The Scanline Sweeper: A Glyph Rendering Algorithm

Scuff3D Rook with *Rook & Possum*

(v0.3)

Abstract

One fundamental building block of Bézier curve rasterization has historically been winding-number computation. Winding-number evaluation per-pixel or per-sample permits rasterization on both the CPU and GPU, but presents implementation challenges due to the need for precise quadratic root-finding. Algorithm failures can result in large discrepancies in the computed coverage estimate for a pixel, and while solutions for these numerical failures exist, a different direction based on continuous coverage estimates is worth exploring.

In this paper, I present the Scanline Sweeper, a distinct paradigm for Bézier curve rasterization that estimates coverage analytically, without computing explicit winding numbers and without any tessellation. While the Scanline Sweeper uses quadratic root-finding, perturbations in the outcome only modestly perturb final coverage estimates, resulting in a numerically robust algorithm.

Author's disclaimer: This is a self-published preprint, not subject to a traditional peer-review, and not submitted to any academic journal. While the ideas here are original in the sense that I have not seen them elsewhere, my background is primarily in 3D graphics programming for AAA games, so it's possible these ideas are already explored in an industry or domain unfamiliar to me. Errors will be corrected and amended versions will be posted as time permits.

Citation: Scuff3D Rook, The Scanline Sweeper: A Glyph Rendering Algorithm, *Rook & Possum*, 2026.

1. Introduction

Rendering font glyphs in games and interactive applications predominantly relies on CPU rasterization. After rasterization is performed on the CPU, applications maintain a *glyph cache* to store texture fonts that are uploaded and sampled later on the GPU. Glyph cache maintenance is a tricky affair, since rasterization results can change based on glyph size and transform. Glyphs are expected to be positioned on pixel boundaries, and text anti-aliasing artifacts are noticeable in UI engines that don't snap glyphs to the pixel grid. Rendering text in a 3D context with CPU-rasterized glyphs is impossible without objectionable aliasing artifacts, because the CPU rasterizer has no awareness of how the glyph will ultimately be projected on the display.

To combat aliasing issues in 3D contexts, a popular preprocessing step is to convert pre-rasterized glyphs into a signed-distance field (SDF) representation. Sampling SDFs reduces aliasing by filtering distances on the GPU at sample time. However, the filtered result, while smooth, may no longer be an accurate representation of the original glyph shape. Multi-channel distance fields (MSDFs) combat this by reserving two additional channels of information to store distances to multiple edges, but even this technique cannot maintain accuracy when glyph shape complexity exceeds a certain threshold. Both SDF and MSDF representations increase the burden of glyph cache maintenance, due to the signed-distance conversion requirement.

More recently, it has become more popular to rasterize glyphs directly on the GPU from Bézier curve data. This approach is attractive for a few reasons:

- A glyph cache is optional, and if maintained, cache entries are never invalidated due to changes in font size, DPI, or glyph transforms.
- As with SVG images, Bézier rasterization is size and orientation-independent, and so is particularly suited for text rasterized in 3D.
- Compared to signed-distance representations, Bézier rasterization preserves all sharp corners and edges at arbitrary levels of magnification.

Bézier curve rasterization is the most versatile of the methods described so far, but pose several implementation challenges. Existing methods operate using the similar principles as CPU-based Bézier rasterizers. Namely, they operate by computing winding numbers for sub-pixel samples. Unlike CPU-based rasterizers, GPU-based rasterizers cannot leverage double-precision arithmetic units portably, nor is excessive Bézier curve subdivision practical on the GPU, so care is needed to properly handle singular events (e.g. curves starting from a sample point, curves tangent to a ray, curves passing through a sample point, etc.). Failures in correct winding number computation due to precision can cause a wide swing in the final result, with catastrophic errors in the case of low sample counts.

1.1. Motivation

The Scanline Sweeper was designed to retain all the benefits of GPU-based Bézier curve rasterization, but simplify implementation significantly by reducing the numerical-exactness burden.

Similarly to existing Bézier curve rasterizers, the Scanline Sweeper operates on glyph curves directly on the GPU, but introduces a different paradigm for pixel coverage estimation. Instead of explicitly computing winding numbers for separate sub-pixel samples as a set of discrete events, the Scanline Sweeper integrates signed areas swept by each curve intersecting the scanline of interest. By remaining in a continuous domain as opposed to measuring discrete events, the sweeper sidesteps the issue of singular-event handling and numerical failure altogether.

Implementation difficulty is, of course, subjective, so I would like to remind the reader that the simplicity claimed in the approach described here is strictly in my own estimation. As this is the only algorithm I have implemented (in this family of GPU Bézier curve rasterizers), I also can't make claims to how the performance of my approach compares to other approaches, except to say that at least on paper, I believe the Scanline Sweeper should certainly no worse, and likely better under various conditions.

2. Related Work

The related work cited here is restricted to algorithms that permit a GPU-oriented implementation, specifically targeting games and other applications with similar interactivity requirements. In particular, tessellation-based solutions which adaptively triangulate glyphs are not considered, in part be-

cause they impose impractical memory or runtime costs for games.

2.1. Texture fonts

For traditional texture fonts, [Rougier 2013][1] provides a useful survey of best-practices, including glyph-packing, texture quad placement, and gamma correction. In particular, Rougier suggests the usage of FreeType or STB's TrueType rasterizer in conjunction with Harfbuzz or Pango to handle accurate glyph positions present in the `GPOS` subtable of an `OpenType` font file. Rougier argues that ahead-of-time rasterization is to be preferred as analytical-methods cannot reasonably respect font hinting and sub-pixel placement. These recommendations bore more significant merit at the time (circa 2013) when low-resolution displays were far more prevalent, but may still be applicable for certain applications.

Author's note: This paragraph was added 2026-07-04, prior to which I was unaware of how STB's TrueType rasterizer functioned. STB's rasterizer, in particular, bears a fair bit of resemblance to the Scanline Sweeper, although this algorithm makes several specific optimizations and changes for more optimal deployment on a GPU. Structurally, the quadrature method is similar.

