Efficient GPU Path Rendering Using Scanline Rasterization

Rui Li Qiming Hou Kun Zhou

State Key Lab of CAD&CG, Zhejiang University*

Abstract

We introduce a novel GPU path rendering method based on scanline rasterization, which is highly work-efficient but traditionally considered as GPU hostile. Our method is parallelized over *boundary fragments*, i.e., pixels directly intersecting the path boundary. Non-boundary pixels are processed in bulk as horizontal spans like in CPU scanline rasterizers, which saves a significant amount of winding number computation workload. The distinction also allows the majority of our algorithmic steps to focus on boundary fragments only, which leads to highly balanced workload among the GPU threads. In addition, we develop a ray shooting pattern that minimizes the global data dependency when computing winding numbers at anti-aliasing samples. This allows us to shift the majority of winding-number-related workload to the same kernel that consumes its result, which saves a significant amount of GPU memory bandwidth. Experiments show that our method gives a consistent $2.5\times$ speedup over state-of-the-art alternatives for high-quality rendering at Ultra HD resolution, which can increase to more than $30\times$ in extreme cases. We can also get a consistent $10\times$ speedup on animated input.

Keywords: path rendering, vector graphics, GPU computing

Concepts: •**Computing methodologies** → **Rendering; Rasterization; Antialiasing;** •**Theory of computation** → **Massively parallel algorithms;**

1 Introduction

Vector graphics have been used in a wide variety of scenarios from typography, web illustrations to stylized artistic design. Such images can be rendered onto any display surface from tiny mobile devices to gigantic building posters, without introducing the pixelated or blurry artifacts characteristic to resampling resolution-dependent bitmaps. However, the costly rendering process has historically discouraged real-time applications.

The majority of modern vector image standards follow the seminal work of Warnock and Wyatt [1982] and rely on *paths* as the basic primitive. Each path is essentially a set of curves, with a filled interior and/or thick lines generated by stroking and optionally dashing the curves. From a rendering perspective, an implementation may choose to approximate strokes as filled paths and for simplicity of discussion we will focus on filled paths in this paper.

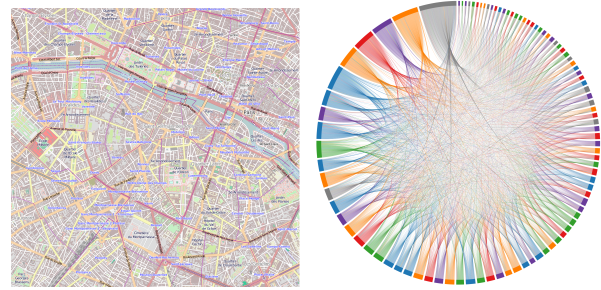
*Corresponding authors: Qiming Hou (hqm03ster@gmail.com), Kun Zhou (kunzhou@acm.org)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. © 2016 ACM.

SA '16 Technical Papers, December 05-08, 2016, Macao

ISBN: 978-1-4503-4514-9/16/12

DOI: http://dx.doi.org/10.1145/2980179.2982434



(a) Paris 30k, 56fps (b) Chords, 36fps

Figure 1: *The two most challenging benchmark images we rendered. Both images are rendered at approximately 2K×2K resolution with* $32\times$ *super-sampling. Our method is able to render the complicated map in (a) at 56fps, which has never been achieved before.*

A key challenge of path rendering is that the input curves specify only the path boundary – the filled interior must be determined from the boundary representation. In modern standards, the interior is defined as the set of points whose *winding number*, i.e., the number of times the path boundary winds counterclockwise around the given point, passes a per-path *fill rule*. On the CPU, widely deployed scanline rasterizers like WebKit [WebKit 2016] intersect the path boundary with scanlines to generate horizontal spans. All points on each span have identical winding numbers and can be processed as a single unit. As mentioned by Kilgard and Bolz [2012], this family of methods is highly work-efficient, as pixels outside the path do not have to be processed and interior pixels do not have to invoke the intense math involved with high-order curves. However, traditionally such methods are also considered primarily sequential and thus GPU hostile.

Consequentially, the majority of state-of-the-art GPU path rendering methods evaluate one individual winding number for each anti-aliasing sample. While such a paradigm makes parallelization easy, it performs more work than traditional CPU methods. In addition, the winding number computation workload can vary considerably across different samples, which would lead to poor load-balancing when directly parallelized at the sample-level.

We propose a novel scanline rasterizer that enables work-efficient path rendering on the GPU. Our method is designed around *boundary fragments*, which are 2×2 pixel squares that intersect with a path boundary. Like in CPU algorithms, we render non-boundary pixels as horizontal spans delimited by boundary fragments. This eliminates individual winding number computation for span-covered samples, which makes our method work-efficient.

Our method is built upon an efficient GPU parallelization scheme with two key properties. First, the distinction of boundary fragments and non-boundary spans allows the majority of our algorithmic steps to focus on boundary fragments only, which leads to highly balanced workload among different GPU threads. Second, our winding number computation algorithm does not perform any memory access on a per-sample basis. Specifically, we first compute the winding numbers at a fixed point on each fragment. Then in a second kernel, we connect samples to this point using a comb-like ray pattern (Fig. 5(b)). We then look up signed cross-

ing numbers from bit-compressed tables, and the complete winding numbers are immediately consumed in the fill rule test. The final results are stored back as per-fragment bitmasks and sent to the hardware rasterizer.

Experiments show that our method gives a consistent $2.5\times$ speedup over state-of-the-art alternatives for high-quality rendering at Ultra HD resolution, which can increase to more than $30\times$ in extreme cases. We can also get a consistent $10\times$ speedup on animated input.

2 Related Work

Here we focus on state-of-the-art GPU rendering methods and the approaches we drew key inspirations from.

A widely-employed GPU solution is to decompose the filled paths into simpler primitives that can be rendered natively. Loop and Blinn [2005] employ a conservative triangulation scheme and reject falsely covered samples in a shader. However, the conservative triangulation has to be generated in a costly CPU preprocess.

Microsoft Direct2D [Kerr 2009] rasterizes pixel space trapezoids and computes fractional coverage at edges. However, their trapezoidal tessellation is performed on the CPU which becomes a bottleneck.

Kokojima et al. [2006] generate potentially self-overlapping triangles and use the stencil buffer to compute winding numbers. Their simpler triangulation can be generated at a significantly lower cost. However, the algorithm can only process one path at a time and the stencil buffer must be reset in-between. The per-path setup overhead becomes a major bottleneck for complicated images with a large number of small paths. Also, their triangulation can generate a considerable amount of redundant pixel fills for high-curvature input like Fig. 1(b), on which even simple stencil tests would accumulate into a significant overhead.

The NVPR (NV_path_rendering) pipeline [Kilgard and Bolz 2012] is a practical adoption of the stencil-based approach, using dedicated driver extensions to mitigate the per-path setup cost. It has demonstrated real-time performance on recent mobile GPUs and has been adopted in commercial software [Kilgard 2014; Batra et al. 2015]. Our method eliminates this setup cost at design level, achieving a higher performance without relying on any feature dedicated to path rendering.

Early CPU scanline methods use edge list structures to lookup scanline-curve intersections required to generated spans [Wylie et al. 1967]. However, such data structures require on-the-fly maintenance and are not suitable for the fine-grain parallelization required by modern GPUs. Newman and Sproull [1979] further developed this method by computing intersections between all edges and all scanlines and then sorting the intersections before rasterizing the spans. The idea of clipping boundary curves against pixels and rasterizing horizontal spans then becomes a common technique in this area and is followed by many works, including those we discuss bellow. Our method implements the same functionality using GPU parallel primitives like sort and scan.

Kallio [2007] proposes a scanline rasterizer based on the edge-flag algorithm [Ackland and Weste 1981], which uses a prefix sum of bitmasks to implement anti-aliasing for triangles. While it is possible to adapt this method to render curves on the GPU, it requires one separate prefix sum per anti-aliasing sample per scanline to compute winding numbers for the non-zero filling mode. Our method adopts an analogous bitmask approach. However, we use a ray shooting pattern that obtains the same winding numbers while only requiring a single prefix sum per scanline, which is significantly more efficient on the GPU.

Manson and Schaefer [2011; 2013] apply a prefix-sum primitive on pixel-sized scanlines to implement analytical shading integration inside pixels. Our scanline formulation is compatible with their integration-based anti-aliasing scheme. However, in this paper we focus on sampling-based anti-aliasing which better implements standard-compliant fill rules.

The recent technique of [Whitington 2015] treats horizontal pixels as rectangle spans. It clips spans against each other, and merges spans together for hidden surface removal. It also takes temporal coherence into account for span updating. While we also see spans as long horizontal lines, we leave clipping, blending and anti-aliasing to hardware for optimal performance.

Our fragment generation step is similar to the previous CPU lattice-clipping scheme presented by Nehab and Hoppe [2008]. We use fixed-sized small fragments for better GPU parallelism. Also, while Nehab and Hoppe [2008] create and store the generated segments and test them against each sample, which consumes considerably more memory bandwidth, we immediately consume relevant crossing numbers after fragment generation and thus reduce memory bandwidth and shader complexity. As a tradeoff, our method sacrifices random sample access capabilities, which limits applications like texture mapping or arbitrary anti-aliasing filters.

