GPU-accelerated Path Rendering

Mark J. Kilgard Jeff Bolz
NVIDIA Corporation*

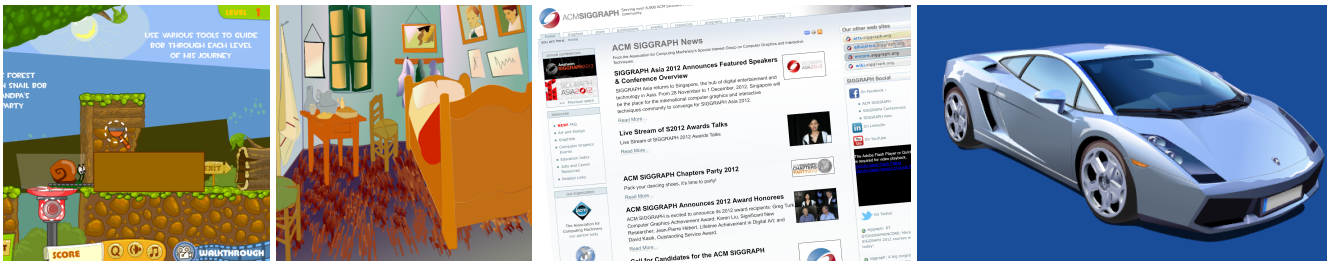

Figure 1: *GPU-accelerated scenes rendered at super-real-time rates with our system: Snail Bob Flash-based game (5ms) by permission of Andrey Kovalishin and Maxim Yurchenko, Van Gogh SVG scene with gradients (5.26ms) by permission of Enrique Meza C, complete (shown clipped) SIGGRAPH web page (4.8ms), and SVG scene with path clipping (1.9ms) by permission of Michael Grosberg, all rendered on a GeForce 560M laptop.*

Abstract

For thirty years, resolution-independent 2D standards (e.g. PostScript, SVG) have depended on CPU-based algorithms for the filling and stroking of paths. Advances in graphics hardware have largely ignored accelerating resolution-independent 2D graphics rendered from paths.

We introduce a two-step "Stencil, then Cover" (StC) programming interface. Our GPU-based approach builds upon existing techniques for curve rendering using the stencil buffer, but we *explicitly* decouple in our programming interface the *stencil step* to determine a path's filled or stroked coverage from the subsequent *cover step* to rasterize conservative geometry intended to test and reset the coverage determinations of the first step while shading color samples within the path. Our goals are completeness, correctness, quality, and performance—yet we go further to unify path rendering with OpenGL's established 3D and shading pipeline. We have built and productized our approach to accelerate path rendering as an OpenGL extension.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Stand-alone systems;

Keywords: path rendering, vector graphics, OpenGL, stencil buffer

Links: DL PDF

1 Introduction

When people surf the web, read PDF documents, interact with smart phones or tablets, use productivity software, play casual games, or in everyday life encounter the full variety of visual output created digitally (advertising, books, logos, signage, etc.), they are experiencing resolution-independent 2D graphics.

While 3D graphics dominates graphics research, we observe that most visual interactions between humans and computers involve 2D graphics. Sometimes this type of computer graphics is called *vector graphics*, but we prefer the term *path rendering* because the latter term emphasizes the path as the unifying primitive for this approach to rendering.

1.1 Terminology of Path Rendering

A *path* is a sequence of trajectories and contours. In this context, a *trajectory* is a connected sequence of *path commands*. Path commands include line segments, Bézier curve segments, and partial elliptical arcs. Each path command has an associated set of numeric parameters known as *path coordinates*. When a pair of path coordinates defines a 2D (x, y) location, this pair is a *control point*. Intuitively a trajectory corresponds to pressing a pen's tip down on paper, dragging it to draw on the paper, and eventually lifting the pen.

A *contour* is a trajectory with the same start and end point; in other words, a closed trajectory. These contours and trajectories may be convex, self-intersecting, nested in other contours, or may intersect other trajectories/contours in the path. There is generally no bound on the number of path segments or trajectories/contours in a path. For a rendering "primitive," paths can be quite complex.

Paths are rendered by either *filling* or *stroking* the path. Conceptually, path filling corresponds to determining what points (framebuffer sample locations) are logically "inside" the path. Stroking is roughly the region swept out by a fixed-width pen—centered on the trajectory—that travels along the trajectory orthogonal to the trajectory's tangent direction.

1.2 History, Standards, Motivation, and Contributions

Seminal work by Warnock and Wyatt [1982] introduced a coherent model for path rendering. Since that time, many standards and programming interfaces have incorporated path rendering constructs

*e-mail: {mjk,jbolz}@nvidia.com

into their 2D graphics framework. Without being exhaustive, we note

- *document presentation and printing*: PostScript [Adobe Systems 1985], PDF [Adobe Systems 2008a]
- *font specification*: PostScript fonts [Adobe Systems 1992]
- *immersive web*: Flash [Adobe Systems 2008b], HTML 5's Scalable Vector Graphics [SVG Working Group 2011a]
- *2D programming interfaces*: OpenVG [Khronos Group 2008]
- *productivity software*: Illustrator, Photoshop, Office

Despite path rendering's 30 year heritage and broad adoption, it has not benefited from acceleration by graphics hardware to anywhere near the extent 3D graphics has. Most path rendering today is performed by the CPU with sequential algorithms, not particularly different from their formulation 30 years ago. Our motivation is to harness *existing* GPUs to improve the overall experience achievable with path rendering.

We present a productized system for GPU-accelerated path rendering in the context of the OpenGL graphics system; see some of our rendering results in Figure 1. Our system works on the three most-recent architectural generations of GeForce and Quadro GPUs—and we expect all recent GPUs can support the algorithms and programming interface we describe.

The primary contributions delivered by our system are:

- A novel "stencil, then cover" programming interface for path rendering, well-suited to acceleration by GPUs.
- Our programming interface's efficient implementation within the OpenGL graphics system to avoid CPU bottlenecks.
- Accompanying algorithms to handle tessellation-free stenciled stroking of paths, standard stroking embellishments such as dashing, clipping paths to arbitrary paths, and mixing 3D and path rendering.

Section 2 reviews prior path rendering systems. Section 3 explains our approach; we cite the crucial prior research that our approach integrates in this section. Section 4 compares our quality and performance to other implementations and highlights our system's novel ability to mix with 3D and GPU-shaded rendering. Section 5 discusses opportunities for future work.

1.3 New Demands on Path Rendering

Historically, applications mostly "pre-render" 2D content specified with paths into bitmaps for glyphs and icons/images for vector artwork, then cache and blit those rasterized results as needed. Rendering directly from the path data generally proved too slow to be viable. Early window systems based on path rendering concepts such as Sun's NeWS [Gosling et al. 1989] and Adobe's Display PostScript [Adobe Systems 1993] were arguably overly ambitious in basing their 2D rendering model around path rendering rather than resolution-dependent 2D bitmap rendering as did the more successful GDI and X11-based systems that proved easier for 2D graphics hardware to accelerate.