2.2. Signed-distance fonts

As an extension to texture fonts, [Green 2007][2] uses a signed-distance representation to leverage hardware texture filtering units to anti-alias glyphs under minification, magnification, and perspective projections. The signed-distances sampled in the shader are used not only to fill in glyph outlines, but also to support various effects such as edge softening, outlining, and drop-shadows.

[Chlumský 2015][3] extends the previous SDF approach by storing signed distances to multiple edges in a multi-component SDF (MSDF). The MSDF is constructed such that the median of the three sampled distances in a shader recovers the closest edge. This method resolves situations of ambiguity in the single-channel SDF representation where the filtered distance is positive in regions of negative space near corners.

2.3. GPU Bézier Rasterizers

[Dobbie 2016][4] introduces a *vector texture* which stores glyph curve data in a texture and performs root-finding to evaluate winding numbers for horizontal rays. To combat precision issues when horizontal rays intersect Bézier curves at glancing angles, [Dobbie 2016] suggests leveraging more rays per pixel and averaging the results, effectively combating precision issues with compute.

Like [Dobbie 2016], [Lengyel 2017][5] leverages root-finding to render glyphs on the GPU from curve data embedded in textures. However, [Lengyel 2017] addresses precision problems by classifying Bézier curves into separate equivalence classes. For each class, the positive and negative roots are considered differently to eliminate numerical failures endemic to the more straightforward root-finding approach. For example, several classes of curves may only increment the winding number, while other curves may only decrement the winding number, and so on. Anti-aliasing is proposed by super-sampling within each pixel in a cross-shape to reuse quadratic and linear coefficients in the quadratic solver, but the technique generalizes in the sense that the implementer is free to use any sample points needed.

Practitioners in the industry have implemented variations of this idea. Some useful references include [Osorio 2025][6] as well as [Lague 2024][7]. Both citations mention difficulties encountered in tackling problems pertaining to floating-point precision, which while solvable with careful implementation as demonstrated by [Lengyel 2017], is in large part what inspired this work.

[Ellis et. al. 2019][8] propose an analytical quadrature method to estimate pixel coverage, similar to this work. [*Author's note: this paper was communicated to the author after the initial publication of this paper.*] Area is estimated per-curve under a weak-perspective approximation which permits quadrature via x-aligned trapezoidal segments. To compute the sampling window, the authors derive a UV transformation that approximates rotation with shear in such a way that horizontal alignment is maintained. The authors also prescribe a method by which each trapezoid can be clipped to the sampled window. Compared to [Ellis et. al. 2019], the Scanline Sweeper avoids the need to clip against left-edges by relying on the Jordan Curve Theorem. Curve monotonicity is also leveraged in this algorithm to accelerate root finding and avoid needing to locate curve critical points.

3. Scanline Sweeper

In the Scanline Sweeper, winding numbers do not play a role in the computation. Instead, each curve contributes a signed coverage estimate additively. Anti-aliasing is an implicit part of the algorithm, and as opposed to integrating the result across separate discrete sample points, the sweeper produces an anti-aliased result for a single rectangular window of the glyph's coordinate space. Linear combinations of these intermediate results may be combined to approximate a coverage estimate for more sophisticated footprints, but are not necessary for the algorithm to work.

In this section, I'll describe the algorithm in its most basic form to facilitate a perfectly serviceable initial implementation. Section 4 will touch on various ways to improve the performance and quality of the algorithm with only a modest amount of additional effort.

3.1. Glyph Preprocessing

As a preprocessing step, all cubic Bézier curves of each rendered glyph must be approximated as sets of directed piecewise quadratic Béziers according to some optimality metric not prescribed by this paper, but an excellent technique for performing this decomposition while preserving C_1 continuity is described in [Truong et. al. 2020][9]. Linear segments (1st order Bézier curves) should also be promoted to quadratic Béziers by adding a control point at the midpoint of the segment endpoints. The exception here are horizontal linear segments that can be removed entirely, as they will not participate in the algorithm.

Next, all quadratic Bézier curves must be subdivided up to 2 times such that each sub-curve of the decomposition is both y-monotonic and x-monotonic over the unit interval. This property inflates storage requirements, but has a marginal effect on many fonts, since the majority of the curves in typical fonts are already monotonic. With the monotonic property, bounds checks can be performed against an axis-aligned bounding box defined by the first and last control point of each curve. This decomposition also ensures that for any curve-line intersection, there is at most one root within the curve's domain.

As an optional step, glyph contours may be decomposed to remove contour overlaps. This step isn't strictly necessary, but improves shader performance and is a practical step to perform ahead-of-time. Section 4.3 describes the adjustment needed if contour overlaps are permitted in contetn.

Because font glyphs are typically defined on a 1024- or 2048-per-em resolution coordinate grid (henceforth referred to as *em-space*), glyph control points should be uploaded to the GPU as `binary16` IEEE floats which can be cheaply unpacked to 32-bit precision in the shader. OpenType font control points are typically snapped directly to the font grid itself, with implicit points occasionally lying directly between grid lines. As we aren't retrieving glyph curves through any texture sampling routine, there is no benefit to storing control points in a texture, and leveraging some sort of storage buffer is recommended for simplicity.

3.2. Pixel Shading

Pixel shading can be done in either a compute shader or a pixel shader. The algorithm requires no special extensions or hardware capabilities, although `SM 6.0` wave intrinsics may be used to reduce `VGPR` pressure by enforcing that every pixel in a wave is accessing the same curve.

Shading begins by computing the size and offset of a rectangular window in em-space corresponding with the pixel. With an orthographic projection, this window can be computed at draw time since the ratio of pixel width and height to a unit in em-space is known. In the general case, this window can be cheaply estimated with the `fwidth` instrinsic. It's possible to compute a coverage estimate of sophisticated footprints by leveraging multiple windows, but the core technique operates on a rectangular coverage primitive, so we'll continue to focus on that in this section. The important point here is that the footprint is dynamically varying per-pixel, as this is what permits anti-aliased glyph shading under arbitrary transformations.

