GPU-Centered Font Rendering Directly from Glyph Outlines

Eric Lengyel
Terathon Software

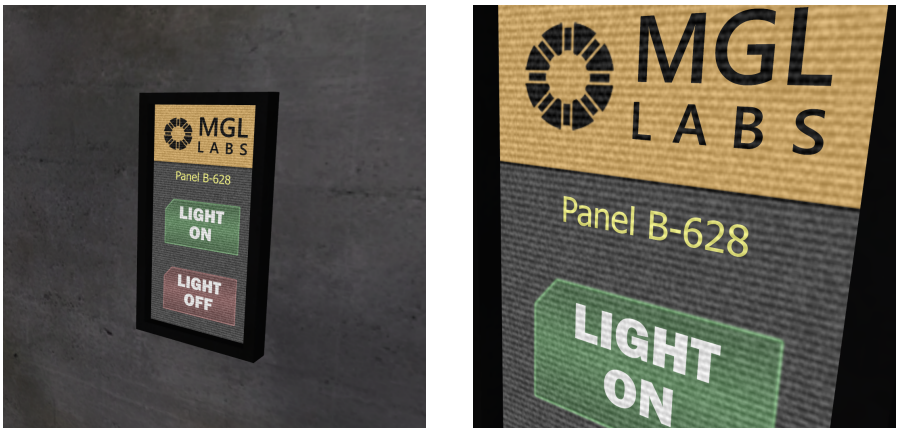


Figure 1. Glyphs are rendered in a game level directly from Bézier curve data extracted from a TrueType font. Because no precomputed images or distance fields are utilized, the results are pixel accurate under any affine or projective transformation, including all scales, rotations, and perspective distortions.

Abstract

This paper describes a method for rendering antialiased text directly from glyph outline data on the GPU without the use of any precomputed texture images or distance fields. This capability is valuable for text displayed inside a 3D scene because, in addition to a perspective projection, the transform applied to the text is constantly changing with a dynamic camera view. Our method overcomes numerical precision problems that produced artifacts in previously published techniques and promotes high GPU utilization with an implementation that naturally avoids divergent branching.

1. Introduction

Games and other real-time 3D applications often have a need to render text on various surfaces inside a virtual environment, as shown in Figure 1. Because the camera is

 ISSN 2331-7418

almost always moving in some way, the glyphs that compose such text are drawn with continuously changing transforms, and thus the size of the glyphs in the viewport are almost never the same from one frame to the next. Furthermore, the glyphs are usually drawn with perspective distortion because the camera isn't pointed straight at the surface to which text is applied. This creates a demand for the ability to dynamically render text with high quality no matter what affine and projective transformations have been applied to it.

Prerendered glyphs stored in a texture atlas have traditionally been used to apply text to in-game surfaces. Despite being simple and extremely fast, this technique suffers from unsightly blurring due to bilinear filtering once the displayed sizes of the glyphs exceed the resolution at which they're stored in the texture atlas. This problem was addressed by the introduction of signed distance field (SDF) methods [Green 2007], which produce crisp boundaries at all scales by storing distances to the glyph boundaries in the texture atlas instead of the final appearance of each glyph at a particular size. However, these methods tend to round off sharp corners and thus do not preserve the true outlines of the glyphs. Multichannel signed distance fields [Chlumský 2015] corrected the corner rounding problem, but required a complicated analysis step in the preparation of the texture atlas and created a new class of difficult-to-avoid artifacts for complex glyphs.

All of the techniques that store data in a texture atlas are inherently using a discrete sampling of what is actually an infinitely precise description of a glyph outline. This inescapably leads to limitations that can be mitigated by increasing the resolution of the texture atlas, but that can never be completely removed. For applications that need to render a wide range of characters at potentially large font sizes, a texture atlas capable of producing glyphs at an acceptable level of quality may have prohibitively large storage requirements.

The limitations of sampling can be avoided altogether by rendering glyphs directly from the mathematical curves that define their shapes. No longer does the source data for each glyph have an intrinsic resolution, because the exact positions of the outline's control points are utilized throughout the rendering process without any prior sampling. The Loop-Blinn [2005] method renders text directly from outline data by constructing a triangle mesh for each glyph that incorporates the control points as vertex positions. Each triangle corresponds to at most one component of a glyph's outline, and a short calculation in the pixel shader determines whether each pixel is inside or outside the boundary with respect to that one component. This method effectively renders precise glyph shapes at all scales, but it requires a complicated triangulation step in which the number of vertices is tied to the number of control points defining each glyph. At small font sizes, these triangles can become very tiny and decrease thread group occupancy on the GPU, reducing performance. The variable and font-dependent numbers of vertices per glyph also make it somewhat difficult to

perform text layout in general, and greater difficulty arises when attempting to apply heavily triangulated glyphs to curved surfaces.