Vector texture methods [Nehab and Hoppe 2008; Parilov and Zorin 2008; Qin et al. 2008; Wang et al. 2010; Ganacim et al. 2014] preprocess the input paths into some data structure, from which one can perform point-in-shape, area coverage, or texel color queries at arbitrary locations. These methods allow highly flexible display transformations, such as mapping vector graphics onto 3D surfaces as a texture, and can achieve high rendering efficiency. On the other hand, the texture construction is typically off-line, which prevents such methods from handling dynamic input. The recent shortcut tree [Ganacim et al. 2014] is notable in that its construction is also accelerated by the GPU. Even if the precomputation cost of vector textures is disregarded, supporting random sample access necessitates a data structure querying cost at each sample, whereas our method does not have such a cost for span-covered samples. In addition, the underlying data structure can have significant variation in the query cost among different texture locations, which leads to poor load-balancing on the wide SIMD threads of modern GPUs.

Some vector texture structures [Ganacim et al. 2014] can cull occluded fragments to avoid over-shading, which is not supported in our current method. However, in practice our improved rasterization efficiency consistently offsets such over-shading as modern vector standards primarily employ very cheap shaders. Should more expensive shading be needed, it is trivial to implement occlusion culling on top of our current algorithm by adding a standard depth prerendering pass with all opaque spans as the occluder geometry.

Diffusion curves is an alternative vector image representation that does not use filled paths [Orzan et al. 2008; Finch et al. 2011; Sun et al. 2012; Sun et al. 2014]. Our work vaguely resembles its principle of processing boundary and non-boundary pixels separately, but solves the unique problem of path rendering.

3 Algorithm Preliminaries

Our method takes SVG images [SVG 2011] as input, though we only render `path` elements. Strokes have to be converted to filled paths beforehand. All SVG curve types are supported, which includes lines, quadratic / cubic Bézier curves, and elliptical / circular arcs. We support both non-zero and even-odd fill rules, both sRGB and linearRGB color interpolation modes, and standard-compliant color gradients and solid fills.

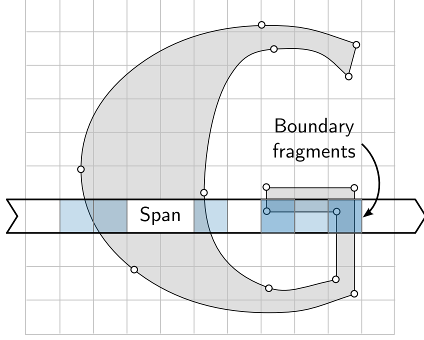


Figure 2: *Pixel-sized scanline and boundary fragments. We use pixel-sized scanline to generate boundary fragments for pixels directly intersecting a curve and horizontal spans for pixels completely covered by a curve. Note that pixels intersecting multiple curves may generate multiple fragments which have to be merged eventually.*

Before applying our GPU rendering algorithm, we parse the SVG file on the CPU and pack the curve vertices and fill attributes and miscellaneous metadata of all paths into a fixed number of flat buffers. This is analogous to standard polygon rendering in OpenGL, where vertex and element arrays from different objects are frequently packed into flat buffer objects for efficiency. The original data ordering in the input file is preserved since it coincides with the back-to-front blending order. The CPU code also packs all color gradients into a single ramp texture and flattens the group transformation hierarchy into a flat list of per-path matrices. The data buffers, the ramp texture, and the matrix list are then copied to the GPU as the input to our main algorithm, which renders everything in a single batch. Unless otherwise specified, all operations described in the following sections are performed on the GPU.

Algorithm 1 Our scanline conversion algorithm

1: Apply view / object transformation to all curve vertices
2: // Generate fragments
3: **for each** curve i {
4: Generate a virtual intersection at $t = 0$
5: **for each** monotonic segment of curve i {
6: **for each** grid line j sequentially {
7: Compute the intersection position $t_{i,j}$ in ascending order
8: }
9: }
10: Generate a virtual intersection at $t = 1$
11: }
12: **for each** curve-grid-line pair (i, j) {
13: Generate a boundary fragment between $t_{i,j-1}$ and $t_{i,j}$
14: }
15: // Compute coverage
16: Sort the fragments by (y, x) ascending, segmented by path id
17: Merge same-pixel-same-path fragments
18: **for each** merged fragment $(\mathrm{path}_k, x_k, y_k)$ {
19: Initialize coverage bitmask $M_C(k)$ to 0
20: **for each** sample point (u, v) {
21: Compute winding number $N(k, u, v)$ at $(x_k + u, y_k + v)$
22: **if** $(N(k, u, v)$ matches the fill rule of $\mathrm{path}_k)$ {
23: Set the corresponding bit in $M_C(k)$
24: }
25: }
26: }
27: // Final rendering
28: **for each** merged fragment $(\mathrm{path}_k, x_k, y_k)$ {
29: Render fragment (x_k, y_k) with coverage $M_C(k)$
30: **if** $(x_k + 1 < x_{k+1}$ && $y_k == y_{k+1}$ && $\mathrm{path}_k == \mathrm{path}_{k+1})$ {
31: **if** $(N(k + 1, 0, 0.5)$ matches the fill rule) {
32: Render span from $(x_k + 1, y_k + 0.5)$ to $(x_{k+1}, y_k + 0.5)$
33: }
34: }
35: }

4 Scanline Conversion

Without loss of generality, in the following discussion we assume the scanlines are horizontal. We also assume that when done sequentially, elements on the same scanline are processed left-to-right and scanlines are processed top-to-bottom.

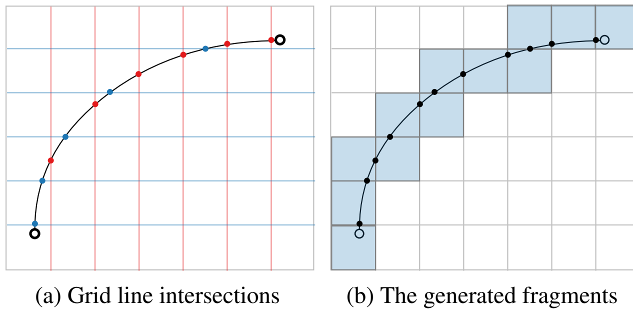
Traditional CPU scanline rasterization works by maintaining an *active edge list*, which incrementally maintains all curve intersections on the current scanline [Wylie et al. 1967]. When scanned sequentially left-to-right, each pair of neighboring intersections forms a horizontal span. As illustrated in Fig. 2, such spans can only be fully covered or fully exposed by the scanned path, and the final result can be produced by rendering all covered spans. Whether a span is covered is determined by testing the *winding number* at an arbitrary point on the span. However, an obvious obstacle working against efficient GPU parallelization is the sequential nature of the left-to-right scan and the incremental maintenance of the active edge list.

Key idea. Our GPU algorithm takes a pixel level approach. Specifically, the "scanlines" in our algorithm are rectangles with a height of one pixel, as in previous work (e.g., [Duff 1989; Whitington 2015]). As illustrated in Fig. 2, the intersections we generate are curve segments produced by clipping a curve with the rectangle corresponding to a pixel. For convenience, we call such intersections *boundary fragments*, as they correspond to the set of fragments that would have been generated by rasterizing the boundary curves.

We also define a related concept called *merged fragment*, which refers to the set of all boundary fragments generated by curves belonging to a given path on a given same pixel. The rightmost covered pixel on the scanline in Fig. 2 illustrates such a situation, where fragments belonging to different curves of the same path end up in the same pixel, and have to be processed as a single unit in several algorithmic steps.

By formulating the problem around such fragments, we are able to aggregate the non-boundary pixels into horizontal spans, thus avoiding the cost of processing them individually. Since the number of boundary fragments and spans is a sublinear function of screen resolution, this aggregation can save a considerable amount of workload on modern high definition displays. The only exception is the final rendering step, which is performed using the highly efficient hardware rasterizer. The fragments themselves can also be generated independently from each curve and processed *en masse*, which makes our method massively parallel thus highly scalable.

Algorithm 1 summarizes our scanline conversion algorithm. There are three major steps. First, we generate the boundary fragments by intersecting curves with pixel grid lines, which corresponds to lines 2-14. The second step in lines 15-26 sorts the generated fragments, merges them, and performs winding number tests to generate the per-merged-fragment coverage masks. Details regarding the winding numbers will be discussed in Sec. 5. The final step, detailed in lines 27-35, renders the final result onto a framebuffer with hardware multi-sample anti-aliasing (MSAA) [Segal and Akeley 2003]. In the following we will describe how we map each step to the GPU and discuss key motivations to our algorithm design.