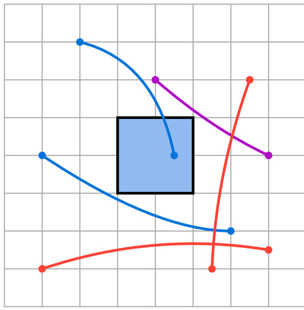


Figure 2: The blue Bézier curves would be considered by the algorithm and result in a non-zero signed area contribution. The red curves would be culled by comparing the outer control points with the scanline locations and right window edge. The purple curve would be considered but would ultimately have a zero contribution because its intersections with the scanlines would be constrained to the window boundary.

Without loss of generality, this algorithm assumes we are operating with horizontal scanlines and performing curve sweeps to the right. This orientation is arbitrary, and the implementer may wish to change this convention if there is a material advantage in doing so.

For each rectangular window, each glyph curve is first considered with respect to the upper and lower bounds of the window. Only curves that intersect the scanline are considered. Next, the curve is conceptually swept to the right. Curves that don't interact with the window are, again, excluded from consideration. As an optimization, these bounds checks need only consider the first and last control point of each curve, due to the monotonic criterion. Every considered curve is thus:

- … intersecting one or both horizontal scanlines, and
- … at least partially to the left of the right window boundary

as depicted in figure Figure 2.

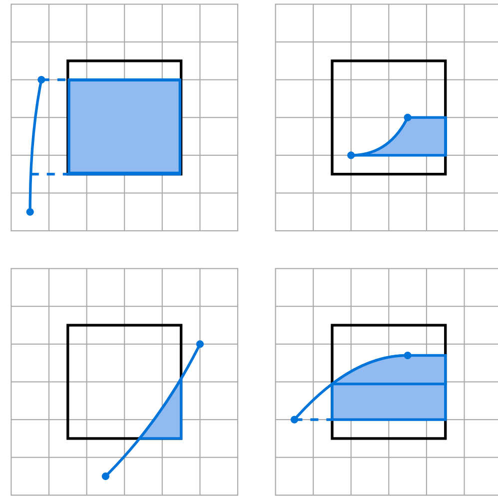


Figure 3: The blue regions shown depict the areas estimated by the sweep operation for each curve. While it may appear that there are many configurations to handle, in reality, all swept regions can be decomposed into a trapezoid-like shape and a rectangle. In some scenarios, the trapezoid may degenerate into a triangle or be missing altogether. In other scenarios, the rectangle may be absent.

Consider the area swept by moving the curve to the right and the *intersection* of this area with the window. The next step is to numerically compute the signed area of this critical region. This quadrature operation proceeds by first decomposing the curve into up to two piecewise regions. One region lies to the left of the window, and its area contribution is simply a rectangular area. The other region lies within the window itself, and its area can be approximated with a trapezoid in the simple case (ref. Figure 3 for a few pictorial representations). The sign of the area is determined by the direction of the curve. Upwards-moving curves correspond to positive areas, and downwards-moving curves correspond to negative areas.

To simplify the implementation, an initial version of this algorithm could simply compute four Bézier line intersections for each line enclosing the window. These intersections may lay outside of the window, but by clamping the results to the pixel window, the area estimation will simply absorb zeros for regions that don't occupy any physical area. This method will also cause triangular regions to be naturally integrated as trapezoidal-like regions with 0 for one of its base lengths.

The HLSL monotonic root-finding implementation is provided in Section 8.2. The HLSL sweep-evaluation implementation is provided in Section 8.3.

The final coverage ratio is computed by summing the signed area contributions of all considered

curves and then dividing by the total window area. This ratio is then modulated by gamma (configured either per font or per glyph) and used as the pixel's alpha value.

3.3. Integral Approximation

When estimating the amount by which a Bézier curve sweep overlaps a window, it isn't practical to compute the integral analytically since curves are parametric paths. There are quite a number of options to tackle this, so we'll cover just a few:

- Linear Approximation
- Linear-y Approximation
- Subdivision

Suppose we have computed two values of t (t_0 and t_1) for which a Bézier curve $B(t)$ either intersects any of the sides of a window or reaches a control point. Furthermore, suppose that $B(t)$ is not a vertical line segment, such that these intersections must occur on two distinct sides of the window (if both points do indeed happen to be intersections). Vertical segments can be easily handled as a separate case by computing its swept area as a rectangle. Then, the area we wish to numerically estimate is the blue area traced by contour shown in Figure 4

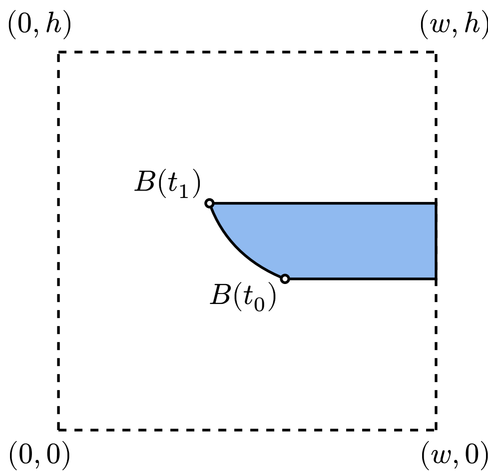


Figure 4: The dashed boundary is a depiction of the window under consideration and the curve from $B(t_0)$ to $B(t_1)$ is the curve being swept. This particular curve happens to lie entirely within the window.

Note that the following discussion can proceed regardless of which segments $B(t_0)$ and $B(t_1)$ happen to fall on. Furthermore, this discussion generalizes to the case where one or both points are in the window interior.

By far, the simplest approximation is to presume that the curve $B(t)$ is approximately linear within the window. This approximation yields the trapezoidal estimate for the area as:

$$h_B = B_y(t_1) - B_y(t_0)$$

$$A_{\text{trapezoid}} = \frac{h_B(2w - B_x(t_0) - B_x(t_1))}{2}$$

For many applications, this choice is perfectly suitable. While the estimate may not be objectively accurate, there will be regions where the estimate is too high, and other regions where the estimate is too low, and on the whole, the result won't possess noticeable issues except under heavy minification. Under heavy minification though, it could be argued that more traditional texture font approaches are more suitable, since these approaches can support hinting. Figure 5 shows a sample edge anti-aliased with this method. Note that unlike rasterizers that compute winding numbers for a set of discrete events, the possible coverage estimate values cover the entire unit interval.