1.3.1 Increasing Screen Density and Resolution

Smart phones and tablets have created new platforms free from legacy display limitations such as relatively low—by today's available technology—display density (measured in *dots-per-inch* or DPI) and the dated visual appearance established by resolution-dependent bitmap graphics. Apple's new iPad display has a display density of 264 DPI, greatly surpassing the 100 DPI density norm for PC screens. These handheld devices are carried directly on one's person so their screen real estate is relatively fixed—so improvements in display appearance is likely to be through increasing screen density rather than enlarging screen area.

Pixel resolutions for conventional monitors are increasing too. Large 2560x1600 resolution screens are mass-produced and readily available. Driving such high resolutions with CPU-based path rendering schemes is untenable at real-time rates. Indeed the very heterogeneity of modern displays in terms of pixel resolution, screen size, and their combination—pixel density—strengthens the case for improving path rendering performance.

1.3.2 Multi-touch Interfaces

Mobile devices also rely on multi-touch screens for input so the user is extremely aware of the latency between touch gestures and the resulting screen update. The user is literally pointing at the pixels they expect to see updated. Multi-touch encourages rotation and scaling. When imagery can easily be rotated, scaled, sub-pixel translated, and even projected, assumptions that all text and graphics will be orthographically aligned to the screen's pixel grid are no longer a given so rendering all path content directly from paths makes sense.

1.3.3 Immersive Web Standards

The proximate HTML 5 web standard exposes path rendering functionality in a standard and pervasive way through both Scalable Vector Graphics (SVG) and the Canvas element.

JavaScript performance has increased to the point that dynamic content can be orchestrated within a standards-based HTML 5 web page such that the system's path rendering performance is often a bottleneck.

1.3.4 Power Wall

Minimizing power consumption has become a mantra for computer system design across all classes of devices—whether mobile devices or not. When power is at a premium, moving CPU- and bandwidth-intensive computations such as pixel manipulation and rasterization to more power-efficient GPU circuitry can reduce overall power consumption while improving interactivity and minimizing update latency. GPU-acceleration of path rendering is precisely such an opportunity.

2 Prior Path Rendering Systems

2.1 CPU-based Path Rendering Systems Critiqued

Path rendering is historically and still typically performed by CPU-based scan line renderers. Paths are converted into a set of edges. These edges are transformed and sorted in Y-order. Then the edges are intersected with each scan line, sorted in X-order, and pixels in the path region are updated.

The scan-line rendering approach is notable for being work-efficient and cache-friendly. No computation is expended on pixels that are obviously outside the path and only active edges are considered when processing a given scan line. Such scan line renderers use a "chunker" strategy—where rather than the chunk being a 2D tile, the chunk is a single scan line. This leads to a reasonably friendly access pattern for CPU caches. Additionally the scan

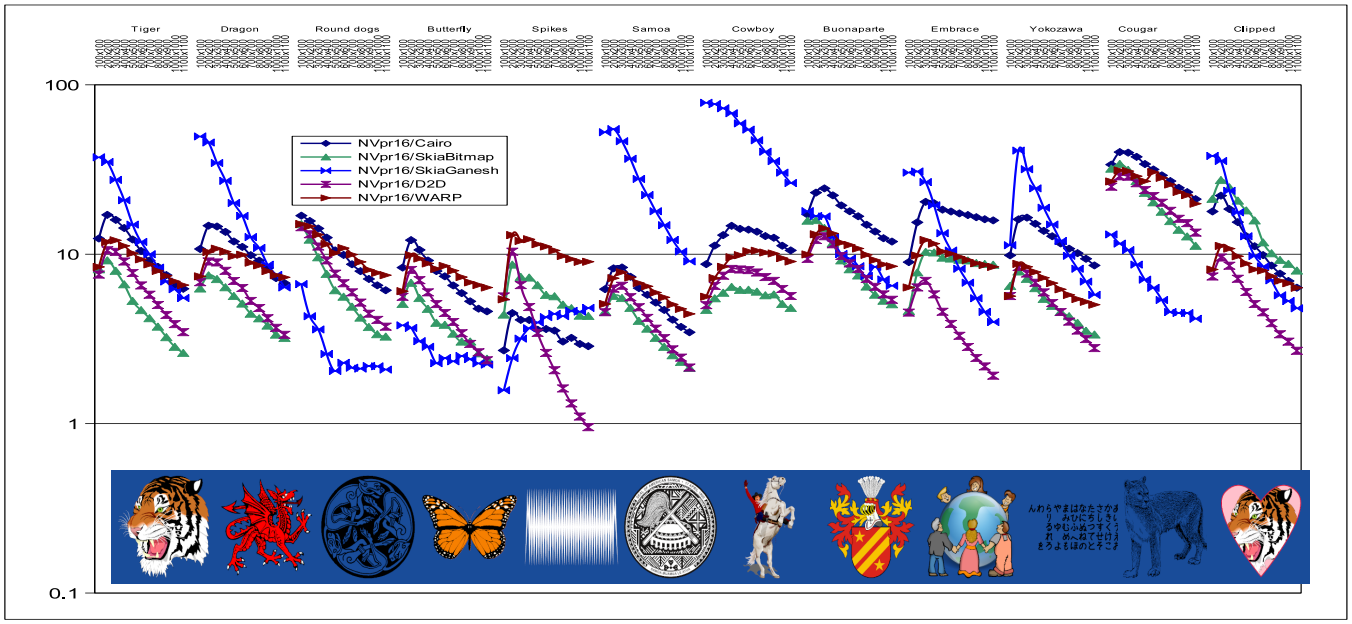

Figure 2: *Performance ratios rendering SVG content at window resolutions from* 100^2 *to* 1100^2. *A ratio of 1.0 means the* `NV_path_rendering` *(16 samples per pixel) performance is equal to the other renderer; higher ratios indicate how many multiples faster* `NV_path_rendering` *is than the alternative. Note the logarithmic Y axis. Scenes were selected for their variety. Benchmark configuration is a GeForce 650 and fast Core i7 CPU.*

line enter/leave counts are transient. In contrast to a window-sized ancillary buffer such as a depth or stencil buffer, the scan line enter/leave counts can live in the cache and have their storage recycled for each processed scan line.

While work-efficient *and* cache-friendly as noted, this CPU-intensive approach is quite sequential. Every path must be transformed into screen space. Every path must be scan line rasterized. Every scan line must be intersected with the active edge list. Every sorted active edge list must be scanned left-to-right. There is not an easy way to pipeline all these tasks or exploit massive parallelism—such as is routine for GPU-accelerated 3D graphics. Hence this is an approach that maps well to the CPU but cannot be obviously accelerated in this form by the GPU.