(a) Grid line intersections (b) The generated fragments

Figure 3: *Generating boundary fragments for an example 7×6 monotonic segment. In (a), the vertical lines and their intersections are colored in red and the horizontal ones are colored in light blue. After sorting, the segment between each pair of neighboring intersections / curve end points becomes a boundary fragment.*

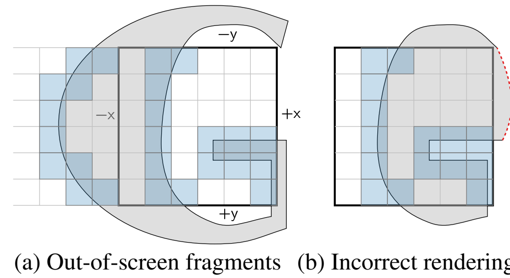
Fragment generation. We start by creating one GPU thread for each input curve, monotizing it, then intersecting each monotonic segment with all relevant pixel grid lines in a loop, as illustrated in Fig. 3(a). We use numerical root finding to compute the t value at each intersection, where t is the variable parameterizing the curve. The horizontal and vertical results are merge-sorted sequentially within each thread, as shown in lines 6-8. Finally, as illustrated in Fig. 3(b), the curve segment between each pair of neighboring intersections is recorded as a boundary fragment, which corresponds to lines 12-14.

While in this step it is possible to increase the parallelism beyond the curve level, the current approach turns out to be the most efficient and the most numerically stable approach among all alternatives we tried. The intra-curve sequential processing allows us to use a bisecting solver initialized from the previous intersection, which enforces monotonicity. The sequential method also requires considerably fewer iterations to converge than independently intersecting each curve-line pair, which outweighs the cost of reduced parallelism. To alleviate load balancing problems created by each curve generating a different number of fragments, we recursively split long input curves in the CPU-side SVG parser until the maximum curve length falls below a fixed threshold.

Sorting. Line 16 in Algorithm 1 sorts the generated boundary fragments in the correct order for horizontal spans to form between neighboring pairs. Specifically, the fragments are sorted by their owning path ids first, then by their y coordinates in ascending order, finally by ascending x coordinates. This is equivalent to the processing order in typical sequential scanline renderers, where each path is processed individually by first sweeping scanlines top-to-bottom, then processing each scanline left-to-right.

The sorting is the only step with a higher-than-linear time complexity. As an optimization, we employ a segmented merge sort [Baxter 2013] and reorganize the input data to make the individually sorted segments as short as possible. Specifically, we assume curves from the same path are provided continuously and paths are specified in back-to-front order. This matches the original input order and is preserved throughout the entire algorithm, which allows us to sort everything by path id for free and restricts the real sorting granularity to the path level. In our benchmarks, the relatively small segments also make merge sort a more efficient choice than the more commonly deployed radix sort.

Rendering. Once the winding numbers are computed, the final step simply renders the fragments and spans using the hardware rasterizer. We generate both fragments and spans as line primitives in a single vertex buffer, then render everything in a single draw call. Each span is rendered as a horizontal line passing through the relevant pixel centers, and each boundary fragment is rendered as a degenerated line one pixel in length. A shader computes a fill color and passes the per-fragment coverage mask to `gl_SampleMask`, a built-in shader output that explicitly controls the set of MSAA samples covered by a raster fragment. Span primitives have their masks set to all-ones. Finally, since all fragments have been sorted by path id, the rendering is naturally back-to-front and blending is performed in the correct order.



(a) Out-of-screen fragments (b) Incorrect rendering

Figure 4: *The $-x$-side fragments we keep and an incorrect rendering caused by discarding them.*

Implementation Details. Two detailed issues have to be handled carefully when implementing Algorithm 1. First, the process illustrated in Fig. 3 may generate fragments outside the viewport. As illustrated in Fig. 4(a), such fragments can be discarded only if the fragment in question lies beyond the $+x$ or $\pm y$ edges, whereas fragments residing beyond the $-x$ edge have to be retained. They contain essential information required to resolve the coverage of in-screen pixels, and removing them may cause the wrong pixels to be filled as illustrated in Fig. 4(b). For viewport culling, we instead perform a simple bounding box test for each path. A path is discarded completely if its bounding box does not overlap with the viewport at all.

The other detail omitted from Algorithm 1 is the dynamic memory allocation. For example, the intersection step in lines 2-11 requires generating a variable amount of data in each thread. We implement such steps using the standard prefix-sum based approach [Sengupta et al. 2007], which is highly efficient. Specifically, we first compute the output size of each thread, then compute a prefix sum of the sizes to compute per-thread addresses, and finally write the generated output in a second kernel.

5 Winding Number Computation

We base our winding number computation on the well-established signed crossing method [Alciatore and Miranda 1995]. The method works by shooting a ray to an infinitely far point and accumulating the ray-curve crossing events, where each event produces a ± 1 count with its sign determined by the curve winding direction relative to the ray. The accumulated result is called a *crossing number*, which is equal to the winding number if the ray leads to infinity and the curves form a closed path. A naive implementation of this method shoots rays from all samples into the same direction. Since efficient GPU utilization requires a parallelization granularity no coarser than boundary fragments, using a per-sample scanline pattern would require storing the winding number change on each sample to memory, which results in an excessive amount of global data dependency and could be prohibitively expensive on GPUs. As illustrated in Fig. 5(a), each ray going in or out of the central pixel represents the need to propagate one winding number across different GPU threads. The propagations have to be implemented using one separate prefix sum per sample per scanline, which would con-

sume a significant amount of memory bandwidth.

Our solution is a comb-like ray pattern that computes the same winding numbers while keeping the data dependency at a minimum. As illustrated in Fig. 5(b), we intercept the horizontal rays shot from sample points at the left side, then make them continue vertically and converge at the center-left point ($(0, 0.5)$ in the figure). Data dependency is limited to a single main ray continuing from there to the next pixel, which maps to one prefix sum per scanline regardless of the number of samples. While our ray pattern may lead to a different set of intersection points, if one traces the path leading from each sample point to infinity and count the signed crossings, one would still obtain exactly the same winding number.

For clarity of discussion, here we focus on one merged fragment k and use a pixel-local coordinate system where the top-left corner of its pixel is at $(0, 0)$. We need to compute the winding number N at MSAA sample points (u, v) and the fragment center-left point $(0, 0.5)$, based on a set of curve segments corresponding to original unmerged boundary fragments. We define $C_i(u_0, v_0; u_1, v_1)$ as the signed crossing number generated by intersecting curve i with a ray segment from (u_0, v_0) to (u_1, v_1). As illustrated in Fig. 5(b), for each sample point (u, v), we first shoot a horizontal ray to $(0, v)$, then shoot a vertical ray to the center-left point $(0, 0.5)$. Finally, all center-left points on a given scanline are connected using a main ray. The sample point winding number $N(k, u, v)$ can be computed by summing up the signed crossing numbers of the three ray segments:

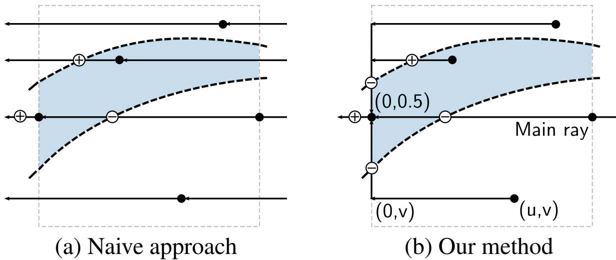
$$N(k, u, v) = N_M(k) + N_V(v) + N_H(u, v), \tag{1}$$

where N_M, N_V, N_H are signed crossing numbers of the main ray, the vertical ray, and the horizontal ray respectively. In the following we analyze the three components one by one and show how they can be computed efficiently on the GPU.

Main ray. Among the three components, only $N_M(k) = N(k, 0, 0.5)$ depends on global information. Since we have already sorted the merged fragments along the scanline direction, N_M can be computed recursively as

$$N_M(k + 1) = N_M(k) + \sum_i C_i(0, 0.5; 1, 0.5), \tag{2}$$

if fragment k and fragment $k + 1$ belong to the same scanline and the same path, and $N_M(k + 1) = 0$ otherwise. Note that Eq. (2) is essentially a prefix sum. We simply compute $C_i(0, 0.5; 1, 0.5)$ for each boundary fragment based on curve endpoint coordinates and then compute N_M using an off-the-shelf segmented scan routine [CUDPP Developers 2014].