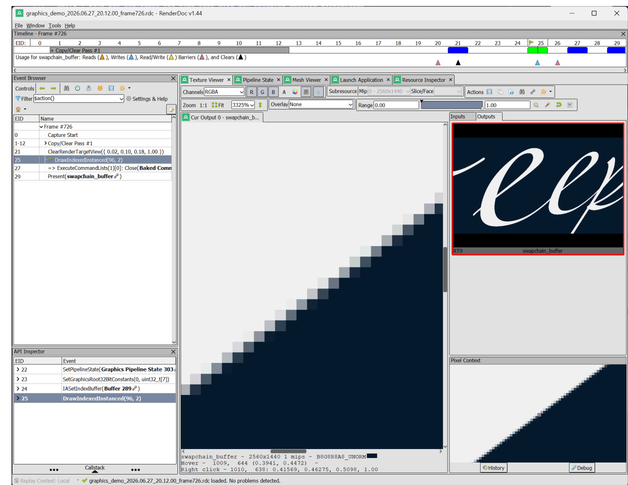


Figure 5: A single antialiased edge from an 'e' glyph rendered with modest perspective.

If more accuracy is desired, a stricter approximation can be made by presuming that $B_y(t)$ (but not $B_x(t)$) varies linearly across the window. In such a case, the parametric Bézier curve can now be represented explicitly as a function of y, which then permits standard integration through a change of variables. This approximation can be thought of as a refinement between a first-order correction and a true second-order evaluation.

$$B_y(t; t_0 \leq t \leq t_1) \approx B_y(t_0) + \frac{t - t_0}{t_1 - t_0} B_y(t_1)$$
$$t \approx (t_1 - t_0) \frac{B_y(t) - B_y(t_0)}{B_y(t_1)}$$
$$\Rightarrow$$
$$\int_{t_0}^{t_1} B_x(t) dt \approx$$
$$\frac{t_1 - t_0}{B_y(t_1)} \int_{B_y(t_0)}^{B_y(t_1)} B_x\left(B_y^{-1}(t)\right) dy$$

Substituting this approximation for t into $B_x(t)$ yields the approximate $B_x(B_y)$ approximation which can be integrated with $B_y(t_0)$ and $B_y(t_1)$ as the limits of integration. Unfortunately, this computation can't be recommended because the evaluation requires too much compute relative to the increase in accuracy.

A more practical method to refine the approximation is with subdivision (ref. Figure 6). This is a far more pragmatic endeavor because the refinement amount can be easily scaled to two or more additional points of refinement without any code changes. Furthermore, because the quadratic Bézier curves were preprocessed to be monotonic, it's likely that the curves are *nearly* linear in most practical scenarios.

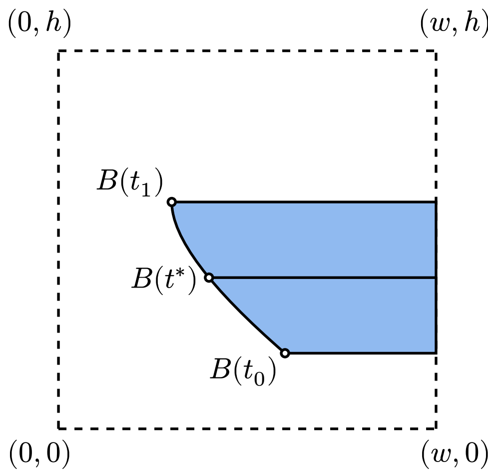


Figure 6: Any trapezoid can be subdivided one or more times to refine the quadrature. In practice, not many refinements are needed because curves were already preprocessed to be monotonic. In most cases, no refinement is needed at all.

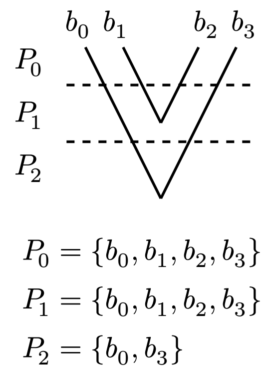
It's important to keep in mind with these approximations that the coverage estimate pertains to the *entire window* subject to analysis. Because we are not computing a winding number for a specific sample point, we are afforded some leeway when it comes to estimating this quantity. Over and under-estimates are not catastrophic by any means – pixels that are fully in the glyph interior will remain fully shaded, and pixels that are fully outside the glyph interior will be fully unshaded.

4. EXTENSIONS

This section describes various ways to improve the performance or flexibility of the algorithm, but nothing mentioned here is strictly necessary for an initial implementation. The subsections describing each extension are listed in subjective priority order. Optimizations should be carefully weighed against content requirements, because as with most optimizations, constant factors play an important role. Optimizations that benefit wildly complex glyphs may not be profitable for simpler glyphs, so my recommendation is to measure and make do with the simplest implementation first, before adding complexity. To that end this section also contains more "editorial" suggestions that may or may not pertain to you, the reader, depending on usage.

4.1. Curve acceleration structure

The SCANLINE SWEEPER benefits from any sort of acceleration structure that allows the shader to avoid loading curves that do not interact with the pixel being shaded. On all hardware tested, the principal bottleneck is not ALU but actually memory, since the computation needed to discard a spatially irrelevant curve is miniscule. As such, the goal for most optimizations in this algorithm should be to *load less data*.



$$P_0 = \{b_0, b_1, b_2, b_3\}$$
$$P_1 = \{b_0, b_1, b_2, b_3\}$$
$$P_2 = \{b_0, b_3\}$$

Figure 7: The simplest curve acceleration structure just partitions the glyph into horizontal stripes, and each partition stores all overlapping curves without any indirection.