A more versatile solution renders each glyph using only two triangles to cover the glyph's bounding box. The pixel shader then accesses a subset of *all* of the components of the glyph's outline to determine whether each pixel is inside or outside the entire boundary. This requires that we have a robust way of dynamically calculating either a signed distance value or a winding number at each pixel. The winding number corresponds to the absolute difference of the number of closed contours wound clockwise around the pixel and the number of closed contours wound counterclockwise around the pixel. Previous attempts at implementing such methods [Esfahbod 2012; Dobbie 2016] have suffered from numerical precision issues that produce a variety of rendering artifacts. These artifacts often begin to appear upon modest magnification and manifest themselves as dropped pixels inside a glyph or incorrectly drawn pixels outside a glyph. Because the artifacts are usually due to round-off errors, they tend to be very position sensitive and thus "sparkle" as the location or scale of a glyph changes. These sparkles also tend to occur along straight lines, producing "streaking" artifacts at various angles inside or outside a glyph.

We present a new technique in this paper that solves the precision problem and completely eliminates all artifacts by taking a different approach. Once robustness is guaranteed, we focus on ways to minimize the number of outline components examined at each pixel for best performance. Our method requires only widely available GPU features and can be implemented on OpenGL 3.x / DX10 hardware.

2. Winding Number Calculation

A glyph is defined by a set of closed contours that are each composed of a continuous piecewise sequence of Bézier curves, as shown in Figure 2. Although cubic curves are supported by other formats, we restrict ourselves to the quadratic curves used by TrueType fonts to keep rendering calculations as simple as possible. A particular point is considered to be inside the glyph outline if the sum of its winding numbers with respect to all of the contours is nonzero. The winding number for each contour is an integer that reflects the number of complete loops the contour makes around a point. A positive number is assigned to one winding direction, clockwise or counterclockwise, and a negative number is assigned to the opposite winding direction.

The winding number is calculated by firing a ray in any arbitrary direction from the point being rendered and looking for intersections with each of the contours belonging to a glyph. The winding number is initialized to zero. When a contour crosses the ray from left to right, one is added to the winding number, and when a contour crosses the ray from right to left, one is subtracted from the winding number (or vice-versa, as long as the choice is consistent). If the final winding number is zero, then the

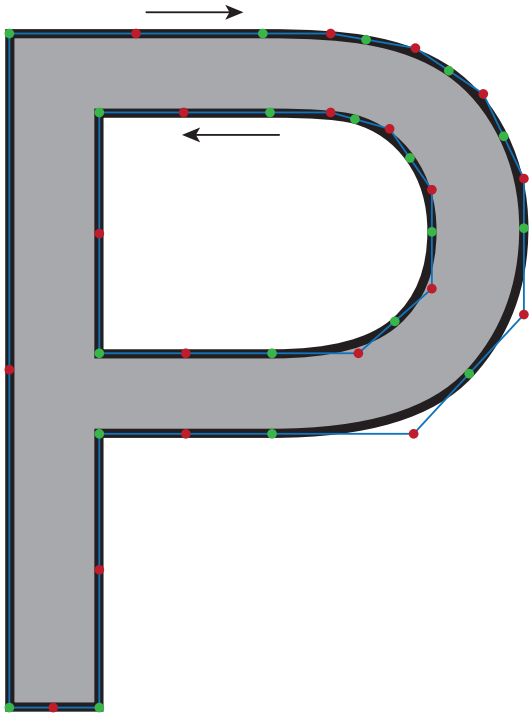


Figure 2. The shape of a glyph is defined by one or more closed contours, each composed of a continuous sequence of quadratic Bézier curves. The green dots represent on-curve control points, the red dots represent off-curve control points, and the blue lines are tangent to the outline. The clockwise winding convention is used in this example, meaning that contours wound in a clockwise direction contribute a positive winding number, and contours wound in a counterclockwise direction, like the interior loop of the letter P, contribute a negative winding number.

point lies in empty space. Otherwise, the point lies inside the glyph outline, and the fact that it can be positive or negative allows contours to employ either a clockwise or counterclockwise convention for defining interior regions of the glyph.

Since the ray directions don't matter, we choose directions that are parallel to the coordinate axes for convenience. A single component of a contour is defined by the parametric function

$$\mathbf{C}(t) = (1-t)^2\,\mathbf{p}_1 + 2t(1-t)\,\mathbf{p}_2 + t^2\,\mathbf{p}_3,$$

which constitutes a quadratic Bézier curve having the 2D control points $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$. The parameter t varies over the range $[0, 1]$. For a ray pointing in the direction of the positive x-axis in a coordinate system in which the point being rendered has been translated to the origin, we can solve for the values of t at which $C_y(t) = 0$ by finding the roots of the polynomial

$$(y_1 - 2y_2 + y_3)t^2 - 2(y_1 - y_2)t + y_1,$$

where we have set $\mathbf{p}_i = (x_i, y_i)$. The roots t_1 and t_2 are then

$$t_1 = \frac{b - \sqrt{b^2 - ac}}{a} \text{ and } t_2 = \frac{b + \sqrt{b^2 - ac}}{a}, \tag{1}$$

where $a = y_1 - 2y_2 + y_3$, $b = y_1 - y_2$, and $c = y_1$. In the case where a is near zero, we instead compute $t_{1,2} = c/2b$. For any values of t in the range $[0, 1)$ such that $C_x(t_i) \geq 0$, we have found an intersection between the ray and a contour. The value $t_i = 1$ is specifically disallowed because it corresponds to an intersection at $t_i = 0$ for the succeeding component in the contour, and we don't want to count an intersection at a shared control point twice.