(a) Naive approach (b) Our method

Figure 5: *Rays used to compute winding numbers. Solid lines are rays and dashed lines are curves. Sample points are shown as black dots. Ray-curve crossings are shown as white circles with their corresponding signs labeled inside.*

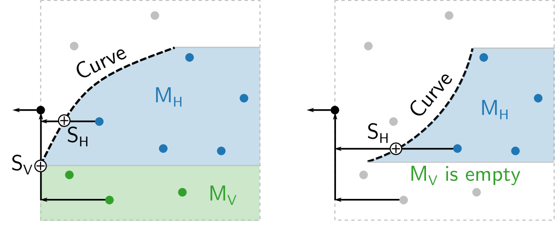
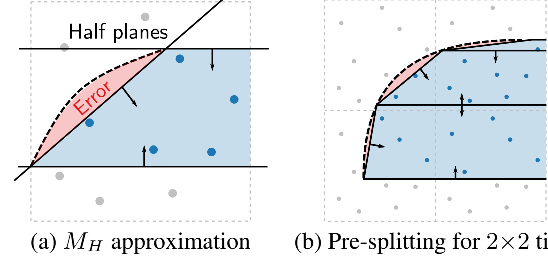


Figure 6: *Bitmasks generated for two different curves. When a curve fragment does not intersect with the left pixel boundary, it does not intersect with our vertical rays at all, leading to an empty M_V. This case typically occurs in boundary fragments that contain a curve end vertex.*



(a) M_H approximation (b) Pre-splitting for 2×2 tiles

Figure 7: *M_H approximation. In (a), the linearized trapezoid is represented as the intersection of three half planes. The rightmost edge coincides with the pixel boundary and can be ignored. In (b), three trapezoids are used to approximate each curve, which leads to a more accurate bitmask when rendering 2×2 pixel tiles.*

Vertical ray. Previous methods such as [Nehab and Hoppe 2008] and [Ganacim et al. 2014] calculate winding numbers using horizontal rays only. They create auxiliary segments to correct winding numbers at fragment boundaries. We perform the same task using vertical rays, which is mathematically equivalent to previous work.

The vertical ray crossing number N_V can be computed as

$$N_V(v) = \sum_i C_i(0, 0.5; 0, v). \tag{3}$$

Note that the curve segments involved are always monotonic as guaranteed by our scanline conversion algorithm. Therefore, each curve can have at most one intersection with the vertical ray regardless of the sample point location. Given a fixed set of MSAA samples $\{u_j, v_j\}$, we can pack the $C_i(0, 0.5; 0, v_j)$ values for all j into a per-curve signed bitmask M_V:

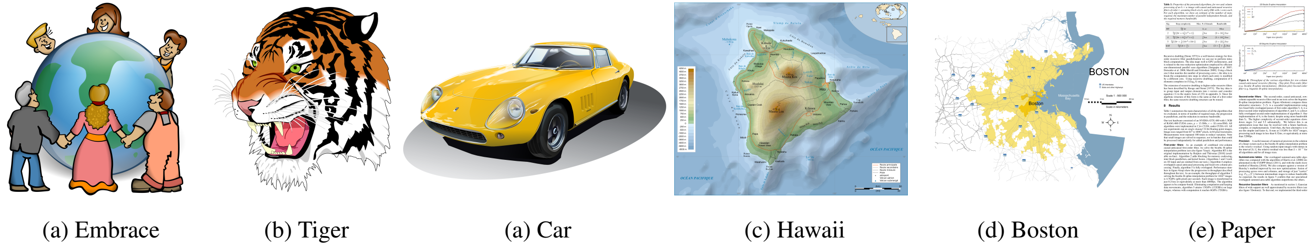
$$C_i(0, 0.5; 0, v_j) = S_V(i)\text{bit}(M_V(i), j), \tag{4}$$

where $S_V(i)$ is the crossing number sign if curve i intersects with the pixel left boundary and 0 otherwise. $\text{bit}(M, j)$ takes the j-th bit of mask M. Fig. 6 provides an intuitive interpretation of the bitmask M_V. As illustrated, M_V only depends on the vertical coordinate of the curve-pixel intersection point, and can be computed efficiently by querying a 1D lookup table prepared off-line. With this bitmask formulation, the final crossing number becomes

$$N_V(v_j) = \sum_i S_V(i)\text{bit}(M_V(i), j). \tag{5}$$

Horizontal ray. The bitmask formulation of N_V can be directly applied to N_H, leading to

$$N_H(u_j, v_j) = \sum_i S_H(i)\text{bit}(M_H(i), j). \tag{6}$$



(a) Embrace (b) Tiger (a) Car (c) Hawaii (d) Boston (e) Paper

Figure 8: *The test images as rendered by our method at Ultra HD resolution using WebKit convention. Full-size images are provided as supplementary material.*

As illustrated in Fig. 6, $S_H(i)$ is an analogously defined crossing number sign and M_H is an analogously defined bitmask. The most significant difference from $M_V(i)$ is that $M_H(i)$ depends on the entire sub-pixel curve geometry, which in turn depends on too many parameters to build a practical lookup table. Therefore, we choose to approximate the curve geometry as a line segment with the same end points, which turns the M_H-covered area into a trapezoid. The trapezoid edges correspond to a poly-line approximation of the original curve. As illustrated in Fig. 7(a), for each trapezoid edge, we first query a lookup table to obtain a bitmask that marks all sample points on the appropriate side of the corresponding half plane. The trapezoid mask are then computed as the *bitwise and* of the three query results.

The lookup table backing our half plane queries is a 2D analogy of the 3D half space lookup table developed by Laine and Karras [2010]. Given a half plane $\mathbf{n} \cdot (x - 0.5, y - 0.5) > c$, where $\mathbf{n}$ is a unit-length normal vector and $c \geq 0$, we store the corresponding bitmask at the point $(0.5 - 0.5c)\mathbf{n} + \mathbf{o}$ in a 2D texture, where $\mathbf{o} = (0.5, 0.5)$ denotes the center of the texture. For each texel of texture coordinate $\mathbf{t} = (u, v)$, the corresponding half plane parameters are given by $\mathbf{n} = normalize(\mathbf{t} - \mathbf{o})$, $c = 1 - 2 * (\mathbf{t} - \mathbf{o}) \cdot \mathbf{n}$. We then get the bitmask corresponding to the half plane and store it into the texel. When querying with a trapezoid line, we obtain $\mathbf{n}$ and c from the line equation, and calculate the texture coordinate for the table lookup. The resolution of the lookup table is 256×256 in our current implementation.

Final summation. A final summation $\sum_i$ is required to turn the per-curve bitmasks into per-sample winding numbers. We implement this summation using an optimistic approach that utilizes CUDA shared memory atomic operations whenever applicable and falls back to a global memory approach otherwise. Specifically, we first launch a kernel with one thread for each boundary fragment curve i, which accumulates the per-curve per-sample N_V and N_H values to a shared memory array. If all curves constituting a merged fragment are found in the same thread block, we can be sure that the accumulated result is complete. In such cases, we combine the shared memory result with N_M and test the complete winding number against the fill rule in the same kernel. That allows us to discard all intermediate results and only store a coverage mask to global memory. For the remaining merged fragments, the per-sample crossing numbers are stored to the global memory and processed normally in a fall-back code path. Since the vast majority of merged fragments only contain a handful of curves, time consumed by the fall-back path is negligible in all our experiments.

Bitmask size. The efficiency of our winding number computation increases as the bitmasks become wider. To maximally exploit this behavior, our final implementation renders each merged fragment as a 2×2 pixel tile using a coverage mask $4\times$ wider than the original anti-aliasing rate. Specifically, we run the entire Algorithm 1 except the final rendering step at half resolution. Once the merged fragment coverage masks have been computed, we render each fragment as a 2×2 line primitive, feeding its wide mask to a shader that extracts the relevant bits as `gl_SampleMask` for each of the four pixels. The corresponding increase in M_H approximation error is countered by uniformly splitting each curve into three sub-pixel segments and approximating them independently during winding number computation, as illustrated in Fig. 7(b). Shading precision is not affected as the hardware rasterizer always executes the color shader once per pixel regardless of the primitive size.

Degenerate cases. We handle degenerate cases by implicitly quantizing them into non-degenerate ones during the table lookup. Specifically, we carefully designed our lookup tables such that no tabulated half plane passes through any of the sample points or coincides with a pixel boundary. To handle imprecisely clipped fragments that generate a small extrusion or gap at the clipping boundary, we maximize our safety margin by basing our ray pattern on pixel center-left points and choosing sample locations away from pixel boundaries.

6 Experimental Results

We implemented our method using CUDA with OpenGL inter-op. The source code is available at *http://gaps-zju.org/pathrendering*. All our experiments were conducted on an NVIDIA GTX 980 GPU. We compare our method with three GPU methods, NVPR [Kilgard and Bolz 2012], the shortcut tree [Ganacim et al. 2014] (abbreviated as MPVG following their source code), and the vector texture method of Nehab and Hoppe [2008] (abbreviated as NH).

Since our method and MPVG only render filled paths, we converted all strokes in the test images to filled paths using off-the-shelf tools, and the same converted data set is used for all algorithms.

The comparison timing and images are generated by running the code or the executable provided by the respective authors on our hardware. For NVPR, we chose to run the "nvpr_svg" sample in the latest official SDK. Due to the lack of a single standard in several detailed aspects of implementation, we adjusted the relevant OpenGL states and shaders in our code to best match the rendered images of each individual comparison target. The NVPR/MVPG-compatible versions use the default `gl_SampleMask` shader as described in Sec. 4. All performance / quality benchmarks are reported for pairs of implementations that produce matching images.