2.2 GPU-based Path Rendering Systems

Over the years, many attempts have been made—with varying degrees of mixed success—to accelerate path rendering with GPUs. We postpone discussion of prior techniques for GPU rendering of curves with the stencil buffer to Section 3 since they are the basis for our approach.

2.2.1 Acceleration of Path Rendering Programming Interfaces

Cairo [Packard and Worth 2003] is an open-source path rendering implementation. An early attempt at GPU-acceleration called Glitz [Nilsson and Reveman 2004] has since been abandoned. Glitz operated at the level of the XRender [Packard 2001] extension so did not accelerate paths directly. Arguably, Glitz was a more GPU-assisted back-end than GPU accelerated. More recently, Cairo has worked on a first-class GPU back-end but the immediate mode nature of the Cairo API and converting CPU-transformed paths to spans limits the acceleration opportunities.

Microsoft's Direct2D [Kerr 2009] API is layered upon Direct3D. Direct2D operates by transforming paths on the CPU and then performing a constrained trapezoidal tessellation of each path. The result is a set of pixel-space trapezoids and additional shaded geometry to compute fractional coverage for the left and right edges of the trapezoids. These trapezoids and shaded geometry are then rasterized by the GPU. The resulting performance is generally better than entirely CPU-based approaches and requires no ancillary storage for multisample or stencil state; Direct2D renders directly into an aliased framebuffer with properly antialiased results. Direct2D's primary disadvantage is the ultimate performance is determined not by the GPU (doing fairly trivial rasterization) but rather by the CPU performing the transformation and trapezoidal tessellation of each path and Direct3D validation work.

Skia is the C++ path rendering API used by Google's Android and Chrome browsers. Skia has a conventional CPU-based path renderer but has recently integrated a new OpenGL ES2-accelerated back-end called Ganesh. Ganesh has experimented with two accelerated approaches. The first used the stencil buffer to render paths. Because of API overheads with this approach, this first approach was replaced with a second approach where the CPU-based rasterizer computes a coverage mask which is loaded as a texture upon every path draw to provide the GPU proper antialiased coverage. This hybrid scheme is often bottlenecked by the dynamic texture updates required for every rendered path.

The Khronos standards body worked to develop an API standard known as OpenVG with the explicit goal of enabling hardware-acceleration of path rendering (the *VG* stands for vector graphics). Various companies and research groups have worked to develop OpenVG hardware designs [FreeScale, Multimedia Applications Division 2010; Huang and Chae 2006; Kim et al. 2008] that, based on available descriptions, are fairly similar to the conventional CPU-based scan line rasterizer scheme, recast as a hardware unit. Reported performance levels are quite low compared to what we report.

2.2.2 Vector Texture Schemes

An unconventional approach to GPU-accelerating path rendering is cleverly encoding path content into GPU memory—typically as a texture—and then using a programmable shader essentially to decode the path content. Nehad and Hoppe [2008] and Qin [2009] adopt variations on this approach. While this approach has some interesting advantages such as being able to directly "texture map" 3D geometry with path rendered content, these approaches suffer from the need to preprocess a static path scene into a specific texture encoding. This makes this approach unsuitable for editable or dynamic path rendering. Additionally, many rendering approximations and authoring limitations are needed to make vector texture schemes tractable.

2.2.3 Discussion of Deficiencies

The norm for CPU-based path rendering systems is maintaining roughly 16 coverage samples per pixel (details vary). This creates a challenge for GPU-based schemes because GPUs often support 1, 2, 4, or 8 samples per pixel through multisampling. This often creates a situation where the GPU-accelerated path rendering is inferior to the CPU-based path rendering quality.

When path rendering schemes are layered upon existing OpenGL or Direct3D APIs, we have observed performance being limited by the state change rate of the underlying 3D API. Often path rendering can result in many state changes per path when scenes can easily consist of 100s or 1000s of paths. In this case, the API overhead can substantially limit the overall performance. Our experience studying prior approaches to using GPUs for path rendering indicates these approaches are often more GPU-assisted rather than GPU-accelerated, with this attributable to continuous CPU involvement or substantial CPU-based preprocessing.

3 Our Approach

In contrast to other systems for accelerating path rendering with GPUs, our approach *explicitly* reveals the coverage determinations. These determinations—for both filling and stroking—appear as stencil buffer updates. A crucial insight underlying our approach is never determining the boundary between the "inside" and "outside" of a stroke or fill. Instead, we rely on point-sampled determinations of whether a particular (x, y) framebuffer location is inside or outside the stroke or fill. For antialiasing, we rely on GPU multisampling to provide multiple sample coverage positions, each with its own sub-pixel stencil value.

3.1 Stencil, then Cover

We perform path rendering in two steps. This is not unique; *all* path rendering schemes involve two steps. The two steps may be "tessellate, then render tessellation" [Kerr 2009] or "intersect with scan line, then paint pixels" [Packard and Worth 2003] or "ray cast, then shade" [Nehab and Hoppe 2008] but each rendering of a path is inherently sequential in the sense that determining what pixels are covered must precede shading and blending those pixels.

What is novel in our approach is *explicitly* decoupling the two steps. We call our approach, with its two decoupled steps, "Stencil, then Cover" (StC). Rather than a single `DrawPath` operation that hides the two-step nature of path rendering within the implementation, an OpenGL application using our extension first "stencils" the path in the stencil buffer [Akeley and Foran 1995], then "covers" the path to shade it.

This explicitly decoupled approach has advantages not available in interfaces that appear to offer a one-step `DrawPath` command. Our two-step approach makes arbitrary path clipping, mixing with 3D graphics, programmable blend modes, and other novel path rendering usages possible.

3.2 Filling

3.2.1 Improvements to Prior Methods

Our approach to filling paths is inspired by the work of Loop and Blinn [2005] who developed an efficient fragment shader-based approach to determining whether or not an (x, y) sample is inside or outside a given quadratic or cubic Bézier hull. In the Loop-Blinn formulation, inexpensive arithmetic on interpolated texture coordinates provides a Boolean predicate which when true indicates the fragment's sample position is not inside the Bézier region.

Our approach is *not* the first time the stencil buffer has been utilized for stenciling paths. Kokojima et al. [2006] applied the Loop-Blinn scheme in conjunction with the stencil buffer to determine the winding number of TrueType glyph outlines. Kokojima et al. showed the general filled polygon algorithm of Lane et al. [1983]—subsequently popularized for use with the stencil buffer [Neider et al. 1993]—can naturally combine with the Loop-Blinn quadratic discard shader to determine the samples inside an arbitrarily complex TrueType outline. After stenciling each glyph into the stencil buffer, conservative geometry based on a convex hull or bounding box can test against the non-zero stencil values, shade those samples, and reset the stencil values back to an initial zero state.