To determine whether a ray intersection corresponds to a positive or negative change in the winding number, we examine the values of $C_y(t)$ before or after a root t_i , but only in the range $[0, 1)$ and only between the two roots if both fall in that range. If $C_y(t) > 0$ for $t < t_i$ or $C_y(t) < 0$ for $t > t_i$, then we add one to the winding number. Conversely, if $C_y(t) < 0$ for $t < t_i$ or $C_y(t) > 0$ for $t > t_i$, then we subtract one from the winding number. Note that these conditions exclude straight lines parallel to the ray from making any contribution to the winding number.

While sound from a purely mathematical standpoint, the method just described is plagued by numerical precision errors in any practical implementation whenever y_1 or y_3 is near zero. The problem is that the finite number of bits in a floating-point value are incapable of producing the exactness needed in calculating t_1 and t_2 when either could be close to zero or one. The result is that a ray passing too close to the point where two contour components are connected may end up counting two intersections or missing both curves altogether, leading to the sparkle and streak artifacts described in the previous section. The situation is especially bad if the Bézier curve is tangent to the ray at its first or last control point. Typical hacks, such as the use of epsilons or coordinate perturbation, may eliminate the problem in some cases, but these measures are not generally effective and do not lead to a robust solution.

We now introduce a different approach that achieves absolute, unconditional robustness over the entire space of finite inputs (i.e., no coordinate value is infinity or NaN). Our new method ignores the values of t_i insomuch as whether they satisfy $t_i \in [0, 1)$ and instead calculates winding numbers based solely on a binary classification of the values y_1, y_2, and y_3, specifically whether each is positive or not positive. Every quadratic Bézier curve then has a three-bit state that reduces the problem domain to exactly eight distinct equivalence classes. For all of the cases belonging to each equivalence class, contributions to the winding number arising from the two roots at t_1 and t_2 are handled in exactly the same way. Furthermore, whenever a contribution is made for the root at t_1, we always add one to the winding number, and whenever a contribution is made for the root at t_2, we always subtract one from the winding number. Thus, all we have to do is turn a three-bit input into a two-bit output, and we have all of the information necessary to properly handle all possible

configurations of a quadratic Bézier curve, including degenerate cases, with respect to a ray pointing in the positive x-direction.

The eight equivalence classes are illustrated in Table 1. In the columns labelled y_i, a one indicates that $y_i > 0$. In the columns labelled t_i, a one indicates that the root at t_i should make a contribution to the winding number if $C_x(t_i) \geq 0$. The contribution is always positive one for t_1, and it is always negative one for t_2, regardless of the order of $C_x(t_1)$ and $C_x(t_2)$. The representative curves shown in the table for each equivalence class cover all 27 cases in which $y_i < 0$, $y_i = 0$, and $y_i > 0$ in order to make it clear what happens in the important instances in which the ray passes directly through a control point. A change is made to the winding number whenever the curve transitions from positive to not positive or vice-versa, and these changes are indicated by green and red dots in the table. A green dot corresponds to a change of positive one occurring when the curve transitions from positive to not positive at the root t_1, and a red dot corresponds to a change of negative one occurring when the curve transitions from not positive to positive at the root t_2.

In equivalence classes A and H, no transitions between positive and not positive ever occur, and thus no change is made to the winding number. In each of the remaining six equivalence classes, the potential for a contribution to the winding number exists. The historically difficult case, in which a contour is tangent to the ray at an endpoint shared by two consecutive curves, is handled without explicit detection or special code. Equivalence class A covers all cases for which a contour is tangent to the ray at an endpoint but is otherwise *negative*, ensuring that the winding number is unaffected. In the similar case that a contour is tangent to the ray at an endpoint but is otherwise *positive*, two equal and opposite contributions are always made to the winding number, and they cancel each other out exactly. This is exemplified by the many combinations of tangent curves shown in the table in which a green dot and red dot would coincide when the curves are connected to each other. (Note that there is no requirement that the curves have a continuous derivative at the endpoint where they are joined.) In equivalence classes C and F, there is a special case in which y_1 and y_3 have the same state but y_2 has the opposite state, and it is possible that $C_y(t)$ has no real roots. In order to handle this case with uniformity, we clamp $b^2 - ac$ to zero, which has the effect of setting $t_1 = t_2 = b/a$. If one root makes a contribution, then the other one does as well in this case because $C_x(t_1) = C_x(t_2)$, so they cancel each other out. This combination of a positive and negative contribution is represented by the yellow dot shown in the table for class F.

The values in the columns labelled t_i in Table 1 form a 16-bit lookup table that can simply be expressed as the number `0x2E74`, with row A corresponding to the two least significant bits and row H corresponding to the two most significant bits. The values in the columns labelled y_i form a shift code such that when the lookup table is shifted right by twice that amount, then the output state for the corresponding

