors formed by the sides of a triangle, the intersection point of those three vectors is the incenter of that triangle as shown in figure 3.

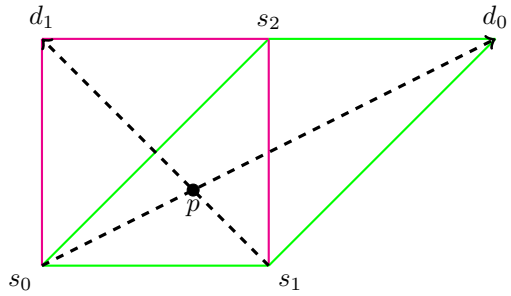


Figure 3: Finding the incenter of the triangle s_0, s_1, s_2 by bisecting the vectors $s_2 - s_0$ and $s_1 - s_0$ and calculating the intersection p.

The two rays d_0 and $d_1$intersect at point p if the following system equations can be solved with the constraint that $u, v \geq 0$.

$$\begin{aligned} p &= s_0 + d_0 u \\ p &= s_1 + d_1 v \end{aligned}$$

These equations can be broken up into their x and y components and set equal to each other, resulting in two equations and two unknowns:

$$\begin{aligned} s_{0_x} + d_{0_x} u &= s_{1_x} + d_{1_x} v \\ s_{0_y} + d_{0_y} u &= s_{1_y} + d_{1_y} v \end{aligned}$$

This system of equations can be rearranged such that u, v are on the left and the constants are on the right:

$$\begin{aligned} d_{0_x} u - d_{1_x} v &= s_{1_x} - s_{0_x} \\ d_{0_y} u - d_{1_y} v &= s_{1_y} - s_{0_y} \end{aligned}$$

Recall that Cramer's rule takes a system of equations in the form:

$$\begin{aligned} au + bv &= c \\ du + ev &= f \end{aligned}$$

and solves them with:

$$\begin{aligned} D &= ae - bd \\ u &= (cd - fb)/D \\ v &= (af - dc)/D \end{aligned}$$

Applying Cramer's rule to the two lines defined by (s_0, d_0) and (s_1, d_1) results in:

$$\begin{aligned} D &= d_{1_x} d_{0_y} - d_{0_x} d_{1_y} \\ \Delta x &= s_{1_x} - s_{0_x} \\ \Delta y &= s_{1_y} - s_{0_y} \\ u &= (d_{1_x} \Delta y - d_{1_y} \Delta x)/D \\ v &= (d_{0_x} \Delta y - d_{0_y} \Delta x)/D \end{aligned}$$

5.4 Determining if a Point Lies to the Left of a Line

One might naively decide to check to see if a ray intersects a Bézier curve to determine if the winding number of that point will change. However, all rays cast by this algorithm are horizontal, meaning that $\Delta y = 0$, so only the starting point is needed. Furthermore, since all of the Bézier curves will be contained within a triangle and the rays we care about will start from outside of the triangle, it is only necessary to discover if the starting point of the ray lies to the left of the line that creates the base of the triangle. Pseudo code is presented to describe this algorithm.

```
float leftOfLine(vec2 s0, vec2 p0, vec2 p1) {
   // Initialize return value to false
   float isLeft = 0; // 0 for false, 1 for true
```

```
    float deltaX = p1.x - p0.x;
    if(deltaX > 0 || deltaX < 0) {
       // In the non-vertical line case, the slope of the line is
       // needed. A divide operation can be eliminated if
       // the reciprocal of the slope is used.
       float rslope = deltaX / (p1.y - p0.y);
       if(((s0.y - p0.y) * rslope) + p0.x >= s0.x) {
          retval = 1;
       } else {
          retval = 0;
       }
    } else {
       // If this is a vertical line, the test is fairly simple. If
       // the point's x value is less than the line's x value,
       // it is to the left
       if(s0.x < p0.x) {
          retval = 1;
       } else {
          retval = 0;
       }
    }
    return isLeft;
}
```

5.5 Determining if a Point is Inside a Triangle

Using Barycentric coordinates, any point, p, in the plane defined by the three points, p_0, p_1 and p_2 can be written as:

$$p = p_0 + s(p_1 - p_0) + t(p_2 - p_0)$$

If $(s,t) \in (0,1)$, then p is located inside of the triangle. In section 6.14, where the fragment shader is described, the point p is known and (s,t) are unknown. (s,t) can be found by creating a system of two equations. First, simplify the Barycentric equation to:

$$v_2 = sv_0 + tv_1$$

Where the vector v_i are defined as:

$$\begin{aligned} v_0 &= p_1 - p_0 \\ v_1 &= p_2 - p_0 \\ v_2 &= p - p_0 \end{aligned}$$

Create two equations by dotting with v_0 and v_1:

$$\begin{aligned} v_1 \cdot v_2 &= s(v_0 \cdot v_1) + t(v_1 \cdot v_1) \\ v_0 \cdot v_2 &= s(v_0 \cdot v_0) + t(v_0 \cdot v_1) \end{aligned}$$

Solving for s and t:

$$s = \frac{(v_1 \cdot v_1)(v_0 \cdot v_2) - (v_0 \cdot v_1)(v_1 \cdot v_2)}{(v_0 \cdot v_0)(v_1 \cdot v_1) - (v_0 \cdot v_1)(v_0 \cdot v_1)}$$

$$t = \frac{(v_0 \cdot v_0)(v_1 \cdot v_2) - (v_0 \cdot v_1)(v_0 \cdot v_2)}{(v_0 \cdot v_0)(v_1 \cdot v_1) - (v_0 \cdot v_1)(v_0 \cdot v_1)}$$

6 Rendering Glyphs on the GPU

To solve the problems with rendering glyphs using FreeType on the CPU, the decision was made to move the rendering into a fragment shader that runs on the GPU. Thiss allow the one-to-one mapping of points on a glyph to screen space to be preserved, thus eliminating the sampling artifacts problem, and by using parameterized texture coordinates, the glyphs could be easily rendered onto arbitrary surfaces.

