Massively-Parallel Vector Graphics

Francisco Ganacim Rodolfo S. Lima Luiz Henrique de Figueiredo Diego Nehab
IMPA — Instituto Nacional de Matemática Pura e Aplicada

Abstract

We present a massively parallel vector graphics rendering pipeline that is divided into two components. The preprocessing component builds a novel adaptive acceleration data structure, the *shortcut tree*. Tree construction is efficient and parallel at the segment level, enabling dynamic vector graphics. The tree allows efficient random access to the color of individual samples, so the graphics can be warped for special effects. The rendering component processes all samples and pixels in parallel. It was optimized for wide antialiasing filters and a large number of samples per pixel to generate sharp, noise-free images. Our *sample scheduler* allows pixels with overlapping antialiasing filters to share samples. It groups together samples that can be computed with the same vector operations using little memory or bandwidth. The pipeline is feature-rich, supporting multiple layers of filled paths, each defined by curved outlines (with linear, rational quadratic, and integral cubic Bézier segments), clipped against other paths, and painted with semi-transparent colors, gradients, or textures. We demonstrate renderings of complex vector graphics in state-of-the-art quality and performance. Finally, we provide full source-code for our implementation as well as the input data used in the paper.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms I.3.7 [Computer Graphics]: Hardware Architecture—Parallel processing;

Keywords: vector graphics, rendering, parallel processing

Links: DL PDF WEB DATA CODE

1 Introduction

Vector graphics are one of the most traditional, prevalent, and versatile forms of visual information representation. They can specify, in a resolution-independent fashion, a wide variety of content, such as scalable fonts, pages of text and illustrations, maps, charts, diagrams, user interfaces, games, and even photo-realistic drawings. Despite recent developments in diffusion-based vector graphics [Orzan et al. 2008; Finch et al. 2011; Sun et al. 2012, 2014], the vast majority of resolution-independent content in use today follows the framework laid out in the seminal work of Warnock and Wyatt [1982].

A vector illustration is composed of multiple *paths* that define shapes to be *painted* in a given order. The outline of a shape is specified by a set of oriented closed *contours*. Each contour is a piecewise polynomial curve, defined by *control points* that specify linear, rational quadratic, or integral cubic Bézier *segments*. The interior of a shape is the set of points that satisfy the path's *fill rule* (*even-odd* or *non-zero winding*). Shapes with disconnected components, multiple holes, and self-intersections are allowed and frequently used in practice. Boolean operations between paths are also supported, in particular intersection between paths (i.e., general *clipping*). Furthermore, the path outlines can be *stroked* to produce thick lines, including dashing patterns, joins, and caps. (In our prototype implementation, strokes are not handled by the pipeline: they are converted to filled primitives in the CPU. Strings of text characters receive the same treatment.) Paints include solid colors, linear and radial gradients, and texture patterns. Semi-transparent paints cause overlapping paths to be composited [Porter and Duff 1984].

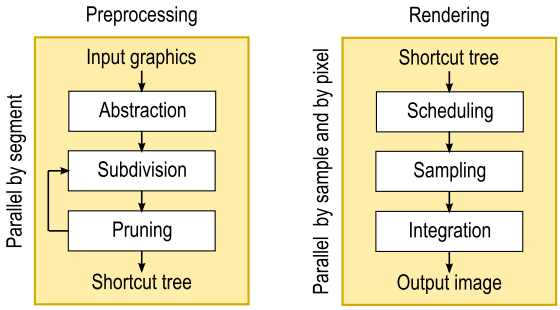


Figure 1: *Our vector-graphics rendering pipeline is divided into a preprocessing component that is parallel at the segment level, and a rendering component that is parallel at the pixel and sample levels.*

Input vector content comes in many similar formats [PostScript 1999; PDF 2006; SWF 2012; OpenXPS 2009; SVG 2011]. Regardless of format, any vector content must be *rendered* (i.e., *scan-converted* or *rasterized*) into an image at a chosen resolution before it can be printed or displayed on screen. Many tools are available for this purpose (e.g., Cairo, OpenVG, NV_path_rendering, Skia, Direct2D, Silverlight, Flash, GhostScript, MuPDF). All rendering algorithms must solve the same fundamental *point-in-shape problem*: identify the set of points that belong to each shape so they can be painted with appropriate colors [Haines 1994].

Although most algorithms exploit spatial coherence to reduce the amount of computation needed for rendering, previous methods involve serial components during processing. For example, shapes may be sequentially added to an acceleration data structure prior to rendering [Nehab and Hoppe 2008], or rasterized one after the other into the output image [Kilgard and Bolz 2012]. To take advantage of the processing power of modern GPUs, we designed our vector graphics rendering pipeline to be massively parallel at every stage. As shown in figure 1, it is divided into a *preprocessing* component that is parallel at the input segment level, and a *rendering* component that is parallel at output sample and pixel levels.

The preprocessing component creates the *shortcut tree*, a novel hierarchical acceleration data structure (i.e., a quadtree) that allows the color of each point in the illustration to be evaluated very efficiently. The *abstraction* stage converts all input segments into implicit and monotonic *abstract segments*. These can be intersected only once by horizontal rays, and can be efficiently queried for the *existence* of such intersections. The shortcut tree is created in breadth-first fashion. At each *subdivision* level, all segments in all cells marked for subdivision are processed in parallel and routed to the appropriate child cells. As subdivision progresses, the *pruning* stage considers all segments in parallel and eliminates those that have been clipped or entirely occluded by other paths. Each leaf cell in the shortcut tree contains a specialized representation of the illustration that includes

only a small fraction of input segments: those that are needed to solve the point-in-shape problem for the paths that contribute to the color of any sample that falls within the area of the leaf cell.