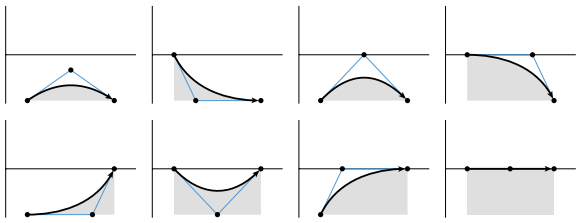
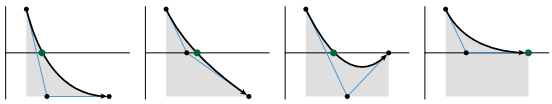
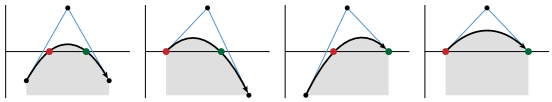
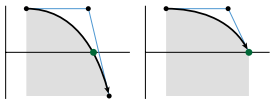
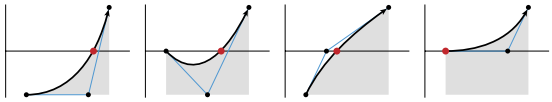
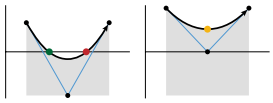
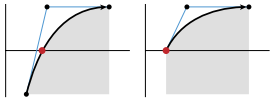
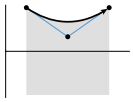
Class	y_3	y_2	y_1	t_2	t_1	Representative curves
A	0	0	0	0	0	
B	0	0	1	0	1	
C	0	1	0	1	1	
D	0	1	1	0	1	
E	1	0	0	1	0	
F	1	0	1	1	1	
G	1	1	0	1	0	
H	1	1	1	0	0	

Table 1. The three bits in the columns labelled y_i constitute an input code based on whether each y_i is positive, and they partition the set of all quadratic Bézier curves into eight equivalence classes. The two bits in the columns labelled t_i constitute an output code specifying whether each intersection with the x-axis should make a contribution to the winding number of the contour to which the curve belongs. Green dots indicate roots at which one is added to the winding number, and red dots indicate roots at which one is subtracted from the winding number. The shaded areas represent the interior of a glyph when a clockwise winding convention is followed.

equivalence class appears in the lowest two bits. Thus, given y_1, y_2, and y_3 for an arbitrary quadratic Bézier curve that has been translated so that the point being rendered is at the origin, all we have to do is calculate the value

$$((y_1 > 0)\ ?\ 2 : 0) + ((y_2 > 0)\ ?\ 4 : 0) + ((y_3 > 0)\ ?\ 8 : 0) \tag{2}$$

and use it to shift the number `0x2E74` rightward. If the lowest bit of the result is set, then one is added to the winding number when $C_x(t_1) \geq 0$. If the second lowest bit of the result is set, then one is subtracted from the winding number when $C_x(t_2) \geq 0$.

Because the precision-sensitive range checks on the values of t_i have been eliminated, it is no longer possible to miscount the number of intersections that a ray makes with a contour. The shift code is an exact calculation based on the translated y-coordinates of the input control points, which are invariant along any horizontal ray. The quadratic and linear terms of $C_x(t)$ are also invariant, leaving only the constant term equal to the translated x-coordinate of the control point $\mathbf{p}_1$ as the quantity that changes as the ray origin is moved left or right. This guarantees that there exists a value x_0 such that for all $x \leq x_0$, a particular contour intersection is counted, and for all $x > x_0$, the same intersection is not counted.

Of course, calculating a discrete inside/outside state at each point being rendered produces a pixelated, black-and-white image. Instead of calculating an integral winding number, we can accumulate coverage values that reflect how close each ray intersection is to the center of the pixel being rendered. The fraction f of a pixel crossed by a ray from left to right before an intersection occurs is given by

$$f = \text{sat}\left(m\, C_x(t_i) + \frac{1}{2}\right), \tag{3}$$

where m is the number of pixels in one em, which corresponds to the font size, and sat is the saturate function that clamps to the range $[0, 1]$. Adding and subtracting these fractions from the winding number has the effect of antialiasing in the direction of the rays. Averaging the final coverages calculated for multiple ray directions antialiases with greater isotropy, but at a performance cost. Considering only rays parallel to the coordinates axes is a good compromise, especially when combined with supersampling, as discussed later.

3. Performance Optimization

A glyph is correctly rendered when we consider every contour component for each pixel intersecting the glyph's bounding box and accumulate the coverage values. Fortunately, no branching is necessary to calculate a coverage value, so a pixel shader runs with high thread-group coherence on the GPU. However, many of the quadratic Bézier curves make no contribution because they never cross a horizontal or vertical ray fired from a particular pixel center, and the best performance is achieved by minimizing the number of components that need to be processed.

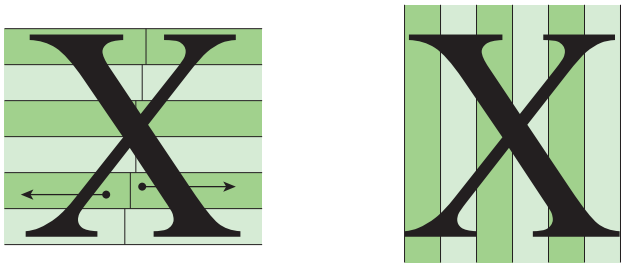


Figure 3. A glyph is divided into equal-width horizontal and vertical bands. Two lists of Bézier curves intersecting each band are created, one sorted in descending order by maximum x- or y-coordinate (for horizontal and vertical bands, respectively) and another sorted in ascending order by minimum x- or y-coordinate. Bands are split along a median position, and rays originating on either side point away from this median to reduce the number of curves that need to be processed.