It was not clear how this could be accomplished. It is known that the data that describes a glyph is defined as a set of curves. Because of the work of Jim Blinn and Charles Loop [3], it is also known that it is possible to render filled quadratic Bézier curves on the GPU. Based on procedure described in their paper, it was also decided to use their method of constrained Delaunay triangulation to break the glyphs up into a set of triangles. These triangles fall into three classes: triangles whose internal points all have non-zero winding numbers, triangles whose internal points all have a winding number of zero, and triangles that contain curves. The Blinn-Loop method then sends these triangles through the graphics pipeline and uses a shader program to render the curves. This method introduces some difficulty when rendering text onto arbitrary geometry. It seems necessary to retesselate the surface so that the triangles that hold the glyphs will fit onto the surface properly. Also, if the text is being combined with another texture map, there may be some trickiness in handling texture coordinates. These problem are not insurmountable, but it seems that graphics hardware has advanced enough that the problems can be avoided and that more of the process can be placed on the GPU.

Instead, this thesis proposes to perform the triangulation and put the list of triangles into a texture map. The fragment shader searches the triangle list to discover if the fragment contributes to the glyph. This allows for texturing of arbitrary surfaces without retesselation and for easy combination with other textures and effects, such as bump mapping. There is no need to fit the triangles into an existing mesh.

In both this thesis and the Blinn-Loop method, there are two high level steps when using a GPU to render text:

- Pre-process the font data and place it into a form suitable for use on the GPU. In this thesis, this step takes place on the CPU.
- Use the data from the previous step to render glyphs using a fragment shader that runs on the GPU.

6.1 Overview of Glyph Pre-Processing

The goal of glyph pre-processing is to assist the GPU in determining which set of fragments make up a glyph.

- Extract font data that describes the segments that make up the contours in each glyph
- Classify the orientation of the contours as either clockwise or counter-clockwise

- Perform constrained Delaunay triangulation
- Compute change in winding numbers for each segment as a ray traverses this segment from left to right
- Find the incenters for each triangle
- Determine the winding number at the incenter of each triangle
- Mark triangles that do not contain a curve whose incenter is zero as non-contributing
- For triangles that contain a curve, determine if the curve is filled above or below.
  - In rare cases, some curves are filled both above and below. Mark this triangles as contributing
- Place glyph data into texture maps

6.2 Extracting Glyph Data Using FreeType

FreeType is used to parse TrueType font files and extract the data needed to render the glyph set. It was initially assumed that the required data would be provided by loading the font using FreeType library calls and then inspecting the data structures associated with the font face. After much frustrating and tedious work, the simpler method of inspecting the glyph outline decomposition process by registering callback functions was chosen. The segments that make up the contours are emitted by the FreeType library, which are then captured and stored in custom data structures for further processing.

The C++ code for starting the decomposition is:

```
// Set up FreeType callback functions for outline decomposition
FT_Outline_Funcs cb;
// Set the pointer to the move callback
cb.move_to  = &moveEmitted_cb;
// Set the pointer to the line callback
cb.line_to  = &lineEmitted_cb;
// Set the pointer to the cubic Bezier curve callback
cb.cubic_to = &cubicEmitted_cb;
// Set the pointer to the quadratic Bezier curve callback
cb.conic_to = &conicEmitted_cb;
cb.shift    = 0;
cb.delta    = 0;

// Start the callback loop to decompose the outline.
FT_Outline_Decompose(&outline, &cb, this);
```

There are four types of segments that are emitted by the FreeType outline decomposition process:

- **Move**: When a move is emitted, a new contour has started. The number of contours is increased and this point is marked as the starting point for the next segment

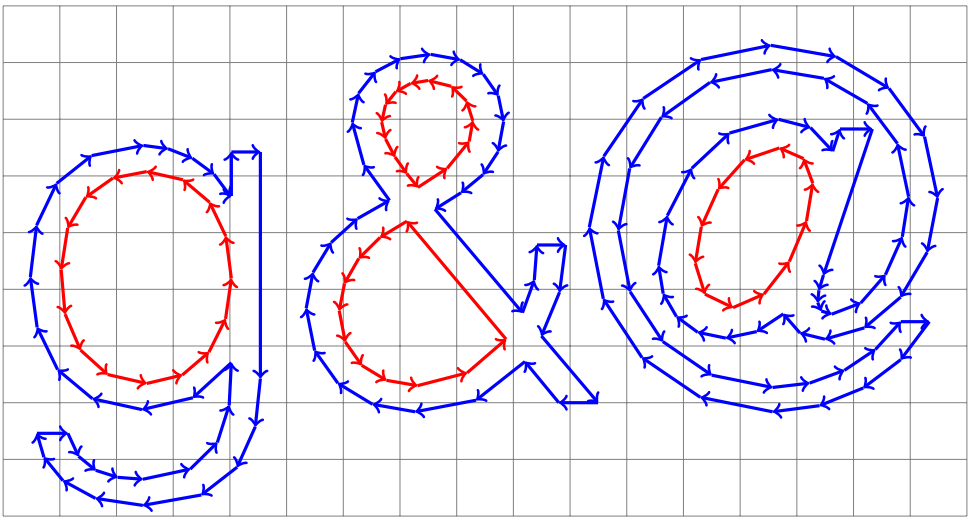


Figure 4: Determining contour orientation using the angle between segments over whole contours

- **Line**: Many glyphs contain straight segments that do not need to be described by a curve. Some glyphs are poorly defined and have contours that consist of a single line segment that starts and ends at the same point. These contours, lines and vertices are discarded.
- **Quadratic Bézier**: This is the interesting part of this thesis. The control points are stored so that the coefficient matrix, Q, can be created.
- **Cubic Bézier**: Cubic Bézier curves are not handled by this library. If font contains any cubic Bézier curves, an exception is thrown.