6.1 Performance Comparison

6.1.1 Comparison with NVPR

Comparing to NVPR (see Table 1), the work efficiency of our method leads to a consistent $2.5\times$ speedup at Ultra HD with $27\times$ MSAA. The ability to process multiple paths in one batch also gives a significant advantage to our method for images composed from

	Paths	Curves	Resolution	S_{NVPR}		$S_{\text{M_NVPR}}$		S_{MPVG}		Our	E_{NVPR}		E_{MPVG}		Our	# frags.
				8×	32×	8×	32×	8×	32×	FPS	8×	32×	8×	32×	mem.	
Embrace	225	5K	1024x1096	0.8	2.1	0.6	1.6	5.6	5.5	530	0.5	0.3	0.5	0.4	3MB	60K
			2048x2192	1.5	3.4	1.2	2.6	9.4	7.7	305	0.4	0.3	0.4	0.3	7MB	113K
Tiger	302	28K	1024x1055	1.1	2.6	0.8	2.1	4.9	5.1	559	0.7	0.4	0.7	0.4	6MB	119K
			2048x2110	1.8	4.6	1.6	3.8	6.9	7.0	338	0.5	0.3	0.5	0.3	11MB	208K
Reschart	723	9K	1024x624	1.6	1.7	0.6	1.3	5.4	6.0	910	0.9	0.6	1.2	0.6	2MB	48K
			2048x1248	1.4	2.7	1.1	2.7	8.6	9.2	662	0.8	0.7	1.4	0.8	5MB	83K
Hawaii	1K	92K	1024x843	1.2	2.4	0.7	1.4	5.9	4.3	296	0.5	0.3	1.1	0.9	19MB	376K
			2048x1686	1.7	4.6	1.1	2.5	7.9	5.6	156	0.4	0.4	1.2	1.0	32MB	596K
Paper	5K	102K	1024x1325	5.9	4.8	0.5	1.2	7.4	7.5	550	1.3	0.8	1.6	0.9	12MB	226K
			2048x2650	4.0	3.0	1.0	2.0	12.7	11.2	375	1.0	0.6	1.4	0.7	18MB	351K
Chords	18K	63K	1024x1024	2.6	8.3	2.2	6.6	4.5	4.1	64	1.3	0.7	1.8	1.0	209MB	4084K
			2048x2048	6.0	28.7	4.6	11.9	5.2	4.1	36	0.9	0.5	1.3	0.8	421MB	8141K
Chords-Black	1	63K	1024x1024	1.7	5.4	1.7	5.4	4.3	4.0	64	1.3	0.7	1.8	1.0	209MB	4084K
			2048x2048	3.4	10.1	3.4	10.0	4.6	4.1	36	0.9	0.5	1.3	0.8	421MB	8141K
Paris 30k	51K	1614K	1096x1060	7.9	6.0	1.2	2.8	3.4	2.7	82	0.8	0.5	1.4	1.0	164MB	2935K
			2192x2120	5.3	3.5	2.5	5.3	5.2	3.4	56	0.6	0.4	1.1	0.9	220MB	4240K
Contour	53K	236K	1024x1024	34.3	30.5	0.7	2.2	6.4	5.8	301	0	0	0.3	0.4	39MB	739K
			2048x2048	20.5	15.3	1.6	4.3	9.9	8.0	183	0	0	0.3	0.3	68MB	1241K

Table 1: *Comparison results. The S_x columns are the speedup factors of our method over algorithm x when producing an image of comparable quality. A number greater than 1 means our method is faster. The MPVG precomputation time is excluded from this comparison. The end-to-end FPS data of our method is measured for the WebKit-compatible version at $32\times$ MSAA. The E_x columns are the RMSE error of our output images to those of algorithm x. The unit is $0.01\times$ output dynamic range. The number in "$n\times$" refers to the number of anti-aliasing samples per pixel. The "# frags." column is the number of boundary fragments generated by Algorithm 1.*

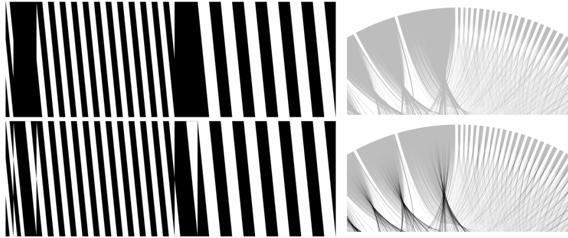


Figure 9: *Error created by merging paths. Left: overlapped paths using the even-odd fill role. Right: overlapped paths with transparent fill.*

many small paths (Paper, Paris, Contour and Chords), where the per-path overhead of NVPR starts to dominate render time.

There is evidence that NVPR could achieve a higher performance if one optimizes the OpenGL call sequence to reduce the per-path API overhead [Batra et al. 2015], though we are unable to reproduce their results based on available vendor documents. Instead, we approximate the overhead-reduced NVPR timing by aggressively merging input paths with identical fill attributes. We denote this path-merging NVPR as M_NVPR. Note that this optimization is not always correct for overlapping paths and an error case is shown in Fig. 9. Leaving correctness aside, we run NVPR with both the original SVG files and their path-merged versions. As illustrated in Table 1, our method still has a remarkable speedup even with path-merging enabled.

Most of our speedup over path-merging NVPR comes from the fact that our method processes less pixels than NVPR. Fig. 10 compares the total number of pixels filled by our fragments and spans, by the NVPR stencil pass, and by the NVPR cover pass. On small cases with short curves like Paper, NVPR processes $1.5\times$ more pixels than our method. On cases with long curves, complex sub-contours or large non-convex areas, NVPR processes $2\times$ to $100\times$ more pixels than our method. The counter-intuitive slower rendering of path-merged Paris 30k in Table 1 can also be explained using the pixel count data, as the merged path generates less efficient anchor and covering geometry than the original separate paths.

This difference in processed number of pixels can also be explained from a complexity analysis point of view. Except for a per-curve preprocess that generates geometry data needed by later passes, the NVPR pipeline primarily consists of a stencil pass and a cover pass. The stencil pass is based on the method proposed by Kokojima et al. [2006]. The method first renders an anchor geometry, which is a triangulation of the path boundary with all curves replaced by linear segments. Then it renders a bounding triangle for each non-linear curve with a discard shader to compensate for areas neglected by the triangulation. The cover pass simply renders the bounding box or the convex hull of an entire path to make sure the color shader gets executed on all path-covered samples. When compared with our method, we can deduct the common cost of shading actually covered samples and only consider the boundary fragment processing cost of our method, and the redundant pixels filled by NVPR in the form of overlapping triangulation, false positives / negatives in curve-discard triangles, and non-covered areas in the bounding box / convex hull. All three sources of NVPR redundant pixels increase quadratically as the resolution increases, whereas the number of boundary fragments in our method only increases linearly. This enabled our method to scale better at high resolution / anti-aliasing rates.

6.1.2 Comparison with Vector Textures

We first compare our method to NH. As illustrated in Fig. 11, our complete pipeline is $2\times$ faster than their rendering stage on the dataset they provided. The majority of this speedup comes from eliminating the redundant winding number computations inside span areas. This is a fundamental cost in vector texture approaches, which trades redundant computations for flexibility. Note that NH also supports a prefiltered rendering mode, which only evaluates one "sample" in each pixel after filtering the fragments. This prefiltered rendering mode of NH can be compared with a non-multisampling version of our method. As shown in Fig. 11, we obtain less performance speedups as the number of samples decreases.

When comparing to MPVG, we ignore their data structure construc-

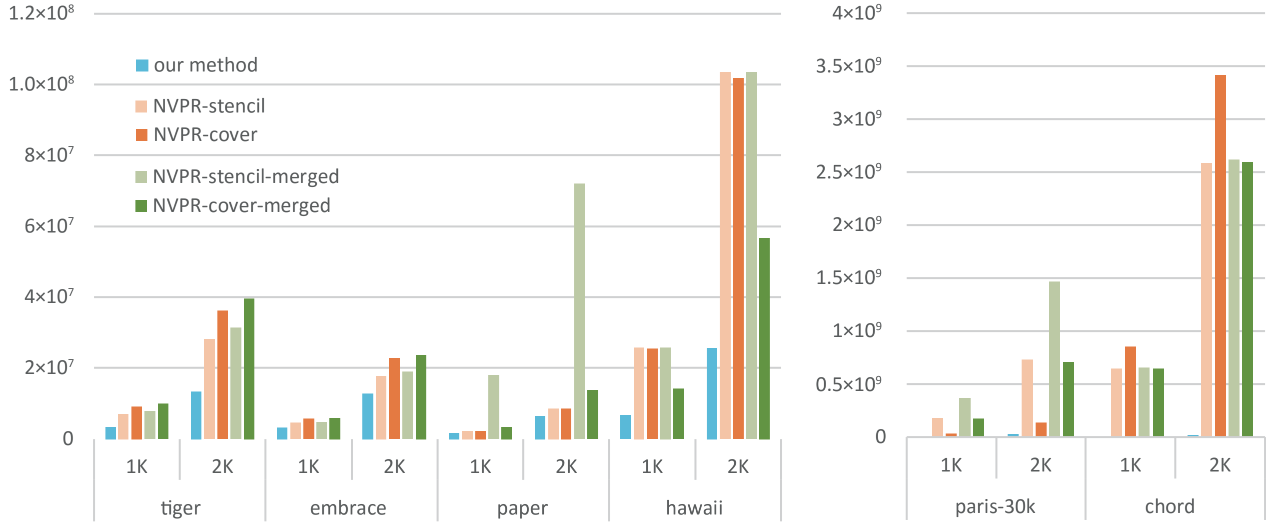


Figure 10: *The number of rendered pixels. For our method, this corresponds to the number of pixels filled by our fragments and spans. For NVPR, it is the number of pixels covered by path anchor geometry in the stencil step and pixels covered by path convex hulls, the tightest option available, in the cover step. The suffix "-merged" indicates that the path-merge optimization is used.*