We divide each glyph into a number of equal-width horizontal and vertical bands, as shown in Figure 3. The number of bands is proportional to the total number of Bézier curves composing the glyph's outline up to a limit of 16 in each direction. For each band, we create a list of the curves that intersect the band and sort them in descending order by their maximum coordinates in the band's direction (x for horizontal and y for vertical). When a pixel is rendered, we first determine which horizontal band contains it and loop over the Bézier curves that are known to intersect that band. As soon as we encounter a Bézier curve for which

$$\max\{x_1, x_2, x_3\}\, m < -\frac{1}{2}, \tag{4}$$

we can break out of the loop because Equation (3) produces a value of zero from that point onward. The process is repeated for the vertical band containing the pixel using a ray that points in the positive y-direction and checking the maximum y-coordinate of each curve for the early-out condition.

For rays pointing in the positive direction along an axis, more curves need to be processed for pixels near the left or bottom sides of a glyph than for pixels near the right or top sides. To reduce the costs of rendering these pixels and make the whole process more symmetric, we split each band into two parts at a location roughly corresponding to the median position of the Bézier curves in that band. As shown in Figure 3, pixels falling on the negative side of the split location fire rays in the negative direction instead of the positive direction. In these cases, the winding number contributions must be negated, so Equation (3) is replaced with

$$f = \text{sat}\left(\frac{1}{2} - m\, C_x(t_i)\right). \tag{5}$$

Because the ray is pointing in the opposite direction, we also need to sort the curves in the opposite order by their minimum coordinates and replace the early-out condition

given by Equation (4) with

$$\min\{x_1, x_2, x_3\}\, m > \frac{1}{2}. \tag{6}$$

Due to the divergence that it introduces in the pixel shader, the band split optimization can have a negative performance impact at small font sizes, so we make it an option for text that is known ahead of time to be rendered at larger sizes.

The most straightforward geometry to render for a glyph is a single quad corresponding to its bounding box. In order to capture all of the pixels for which Equations (3) and (5) could generate fractional values, the box needs to be expanded by half the width of a single pixel. The bounding boxes for uppercase letters are shown in the top row of Figure 4 to demonstrate how many glyphs contain nothing but empty space near the corners. We don't want to run the pixel shader for all of those pixels that end up being completely transparent, so we eliminate some of this empty space by clipping the corners of the bounding boxes where possible, as shown in the bottom row of Figure 4. Where to clip is determined by considering several normal directions at each corner, calculating the support plane for each normal direction with respect to all of the glyph's control points, and choosing the plane that clips off the triangle having the greatest area above some minimum threshold. As with the band split optimization, this geometry clipping optimization is more effective at larger font sizes. The tiny triangles that it can produce at small font sizes reduce occupancy on the GPU, which can result in slightly worse performance.

Figure 4. To reduce the number of pixels rendered for most glyphs, simple quads are replaced with polygons having up to eight sides after the corners of the bounding boxes are clipped.

4. Implementation

Our implementation stores the control points for every glyph in a four-channel 16-bit floating-point texture map. The first and second control points belonging to each Bézier curve are stored in the (x, y) and (z, w) components of a single texel. The third control point is stored in the (x, y) components of the next texel in the same row. Since the third control point of one curve is identical to the first control point of the next curve in the same contour, it is usually the case that the second texel is shared by two curves, and thus the total storage requirements for the control point data is slightly larger than eight bytes per curve.

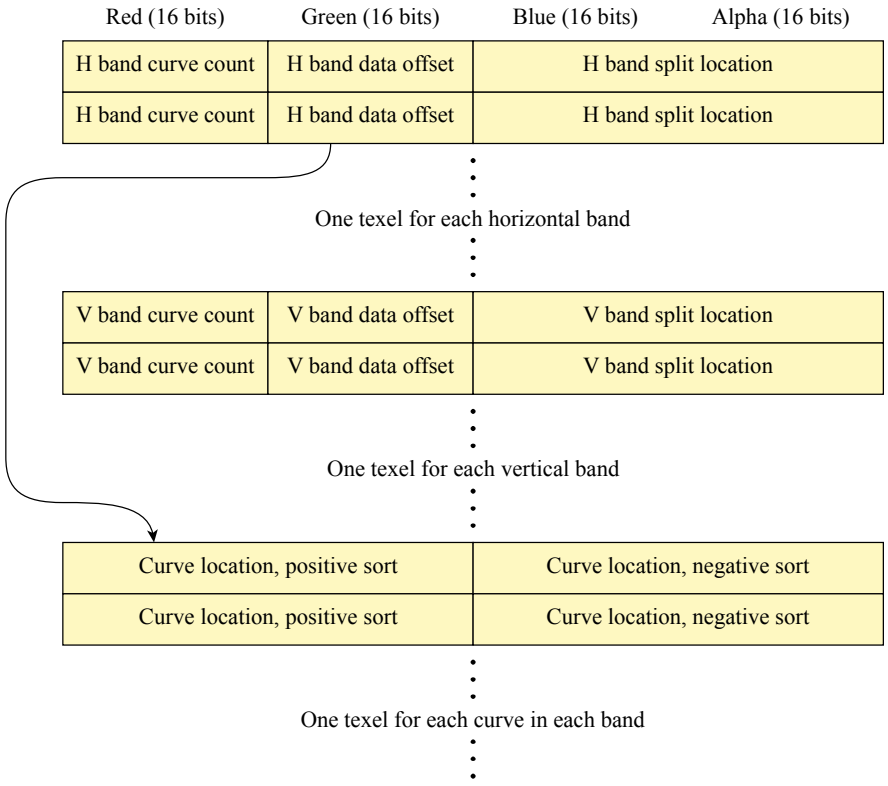


Figure 5. For every glyph, the band data texture includes a header texel for each horizontal and vertical band, and those are followed by the locations of the curves belonging to each band in the control point texture.