Lines that start and end at the same vertex are quite common. These contours are detected and eliminated to simplify processing.

6.3 Contour Orientation Classification

Knowing the orientation of a contour is critical component in determining the winding number of a point. By treating all of the points in a contour as forming a closed loop, the orientation can be determined. This is computed by iterating over each pair of joined segments in a contour and calculating the difference in angle between the previous two segments. Each of the differences is summed. If the sum of the differences is less than zero, the orientation is clockwise. Otherwise, the orientation is counter-clockwise.

6.4 Delaunay Triangulation

Glyphs are decomposed into a set of triangles that covers the entire area of the glyph. In this thesis, a constrained Delaunay triangulation is performed. The segments and curve containing triangles extracted from the font data are passed to the triangulation library as existing connections between

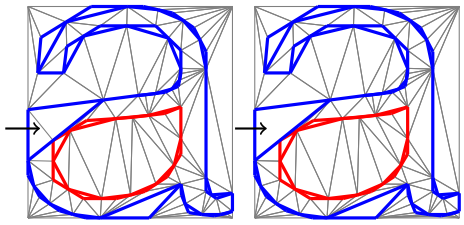


Figure 5: Constraining the Delaunay triangulation: on the left, no holes specified for the triangles that contain curves results in an unneccesary triangle. On the right, the desired result when holes are added to the triangles that contain curves

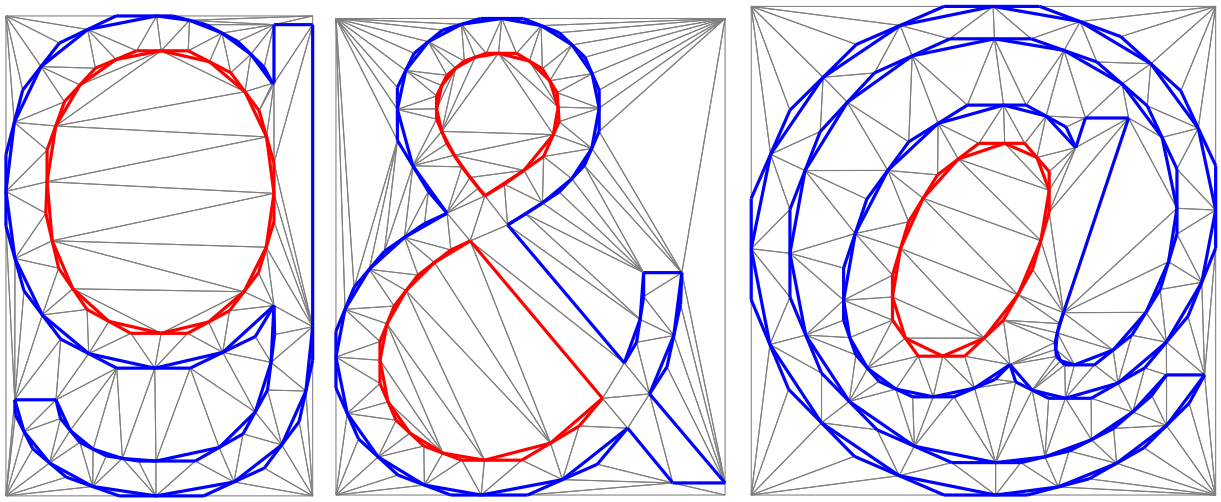


Figure 6: Delaunay triangulation applied to glyph outline data

vertices. A very important, but subtle step, is to pass in a list of holes that reside inside of the curve-containing triangles. If this step is not performed, the triangulation may subdivide curve-containing triangles, which may result in an improperly rendered glyph. See Figure 5 for an example of what can happen when a list of holes is not generated. The list of holes is created by finding the incenters of the curve-containing triangles.

An external program, *Triangle* [8], is used to generate the triangulation. This generates an easily parsed file. Triangle can be built as a library and linked to the executable. It was decided to call the program and not link to the library so as to not be bound by its licensing terms. The licensing terms are not onerous, but it is an undesired complication.

The Delaunay

Δy	**Orientation**	**Crossing**	**Δ Winding Number**
0	clockwise	no crossing or 2 opposite crossings	0
0	counter-clockwise	no crossing or 2 opposite crossings	0
negative	clockwise	left $\rightarrow$ right	-1
negative	counter-clockwise	left $\rightarrow$ right	-1
positive	clockwise	right $\rightarrow$ left	1
positive	counter-clockwise	right $\rightarrow$ left	1

Figure 7: Calculating change to winding number as a ray passes through a triangle that contains either a curve or a line.

6.5 Computing Change in Winding Number for each Triangle

To make later preprocessing stages faster, the results of a ray passing through a triangle that contains a segment of a contour is calculated and stored. Because rays always pass from left to right through contour segments, the change to the winding number for a point is always the same regardless of where the ray passes through the containing triangle. Figure 8 displays this visually.

The calculation is performed by iterating over each triangle and checking the change in the y-coordinate of line segment and the orientation of the contour in that segment. In the case of a curve, this line segment is formed by creating a line from the two outer control points. Figure 7 shows the possible contour orientations and line slopes and how that effects the change, or delta winding number. A slope of zero is not considered because there is no change to the winding number either because two opposing intersections will occur, no intersection at all, or the ray will run tangent to the line segment which also results in no change to the winding number.