When implementing an acceleration structure, it is important not to add any indirection to the shader. That is, an acceleration scheme that stores curve indices for different regions to the glyph will perform

strictly worse than the completely unaccelerated storage scheme. If the glyph is to be partitioned according to some scheme, it's suggested that each partition loaded contains the entire set of curves pertaining to the partition as seen in Figure 7.

This necessarily inflates the data size, but this is easy to manage because glyph curve data can be suballocated on an as-needed basis. On the topic of memory, it's possible to compress the data set by exploiting the fact that many control points are shared between curves. One expression of this compression is to reserve two `binary16` floats per curve stored in each partition. In this scheme, the first control point is implicitly the last control point encountered in the stream, and a sentinel value is used to reset this value. I found that this optimization saved roughly 25% of the memory used, but nominally increased the cost of the shader by ~5%, which may or may not be a good tradeoff depending on the application.

4.2. Compute Shader Implementation

When implemented in a pixel shader, each pixel loads curve data into non-uniform registers. This could be scalarized by looping over a superset of all curves that must be considered in a wave, similar to light-loop scalarization. However, this change isn't always profitable because processing each curve didn't require many registers to begin with. Furthermore, such scalarization is at odds with a partitioning scheme in scenarios where a wave straddles a partition boundary.

A more fruitful endeavor is to rasterize glyphs in a compute shader, assigning thread groups to scanlines. This way, the data loaded in a wave is minimized to the set of curves that overlap the scanline partition (assuming some acceleration scheme is used). In this scenario, it's impossible for a wave to straddle a partition boundary, and scalarization is far more natural.

In addition, a compute shader can process curves within each partition cooperatively. For example, each thread in the wave can be assigned to separate curves to compute any values that pertain to the entire scanline, before transitioning to a separate loop for each pixel rasterized. Similarly, the second degree coefficients of the quadratic Bézier expansion could be computed cooperatively for each curve in parallel and shared either through wave access intrinsics, or via `groupshared` memory.

As an aside, I do not recommend optimizing this shader by attempting to identify pixels on the glyph boundary and treating these pixels separately from pixels entirely on the glyph interior or exterior. While this may seem like a useful optimization to pursue from an algorithmic-complexity standpoint, sweeping a curve is so cheap arithmetically that elaborate schemes to process glyph boundaries are more liable to be a pessimization to the total runtime. This is in line with conventional wisdom for graphics programmers, where the "superficially more efficient" but simultaneously "more divergent" strategy is often not the correct approach due to constant factors, synchronization costs, pipeline context rolls, and other sources of inefficiency.

4.3. Permitting Contour Overlaps

The algorithm as described will not function correctly if there are overlaps in the glyph contours because the area ratio denominator will exceed 1 for windows that intersect the overlap. This restriction can be relaxed by simply considering each contour separately in the shader. That is, instead of having a global coverage estimate, the shader can simply compute the coverage estimate for each contour separately, and then average the results at the end. This changes the storage scheme somewhat, but otherwise, the algorithm is unchanged.

For a game that can ship content with assets that undergo a processing stage before final packaging, I would suggest simply removing the contour overlaps ahead-of-time. Many font processing tools handle this without too much issue, and the operation can finish in seconds for most fonts tested. Note that glyph contours in `TrueType` fonts may advertise the `OVERLAP_SIMPLE` flag if overlaps are present, but it may or may not be wise to rely on this flag since its possible the flag is incorrectly unset.

For applications that prefer to do minimal font preprocessing before rasterization, storing contours separately and adding an additional outer loop is an easy adjustment that adds flexibility without increasing runtime costs too much.

4.4. Accurate Footprint Assembly

If each pixel is associated with a single rectangular footprint and coverage estimate, the outcome is a simple box-filtered result. Conceptually, this will result in blurring in cases where the window overlaps the exterior of the true footprint. Conversely,

aliasing is expected in cases where the window is smaller than the true footprint.

If more accurate anti-aliasing is needed, we have many options at our disposal. The simplest option is to continue to simply rely on `fwidth` to derive a rectangular footprint and do nothing else in particular, with the expectation that results will look the best when viewed orthographically, and look passable in other scenarios with blurring or aliasing depending on camera pose. This option, while sounding unappealing, is actually fairly passable with this technique. Because the algorithm doesn't compute binary coverage results for sets of discrete samples, blurring and aliasing effects aren't nearly as noticeable, and the question of *necessity* should be posed first. Would the player appreciate sharp edges in what is likely unreadable text anyways? Or would they tolerate slight blurriness for text at extreme glancing angles in exchange for better performance in all scenarios?

If in spite of that, more accurate filtering is still desired, `ddx(uv)` and `ddy(uv)` can be used to construct vectors in em-space that constitute the major and minor axes of the ellipse of anisotropy, at which point any number of methods can be employed to decompose this ellipse into a set of rectangular windows. A similar technique is employed by [Mavridis and Papaioannou 2011][10] to decompose elliptical footprints, and this method can be easily extended to axis-aligned rectangular windows instead.

If the "weak-perspective approximation"[1] is acceptable (likely the case in all practical scenarios), this method is also compatible with applying shear transforms to Bézier control points, similar to the domain transformation described by [Ellis et. al.][8]. I have not explored this direction myself, but think it's a useful extension to consider, with a nominal 2×2 QR decomposition and shear transformation cost per control point. The main downside of this approach is that curves do not retain monotonicity after shear transforms in the general case, so splitting curves at critical points will need to happen in the shader itself as opposed to at font processing time. In other words, we can gain some anti-aliasing accuracy at the expense of some runtime performance.

Finally, some combination of all the above techniques are possible. The Scanline Sweeper algorithm is ultimately concerned with demonstrating one method to estimate an unweighted box-filter coverage value in em-space, which on its own does not stipulate any particular footprint assembly. Practitioners should way the costs of footprint assembly against content requirements.

5. Limitations

Even with the extensions prescribed, there are some limitations to this technique that are worth considering before implementation and deployment.

5.1. Small Font Rendering

In the case of small font rendering, some clarity is lost compared to rasterizers that can take advantage of the hinting interpreter on the CPU. This is best seen in Figure 8, where various features that would normally snap to pixel boundaries like vertical stems end up straddling the boundaries instead. This is likely not a deal-breaker on many modern displays, but can pose an issue for deployments that render on low-resolution displays.