A second texture map containing four-channel 16-bit integer data holds the location of every Bézier curve intersecting the horizontal and vertical bands belonging to each glyph. The layout of this data is shown in Figure 5. The data for a glyph begins with a table of band headers for all of the horizontal bands followed by all of the vertical bands. The header fits into one texel and contains the number of curves intersecting the band, the offset to the list of curve locations, and the coordinate value at which the band is split between negative rays and positive rays. The list of curve locations is actually two lists that occupy different channels in the same set of texels. One set of (x, y)-coordinates holding the location of a Bézier curve in the control-point texture is stored in the red and green channels, and another set of (x, y)-coordinates is stored in the blue and alpha channels. The list of curves is sorted in descending order by maximum coordinate in the red and green channels for positive rays, and it is sorted in ascending order by minimum coordinate in the blue and alpha channels for negative rays.

The pixel shader that renders a glyph can be found in the supplemental materials. For the sake of brevity, the band split optimization is omitted, and rays are always

fired in the positive direction along each of the x- and y-axes. The pixel's position in em-square coordinates is interpolated and passed into the pixel shader through the `texcoord` input. The starting location of the band data is passed to the pixel shader from the vertex shader in the lower 12 bits of the x- and y-components of the `glyphParam` input. (This limits the band data texture to 4096×4096 texels.) The upper four bits contain the maximum band indexes for vertical bands in the x-component and horizontal bands in the y-component. Scale and offset parameters are passed in through the `bandParam` input, and they are used to calculate band indexes for each pixel. All bands, both horizontal and vertical, have the same width, so a single scale value is passed in through the z-component of `bandParam`. The x- and y-components contain separate offsets for vertical and horizontal bands, respectively. Once the scale and offsets have been applied, the resulting band indexes are clamped to the maximum values specified in the `glyphParam` input.

After the band indexes have been determined, the shader reads the headers from the band data texture, locates the per-band curve lists, and calculates a coverage value for each curve until the early-out condition is satisfied or all of the curves have been processed. For each curve, Equation (2) is used to calculate a shift code that is then applied to the lookup table `0x2E74` to move the root contribution code into the lowest two bits. Although not strictly necessary, the shader then tests whether these two bits are nonzero before continuing with the coverage calculation because we have found that doing so provides a small speed increase. If a contribution could be made, then the roots of the curve are calculated with Equation (1), and the cumulative coverage value is increased or decreased by the value given by Equation (3) as directed by the two-bit contribution code.

5. Results

The glyph rendering method described above has been integrated into our professional-grade game engine [Lengyel 2016], and it is used to handle all text drawing needs, including user interface widgets, heads-up displays, debugging facilities, and interactive panels rendered inside the game world. Samples of these usages are shown in Figures 6 and 7. The resolution independence of our method allows text to be crisply rendered at a constant physical size in DPI-aware applications across monitors having different pixel densities. It also provides a way to render crisp text when the pixel density may not even be constant over a single glyph, as is the case for text drawn inside a game world where the camera may be viewing it at an oblique angle.

Compared to a basic text shader that does nothing more than sample glyphs from a prerendered texture map, containing either final coverage values or a signed distance field, it should be clear that our method requires considerably more computation. The cost of the flexibility and scalability provided by rendering directly from outline

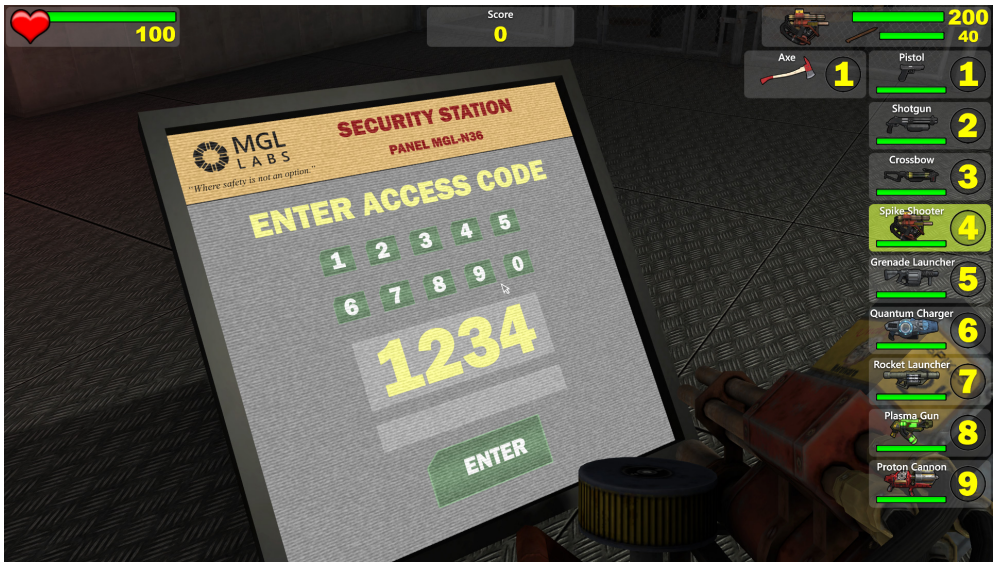


Figure 6. Text is rendered with our method for the heads-up display, showing information about health, score, and weapons, as well as the interactive panel embedded in the game world itself and viewed at an oblique angle.