Kokojima's approach does not immediately extend to cubic Bézier segments because the inside region within a cubic Bézier hull is not necessarily convex. Rueda et al. [2008] addressed this by providing simple topological strategies to subdivide cubic Bézier hulls using Bézier subdivision to guarantee convexity, but used an overly expensive discard fragment shader based on Bézier normalization rather than applying the Loop-Blinn cubic formulation.

Our approach to handling cubic Bézier segments builds on all this work by combining cubic Bézier convex subdivision rules with the Loop-Blinn cubic formulation. We also perform the discard shaders at sample-rate rather than pixel-rate for improved coverage determinations and antialiasing. We use interpolation at explicit sample positions and our target GPU's sample mask functionality to evaluate multiple samples within a pixel in a single shader instance.

PostScript, SVG, and other standards support partial circular and elliptical arcs so an additional discard shader, expressed in Cg, handles these cases:

```
void roundCoverage(float2 st : TEXCOORD0.CENTROID)
{
  if (st.s*st.s + st.t*st.t > 1) discard;
}
```

with the (s, t) texture coordinates assigned so (0,0) is centered at the origin of roundness to discard samples outside the arc region contained in a sequence of one or more polygonal hulls bounding such arcs.

3.2.2 Baked Form of Filled Paths

In order to render a filled path, we "bake" the path into a resolution-independent representation from which the path can be stenciled under arbitrary projective transforms. This baking process takes time linearly proportional to the number of path commands. The resulting baked path data resides completely on the GPU. The required GPU storage is also linearly proportional to the number of

Figure 3: *Filled path, with control points, with anchor geometry, and with cubic Bézier discard hulls, and conservative cover geometry.*

path commands. For a static path, the baking process needs to be done just once; the baking process must be repeated if the path's commands or coordinates change, but edits to the path, including insertions and deletions of commands, require just re-baking the path segments at or immediately adjacent to the edits.

Once baked, a filled path is reduced to five sets of primitives:

1. Polygonal anchor geometry (structured as triangle fans), rendered with no shader.
2. Quadratic discard triangles, rendered with a Loop-Blinn quadratic discard shader.
3. Cubic discard triangles (if the cubic Bézier hull is a triangle) and quadrilaterals, rendered with a Loop-Blinn cubic discard shader.
4. Arc discard triangles, rendered with the `roundCoverage` discard shader shown above.
5. Conservative covering geometry, typically a triangle fan or quadrilateral.

Primitive sets #1 through #4 are rendered during the stencil fill step. Two-sided stencil testing increments non-discarded stencil samples of front-facing primitives; back-facing primitives instead decrement non-discarded stencil samples. Primitive set #5 is rendered during the cover fill step. Primitive sets #2 through #4 have properly assigned texture coordinates that drive each set's respective discard shader. Figure 3 visualizes the baked anchor, discard, and cover geometry.

This approach to path filling is theoretically sound because the stencil rendering reduces to a winding number computation consistent with a discrete formulation of Jordan's Theorem [Fabris et al. 1997].

All the data for a baked path can be stored within a single allocation of GPU memory to minimize the expense of stenciling or covering the path. Because the baked representation is completely resolution-independent, robust, and entirely on the GPU, the CPU overhead to launch the stenciling and/or covering of an already baked path object is minimal.

We implement our approach as an OpenGL extension named `NV_path_rendering` [Kilgard 2012]. Performing the stencil and cover steps within the graphics driver avoids the API and driver validation overhead (see Section 4.2) that plagued other GPU-based approaches. Figure 4 shows how our new path pipeline co-exists with the existing pipelines in OpenGL for pixel and vertex processing.

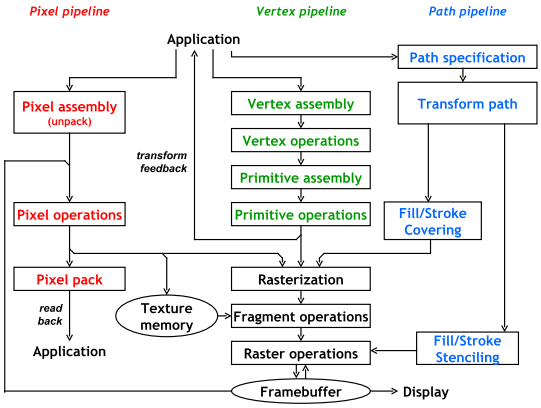

Figure 4: *High-level data flow of OpenGL showing pixel, vertex, and new path pipelines.*

3.3 Stroking

Our stroking approach operates similarly to our filling approach whereby we stencil, then cover stroked paths from a baked resolution-independent representation residing on the GPU that requires minimal CPU overhead to both stencil and cover.

3.3.1 Quadratic Bézier Stroking

Analytically determining the points contained by a stroke curved segment is not easy. The boundary of the stroked region of a quadratic Bézier corresponds to an offset curve. While the quadratic Bézier curve generating the offset curve is 2nd order, the offset curve for this generating segment's boundary is 6th order [Salmon 1960]. This makes exactly determining an intersection with this boundary unfeasible, particularly within the execution context of a GPU's fragment shader. The boundary becomes even more vexing for partial elliptical arcs and cubic Bézier segments. The boundary for a general cubic Bézier curve is 10th order [Farouki and Neff 1990]!

Quadratic Bézier Segment Point Containment Hence our approach involves simply determining if a given (x, y) point is inside or outside the stroked region of a quadratic Bézier segment. This can be reduced to solving a 3rd order equation.

A quadratic Bézier segment Q—defined by the segment's three control points C_0, C_1, and C_2—can converted to monomial form $Q(t) = At^2 + Bt + C = 0$ for $t \in [0, 1]$. A point P is judged within the stroke of Q when there is a parametric value s on Q such that $Q'(s) \cdot (P - Q(s)) = 0$ and the squared distance between $Q(s)$ and P is within the squared stroke radius. (The dot product of a quadratic function and the derivative of a quadratic function is 3rd order.) Intuitively this corresponds to finding the 1 or 3 points $Q(s)$ with a tangent direction orthogonal to the segment connecting P and $Q(s)$. Such solutions s will be local minima or maxima for the distance between P and points on Q so computing the squared distance $d = (Q(s) - P) \cdot (Q(s) - P)$ for each solution s indicates if P is within the stroke of $Q(t)$ for $t \in [0, 1]$ when both $s \in [0, 1]$ and d is less than or equal the square of half the path's stroke width. Figure 5 visualizes this procedure.