Figure 8: Without hinting, features like stems cannot be aligned on pixel grid boundaries

As a reminder, a rasterizer that respects hinting cannot produce results that are correctly antialiased once the texture-sampling quad is translated off the pixel grid regardless, so this limitation should be weighed against the need for properly hinted statically placed text.

5.2. Preprocessing Requirements

As mentioned in Section 3.1, some modest Bézier curve processing is needed to properly feed the Scanline Sweeper shader. All of the preprocessing requirements are fairly efficient, and an entire font file can be parsed and processed on one thread within a few frames if needed. The exception here would be contour overlap removal. If your appli-

[1] In general, squares mapped with a perspective projection are mapped to a trapezoid. The weak-perspective approximation approximates the projected regions as parallelograms.

cation cannot process font files ahead-of-time, its recommended that the shader operate on separate contours independently with a small increase in runtime cost, as described in Section 4.3.

5.3. Performance Considerations

A more detailed breakdown of some performance measurements are available in the next section (Section 6), but this technique (and other similar techniques) will be more expensive relative to a baseline implementation that simply samples a texture rasterized on the CPU. UI renderers already benefit from some form of a compositor that can cache partial UI layouts in rendered textures, but if no compositor is available, this technique may need to be supplemented with a glyph cache instead. In fact, both a glyph cache and a UI compositor can ship along with this solution in a complementary fashion.

6. EVALUATION

The tests here are not exhaustive in any way, but are here simply to give a representation of expected memory and runtime characteristics you will need to extrapolate for your content. For this test, I evaluated only two separate fonts:

- Geist as a base case font (Figure 9).
- Miama as a difficult font with 4 times as many curves per glyph as the other fonts (Figure 10).

Figure 9: Pangrams rendered in the Geist font at 1440p.

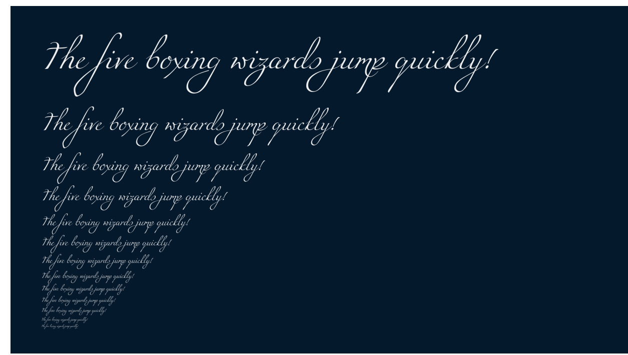


Figure 10: Pangrams rendered in the Miama font at 1440p.

Font	RTX 2060	RTX 4090	9070 XT
Geist	240 µs	50 µs	53 µs
Miama	370 µs	70 µs	87 µs

Table 1: Timings recorded for Figure 9 and Figure 10 at 1440p with stable power states. NVIDIA timings are reported with 10 µs granularity.

For each font, I rendered a pangram at varying sizes across a 1440p display and captured timings on an NVIDIA RTX 2060 budget card, an NVIDIA RTX 4090, and an AMD 9070 XT with stable power states active. The shader profiled was a pixel shader with a basic horizontal partitioning scheme. Timings are reported in Table 1 with microsecond granularity, but note that NVIDIA's Nsight Graphics profiler snaps to 10 microsecond boundaries.

While I have not performed the same test with ray-stabbing Bézier rasterizers, my intuition is that performance should be similar, based on the following assumptions:

- The evaluation of a curve's coverage estimate is similar to evaluating the winding number of roughly four sample points in a cross-pattern, since the underlying math is identical.
- Accumulating a coverage area is similar to incrementing and decrementing winding point values.

Furthermore, I would expect that SCANLINE SWEEPER performance may outpace other solutions in a good implementation where curves are loaded and cooperatively processed in a wave-uniform fashion, as described in Section 4.2, with the caveat being that many these optimizations may not be unique to this algorithm. The other point to consider is that the SCANLINE SWEEPER effectively takes many implicit samples with a single evaluation, so there is a performance crossover point where this algorithm likely performs better when compared to a ray-stabbing implementation when the sample count exceeds some threshold.

Font	VRAM bytes
Geist	17.3 KiB
Miama	43.4 KiB

Table 2: Video memory needed to render all ASCII characters of each associated font.

Table 2 shows video memory usage for the same fonts with the entire ASCII character set uploaded. In a general production setting, glyph data could be suballocated and uploaded on demand, which may be necessary for languages with many more glyphs. Even then, storing 100 times as many glyphs would still impose only a modest footprint.

7. Conclusion

In this paper, I've presented what I believe to be a different take on Bézier curve rasterization on the GPU. The primary goal was to derive a technique that avoided needing to track discrete events to estimate coverage. Instead, by staying in a continuous domain, the SCANLINE SWEEPER sidesteps issues pertaining to singular events while offering high-quality anti-aliasing with relatively little effort.

Extensions to the technique are fairly natural, since linear combinations of rectangular coverage estimates can be quickly assembled to form more exotic footprints.

Future work in this space may include but is not limited to:

- Adaptive subdivision that can subdivide curves only if the curvature is high relative to the window size.
- Basic hinting to align vertical line segments or other glyph features to pixel boundaries.
- Alternative acceleration structures that either provide faster glyph curve access or tighter memory requirements.
- Optimizations that leverage `FP16` units on modern GPUs in cases where full 32-bit precision is not needed.

Bibliography

[1] N. P. Rougier, "Higher Quality 2D Text Rendering," *Journal of Computer Graphics Techniques (JCGT)*, vol. 2, no. 1, pp. 50–64, Apr. 2013.

[2] C. Green, "Improved Alpha-Tested Magnification for Vector Textures and Special Effects," Aug. 05, 2007, *Valve Corporation*. doi: 10.1145/1281500.1281665.

[3] V. Chlumský, "Shape Decomposition for Multi-channel Distance Fields," Master's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2015.

