GPU Path Rendering

RAPH LEVIEN and ARMAN UGURAY, Google, USA

While rendering of 3D graphics primitives has advanced enormously, getting good performance from 2D vector graphics, particularly path rendering, has remained challenging, and most renderers in production still do a signficant amount of work on the CPU.

This paper presents a high performance path rendering running entirely as a pipeline of compute shaders, implemented in WebGPU so it can run portably on a wide range of GPU hardware. The algorithm is sparse and work efficient, based on a hierarchy of spatial subdivisions tuned to the characteristics of GPU hardware. Special attention is given to rendering quality, and in particular multisampling factor is not limited by the hardware. Numerical robustness is challenging, as the processing at different levels of the hierarchy must be consistent, and we provide a technique for resolving those ambiguities robustly. We conclude with a detailed performance comparison with existing techniques.

ACM Reference Format:
Raph Levien and Arman Uguray. 2023. GPU Path Rendering. 1, 1 (September 2023), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

GPU path rendering is awesome.

2 CONFLATION ARTIFACTS

Path renderer antialiasing falls into two general categories: analytical area calculation and multisampling. The former has good quality for certain applications, particularly text rendering, but has one serious downside, known as conflation artifacts.

Pathfinder (from which Vello derives) is based on analytical area rendering.

Note that there are two categories of conflation artifacts: those inherent to the rendering of a single path (which is what we'll be addressing), and those resulting from the compositing of multiple paths. The latter are interesting, for sure, but out of scope.

3 RELATED WORK

Loop-Blinn. Kilgard and Bolz. Nehab and Hoppe RAVG. Ganecin et al MPVG. Li et al scanline. Dokter et al. (Pathfinder? Spinel?)

4 GENERAL THEORY OF PATH RENDERING

Whether a particular point is filled or not filled is a function of the *winding number* of that point relative to the path. Rendering a path can be seen as determining the winding number of some very large number of points (a multiple of the pixel resolution, in the case of multisampling), then sampling down to produce the antialiased image. Calculating lots of winding numbers can be modeled as a brute force computation, but the problem also has lots of structure to allow it to be

Authors' address: Raph Levien; Arman Uguray, Google, San Francisco, CA, USA.

XXXX-XXXX/2023/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

computed sparsely. Given that we're evaluating this on a GPU, our goal is to exploit this sparseness while *also* preserving the parallelism in the problem.

We assume that our input paths are properly closed. This is a 2D equivalent of a 3D triangle mesh being watertight, but is a simpler property; basically it means that each start point of a line segment has a corresponding unique end point of another line segment with the same coordinates.

Given this assumption, winding numbers have a somewhat surprising and very powerful property: the winding number of a point relative to a path can be computed by adding up the crossings on *any* sequence of rays starting outside the path and ending at that point. No matter where you start or what intermediate point you choose, the computed winding number will be the same. Traditionally in software rendering techniques, winding numbers are computed by shooting rays from the left; since the image only changes at crossings with the path, that effectively gives a run length encoded image, which can be very efficient. But for parallel computation, choosing only scanline rays is limiting.

4.1 Spatial hierarchy

A more general theory of path rendering is based on a subdivision of space into an arbitary hierarchy. We then freely choose the hierarchy for maximum computational efficiency and convenience. Levels of the hierarchy can include horizontal strips of 16x16 tiles, the 16x16 tiles, horizontal strips of 16 pixels within a tile, individual pixels within a strip, and subpixel samples within a pixel. Indeed, that happens to be the hierarchy we choose.

At each level of the hieararchy, we start with the winding number at some representative point inside the spatial subdivision, conveniently the top left corner of the corresponding rectangle. We then consider only path segments inside that region, counting crossings to assign winding numbers to the corresponding representative points of each subdivision in the lower level of the hierarchy.

This induces a tree structure on the space for the rendering viewport (not unlike a quadtree). The winding number at each point is then the sum of winding number contributions on each edge up the path to the root. Seen this way, it is clear how parallelism can be preserved (while of course allowing sequential evaluation if that's convenient).

The approach is efficient. At higher levels of the hierarchy, only small numbers of winding numbers need be computed. At lower levels, lots of winding numbers are needed, but only line segments within that local region need be considered. As an important special case, if there are no line segments intersecting the region, then all the computation in lower levels of the hierarchy can be skipped; the winding numbers are all the same within that region. Note that this is a generalization of the run length approach for scanline rasterization, but it works for all rectangular regions.

Subdivision into horizontal strips can be seen as an instance of prefix sum. A ray is broken into equal-length chords, the winding number is computed for each such chord, then summed up using prefix sum. It's very well understood how to compute prefix sums efficiently on GPU; it's been a theme of my work.