Pseudo-code is provided to better understand how the change in winding number is calculated:

```
for(tri : triangle_set) {
  if(tri contains a curve or a line segment) {
    slope = slope of line that forms base of triangle;
    if(slope < 0) {
      if(tri->orientation() == CW) {
        tri.crossing = RIGHT_TO_LEFT;
      } else {
        tri.crossing = LEFT_TO_RIGHT;
      }
    } else if(slope > 0) {
      if(tri->orientation() == CW) {
        tri.crossing = LEFT_TO_RIGHT;
      } else {
        tri.crossing = RIGHT_TO_LEFT;
      }
    } else { // slope == 0
      // ray travels parallel to the base and either transitions on/off,
      // off/on or not at all
      // resulting in no change to the winding number
```

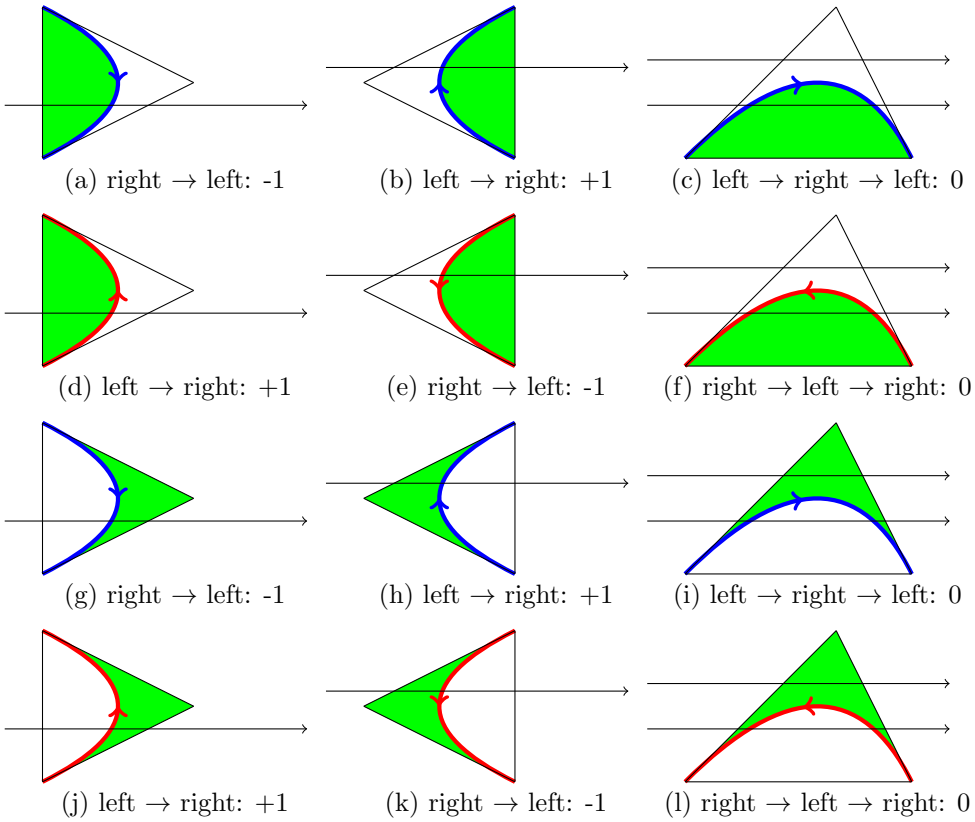


Figure 8: Marching rays from left to right through triangles that contain curves and resulting the change to the winding number. The top row (a,b,c) shows curves that have a clockwise orientation and the second row (d,e,f), shows curves with a counter-clockwise orientation. The bottom two rows (g,h,i) and (j,k.l), show the how the side of the curve that is filled has no effect on the winding number calculation.

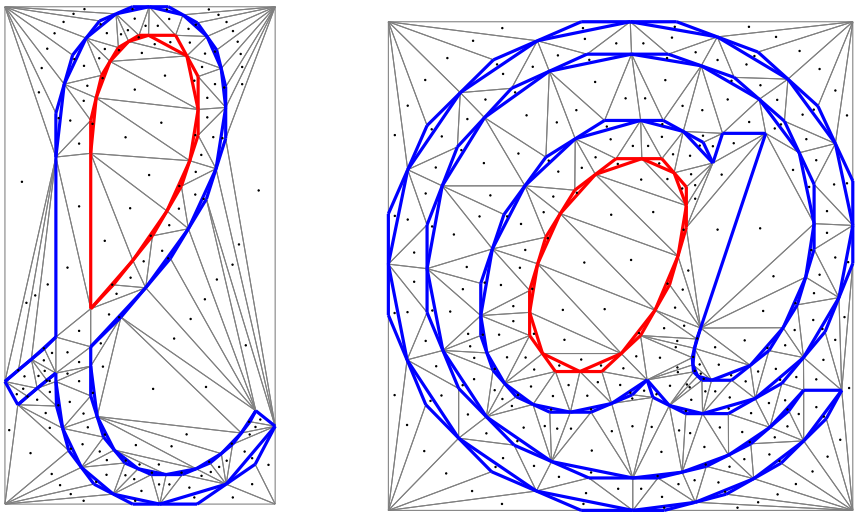


Figure 9: Finding the incenter of each of the triangles in a glyph outline

```
      tri.crossing = ZERO_SUM;
    }
  }
}
```

6.6 Finding Incenters for all Triangles

In the following step, calculating the winding number for all triangles, requires that the incenter for *all* triangles be found. Previously, only curve containing triangles had their incenters calculated. In this step, the incenter algorithm as described in section 5.3 is applied to all triangles. The result of applying this equation to all of the triangles in a glyph can be seen in figure 9.