tion time and compare our entire pipeline to their rendering step, see Table 1 for timing comparison. Querying a sample in their shortcut tree has a cost roughly proportional to the tree depth, which can reach over 10 for real scenes. In addition, traversing each tree level requires at least one random memory fetch and one branch. In contrast, our method operates entirely on boundary fragments except for the final rendering step. Our winding number computation has no significant branching and only performs 6 memory fetches for each *fragment*, which averages to less than one fetch per sample. Finally, MPVG is limited to per-sample shading by design, whereas our method, like NVPR, can utilize the more efficient per-fragment shading natively provided by modern graphics pipelines.

The Chords image is a chord diagram visualization of a randomly generated matrix, which mainly consists of chords formed by long cubic curves connecting random points on the outer circle. Despite its small size, this example is a challenging case for all three algorithms. To our method, the thin chords barely generate any meaningful spans and almost the entire image is rendered as boundary fragments. To NVPR, the long curves lead to highly inefficient triangulations and each transparent chord has to be rendered as a different path to produce the correct blending. The chord-black variant listed in Table 1 replaces the fill color of all paths to solid black, allowing the entire image to be merged into one single path for NVPR. That removes the impact of the per-path overhead and provides additional evidence for our cost analysis. To MPVG, the long curves are very hard to separate in their spatial subdivision structure. Despite the challenge, our method still scales gracefully and delivers a reasonable performance, while still being faster than the similarly scalable MPVG. On the other hand, both our method and MPVG have to consume a considerable amount of memory to achieve this scalability.

Regarding memory consumption, our method starts directly from raw curve vertices and indices and does not store any information persistently (save for the lookup tables that are approximately 1MB in size). The numbers reported in Table 1 are the *peak* memory consumption excluding input and output, which is reached just before the final OpenGL draw call. Overall, our memory consumption is roughly proportional to the number of boundary fragments, which only increases sub-linearly as the resolution increases. This also matches our performance scaling behavior.

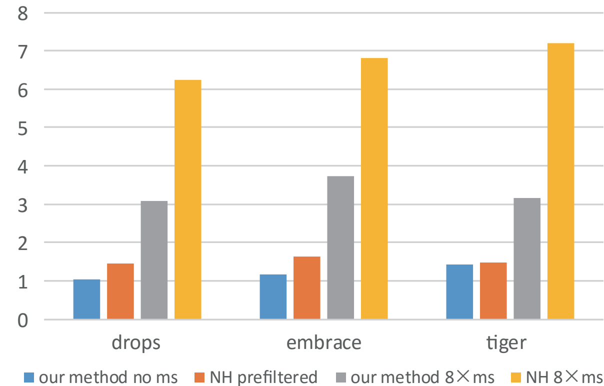


Figure 11: *Rendering time of our method and NH in millisecond. We compare our non-multisampling configuration with NH's prefiltered rendering mode, and also compare the two methods with* $8\times$ *multisampling.*

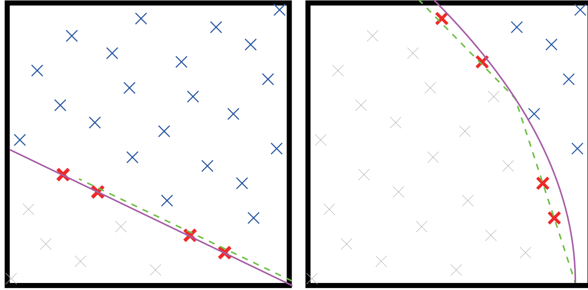
6.2 Rendering Quality

Fig. 14 provides a visual comparison between our method and a ground truth rendering using the Reschart image, which is a test case designed to highlight sub-pixel anti-aliasing issues. As illustrated, our result appears identical to the reference. Table 1 shows the root mean squared error (RMSE) of our rendering result to each comparison target. This difference mainly represents the approximation errors caused by our linearization and table lookup steps. The overall error is at approximately one percent of the output dynamic range, which is very modest.

For a quantitative analysis of our approximation errors, we have constructed dedicated test cases to better exercise our two error sources – look-up table quantization and sub-pixel curve linearization. We compare our method using MPVG as a reference. We modified both implementations to use exactly the same set of sampling positions and moved all operations to the linear color space. The test cases are designed to only contain fully-saturated color values and no overlapping / blending. Such a setup allows the number of differing samples in each pixel to be directly calculated from the

	RMSE	ME	NE/NB	
			1	≥ 2
line	0.0106	2	1.066%	0.0006%
fan	0.0132	10	4.322%	0.6633%
thin-quad	0.0029	3	1.283%	0.0311%
fat-quad	0.0035	2	1.139%	0.0038%
quad-in-pixel	0.0013	2	0.004%	0.0004%
quad-in-frag	0.0026	3	0.005%	0.0014%
cubic	0.0105	6	3.252%	0.4956%

Table 2: *Approximation error analysis. ME is max number of incorrect samples in one pixel. NE/NB is the percentage of curve-boundary pixels containing a given number of incorrect samples.*



(a) Lookup table error (b) Curve approximation error

Figure 12: *Two example pixels. Incorrect samples are shown as red $\times$. The solid purple curves show the correct path boundary, and the green dashed lines show the line segments corresponding to the actually fetched mask table entries.*

color difference between the two output images.

Table 2 shows the distribution of approximation errors in various test cases (see the test images in the supplementary material). The "lines" image contains a series of right-angled triangles with a continuously varying hypotenuse slope, which is designed to test the look-up table texels where the quantization error is the highest. The "fan" image is a fan of 180 triangles sharing the same tip, which is designed to test the error accumulation when many curves share the same pixel. "thin-quad", "fat-quad", "quad-in-pixel", "quad-in-frag" and "cubic" enumerate quadratic and cubic curves with continuously changing curvatures, which are designed to exercise our sub-pixel curve linearization. The "transparent-fan" in Fig. 18 is a transparent disk formed by 360 triangles, which demonstrates that our approximation remains consistent at coinciding edges.

As shown in the table, our approximation is able to correctly render 95% boundary fragments in the test images. 99.5% fragments are rendered with one or less wrong samples. This is well below the variance that can be introduced by varying sampling positions, and the difference is imperceptible on the final output image. In the rare cases where we do introduce noticeable error, the approximation remains visually plausible. Fig. 12 shows two single-curve worst cases with 4 incorrect samples inside one pixel at $32\times$ MSAA. As illustrated, the approximation errors correspond to a subtle displacement of the underlying curves and normally would not become objectionable when visually inspected on the final image. Fig. 13 shows one case that could introduce cracks due to the poly-line approximation, which may cause background color leak in fully-covered areas. Finally, as illustrated in the "transparent-fan" image, our approximation always produces exactly complementing bit-masks for pairs of coinciding edges and do not generate conflation artifacts.

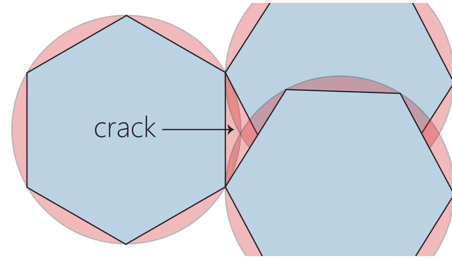


Figure 13: *Illustration of the crack case. The indicated central area should have been completely covered by the three circles. However, with our poly-line approximation, this small area becomes exposed, resulting in background color leak. Here we draw a 6-segment approximation to better visualize the crack. In practice, a pixel-size circle would have been approximated with more segments in our implementation due to the circle-to-rational conversion and monotizing processes. Adding a curvature-based adaptive splitting step can also reduce such artifacts.*

Image	Resolution
Embrace	1024×1096
	2048×2192
Tiger	1024×1055
	2048×2110
Car	1024×682
	2048×1364
Reschart	1024×624
	2048×1248
Hawaii	1024×843
	2048×1686
Boston	1024×917
	2048×1834
Paper	1024×1325
	2048×2650
Chords	1024×1024
	2048×2048
Paris 30k	1096×1060
	2192×2120
Contour	1024×1024
	2048×2048