Solving the cubic equation at every rasterized sample is expensive, but the computation can be simplified somewhat. The cubic equation can be rearranged into an easier-to-solve depressed cubic [Cardano 1545] of the form $t^3 + G(x, y)t + H(x, y) = 0$. While the

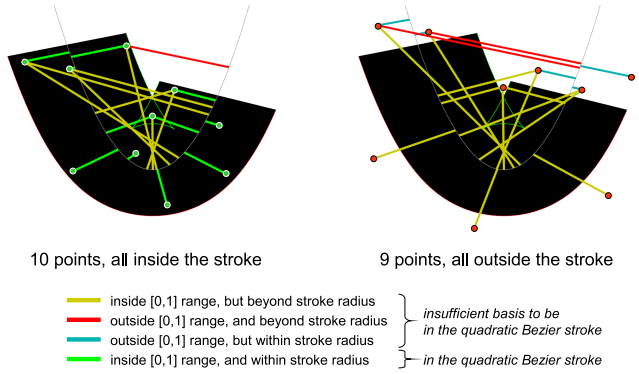

Figure 5: *Visualization of points within and outside the stroked region of a quadratic Bézier segment and their basis for inclusion or not.*

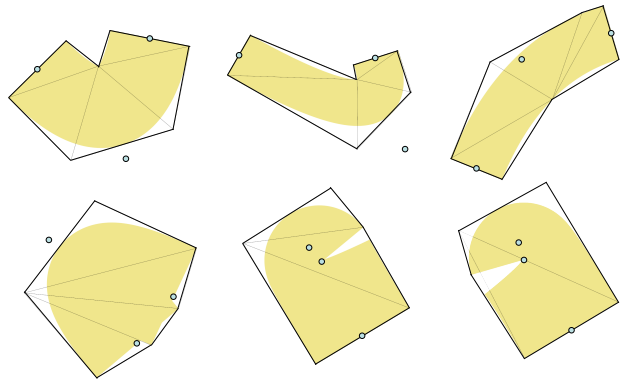

Figure 6: *Examples of concave (top row) and convex (bottom row) stroked quadratic Bézier segment hulls.*

coefficients $G(x, y)$ and $H(x, y)$ are different for every path-space (x, y) location, the functions G and H are linear in terms of (x, y) so a vertex shader can evaluate $G(x, y)$ and $H(x, y)$ at hull positions and exploit the GPU's ability to interpolate linearly G and H at positions within the hull.

Care is taken when an arrangement of quadratic Bézier control points is collinear, collocated, or very nearly so. In such cases, we demote the quadratic Bézier segment to its linear degenerate equivalent for robustness.

Stroked Quadratic Segment Hull Construction To harness this approach for rendering, we construct a hull around the quadratic Bézier stroked segment. As shown in Figure 6, the hull is typically concave, consisting of seven vertices—though the hull may be convex when the quadratic stroke's width is wide relative to its arc length. Ruf [2011] has addressed the problem of a tight bounding representation for quadratic strokes, but his approach involves parabolic edges with the assumption the CPU can evaluate such edges efficiently; for our purposes, we want a triangular decomposition of the hull suitable for GPU rasterization.

While solving the cubic equation—even in depressed form—is expensive, we note that stroked regions are typically small and narrow in screen space so this expensive process is used sparingly in practice. Even when strokes are wide, the massively parallel nature of the GPU makes this approach quite fast. Most important to us, once a quadratic stroke is "baked" for rendering, it can be rendered under an arbitrary linear transformation—including projection—without any further CPU re-processing. The hull vertices and their coefficients for G and H can be stored in GPU memory so that stencil-only rendering the quadratic stroke involves simply configuring the appropriate buffers, the appropriate vertex and fragment shader pair, and rendering the hull geometry of the quadratic stroke.

Higher-order-than-Quadratic Stroking Path rendering standards incorporate cubic Bézier segments and partial elliptical arcs; these involve cubic and rational quadratic generating curves for rasterized offset curve regions. The direct evaluation approach applied to generating quadratic Bézier curves is not tractable.

Instead we subdivide cubic Bézier segments and partial elliptical arcs into an approximating sequence of quadratic Bézier segments. To maintain a curved boundary appearance at all magnifications, our subdivision approach maintains G^1 continuity at quadratic Bézier segment boundaries. No matter how much you zoom into the boundary of higher-order stroked segments, there is never any sign of linear edges or even a false discontinuity in the curvature.

Following the approach of Kim and Ahn [2009], we bound the subdivision such that the true higher-order generating curve never escapes a specified percentage threshold of the stroke width of the approximating quadratic stroke sequence. We also subdivide at key topological features, specifically points of self-intersection and minimum curvature.

3.3.2 Stroking Embellishments

Stroking of line segments, end caps, and joins is straightforward. Stroked line segments are drawn as stencil-only rectangles. Polygon caps (square and triangular) and joins (bevel and miter) are likewise drawn as stencil-only triangles. This geometry can be drawn without any fragment shader. Round caps and joins are drawn with the same `roundCoverage` stencil-only sample-rate discard shader (Section 3.2.1) used for partial circular and elliptical arcs for filling with the (s, t) texture coordinates assigned appropriately to discard samples outside the circular region of the round cap or join. The baking process for stroked paths includes generating the rectangles and triangles for line segment and polygonal caps and joins. Geometry for round caps and joins is generated along with the texture coordinates to drive the round coverage discard shader.

3.3.3 Dashing

Dashing is a feature of all major path rendering standards except Flash. Dashing complicates stroking by turning on and off the stroking along a path based on an application-specified repeating on-off pattern specified in units of arc length. Our stroke baking process applies the dash pattern while gathering the geometry for the stroked path. While complicated in its details, our dashing process is similar to other path rendering implementations in its high-level structure. The primary difference is curved path segments are reduced to quadratic Bézier segments in our approach instead of line segments. Whereas the arc length computations in conventional path rendering systems typically involve recursive subdivision until the curved segment approximates a line segment, our approach can stop subdividing at quadratic Bézier segments. Unlike higher order curves, the arc length of a quadratic Bézier segment (a

segment of a parabola) has a closed form analytical solution:

$$\int_0^1 \sqrt{Q_x(t)^2 + Q_y(t)^2}dt = \frac{\ln(\frac{b+2\sqrt{ac}}{b+2c+2\sqrt{c(c+a+b)}})\left(b^2-4ac\right)+2(b+2c)\sqrt{c(c+a+b)}-2b\sqrt{ac}}{8c^{3/2}}$$

with copious common subexpressions and where $a = B \cdot B$, $b = 2B \cdot C$, and $c = C \cdot C$. Our interest in this approach is our desire to minimize use of expensive recursive subdivision algorithms while baking stroked paths, particularly during dashing. Some numerical care must be taken to avoid negative square roots, negative logarithms, and division by zero, but these cases occur when quadratic segments are nearly linear.