6.7 Calculating the Winding Number for Incenters of the Triangles

The information calculated in sections 6.5 and 6.6 are used when calculating the winding number for the incenters of the triangles. The triangles that do not contain curved contour segments define an area of points whose winding numbers fall into one either the zero or non-zero category.

When rendering a filled glyph, knowing if a triangle's interior points fall into the non-zero or zero category corresponds to a triangle that is in the interior or exterior of a glyph.

```
// Iterate over all triangles generated by the Delaunay triangulation.
// None of these triangles contain curved contour segments
for(tri : triangle_set) {
   int winding_number = 0;
   vec2 incenter = tri->getIncenter();
```

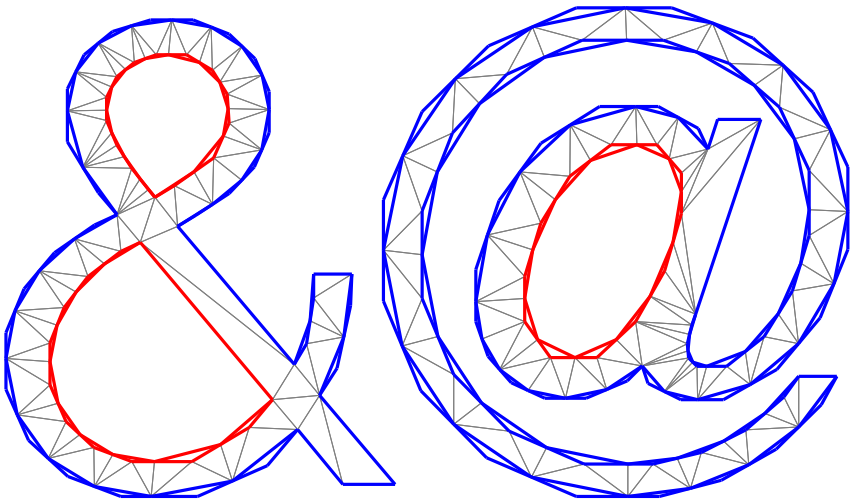


Figure 10: Triangulation after non-contributing triangles have been removed

```
    // Iterate over all of the segments that contain curved contours. Check to see
    // if the incenter is to the left of the segment.
    for(seg : segment_set) {
        if(rayIntersects(incenter, seg)) {
            // The delta winding was computed during contour classification
            windingNumber += seg->getDeltaWinding();
        }
    }

    // This is the method that matches the spec. A point in a glyph is colored
    // if its winding number is non-zero
    if(windingNumber != 0) {
        tri->setContributing(true);
    }
}
```

After the winding numbers for all incenters are found, verification of the algorithm up to this point was performed by creating figure 10. In this figure, all of the non-contributing triangles have been removed. This verification step is performed to help with the debugging process. In section 6.14, a fragment shader has been created that uses the triangle data to determine the shading of a fragment. Fragment shaders can be difficult to debug. Part of the difficulty lies in determining if the data being sent to the GPU is incorrect or if the shader is incorrect. This step of generating a figure that shows the non-contributing triangles being removed helps to verify the data prior to passing it to the GPU.

(a) Below curve filled (b) Above curve filled

Figure 11: Filling the above and below the curve

6.8 Side of Curve to Fill

After calculating the winding numbers for the incenters of all triangles, the side of the curves to be filled is determined. Curves contained within triangles need to be characterized as either having the area above or below the curve filled. The difference between filling above and below the curve can be seen in figure 11. Filling below the curve is defined as filling the area between the curve and the base of the triangle formed by the outer two control points. Filling above the curve is defined as filling in the opposite.

To determine the side to fill, a ray is cast from left to right starting at the center of the base of the curve containing triangle. If the ray intersects any of the contour segments, including the contour segment included in this triangle, the winding number for that point is changed by the amount calculated in section 6.5. A non-zero winding number may be an indicator that this triangle contains a curve that is filled below. A winding number of zero definitely indicates that this triangle contains a curve that is filled above.

A ray is also cast from the point opposite the base of the triangle. If this point also has a non-zero winding number, then both sides of the curve are filled and the triangle is reclassified from being a curve containing triangle to simply a contributing triangle. It is also possible that the winding number of both points is zero, which turns this into a non-contributing triangle.

Psuedo-code is included to better describe this algorithm:

```
for(curve : curve_set) {
   // Winding number
   int wnBase = 0;

   // Create a ray that starts at the point opposite the base of the triangle
   vec2 rayAbove = _vertices[(*tri)->indices()[1]];

   // Create a ray that starts at the midpoint of the base of the triangle
   vec2 rayBelow = _vertices[(*tri)->indices()[2]] - _vertices[(*tri)->indices()[0]];
   rayBelow *= 0.5;

   // Check to see if
   for(seg : segment_set) {
      if(seg->rayIntersects(rayAbove)) {
            wnAbove += seg->getDeltaWinding();
      }

      if(seg->rayIntersects(rayBelow)) {
         wnBelow += (*testSeg)->getDeltaWinding();
```

Figure 12: Glyphs with their curves and triangles filled using the proper above and below filling schemes

```
      }
   }

    if(wnAbove != 0) {
       if(wnBelow != 0) {
          curve->setFillCurve(ABOVE_AND_BELOW)
          curve->setContributing(true);
       } else {
          curve->setFillCurve(ABOVE);
       }
    } else if(wnBelow != 0) {
       curve->setFillCurve(BELOW);
    } else {
       // Both winding numbers are zero, this is a strange case,
       // but could be possible. Do not fill above or below, mark
       // this triangle as non-contributing
       curve->setFillCurve(NONE);
       curve->setContributing(false);
    }
}
```

6.9 Verification of Triangle and Curve Data

An additional debugging step was performed prior to writing the shader program. In figure 12, it is verified that non-contributing and contributing triangles are properly classified and that curve fill is proper classified.