Curve intersection Sorting
Winding numbers OpenGL rendering

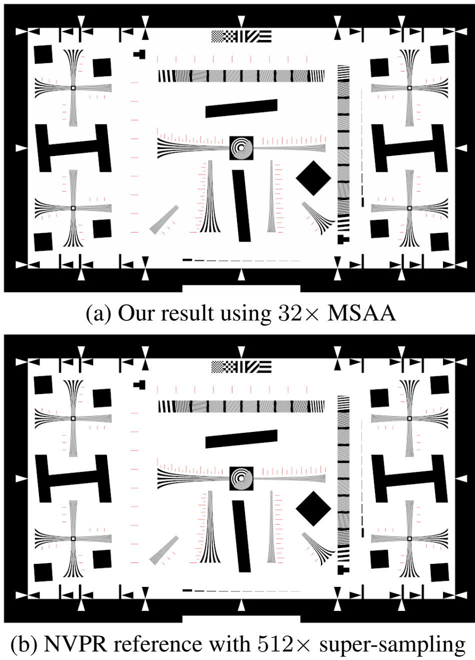
Table 3: *Relative timing visualization for each algorithm step for NVPR-compatible rendering at $32\times$ MSAA. The length of each bar is proportional to the time spent on the corresponding step.*

6.3 Memory

Fig. 15 shows the memory footprint of our method as a function of the output resolution. We vary the resolution from 128×128 to 2048×2048. Our method allocates memory to store curve fragments, merged fragments and spans, which increase in proportional to the total curve length in pixels. The result shows that our memory consumption grows linearly as the square root of total number of pixels, with a slope that depends on the scene complexity.

6.4 Performance Analysis

Time breakdown. Table 3 visualizes the per-step breakdown of our rendering time at $32\times$ MSAA. For the smaller test images, curve intersection consumes a higher fraction of total render time. This is caused by our curve-level parallelization, which results in GPU under-utilization for small images. Nevertheless, the absolute overhead remains tolerable. It is also notable that the time spent in the fragment-centric non-GL steps reduces in proportion to the



(a) Our result using 32× MSAA

(b) NVPR reference with 512× super-sampling

Figure 14: *The Reschart image rendered at its original* 1024×624 *resolution. The reference image is generated by down-sampling a* 4×4 *larger image rendered using NVPR with* 32× *MSAA.*

GL rendering time as the resolution increases, which is a natural consequence of the sub-linear fragment count scaling.

Algorithm validation. To demonstrate the optimization brought by each individual design choice, we implemented a naive GPU scanline path renderer and progressively added features / optimizations until it matches our final algorithm. Fig. 16 illustrates the performance differences among the following implemented variants:

- **no-ms**. A naive scanline rasterizer with 1×1 boundary fragments and no multi-sampling. It only tests one sample at the pixel center using implicit function test, then calculates winding number use per-pixel-scanline prefix sum. The implicit function code is taken from MPVG.

- **i-sample-scan**. A naive rasterizer with 32× multi-sampling. 32 prefix sums are needed for each horizontal pixel line. Samples are still tested using the implicit function, which is indicated by the prefix **i**.

- **mask-sample-scan**. This is based on **i-sample-scan**. The implicit function tests are replaced with our bitmask lookup tables, which is indicated by the perfix **mask**.

- **i-pixel-scan** and **mask-pixel-scan**. These are based on **i-sample-scan** and **mask-sample-scan** respectively. The comb-like ray pattern is added, which replaces the 32 prefix sums per scanline with a single one per pixel that computes N_M. The difference is compensated by the N_V term.

- **i-frag-scan** and **mask-frag-scan**. These are based on **i-pixel-scan** and **mask-pixel-scan** respectively. The boundary fragments are enlarged to 2×2 pixels.

The result shows that each of our design choices makes measurable improvements to a few rendering steps. Combining all of them together has a significant net effect, where the end-to-end rendering speed becomes comparable with the non-antialiased version.

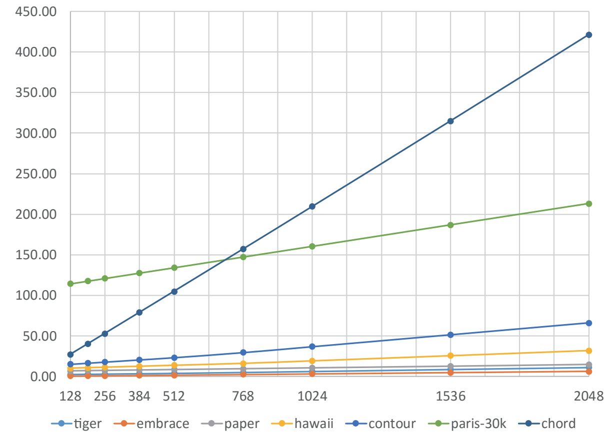


Figure 15: *Memory consumption. The horizontal axis shows output image height in pixels. The vertical axis shows our memory usage in MB.*

	data packing	copy to gmem	1K rendering
Embrace	0.14	0.29	1.87
Tiger	0.79	0.62	1.79
Reschart	0.25	0.35	1.10
Hawaii	7.58	3.21	3.38
Paper	3.92	1.79	1.82
Chords	0.58	0.64	15.63
Paris-30k	56.59	17.57	12.20
Contour	7.13	2.74	3.32

Table 4: *The CPU preprocessing time and 1K resolution rendering time in milliseconds. The "date packing" column represents the cost of packing an object-oriented representation SVG data into a structure of bit-compressed flat arrays. The "copy to gmem" column corresponds to the time spent inside cudaMemcpy during the preprocess.*

From **i-sample-scan** to **mask-sample-scan**, the table-lookup optimization makes the curve intersection step 2∼4× faster at little quality loss. Similar speedups can be observed under the pixel-scanline and fragment-scanline configurations. From **mask-sample-scan** to **mask-pixel-scan** to **mask-frag-scan**, the comb-like ray pattern reduces the number of prefix sums per scanline from 32 to 1 to 0.5. This significantly reduces the net memory bandwidth consumed during winding number computation, which results in a considerable speedup. Finally, from **mask-pixel-scan** to **mask-frag-scan**, switching to 2×2 fragments reduces the total number of fragments we have to generate, which makes the curve intersection and sorting steps faster with little performance loss in the final rendering step.

6.5 Setup cost

Our implementation stores vector graphics data as a structure of arrays. We perform a data packing and rearranging step on the CPU after parsing the SVG file and then copy the data to the GPU for rendering. CUDA and GPU launch overheads, plus miscellaneous per-frame tasks such as passing transformation matrices to the GPU, have been included in the reported rendering time. Table 4 shows the CPU preprocessing timing together with the rendering time for a single frame. The preprocess cost is negligible on small inputs but costly on large scenes, which might limit applications that require low setup latency (e.g., a web browser). Also, implementing the data packing step on the GPU may improve the performance.

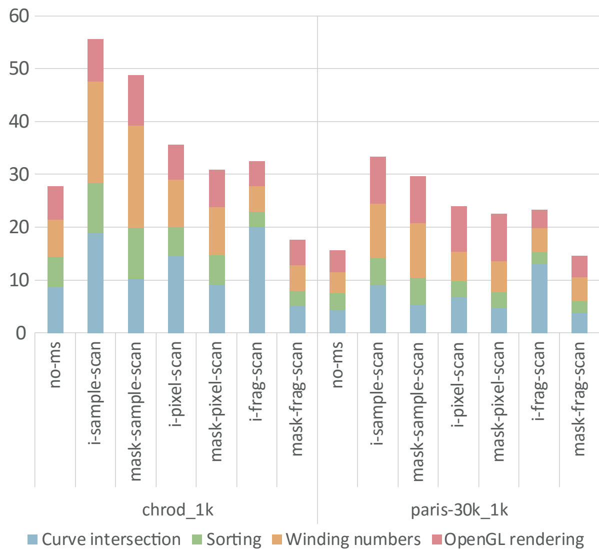


Figure 16: *Scanline comparison time break down in milliseconds. We show the performance differences among several implementation variants of our algorithm.*

Figure 17: *Paris-30k clipped to the union of all paths in Tiger. Our algorithm can use NVPR to populate the stencil buffer for clip-path support.*

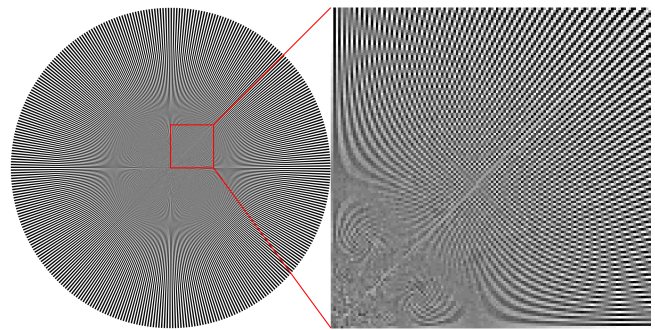
6.6 Vector Animation

Our fully-GPU scanline pipeline allows us to easily support arbitrary vector animation. We allow each frame to be generated from scratch with changing path topology, vertex position or shading attributes. In the supplementary video we implement an animated visualization similar to that provided by *http://blog.csaladen.es/refugee/*. Our method runs $14.8\times$ faster than NVPR and $11.8\times$ faster than MPVG on this animation, which allowed us to render this sophisticated demo in real-time.