Our dashing approach results in a resolution-independent baked form of the dashed stroked path. Once dashed and baked, no further CPU-based processing is necessary to render dashed paths. This is in contrast to other implementations of dashed stroking where dashing has a considerable CPU processing expense during rendering. While our implementation must of course represent each segment resulting from dashing, our render-time algorithm is completely oblivious to whether the original path was dashed.

3.3.4 Baked Form of Stroked Paths

Once baked, a stroked path is reduced to four sets of primitives:

1. Polygonal geometry (line segments, bevel and miter joins, square and triangular end caps) with no shader.
2. Triangle fans corresponding to quadratic Bézier segment hulls (curved path segments), rendered with a stroked quadratic discard shader.
3. Triangle fans corresponding to round hulls (round end caps and joins) rendered with a round coverage shader.
4. Conservative covering geometry, typically a triangle fan or quadrilateral.

Primitive sets #1 through #3 are rendered during the stencil stroke step. Primitive set #4 is rendered during the cover stroke step.

The `REPLACE` stencil operation used for stroking is order-invariant. Therefore we select a static rendering order during the baking process that minimizes GPU state changes during rendering.

The geometry, texture coordinates, and per-hull quadratic discard shader coefficients are all packed into a single GPU buffer allocation. The rendering process for stenciling the baked path is very straightforward, requiring no more than three GPU state reconfigurations, one per primitive set above.

The GPU storage for the linear and quadratic path segments in a baked stroked path is linearly proportional to the number of segments (post-dashing). For cubic Bézier segments and partial elliptical arcs, the storage depends on their required level of subdivision. Because this subdivision is tied to the stroke width, narrower stroke widths require more storage while wider stroke widths require less storage.

3.4 Clipping to Arbitrary Paths

All major path rendering standards support clipping a *draw* path to the filled region of a *clip* path. Our two-step "stencil, then cover" approach readily supports clipping to arbitrary paths. We briefly describe the process assuming an 8-bit stencil buffer, initially cleared to zero:

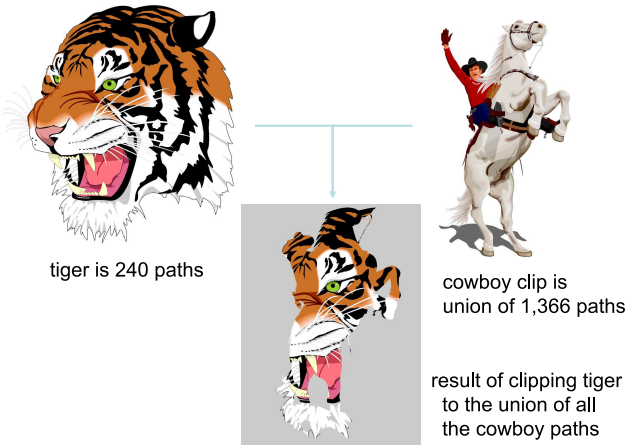

Figure 7: *Complex clipping scenario. Our approach: 8.69ms @ 1000x1000x16. Cairo: 909ms @ 1000x1000. System: Core i7 + GeForce 560M GPU.*

1. Stencil the clip path into the stencil buffer with a "stencil fill" operation.
2. Perform a "cover fill" operation to coalesce the samples matching the fill rule so that the most-significant stencil bit is set and all the lower bits are cleared. For example, if a sample's stencil value is non-zero, replace the stencil value with 0x80. This step updates only the stencil buffer (disable any color writes).
3. Stencil the draw path into the stencil buffer with a "stencil fill" operation, but (a) modify only the bottom 7 bits of the stencil buffer, and (b) discard any rasterized samples without the topmost bit of the stencil buffer set.
4. Perform a shaded "cover" operation on the draw path. Update any color sample whose stencil value's bottom 7 bits are non-zero and zero the bottom 7 bits of the sample's stencil value. Write shaded color samples during this step; due to the stencil configuration, only samples within both the clip and draw paths get shaded and updated.
5. Finally to undo the clip path's stencil manipulation from step 1, perform a "cover" operation on the clip path to reset the most significant stencil bit back to zero.

Many variations on this approach are possible. For example, steps 3 and 4 can be repeated for each path in a layered group of paths. This avoids having to re-render the clip path for each and every path in a group of paths.

Most standards allow nested clipping of paths to other paths. Clever manipulation of the stencil bit-planes allows such nested clipping. Standards such as SVG allow for clipping to the union of an arbitrary number of paths as shown in Figure 7. Again, we can accomplish this by clever use of stencil bit-planes and re-coalescing coverage from different clip paths.

3.5 Painting

What path rendering standards often call "painting" a filled or stroked path is called shading in 3D graphics. Our goal is to allow the full generality of GPU-based programmable shading to be exposed when painting paths.

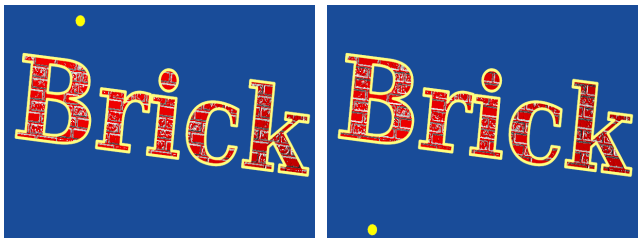

Figure 8: *Bump map shader applied to path rendered text, rendered from to different light positions, shown in yellow.*

During the cover step where a conservative bounding box or convex hull is rendered to cover fully the stenciling of the path, the application can configure arbitrary OpenGL shading. This could be fixed-function shading, assembly-level shaders, or shaders written in a high-level language such as Cg or GLSL.

In conventional path rendering systems, linear and radial gradients are a common form of paint for paths. We note how straightforward radial gradient paint can be implemented, including mipmapped filtering of the lookup table accesses, with the following Cg shader:

```
void radialFocalGradient(float3 str : TEXCOORD0.CENTROID,
                         float4 c   : COLOR.CENTROID,
                    out float4 color   : COLOR,
                uniform sampler1D ramp : TEXUNIT0)
{
  color = c*tex1D(ramp, length(str.xy) + str.z);
}
```

The texture coordinates needed for this shader can be generated as a linear function of the path-space coordinate system. Painting need not be limited to conventional types of path rendering paint. Arbitrary fragment shader processing can be performed during the cover step (see Figure 8).

3.6 Blending and Blend Modes

OpenGL blending is sufficient for most path rendering where the default path compositing operation is the "over" blend mode, assuming pre-multiplied alpha. Color writes during our cover step apply the currently configured OpenGL blend state. Modern GPUs also have efficient first-class support for blending in the widely-used sRGB device color space.

Sophisticated path rendering systems have additional blend modes [SVG Working Group 2011b] beyond the standard Porter-Duff compositing algebra [1984]. Digital artists are familiar with these modes with names such as *ColorDodge*, *HardLight*, etc. However GPU blending does not support these blend modes because they are rare, complex, and not used by 3D graphics. While some of these blend modes can be simulated with multiple rendering passes, many of these modes are impossible to construct from conventional GPU blending operations.