[4] W. Dobbie, "GPU text rendering with vector textures." [Online]. Available: https://wdobbie.com/post/gpu-text-rendering-with-vector-textures/

[5] E. Lengyel, "GPU-Centered Font Rendering Directly from Glyph Outlines," *Journal of Computer Graphics Techniques (JCGT)*, vol. 6, no. 2, pp. 31–47, Jun. 2017, [Online]. Available: https://jcgt.org/published/0006/02/02/

[6] R. L. Osorio, "Rendering Crispy Text on the GPU." Accessed: Jun. 24, 2026. [Online]. Available: https://osor.io/text

[7] S. Lague, *Coding Adventure: Rendering Text*, (Apr. 13, 2024). [Online Video]. Available: https://www.youtube.com/watch?v=SO83KQuuZvg

[8] A. Ellis, W. Hunt, and J. Hart, "Real-Time Analytic Antialiased Text for 3-D Environments," *Computer Graphics Forum*, vol. 38, no. 8, pp. 23–32, Nov. 2019, [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13757

[9] N. Truong, C. Yuksel, and L. Seiler, "Quadratic Approximation of Cubic Curves," *Proc. ACM Comput. Graph. Interact. Tech.*, no. 16, pp. 1–17, Aug. 2020, doi: 10.1145/3406178.

[10] P. Mavridis and G. Papaioannou, "High quality elliptical texture filtering on GPU," *I3D '11 Symposium on Interactive 3D Graphics and Games*, pp. 23–30, Feb. 2011, doi: 10.1145/1944745.1944749.

[11] T. W. Sederberg, *Computer Aided Geometric Design*. BYU ScholarsArchive, 2012. [Online]. Available: https://scholarsarchive.byu.edu/cgi/viewcontent.cgi?article=1000&context=facpub

[12] F. Goualard, "The Ins and Outs of Solving Quadratic Equations with Floating-Point Arithmetic," 2023.

8. Appendix

Code listings in this section are compiled with `DXC` with `HLSL2021` as the language mode. While not required for code listings presented here, the shaders benchmarked for this paper were compiled with `SM6.7`. Absent from these code listings are code snippets needed to load glyph data from buffers or convert pixel texture coordinates to em-space, both of which are considered details that will vary for each implementer.

For a general background on the mathematics needed to manipulate Bézier curves numerically, I suggest consulting [Sederberg 2012][11] as a comprehensive reference.

All source code is subject to the terms of the Mozilla Public License, v. 2.0. This license can be obtained at https://mozilla.org/MPL/2.0/.

8.1. Quadratic Bézier form

The code listings here relies on the following re-expression of a quadratic Bézier curve evaluation and the corresponding gradient:

$$
\begin{aligned}
B(t) &= (1-t)((1-t)p_0 + tp_1) + t((1-t)p_1 + tp_2) \\
&= (p_0 + p_2 - 2p_1)t^2 + 2(p_1 - p_0)t + p_0 \\
\frac{dB(t)}{dt} &= 2(p_0 + p_2 - 2p_1)t + 2(p_1 - p_0)
\end{aligned}
$$

The code listings also rely on this simple helper function for evaluating a Bézier curve for a given parameter t:

```
// p0 and p2 always refer to the curve endpoints.
// p0, p1, and p2 are the curve control points in order of increasing t.
float2 evaluate_bezier(float2 p0, float2 p1, float2 p2, float t)
{
  float2 a = lerp(p0, p1, t);
  float2 b = lerp(p1, p2, t);
  return lerp(a, b, t);
}
```

8.2. Monotonic Root Finding

This routine computes the parameter of intersection for a specific dimension of a Bézier curve. This can compute vertical or horizontal intersections. A general quadratic root-finder needs to apply extraodinary care to be numerically robust (ref [12] for an exhaustive treatment of numerically robust quadratic root-finding). However, we can sidestep many of the issues that plague a general solver:

- Our control points are constrained to the vicinity of the em-square, so discriminant overflow isn't possible.
- Errors due to discriminant underflow are absorbed in the final coverage estimate, since the alpha value will be quantized to 8 bits anyways.
- We could use the *citardauq* form of the quadratic formula to avoid catastrophic cancellation when $-b$ and $\sqrt{b^2 - 4ac}$ have opposing signs, but this isn't necessary due to the same quantization.
- Because we only deal with monotonic curves, we never need to solve for both roots. As such, approaches to improve precision by computing the precise root and leveraging Vieta's formula to derive the other root aren't useful.

```
// Preconditions:
// - qa is the second-degree coefficient.
// - c0, c1, and c2 are the coordinate values of the first,
//   second, and third control points respectively.
// - target is the desired line of intersection.
//
// This routine assumes that c0 <= target <= c2. That is, the
// caller has done bounds checks that guarantee that this
```

```
// routine is needed already.
//
// This routine also assumes monotonicity:
// c0 <= c1 <= c2.
//
// Returns the value of t where the intersection occurs.
float intersect_monotonic(
  float qa,
  float c0,
  float c1,
  float c2,
  float target)
{
  if (abs(qa) < 1e-3f)
  {
    // Approximately linear case. Threshold can be adjusted
    // as needed.
    return (target - c0) / (c2 - c0);
  }

  // First-degree coefficient.
  float qb = mad(2.f, c1, -2.f * c0);

  float qc = c0 - target;

  float d = mad(qb, qb, -4.f * qa * qc);

  // Clamping d above 0 is OK because of our established
  // precondition that c0 <= target <= c2.
  float sqrt_d = d < 0.f ? 0.f : sqrt(d);

  float inv_2a = 0.5f / qa;

  // The sign dictates whether we need the positive or
  // negative root. Because of the monotonic assumption,
  // only one root ever needs consideration for each
  // curve.
  return mad(-qb, inv_2a, sign(c2 - c0) * sqrt_d * inv_2a);
}
```