6.10 Mapping Control Points into (0,1)

After decomposition is complete and before the triangle and curve data can be sent to the GPU, this data needs to be mapped from glyph space, which is measured in 1/64ths of an inch into the range (0,1). Care needs to be taken during this step to preserve the glyph's aspect ratio while at the same time mapping into this space such that glyphs are scaled and translated relative to each other. For example, the character 'f' has it's bottommost point at the baseline, but the character 'g' has a descender that goes below the baseline. See figure 1 in section 4.1 for a description of glyph metrics.

When this mapping is performed, the minimum and maximum positions of points in all of the glyphs in a font need to be taken into account. An affine transformation that maps the y range (minimum_y, maximum_y) to (0,1). It is tempting to do use minimums and maximums on a per glyph basis, but this does not preserve the descenders.

After the y-range is mapped, the x-range needs to be mapped to (0,1) such that the aspect ratio and the bearing is preserved.

6.11 Creating Inverse Matrices

Section 5.1 describes the math for determining if a point lies under a curve. The matrix $\mathbf{Q}^{-1}\mathbf{P}^{-1}$ is calculated and passed to the GPU so that the fragment shader can determine if a fragment that lies inside of a segment containing triangle should be shaded.

```
// The matrix Q as described in the Bezier Curves
// section of the thesis
vec3 q0(1.0f, -2.0f,  1.0f);
vec3 q1(0.0f,  2.0f, -2.0f);
vec3 q2(0.0f,  0.0f,  1.0f);
mat3 Q = inverse(mat3(q0, q1, q2)));

mat3 P(vec3(v[0], 1), vec3(v[1], 1), vec3(v[2], 1));
mat3 inv = Q * inverse(P);

_invData.push_back(inv);
```

6.12 Placing Data into Texture Map

Two texture maps are created for each glyph. The first texture map contains information about the triangles, specifically the points that make up those triangles and whether or not the triangle contributes to the glyph. The second texture map contains data that defines inverse matrices described in section 6.11

In figure 13, data is packed tightly into two texels. This data contains the vertices of the triangle, whether or not it is a contributing triangle, and if the triangle contains a curve, the alpha component of the second texel indicates if the triangle is filled above or below the curve. The blue component of the second texel takes on the following values:

- -2 if the triangle is non-contributing and is not filled
- -1 if the entire triangle contributes to the glyph

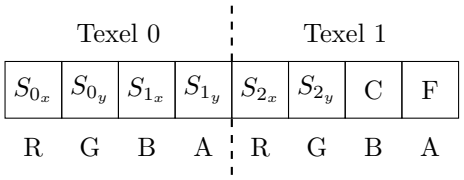


Figure 13: Triangle data packed into two contiguous RGBA texels. The first texel holds data for two of the points of a triangle, the second texel holds the third point. C determines if the triangle is non-contributing, contributing, or curve containing. If $C \geq 0$, the triangle contains a curve and C is an index into the inverse matrix texture map. $F = -1$ if a curve-containing triangle is filled below the curve, $F = 1$ if filled above the curve.

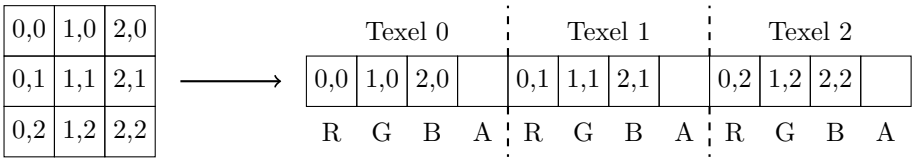


Figure 14: Copying the inverse matrix data from CPU memory to GPU memory. Each row in the 3x3 matrix is placed into a RGBA texel. The alpha element of each texel is unused.

- 0 or greater if the triangle contains a curve. In this case, this value is used as an index into the texture map that contains the inverse matrices.

In figure 14, it can be seen that the data is not tightly packed. Each row of the 3x3 inverse matrix is packed into three separate RGBA texels. The alpha component is unused. This matrix is used by the fragment shader to determine on which side of the curve a fragment lies.

6.13 Vertex Shader

The vertex shader is very simple. The vertex positions are transformed into the canonical view volume and the texture coordinates are passed through untouched. The input texture coordinates range from (0,0) to (1,1)

```
#version 150

in vec4 vertex;
in vec2 texcoord;
out vec2 tc;
out vec4 pos;
uniform mat4 mvp;

void main(void) {
   gl_Position = mvp * vertex;
   pos = gl_Position;
```

```
   tc = texcoord;
}
```

6.14 Fragment Shader

The fragment shader shader is where the majority of the GPU-based computation occurs. This is where it is determined whether or not a fragment contributes to the glyph. The fragment shader consists of five parts:

- Main loop
- Triangle Search
- Determine if the fragment is inside of a triangle
- Determine if the fragment is part of a curve
- Antialiasing