Our "stencil, then cover" approach makes it possible to implement these blend modes despite their lack of direct GPU hardware support. Normally, GPUs do not reliably support reading-as-a-texture a framebuffer currently being rendered. However a recent OpenGL extension called `NV_texture_barrier` [Bolz 2009] provides a reliable memory barrier under restrictive conditions. A fragment shader must ensure there is a single read and write for any particular pixel done from that pixel's fragment shader instance.

The "stencil, then cover" approach provides precisely such a "no double blending" guarantee. So by preceding each cover operation (whether fill or stroke) with an OpenGL `glTextureBarrierNV` command and reading the pixel's color value as a fetch to the framebuffer bound as a texture, reliable programmable blending with the fragment shader is possible.

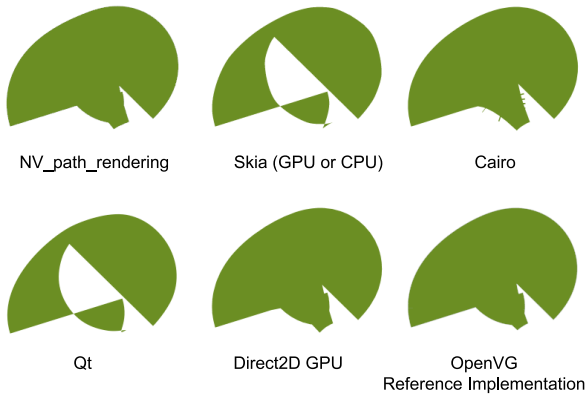

Figure 9: *Various path rendering implementations drawing a difficult cubic Bézier curve (the centurion head).*

4 Discussion

4.1 Quality

Our system's rendering quality is directly tied to how many color and stencil samples the framebuffer maintains per pixel. This determines the quality of our antialiasing. Our GeForce GPUs support up to 16 samples per pixel while our Quadro GPUs support 32 and 64 samples per pixel as well.

At 16 samples per pixel, our rendering quality compares quite favorably with CPU-based path renderers. Because our GPUs have 8 bits of sub-pixel precision, irregular coverage sample positions, and our point containment determinations are numerically sound, we are well-justified in stating our quality exceeds what can reasonably be expected for CPU-based path renderers. We focus on two aspects of path rendering quality where our implementation has superior quality.

4.1.1 Stroking Quality

For stroking, our quadratic Bézier stroke discard shader is mathematically consistent with the sweep of an orthogonal pen traversing the path's trajectory. In Figure 9 we compare our very fast stroking result to alternatives that are generally substantially slower on a difficult cubic Bézier stroke test case. Notice three path rendering implementation get this test case quite wrong—whereas `NV_path_rendering` matches the OpenVG reference implementation and Direct2D version.

4.1.2 Conflation Avoidance

Conflation is an artifact in path rendering systems that occurs when coverage (a Boolean concept) is *conflated* with opacity. This generally occurs when sub-pixel coverage is converted to a fractional value and multiplied into the alpha color component for compositing. While this approach is standard practice, it can result in noticeably incorrect colors.

Conflation is particularly noticeable when two opaque paths *exactly* seam at a shared boundary. Say path A covers 40% of the pixel and an adjacent path B covers the other 60%. But if A is drawn first,

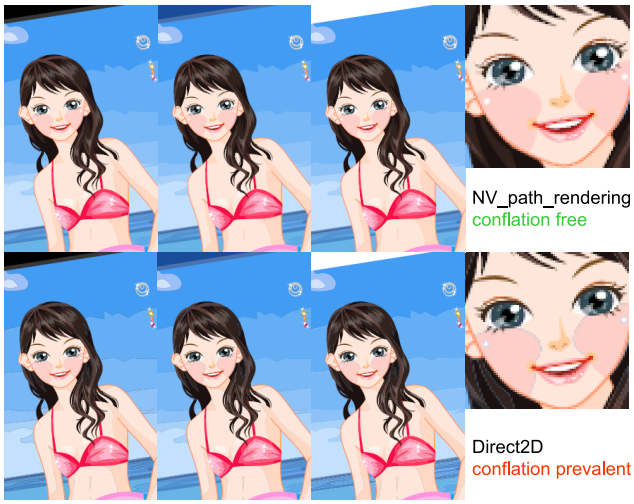

Figure 10: *Flash scene with shared edges.* `NV_path_rendering` *shows no conflation while Direct2D (and Cairo, Skia, Qt, and OpenVG) shows conflation. Upper left corner shows the background clear color; conflation is tinted by this color in the bottom scenes. Notice the conflated blue tint on the girl's cheek.*

the pixel picks up 40% of A's color and 60% of the background color. Now when B is drawn, the pixel gets 60% of B's color and 40% of the combination of 40% of A's color and 60% of the background color. The result is some fraction of the background color has leaked into the pixel when a more accurate assessment of coverage would have no background color.

Flash content is particularly prone to conflation artifacts because path edges are typically authored for exact sharing of edges. Adobe's Flash player specifically works to avoid conflation artifacts. This is possible because Flash player has complete knowledge of all the paths in a Flash shape and how those path edges are shared. Exact sharing of edges is helpful from a content creation standpoint because a shared edge can be stored once and used by two paths (more compact) and reduces the overall layered depth complexity of the scene by avoiding overlaps. Because `NV_path_rendering` maintains distinct sub-pixel color samples, the scene in Figure 10 renders free of conflation artifacts.

4.2 Performance

The rendering performance of `NV_path_rendering` scales with GPU performance. Because the baked paths reside on the GPU and are resolution-independent, once baked, path rendering performance is decoupled from CPU performance. Figure 2 charts the performance of `NV_path_rendering` relative to alternatives—including GPU-accelerated alternatives such as Direct2D and Skia's Ganesh approach.

Our performance advantage is attributable to the overall rendering and shading performance of our underlying GPUs. Several aspects are particularly noteworthy. Our underlying GPUs support a fast stencil culling mode so hundreds of pixels can be culled in a single clock if a coarse grain test can determine the stencil test for all the pixels would fail. This mitigates much of what might otherwise seem very inefficient about the "stencil, then cover" approach. Also stencil processing generally is very well optimized. The 8-bit memory transactions during the stencil and cover steps can often run at memory bus saturating rates. Our OpenGL driver

Figure 11: *Mixing 3D and path rendering in a single window.*

implementation makes use of a configurable front-end processor within the GPU—not otherwise accessible to applications—to transition quickly between the stencil step and cover step and back. This avoids the driver performing expensive revalidations of CPU-managed state so our rendering stays GPU-limited rather than CPU-bottlenecked, even when presented with otherwise overwhelming numbers of small paths.