8.3. *Scanline Sweep Implementation*

This code listing provides the core routine of the algorithm. The caller is expected to provide a window size and offset, as well as the control points of a monotonic quadratic Bézier curve. The function sweeps the curve to the right and estimates the area covered by the sweep constrained to the window. The approximation used is a trapezoidal approximation, but extensions to this are possible, as described in Section 3.3.

```
// Preconditions:
// - size is the size of the window to be considered
// - offset is the position of the lower-left window corner
// - p0, p1, and p2 are control points of a monotonic Bézier curve
//
// The curve must be monotonic, such that all(p0 <= p1) and
// all(p1 <= p2) are true.
//
// Returns the estimated covered area of the window if the curve
// is swept to the right. This is a signed quantity that is
// positive for upwards moving curves, and negative for downwards
// moving curves.
```

```
float scanline_sweep(
  float2 size,
  float2 offset,
  float2 p0,
  float2 p1,
  float2 p2)
{
  // Discard curves above or below the scanline.
  if (max(p0.y, p2.y) <= offset.y || min(p0.y, p2.y) >= offset.y + size.y)
  {
    return 0.f;
  }

  float2 delta = p2 - p0;

  // Shift all control points to a coordinate system with the
  // window at the origin.
  p0 -= offset;
  p1 -= offset;
  p2 -= offset;

  // Fast path for strictly vertical segments, common in many fonts.
  if (p0.x == p1.x && p0.x == p2.x)
  {
    if (p0.x >= size.x)
    {
      // Segment is to the right of the window. Nothing to do.
      return 0.f;
    }

    float top = min(max(p0.y, p2.y), size.y);
    float bottom = max(min(p0.y, p2.y), 0.f);

    float h = top - bottom;
    float b = min(size.x, size.x - p0.x);

    // Signed area of the swept rectangle.
    return sign(delta.y) * b * h;
  }

  // qa is the second-degree coefficient for the y-coordinate
  // quadratic.
  float qa = mad(-2.f, p1.y, p0.y + p2.y);
  float bt = intersect_monotonic(qa, p0.y, p1.y, p2.y, 0.f);
  float tt = intersect_monotonic(qa, p0.y, p1.y, p2.y, size.y);

  // v_min_t and v_max_t are the crossings where the curve enters
  // and exits the scanline.
  float v_min_t = delta.y > 0.f ? bt : tt;
  float v_max_t = delta.y > 0.f ? tt : bt;

  float2 v_min = evaluate_bezier(p0, p1, p2, saturate(v_min_t));
  float2 v_max = evaluate_bezier(p0, p1, p2, saturate(v_max_t));

  if (max(v_min.x, v_max.x) <= 0.f)
  {
    // Fast path for curves entirely to the left of the window
    // within the scanline. Note that the area sign is
    // incorporated in the result.
```

```
  return (v_max.y - v_min.y) * size.x;
}

if (min(v_min.x, v_max.x) >= size.x)
{
  // The curve is entirely to the right of the window within
  // the scanline, so it can be ignored.
  return 0.f;
}

// Solve for roots along x.
qa = mad(-2.f, p1.x, p0.x + p2.x);

// As with v_min_t and v_max_t, we now need the values of t where
// the curve enters and exits the window moving horizontally.
float h_min_t;
float h_max_t;

// This check vector stores the following quantities in each component:
// - lower x bound
// - upper x bound
// - target value
// - parameter associated with the lower x bound (0 or 1)
//
// Packing the values in this way simplifies bounds checks and intersection
// testing, and the values depend on the direction the curve moves.
float4 h_check = delta.x > 0.f
  ? float4(p0.x, p2.x, 0.f, 0.f)
  : float4(p2.x, p0.x, size.x, 1.f);

if (h_check.x >= h_check.z)
{
  h_min_t = h_check.w;
}
else if (h_check.y <= h_check.z)
{
  h_min_t = 1.f - h_check.w;
}
else
{
  h_min_t = intersect_monotonic(qa, p0.x, p1.x, p2.x, h_check.z);
}

h_check.z = size.x - h_check.z;

if (h_check.x >= h_check.z)
{
  h_max_t = h_check.w;
}
else if (h_check.y <= h_check.z)
{
  h_max_t = 1.f - h_check.w;
}
else
{
  h_max_t = intersect_monotonic(qa, p0.x, p1.x, p2.x, h_check.z);
}

// Now, we can compute the values of t for which the curve enters
```

```
  // and leaves the window in any direction. Note that these values
  // are constrained to the unit interval, so it's ok if the curve
  // stops or ends within the window.
  float min_t = saturate(max(v_min_t, h_min_t));
  float max_t = saturate(min(v_max_t, h_max_t));

  // Evaluate the curve at new intersection points if needed based
  // on the newly constrained interval.
  float2 q0 = v_min_t >= h_min_t ? v_min : evaluate_bezier(p0, p1, p2, min_t);
  float2 q1 = v_max_t <= h_max_t ? v_max : evaluate_bezier(p0, p1, p2, max_t);

  float coverage = 0.f;

  if (min_t > 0.f && delta.x > 0.f)
  {
    // We enter the pixel from the left, so we need to integrate the
    // swept rectangle below the entry point.
    float h = delta.y > 0.f
      ? q0.y - max(0.f, p0.y)
      : min(size.y, p0.y) - q0.y;

    coverage = sign(delta.y) * h * size.x;
  }

  if (max_t < 1.f && delta.x < 0.f)
  {
    // We exit the pixel on the left side, so we need to integrate the
    // swept rectangle after the exit point.
    float h = delta.y > 0.f
      ? min(size.y, p2.y) - q1.y
      : q1.y - max(0.f, p2.y);

    coverage += sign(delta.y) * h * size.x;
  }

  // This implements the simple trapezoidal approximation for the
  // portion of the curve within the window.
  float h = q1.y - q0.y;

  // Sum of trapezoid bases divided by two. If q0.x or q1.x happen
  // to equal size.x, the trapezoidal area is effectively a triangle.
  float b = mad(-0.5f, q0.x + q1.x, size.x);

  coverage += b * h;

  // The caller is expected to accumulate this coverage for each curve,
  // and divide the final result by the window area.
  return coverage;
}
```