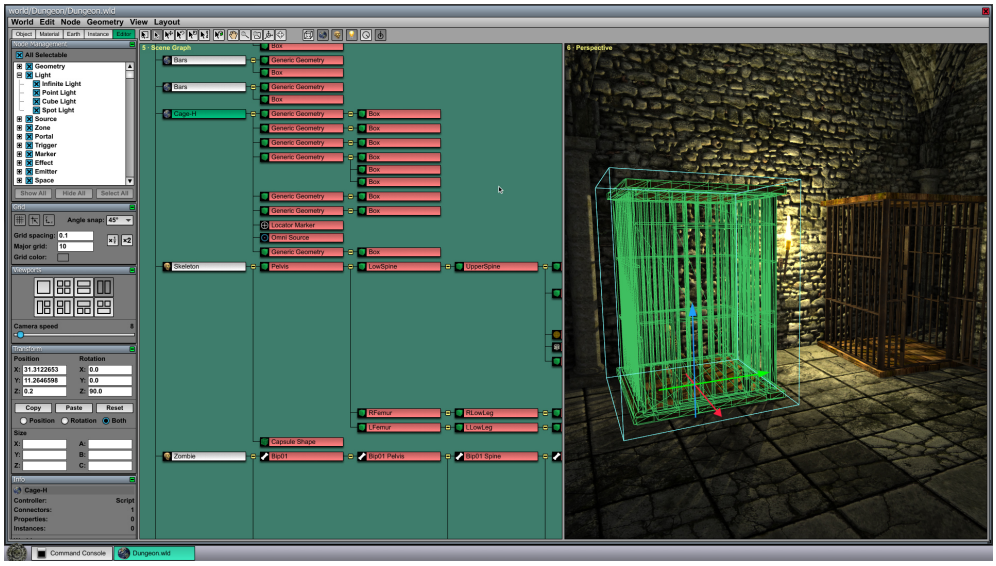


Figure 7. Our method is used to render text in a world editor comprising a complex user interface that contains windows, menus, lists, check boxes, push buttons, and a variety of additional widgets. Resolution independence allows the glyphs to cleanly scale to properly match system DPI settings.

Font	Sample	Complexity	Time
Arial	ABCDEFG	20	1.1 ms
Centaur	ABCDEFG	48	1.3 ms
Halloween	ABCDEFG	72	3.8 ms
Wildwood	ABCDEFG	546	13.3 ms

Table 2. A two-megapixel area was filled with 50 lines of text rendered at 32 pixels per em using a variety of fonts and timed on a GeForce GTX 1060 graphics chip. The complexity value represents the typical number of Bézier curves composing an uppercase letter in each font.

data was measured by filling a two-megapixel area with text drawn in a variety of fonts and recording the time needed to render it on a GeForce GTX 1060 graphics chip. In the best-performing case, a conventional shader sampling prerendered glyph images (without an SDF) requires 26 µs to render the text, and the time required by our method is 1.1 ms, roughly 40 times as long. The performance of our method strongly depends on the complexity of the font, as determined by the typical number of Bézier curves composing a glyph, so the best case is achieved using a font like Arial that has simple outlines and no serifs. Timings for more complex fonts are listed in Table 2.

The Dobbie [2016] method, which is the previously published method closest to ours in terms of algorithm design, requires 5.2 ms to render the same text in the same area using the Centaur font. Its measured time of 10.4 ms was cut in half to account for the fact that it evaluates four rays, although not a requirement of the algorithm, instead of the two used by our method. Even after this adjustment, our implementation is four times as fast, requiring only 1.3 ms.

A TrueType font needs to be preprocessed in order to generate the data format that is consumed by the glyph shader. The glyph outlines contained in a TrueType font usually have many implicit control points that require no storage. (An implicit on-curve or off-curve control point is one that falls exactly halfway between two explicit control points of the opposite type.) Because the glyph shader must be able to fetch all three controls points belonging to any quadratic Bézier curve, every control point must be included in the final data, which increases storage requirements. Furthermore, the band data needed for efficient rendering adds considerable storage requirements beyond what is found in a TrueType font. In general, we have found that the preprocessed data needed by our method is roughly twice as large to several times as large as the TrueType font from which it is derived. The size differences for a

Font	Glyph Count	TTF Size (KB)	Data Size (KB)	Curve Texture Size	Band Texture Size
Wildwood	64	118	631	4096×6	4096×23
Halloween	131	57	294	4096×2	4096×17
Centaur	241	81	201	4096×2	4096×9
Arial	3223	894	1549	4096×11	4096×55
Times	3502	1085	2201	4096×16	4096×85
JhengHei	29,386	20,650	52,842	4096×507	4096×2350

Table 3. This table lists the number of glyphs contained in a variety of TrueType fonts and the sizes of the original `.ttf` files. The Data Size column lists the storage requirements of the glyph data after processing to generate the curve and band texture maps. This size also includes a small amount of per-glyph data and information about kerning, ligatures, and combining diacritical marks. The last two columns give the dimensions of the curve and band texture maps that were generated.

variety of fonts containing a wide range of characters are listed in Table 3 along with the dimensions of the texture maps that were generated to hold the final data.