4.3 New Functionality

Because `NV_path_rendering` is integrated into the OpenGL pipeline and the coverage information is accessible through the stencil buffer, we are able to implement unconventional algorithms such as mixing path rendering with arbitrary 3D graphics.

Figure 11 demonstrates an example of this capability. No textures are used in this scene. Arbitrary zooming into the tigers' detail is supported. Notice how the tigers properly occlude each other and the teapot. Due to the perspective 3D view, the path rendering is properly rendered in perspective as well.

5 Future Work

We believe our performance can be further improved. We are investigating hardware improvements to mitigate some of the memory bandwidth expense involved in our underlying stencil-based algorithms. In particular, we are seeking to reduce the GPU memory footprint.

Web browser architecture should change to incorporate GPU-accelerated path rendering. Today web browsers respecify paths every time a web page with path content is re-rendered assuming respecifying paths is cheap relative to the expense of rendering them. When path rendering is fully GPU-accelerated, a retained model of rendering is more appropriate and efficient. We believe web browsers should behave more like video games in this respect to exploit the GPU.

Mobile devices are power constrained so off-loading path rendering to a graphics processor designed for efficient pixel processing makes good sense. Mobile devices in particular prize a low-latency experience for the user so the sooner the device can complete its resolution-independent 2D rendering, the better the user experience and the sooner the device can power down to a low power state.

Acknowledgements

Michael Toksvig corrected Mark's 3D bigotry and insisted 2D rendering deserved acceleration. Chris Dalton assisted building our test bed. Tero Karras provided crucial math insights. Barthold Lichtenbelt supported this work throughout.

References

ADOBE SYSTEMS. 1985. *PostScript Language Reference Manual*, 1st ed. Addison-Wesley Longman Publishing Co., Inc. 2

ADOBE SYSTEMS. 1992. *Adobe Type 1 Font Format*, 2nd ed. Addison-Wesley Longman Publishing Co., Inc. 2

ADOBE SYSTEMS. 1993. *Display PostScript System–Introduction: Perspective for Software Developers*. 2

ADOBE SYSTEMS. 2008. *Document management–Portable document format–Part 1: PDF 1.7*. Also published as ISO 3200. 2

ADOBE SYSTEMS. 2008. *SWF File Format Specification, version 10*. 2

AKELEY, K., AND FORAN, J., 1995. Apparatus and method for controlling storage of display information in a computer system. US Patent 5,394,170. 4

BOLZ, J., 2009. `NV_texture_barrier`. http://www.opengl.org/registry/specs/NV/texture_barrier.txt . 8

CARDANO, G. 1545. *Artis magnae sive de regulis algebraicis, liber unus*. 5

FABRIS, A., SILVA, L., AND FORREST, A. 1997. An efficient filling algorithm for non-simple closed curves using the point containment paradigm. In *Proceedings of X Brazilian Symposium on Computer Graphics and Image Processing*, 2 –9. 5

FAROUKI, R., AND NEFF, C. 1990. Algebraic properties of plane offset curves. *Computer Aided Geometric Design 7*, 101–127. 5

FREESCALE, MULTIMEDIA APPLICATIONS DIVISION. 2010. i.MX35 accelerated 2D graphics: Optimizing 2D graphics with OpenVG and i.MX35, application note, doc. # an3975. 3

GOSLING, J., ROSENTHAL, D. S. H., AND ARDEN, M. J. 1989. *The NeWS book: an introduction to the network/extensible window system*. Springer-Verlag. 2

HUANG, R., AND CHAE, S.-I. 2006. Implementation of an OpenVG rasterizer with configurable anti-aliasing and multi-window scissoring. In *Proceedings of the 6th IEEE International Conference on Computer and Information Technology*, IEEE Computer Society, CIT '06, 179. 3

KERR, K. 2009. Introducing Direct2D. *MSDN Magazine* (June). 3, 4

KHRONOS GROUP, 2008. OpenVG specification version 1.1. 2

KILGARD, M., 2012. `NV_path_rendering`. http://www.opengl.org/registry/specs/NV/path_rendering.txt . 5

KIM, Y., AND AHN, Y. 2009. Explicit error bound for quadratic spline approximation of cubic spline. *Journal of the Korean Society for Industrial and Applied Mathematics 13*, 4, 257–265. 6

KIM, D., CHA, K., AND CHAE, S.-I. 2008. A high-performance OpenVG accelerator with dual-scanline filling rendering. *Consumer Electronics, IEEE Transactions on 54*, 3 (August), 1303 –1311. 3

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*, SIGGRAPH '06. 4

LANE, J. M., MAGEDSON, R., AND RARICK, M. 1983. An algorithm for filling regions on graphics display devices. *ACM Trans. Graph. 2*, 3 (July), 192–196. 4

LOOP, C., AND BLINN, J. 2005. Resolution independent curve rendering using programmable graphics hardware. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, 1000–1009. 4

NEHAB, D., AND HOPPE, H. 2008. Random-access rendering of general vector graphics. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, 135:1–135:10. 4

NEIDER, J., DAVIS, T., AND WOO, M. 1993. *OpenGL Programming Guide, 1st edition*. See "Drawing Filled, Concave Polygons Using the Stencil Buffer", 398–399. 4

NILSSON, P., AND REVEMAN, D. 2004. Glitz: hardware accelerated image compositing using OpenGL. In *Proceedings of the FREENIX Track: 2004 USENIX Annual Technical Conference*, 29–40. 3

PACKARD, K., AND WORTH, C. 2003. A realistic 2D drawing system. A rejected *SIGGRAPH 2003* paper submission☺. 3, 4

PACKARD, K. 2001. Design and implementation of the X Rendering Extension. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, USENIX Association, 213–224. 3

PORTER, T., AND DUFF, T. 1984. Compositing digital images. In *Proceedings of the 11th annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, 253–259. 8

QIN, Z. 2009. *Vector Graphics for Real-time Rendering*. PhD thesis. University of Waterloo. 4

RUEDA, A. J., RUIZ DE MIRAS, J., AND FEITO, F. R. 2008. GPU-based rendering of curved polygons using simplicial coverings. *Computer Graphics 32*, 5 (Oct.), 581–588. 4

RUF, E. 2011. An inexpensive bounding representation for offsets of quadratic curves. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, 143–150. 6

SALMON, G. 1960. *A Treatise on Conic Sections*. Chelsea New York (reprint). 5

SVG WORKING GROUP, 2011. Scalable Vector Graphics (SVG) 1.1 (2nd edition). 2

SVG WORKING GROUP, 2011. SVG compositing specification. W3C working draft March 15, 2011. 8

WARNOCK, J., AND WYATT, D. K. 1982. A device independent graphics imaging model for use with raster devices. In *Proceedings of the 9th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '82, 313–319. 1