6.14.1 Main Loop

In the main loop of the fragment shader, the triangle list is searched to determine if a fragment contributes to the glyph. If the fragment is in a non-contributing triangle, return 0. If the fragment is inside of a contributing triangle, return 1. If the fragment is within a triangle that contains a curve, an additional test is performed to determine if the fragment is below or above the curve, and depending on the side to fill, a 0 or a 1 is returned.

```
float searchTriangles(vec2 s0) {
   for(int i = 0; i < numTriangles; ++i)    {
      vec4 part0 = texelFetch(triangleData, ivec2(i * 2 + 0, 0));
      vec4 part1 = texelFetch(triangleData, ivec2(i * 2 + 1, 0));
      float inside = insideTriangle(s0, part0.rg, part0.ba, part1.rg);

      if(inside > 0) {
         if(part1.b <= -2)  { // Non-contributing triangle
            return 0;
         } else {
            if(part1.b <= -1) { // Contributing triangle
               return 1;
            } else { // Contains curve
               // Index into the texture map that contains the inverse matrices
               // using part1.z and pass in the inflection parameter from part1.w
               return underCurve(s0, int(part1.z), part1.w);
            }
         }
      }
   }
```

```
   // If flow gets here, the fragment is in an area that is not
   // covered by a triangle. Do not fill in this area
   return 0;
}
```

6.14.2 Checking Triangle Boundaries

This function uses the math described in section 5.5 to determine if a fragment lies within a triangle.

```
/**
 * Check to see if a point is inside a triangle
 *
 * @param p0,p1,p2
 *    The points the define the triangle
 * @param x
 *    The point to test
 *
 * @return  1.0 if the point is inside the triangle, 0.0 otherwise
 */
float insideTriangle(vec2 x, vec2 p0, vec2 p1, vec2 p2)
{
   vec2 v1 = p1.xy - p0.xy;
   vec2 v2 = p2.xy - p0.xy;

   float denom = 1.0 / (v1.x * v2.y - v1.y * v2.x);

   float det_x_v2  =    x.x * v2.y -    x.y * v2.x;
   float det_p0_v2 =   p0.x * v2.y -   p0.y * v2.x;
   float det_x_v1  =    x.x * v1.y -    x.y * v1.x;
   float det_p0_v1 =   p0.x * v1.y -   p0.y * v1.x;

   float a = (det_x_v2 - det_p0_v2) * denom;
   float b = (det_p0_v1 - det_x_v1) * denom;

   float retval;
   if(a > 0 && b > 0 && a + b < 1) {
      retval = 1.0;
   } else {
      retval = 0.0;
   }

   return retval;
}
```

6.14.3 Checking Above and Below the Curve

If a fragment lies within a triangle that contains a curve, then it needs to be determined if the fragment is contributing to the glyph by using the math described in section 5.1. The side of the curve that is checked is determined by the inflection parameter. This parameter is precomputed in section 6.8 and placed into the inverse matrices texture map.

```
/**
 * Check to see if a point is under (or above the curve)
 *
 * @param s0
 *    The point to check
 * @param idx
 *    Index into the array of inverse matrices
 * @param inflection
 *    -1 to check under the curve, 1 to check above the curve
 * @return
 *    If inflection is -1, return value is 0 if the point is above
 *    the curve, 1 if the point is under the curve. If the inflection
 *    is 1, the return value is 1 if the point is above the curve,
 *    0 if below
 */
float underCurve(vec2 s0, int idx, float inflection) {
   vec3 q0 = texelFetch(invMatrices, ivec2(idx * 3 + 0, 0)).rgb;
   vec3 q1 = texelFetch(invMatrices, ivec2(idx * 3 + 1, 0)).rgb;
   vec3 q2 = texelFetch(invMatrices, ivec2(idx * 3 + 2, 0)).rgb;
   mat3 invMat = mat3(q0, q1, q2);

   vec3 t = invMat * vec3(s0, 1);

   // Does the solution have the form [1, t, t^2]? If it does,
   // Then t.z == t.y * t.z, or t.z - t.y * t.y will be close to zero
   float diff = inflection * (t.z - t.y * t.y);

   float retval;

   if((diff >= 0 && t.y >= 0 && t.y <= 1)) {
      retval = 1;
   } else {
      retval = 0;
   }
   return retval;
}
```

6.14.4 Antialiasing

Antialiasing is performed by sampling the glyph at multiple sub fragments. Acceptable results were achieved by sampling on a 3x3 grid for a total of nine samples. The built in GLSL functions, dFdx and dFdy were used to find the distance from the current fragment to the neighboring fragments in texture space.

```
float glyph(vec2 s0) {
   // Find the derivative with respect to x and y for the coord
   // variable by considering coord's values in the adjacent
   // fragments.
   vec2 deltaToNextPixelX = dFdx(s0);
   vec2 deltaToNextPixelY = dFdy(s0);

   float width = 3;
   float alpha = 0;
   int count = 0;
   int idx = 0;
   // Take width*width number of samples
   vec2 stepX = 2 * deltaToNextPixelX / width;
   vec2 stepY = 2 * deltaToNextPixelY / width;
   vec2 xpos = tc - deltaToNextPixelX;
   for(int i = 0; i < width; ++i)
   {
      vec2 ypos = - deltaToNextPixelY;
      for(int j = 0; j < width; ++j)
      {
         vec2 cpos = xpos + ypos;
         alpha += glyphTriangle(cpos);
         ypos += stepY;
      }
      xpos += stepX;
   }

   // Normalize the color value
   alpha /= (width * width);

   return alpha;
}
```

7 Rendered Glyphs

In this figure 15, the result of the final program is seen. A selection of Unicode characters are presented to demonstrate capabilities of this software. Figure 16 shows the results of placing a glyph onto non-flat geometry. As expected, resolution and geometry independence with high-quality rendering are achieved.

Figure 15: Various glyphs rendered using the GPU-based algorithm described in this thesis

8 Future Work

Future work should involve speeding up the algorithm. In the algorithm's current form, all triangles may need to be searched to find out if the point is contributing to the glyph. However, it should be possible to do a low resolution rendering of a glyph, for example on an 8x8 square, and record the list of triangles that are within each of those 64 texels. This information could be used in more detailed renderings by determining which of the 8x8 texels are being drawn an only search the triangles within this restricted set.