Seen through this lens, many of the existing GPU path rendering algorithms in the literature make more sense, and can be seen as different instances of the same unifying concept, just with different choices of subdivision and techniques for knitting the tree together. The RAVG paper uses 16x16 tiles as its primary subdivision, doing brute-force calculation of pixels within each tile. The Pathfinder and original Vello algorithms derive from that and are architecturally similar. The Li et al scanline rasterization approach uses scanlines (actually strips two pixels tall to more efficiently utilize rasterization hardware, organized into 2x2 pixel quads), plus subpixel samples. [Describe MPVG, maybe one or two others]

The choice of subdivision hierarchy is not arbitrary. Finer grain is better at exploiting sparseness, but may contribute overhead. And, especially on GPU, the amount of subdivision should be tuned so inter-thread communication fits neatly into workgroup memory. This is why there's a 16x16 tile structure at several levels, as it maps well to 256 threads per workgroup, generally an efficient and well-supoprted size (all WebGPU implementations are required to support a minimum workgroup of 256 threads).

It is my hope that new, highly parallel computers will emerge, fulfulling the original promise of the Connection Machine. I'm impressed at the perforamance and practicality of mainstream GPU hardware, but also dissatisfied with the overall complexity and dysfunction in the ecosystem (perhaps a topic for a future blog post). As such new computers become available, this general theory should be useful for choosing a spatial subdivision hierarchy tuned for the specific hardware.

5 PARALLEL CONSERVATIVE LINE RASTERIZATION

Line rasterization has a long and rich tradition. Bresenham's algorithm is one of the most famous; at each step, a decision is made whether to step to the right, or diagonally one pixel right and one pixel down (assuming a line slope in the first octant). For sequential machines, this algorithm is elegant and efficient. However, for our purposes there are two shortcomings.

The most obvious is that it is sequential rather than parallel, and we want to run our algorithms on highly parallel computers (GPUs in particular). A parallel Bresenham-like algorithm is fairly straightforward. Again assuming our line is in the first octant, the number of pixels is the width, and each pixel can be computed in parallel. The x coordinate is just x0 plus the index of the parallel enumeration, and the y coordinate can be computed as ‘round(a*x+y0)‘, where ‘a‘ is the slope.

The second problem is that Bresenham computes a "thin" line, or one with 8-neighbor connectivity, while we require 4-neighbor connectivity, also known as conservative rasterization. This is because we're not simply drawing the line, but identifying all square tiles that the line passes through.

image of 4-neighbor vs 8-neighbor line

One of the core techniques in Vello's new multisampled path rendering algorithm is a parallel approach to conservative line rasterization. I'm not aware of this technique being published anywhere, but I would also not be surprised if it did exist somewhere. It's slightly tricky to try to explain, but I'll do my best.

One way to construct a 4-neighbor connected (conservatively rasterized) line is to take an 8-neighbor line and shift every row by 1, so a move down and to the right simply becomes a move down.

Winding numbers. Spatial hierarchy. Winding number is sum along tree. Prefix sum a special case.

6 CONSERVATIVE LINE RASTERIZATION

Line rasterization has a long and rich tradition. Bresenham's algorithm is one of the most famous; at each step, a decision is made whether to step to the right, or diagonally one pixel right and one pixel down (assuming a line slope in the first octant). For sequential machines, this algorithm is elegant and efficient. However, for our purposes there are two shortcomings.

The most obvious is that it is sequential rather than parallel, and we want to run our algorithms on highly parallel computers (GPUs in particular). A parallel Bresenham-like algorithm is fairly straightforward. Again assuming our line is in the first octant, the number of pixels is the width, and each pixel can be computed in parallel. The x coordinate is just x0 plus the index of the parallel enumeration, and the y coordinate can be computed as ‘round(a*x+y0)‘, where ‘a‘ is the slope.

The second problem is that Bresenham computes a "thin" line, or one with 8-neighbor connectivity, while we require 4-neighbor connectivity, also known as conservative rasterization. This is because we're not simply drawing the line, but identifying all square tiles that the line passes through.

image of 4-neighbor vs 8-neighbor line

One of the core techniques in Vello's new multisampled path rendering algorithm is a parallel approach to conservative line rasterization. I'm not aware of this technique being published anywhere, but I would also not be surprised if it did exist somewhere. It's slightly tricky to try to explain, but I'll do my best.

One way to construct a 4-neighbor connected (conservatively rasterized) line is to take an 8-neighbor line and shift every row by 1, so a move down and to the right simply becomes a move down.