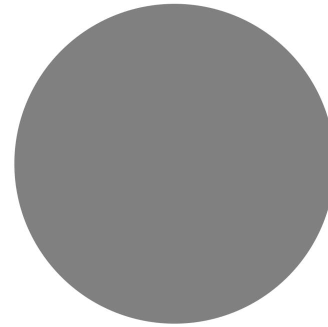
Compared with MPVG, our method eliminates the need to construct a tree for acceleration, which is still expensive even when parallelized on the GPU. NVPR has a baking step that constructs the various geometry data required by the stencil and cover passes. This step currently requires the path data to be updated on the CPU using `glPathSubCommandsNV` or `glPathSubCoordsNV`, which turned out to be very expensive in our experiments.

7 Limitations and Future Work

Our main limitation is the lack of native support for strokes and more advanced features like clip paths. We have implemented a simple recursive subdivision stroke-to-fill convert algorithm [Tiller and Hanson 1984] on the GPU, which consumes an additional $\sim$15% rendering time on our current test scenes. However, more efforts are still needed to render dashed strokes. On the other hand, since our method implements the final rendering pass in the conventional GPU pipeline, it is compatible with the majority of NVPR-based application-level techniques [Batra et al. 2015] and relevant OpenGL extensions, including NVPR. That allows the relevant features to be implemented outside our algorithm. For example, to implement clipping, we can simply invoke NVPR to populate the stencil buffer with the clip-path, then render the draw-path using our method with a stencil test enabled. A simple result is illustrated in Fig. 17. We also implemented a prototype hybrid renderer that handles filled paths using our method and strokes using NVPR, which avoids the stroke-to-fill conversion. This approach has not yet reached a competitive performance due to the overhead we experienced when mixing GL draw calls with NVPR strokes. However, such overhead would likely see a significant reduction if NVPR strokes could be made compatible with the NVIDIA command list extension.



(a) Triangle fan.



(b) Transparent triangle fan.

Figure 18: *Quality test results. (a) A fan of 360 triangles with alternating black and white colors, which shows the sampling artifacts. (b) A semi-transparent disk formed by a fan of 360 black triangles, which demonstrates that our approximation remains consistent at coinciding edges.*

The constant overhead of CUDA kernel launches, CPU-GPU synchronization and CUDA-GL context switches amounts to approximately 1.5ms in our method. This gives us a considerable disadvantage on the smaller test cases. A sort/scan library optimized for indirect kernel launching could likely reduce this overhead.

The most important future work direction would be providing native support for strokes. It would be also interesting to try to replace the winding number tests with an analytical integration approach [Manson and Schaefer 2011; Manson and Schaefer 2013; Kallio 2007]. Finally, adapting our method to mobile platforms may require some non-trivial changes, since mobile GPUs may have different performance characteristics than their desktop counterparts.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments, and Stephen Lin and Yiying Tong for proofreading the paper. This work is partially supported by the National Key Research & Development Plan of China (No. 2016YFB1001403), the NSF of China (No. 61272305, No. 61472352), and the National Program for Special Support of Eminent Professionals of China.

References

ACKLAND, B. D., AND WESTE, N. H. 1981. The edge flag algorithm: A fill method for raster scan displays. *IEEE Trans. Comput. 30*, 1 (Jan.), 41–48.

ALCIATORE, D., AND MIRANDA, R. 1995. A winding number and point-in-polygon algorithm. *Department of Mechanical Engineering Colorado State University, Fort Collins, CO*.

BATRA, V., KILGARD, M. J., KUMAR, H., AND LORACH, T. 2015. Accelerating vector graphics rendering using the graphics hardware pipeline. *ACM Trans. Graph. 34*, 4 (July), 146:1–146:15.

BAXTER, S., 2013. Modern GPU. https://nvlabs.github.io/moderngpu/.

CUDPP DEVELOPERS, 2014. CUDPP. http://cudpp.github.io/, version 2.2.

DUFF, T. 1989. Polygon scan conversion by exact convolution. In *International Conference On Raster Imaging and Digital Typography*, 154–168.

FINCH, M., SNYDER, J., AND HOPPE, H. 2011. Freeform vector graphics with controlled thin-plate splines. *ACM Trans. Graph. 30*, 6 (Dec.), 166:1–166:10.

GANACIM, F., LIMA, R. S., DE FIGUEIREDO, L. H., AND NEHAB, D. 2014. Massively-parallel vector graphics. *ACM Trans. Graph. 33*, 6 (Nov.), 229:1–229:14.

KALLIO, K. 2007. Scanline Edge-flag Algorithm for Antialiasing. In *Theory and Practice of Computer Graphics*, The Eurographics Association, I. S. Lim and D. Duce, Eds.

KERR, K. 2009. Introducing Direct2D. *MSDN Magzine 3*, 4.

KILGARD, M. J., AND BOLZ, J. 2012. GPU-accelerated path rendering. *ACM Trans. Graph. 31*, 6 (Nov.), 172:1–172:10.

KILGARD, M. J., 2014. NVIDIA path rendering: Accelerating vector graphics for the mobile web. http://www.slideshare.net/Mark_Kilgard/gtc-2014-nvidia-path-rendering.

KOKOJIMA, Y., SUGITA, K., SAITO, T., AND TAKEMOTO, T. 2006. Resolution independent rendering of deformable vector objects using graphics hardware. In *ACM SIGGRAPH 2006 Sketches*, ACM, New York, NY, USA, SIGGRAPH '06.

LAINE, S., AND KARRAS, T. 2010. Two methods for fast ray-cast ambient occlusion. In *Proceedings of the 21st Eurographics Conference on Rendering*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR'10, 1325–1333.

LOOP, C., AND BLINN, J. 2005. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph. 24*, 3 (July), 1000–1009.

MANSON, J., AND SCHAEFER, S. 2011. Wavelet rasterization. *Computer Graphics Forum (Proceedings of Eurographics) 30*, 2, 395–404.

MANSON, J., AND SCHAEFER, S. 2013. Analytic rasterization of curves with polynomial filters. *Computer Graphics Forum (Proceedings of Eurographics) 32*, 2, 499–507.

NEHAB, D., AND HOPPE, H. 2008. Random-access rendering of general vector graphics. *ACM Trans. Graph. 27*, 5 (Dec.), 135:1–135:10.

NEWMAN, W. M., AND SPROULL, R. F. 1979. *Principles of interactive computer graphics*. McGraw-Hill, Inc.

ORZAN, A., BOUSSEAU, A., WINNEMÖLLER, H., BARLA, P., THOLLOT, J., AND SALESIN, D. 2008. Diffusion curves: A vector representation for smooth-shaded images. *ACM Trans. Graph. 27*, 3 (Aug.), 92:1–92:8.

PARILOV, E., AND ZORIN, D. 2008. Real-time rendering of textures with feature curves. *ACM Trans. Graph. 27*, 1 (Mar.), 3:1–3:15.

QIN, Z., MCCOOL, M. D., AND KAPLAN, C. 2008. Precise vector textures for real-time 3d rendering. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '08, 199–206.

SEGAL, M., AND AKELEY, K., 2003. The OpenGL graphics system: A specification. http://www.opengl.org/documentation/specs/version1.5/glspec15.pdf.

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for gpu computing. In *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, GH '07, 97–106.

SUN, X., XIE, G., DONG, Y., LIN, S., XU, W., WANG, W., TONG, X., AND GUO, B. 2012. Diffusion curve textures for resolution independent texture mapping. *ACM Trans. Graph. 31*, 4 (July), 74:1–74:9.

SUN, T., THAMJAROENPORN, P., AND ZHENG, C. 2014. Fast multipole representation of diffusion curves and points. *ACM Trans. Graph. 33*, 4 (July), 53:1–53:12.

SVG, 2011. Scalable vector graphics, v. 1.1. W3C, second edition.

TILLER, W., AND HANSON, E. G. 1984. Offsets of two-dimensional profiles. *IEEE COMP. GRAPHICS APPLIC. 4*, 9, 36–46.

WANG, L., ZHOU, K., YU, Y., AND GUO, B. 2010. Vector solid textures. *ACM Trans. Graph. 29*, 4 (July), 86:1–86:8.

WARNOCK, J., AND WYATT, D. K. 1982. A device independent graphics imaging model for use with raster devices. *SIGGRAPH Comput. Graph. 16*, 3 (July), 313–319.

WEBKIT, 2016. Webkit, open source web browser engine. https://webkit.org/.

WHITINGTON, J. G. 2015. Two dimensional hidden surface removal with frame-to-frame coherence. In *Proceedings of the 31st Spring Conference on Computer Graphics*, ACM, New York, NY, USA, SCCG '15, 141–149.

WYLIE, C., ROMNEY, G., EVANS, D., AND ERDAHL, A. 1967. Half-tone perspective drawings by computer. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*, ACM, New York, NY, USA, AFIPS '67 (Fall), 49–58.