Also, typesetting entire strings should be added to the library. The current version of the library only renders a single glyph at a time. Instead, an entire string of glyphs could be rendered. The method I propose is to lay out the text on the CPU and to pass in a texture map that contains indices to the glyphs to be rendered at that position in the texture map.

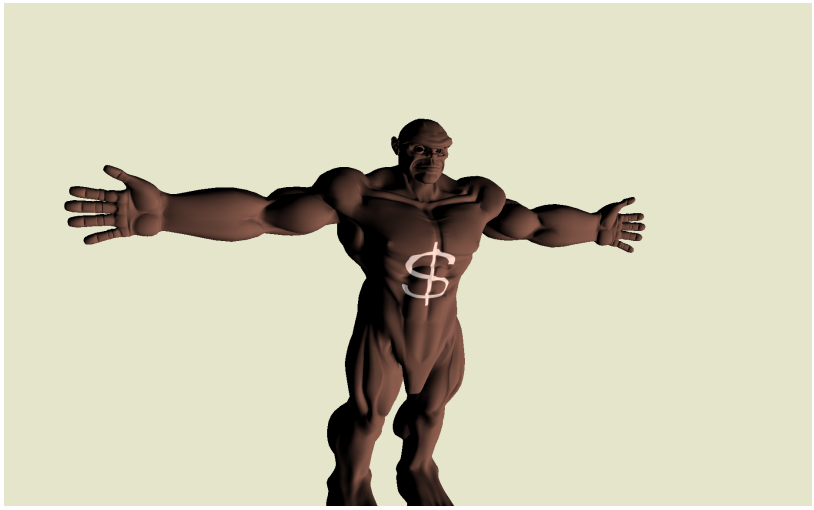
Another interesting option to try is to completely eliminate the Delaunay triangulation. Instead of searching the set of triangles, search for ray/contour intersections in the fragment shader to determine the winding number. This may require a greater amount of computation within the fragment shader, but it could also be sped up in the same manner that was previously mentioned.

The most promising idea is to creating a texture map that samples a signed distance function. This distance function would take an (s, t) texture coordinate as input and return the distance to the contour. The sign can be used to determine if the texel lies with the glyph outline. This may also require an encoding of the derivative of the slope of the curves and lines of the contours to ensure that sharp edges are preserved.

There are two problems that I believe exist with the approach presented in this thesis. Outlines of glyphs cannot be rendered and the fragment shader runs in $O(n)$ time. I believe that the signed distance function approach would eliminate both of these problems. Sampling the distance to the contour and storing the result in a texture map would allow the rendering of glyph outlines and would also change the running time to $O(1)$.

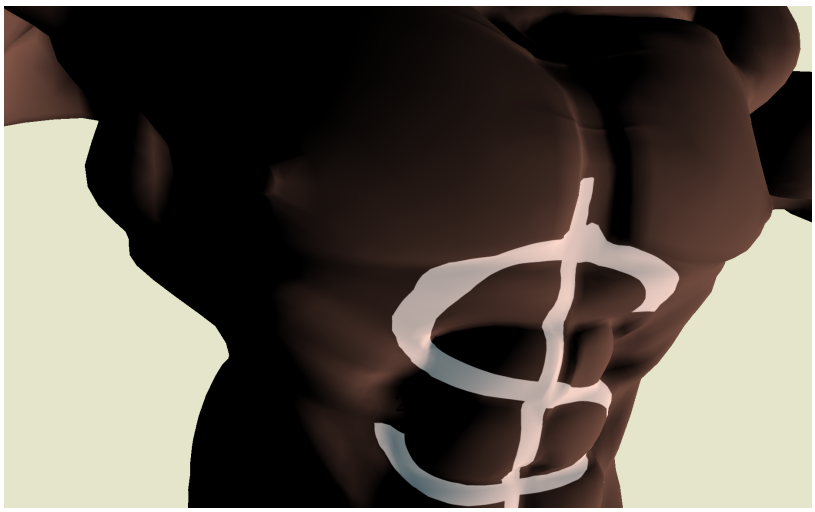
9 Conclusion

High quality, resolution and transformation rendering of glyphs defined by the contours a font face is possible to do within a fragment shader. Testing showed that when the glyphs took up a small amount of screen space, performance was quite good, but when a glyph filled the screen, performance became slow. This method should be investigated more carefully and turned into a full featured freely available library to provide graphics developers with easy access to this method of high quality rendering.



(a) Zoom level 1

(b) Zoom level 2



(c) Zoom level 3

Figure 16: A glyph applied to non-flat geometry. No loss of quality is experienced at all zoom levels.

References

[1] Apple. Truetype reference manual. `https://developer.apple.com/fonts/TTRefMan/index.html`.

[2] John Kessenich. *The OpenGL Shading Language*. July 2009. Language Version 1.50.

[3] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph.*, 24(3):1000–1009, July 2005.

[4] The FreeType Project. What is freetype 2? `http://freetype.sourceforge.net/freetype2/index.html`.

[5] Z. Qin, M.D. McCool, and C.S. Kaplan. Real-time texture-mapped vector glyphs. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 125–132. ACM, 2006.

[6] Nicolas Ray, Xavier Cavin, and Bruno Lvy. Vector texture maps on the gpu. Technical report, 2005.

[7] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 3.2 (Core Profile))*. December 2009.

[8] Jonathan Richard Shewchuk. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996. From the First ACM Workshop on Applied Computational Geometry.

[9] T. Wright. History and technology of computer fonts. *Annals of the History of Computing, IEEE*, pages 30–34, 1998.