7 NUMERICAL ROBUSTNESS

A particular challenge with a clever algorithm such as this is numerical robustness. Pure scanline rasterization is relatively simple to make robust, though it does take some care. However, with multiple levels of hierarchy, it's possible for line segments to be ambiguous which tile edges they cross, and if that's not accounted consistently, artifacts (such as 16x16 blocks of pixels being set to the wrong color) are possible.

Here we present a systematic approach to numerical robustness. First, we give a definition of robust rendering, as that's not obvious. Then we give a strong argument for correctness, one that could likely be developed into a mathematical proof. Each level of the hierarchy has its own possibility of an ambiguous crossing. The goal is that when they're all summed up, they come to a robust answer. For example, a vertical line segment aligned to the left side of 16x16 tile boundary also crosses the pixel boundary of the leftmost pixel in the tile. It is essential not to count that twice. Similar considerations apply to line segments that pass near corners.

An overly strict definition of robustness would be *exact* rendering. Every crossing determination is fundamentally an instance of an orientation problem. To determine the orientation of a point relative to a line, where all coordinates are rational numbers (and floating point numbers are a subset of the rationals) can always be computed exactly given sufficiently precise numbers. Shewchuk's approach [TODO link] is to determine whether ordinary floating point arithmetic is sufficient to unambiguously determine an orientation, and then upgrade to higher precision when not. If such cases are rare, then the resulting algorithm can be both efficient and very robust.

However, this is unappealing. Arbitrary precision arithmetic is not practical to implement on GPUs, as accounting for the storage of dynamically sized numbers alone is a massive jump in complexity. It's also overkill, we don't need results that precise, we just need it to be visually indistinguishable from exact rendering.

Instead, we propose a subtle but practical correctness criterion. Essentially, we are allowed to perturb control points by a small amount, then compute exact winding number based on the perturbed path. This seems like a small tweak, and that computing exact winding number would be challenging in any case, but the general principle is that after clipping to tile boundaries, it's possible to reliably determine the orientation relative to points on the boundary.

7.1 Numerical robustness for scanlines

Before getting into the full generality of the numerical robustness approach, let's consider a much simpler case: the traditional scanline approach. Conceptually, this approach casts infinite rays from the left to each pixel location to render, counting the crossings of that ray against the path to render. This is not extremely challenging to do robustly, but some care is needed to avoid

double-counting or skipping, which render as stripes across the image – an extremely familiar sight to those developing such algorithms.

Consider the unit square, (0, 0) to (1, 0) to (1, 1) to (0, 1). Is the pixel (0, 0) considered inside this square? What about the other four corners? By one convention, the first is considered inside, and the other three corners not. Similiarly, if we have a line from (0, -1) to (0, 0), and another line from (0, 0) to (0, 1), then a crossing through y = 0 should be counted once, as if there were a single line from (0, -1) to (0, 1), even though both lines actually touch the ray. How to keep it from being double counted?

A powerful analytical technique is to consider the ray not through y = 0 exactly, which has ambigous winding number in all these cases, but rather $y = \epsilon$, where ϵ is chosen to be some number finer than the resolution of floating point arithmetic.

It quite tractable to determine reliably and unambigously whether a line at $y + \epsilon$ (where y is a floating point number) intersects a line segment with floating point coordinates (x_0, y_0) to (x_1, y_1). In particular, there is an intersection exactly when $\min(y0, y1) \leq y < \max(y_0, y_1)$. One consequence is that a purely horizontal line ($y_0 = y_1$) has no such intersection. When there is an intersection, its location is $x_0 - y_0/(y_1 - y_0)$. With floating point arithmetic, this value can only be approximated, but an approximate value is sufficient. Also note that discarding perfectly horizontal lines avoids a divide by zero error; this will be a theme.

7.2 Generalization to two dimensions

Path rendering with general spatial subdivision requires intersections of both horizontal and vertical rays, which certainly makes reasoning about numerical robustness more challenging. And in general, when the winding number test path is broken down into multiple segments, it's important not to double-count or skip crossings.

Consider the triangles (0, 0) to (1, 0) to (1, 1) and (0, 0) to (1, 1) to (0, 1). Their winding numbers must sum to the same as the unit square, but that leaves the question of *which* triangle includes (0, 0). Either is reasonable, as long as it's consistent. If there are different choices at different levels of the hierarchy, the inconsistent rendering can result.

The above technique must be refined. In particular, sampling the winding number at (ϵ, ϵ) won't work, because the diagonal line goes right through it, making orientation ambiguous. We can again borrow techniques from the hyperreals, and choose to sample at (ϵ^2, ϵ). Using this criterion, the first triangle is clearly excluded, and the second is clearly included.

8 PERFORMANCE EVALUATION

Measurements. Lots of measurements.