6. Extensions

There are a number of ways in which our glyph rendering method can be extended. In particular, it is straightforward to implement techniques that utilize multiple samples per pixel. As the ray origin is moved perpendicular to the direction in which the ray points, the values of a and b used to calculate roots in Equation (1) are invariant, so a significant amount of work can be shared over multiple samples. A simple box filter can be implemented by shifting the ray origin up and down within a pixel's footprint for horizontal rays, or right and left for vertical rays, and averaging the coverage values calculated for each Bézier curve. Figure 8 shows the result of supersampling in this manner. Because the pixel size grows larger in em space as a font is rendered at smaller sizes, care must be taken to expand the width of the bands by half of the largest pixel size, equal to the reciprocal of the smallest font size, when collecting the curves that intersect each band. Otherwise, the lists of curves belonging to each band may not be valid for all sample positions within a pixel.

By inflating the pixel footprint and applying a more sophisticated filter, effects such as a glow or drop shadow can be implemented. As with the supersampling technique, sample positions are always arranged on a line perpendicular to the ray direction. An increase in the pixel size corresponds to a decrease in the value of m used by Equations (3) and (5), and this causes the coverage gradient to be spread out over a longer distance for a softer appearance, as shown in the drop shadow in Figure 8. Again, the bands must be expanded to account for the largest effect radius so that the lists of curves are valid for each sample point.

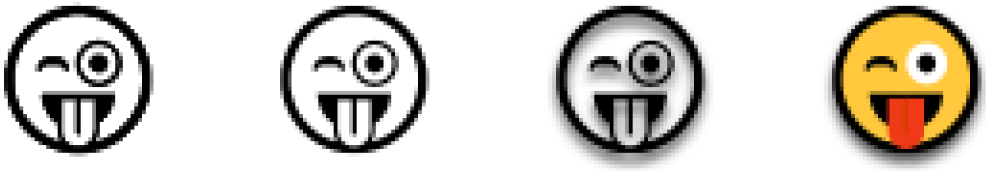


Figure 8. Emoji glyph with Unicode value U+01F61C rendered at 34 pixels per em. The leftmost image is rendered with the average coverage calculated for a horizontal ray and a vertical ray. The second image applies supersampling with three samples in each direction. The third image adds a drop shadow, and the fourth image renders six color layers.

An extension to the TrueType format facilitates multicolor glyphs by defining outlines for multiple layers that are each rendered in a different color and stacked on top of each other. (This data appears in `'COLR'` and `'CPAL'` tables inside a font.) This can be implemented by adding an outer loop to our pixel shader and including some extra color data in our texture maps. The result is the ability to render multicolor emoji and pictographs, as shown in Figure 8.

References

CHLUMSKÝ, V. 2015. *Shape Decomposition for Multi-channel Distance Fields*. Master's thesis, Czech Technical University. URL: https://dspace.cvut.cz/bitstream/handle/10467/62770/F8-DP-2015-Chlumsky-Viktor-thesis.pdf. 32

DOBBIE, W., 2016. Gpu text rendering with vector textures. Web post. URL: http://wdobbie.com/post/gpu-text-rendering-with-vector-textures/. 33, 44

ESFAHBOD, B., 2012. Glyphy. Software library. URL: https://github.com/behdad/glyphy. 33

GREEN, C. 2007. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, ACM, New York, NY, USA, SIGGRAPH '07, 9–18. URL: http://www.valvesoftware.com/publications/2007/SIGGRAPH2007_AlphaTestedMagnification.pdf. 32

LENGYEL, E., 2016. Tombstone engine. Software game engine. URL: http://tombstoneengine.com. 42

LOOP, C., AND BLINN, J. 2005. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph. 24*, 3 (July), 1000–1009. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2005/01/p1000-loop.pdf. 32

Index of Supplemental Materials

The file `GlyphShader.glsl` contains a GLSL pixel shader that renders a glyph using the control point data and the band data stored in textures named `bandTex` and `curveTex`. The coverage calculated at each pixel becomes an alpha value that is combined with an interpolated vertex color.

Author Contact Information

Eric Lengyel
Terathon Software
lengyel@terathon.com

Eric Lengyel, GPU-Centered Font Rendering Directly from Glyph Outlines, *Journal of Computer Graphics Techniques (JCGT)*, vol. 6, no. 2, 31–47, 2017
http://jcgt.org/published/0006/02/02/

Received:	2017-03-27		
Recommended:	2017-04-26	Corresponding Editor:	John Hable
Published:	2017-06-14	Editor-in-Chief:	Marc Olano