The rendering component finds the leaf cell that contains each sample, loads the appropriate cell contents, and performs the required computations to evaluate the sample colors. To enable efficient support for user-defined warps, the samples are *scheduled* so that those falling within the same leaf cell are grouped together. This allows the *sampler* to evaluate these samples in parallel, without control-flow divergence, while reusing the bandwidth required to load cell contents. Moreover, samples falling under the overlapping supports of antialiasing filters associated with neighboring pixels are evaluated only once and shared between them. *Integration* of sample colors happens in fast local memory, before pixels are written to global memory. This setup enables antialiasing filters with wide support (e.g., 4×4) and large sampling rates (e.g., 512 samples/pixel) for sharp, noise-free renderings.

Since both preprocessing and rendering are efficient and fully parallel, we can render complex illustrations surpassing state-of-the-art quality and performance.

Our contributions include:

- The *shortcut tree*, a novel hierarchical acceleration data structure that enables efficient random access to the color of each point in the illustration;
- *Fully parallel* construction of the shortcut tree, including novel subdivision and pruning algorithms;
- New *segment abstraction* that eliminates the need for *all* intersection computations throughout the pipeline. Conversion from input segments during preprocessing requires only monotonization, splitting at double and inflection points, and implicitization;
- The *flat clipping* algorithm that supports arbitrary nesting of clip-paths without resorting to recursion or even a stack.
- The pipeline renders *front-to-back* and aborts computation when full opacity is reached. This is done independently per sample;
- Support for user-defined warps, sharing samples under wide antialiasing filters, using large sampling rates, minimizing control-flow divergence as well as memory and bandwidth usage.

2 Related work and motivation

In order to render complex paths, Loop and Blinn [2005] start with a constrained triangulation of the control polygons that define the path. Internal triangles are rasterized normally. Boundary triangles employ a fragment shader that tests the sign of an implicit function to discard fragments that are outside of piecewise polynomial boundaries. Although we use the same implicitization method as Loop and Blinn [2005] (see also Salmon [1852]), the similarity between the two approaches ends there. Our pipeline manipulates segments directly and uses implicit tests exclusively to detect intersections with axis-aligned rays during preprocessing and rendering.

Kokojima et al. [2006] simplified the method of Loop and Blinn [2005] and made it more practical by replacing the constrained triangulations with line-edged triangle fans and the stencil buffer [Neider et al. 1993]. This idea was then adopted by Kilgard and Bolz [2012] for the NV_path_rendering [2011] pipeline. When there is potential spatial overlap, different paths *cannot* be rasterized in parallel, lest they interfere in the stencil buffer. Interestingly, in the original work by Loop and Blinn [2005], all paths *could* be drawn in parallel, at least in scenarios where the necessary constrained triangulations can be precomputed (e.g., in text). Unfortunately, a parallel, efficient, and general way to obtain such triangulations is not available.

The methods described so far fall into a category that exploits spatial coherence by subdividing complex input shapes into primitives that are easier to fill. Scanline-based algorithms follow the work of Wylie et al. [1967] and break shapes into horizontal spans that cover interior pixels. This is the approach followed by the Cairo and Skia renderers, and their parallelization to GPU architectures has so far met with little success. The trapezoidal decomposition used by Direct2D [Kerr 2009], the constrained triangulations used by Loop and Blinn [2005], and the triangulation+stencil method adopted by Kilgard and Bolz [2012] share the same underlying motivation. These methods tend to amortize the rendering cost across entire paths. In contrast, one of our goals was to support parallel access to the color of individual samples, as in *vector textures*.

Vector textures [Kilgard 1997; Frisken et al. 2000; Sen et al. 2003; Sen 2004; Ramanarayanan et al. 2004; Ray et al. 2005; Lefebvre and Hoppe 2006; Qin et al. 2006; Nehab and Hoppe 2008; Qin et al. 2008; Parilov and Zorin 2008; Rougier 2013] combine the resolution independence of vector graphics with the random-access sampling of images. This enables a range of new applications, such as direct mapping of vector graphics onto 3D surfaces and creative warping effects. Unfortunately, most of these methods restrict the complexity of the input, which is not acceptable in a general-purpose rendering scenario. More importantly, these methods build acceleration data structures during expensive preprocessing stages that are sequential in nature, and this precludes their use with dynamic content.

To accelerate rendering, we build the shortcut tree, a quadtree data structure inspired by the seminal work of Warnock [1969]. Rather than including references to entire paths in each cell, we follow the approach of Nehab and Hoppe [2008] and include in each cell only the segments needed to correctly compute winding numbers when shooting horizontal rays. Since segment ordering becomes irrelevant, we can consider all segments simultaneously. Our shortcut tree improves on the work of Nehab and Hoppe in many ways. First, it is a quadtree rather than a regular grid, so our pipeline automatically adapts to the input. Second, it is created very efficiently, in parallel, by novel algorithms. Third, by splitting segments into monotonic components, we can use an implicit test to *detect* intersections, rather than explicitly *computing* them. This eliminates the need for repeatedly solving quadratic and cubic equations during shortcut tree creation and during rendering. (Note that Qin et al. [2008] also use monotonic segments, but to improve the accuracy of approximate signed-distance computations.) Finally, instead of clipping segments against cell boundaries, we include references to the entire monotonic segments into each cell. This increases numerical robustness and reduces the number of segments in the tree.

Our algorithms require the splitting of segments into monotonic components. Since the monotonization of *rational cubic* segments requires the solution of *quartic* equations, our pipeline does not support such segments. As a consequence, there is no support for the independent projective transformation of paths that contain integral cubic segments. To obtain projective effects on an entire scene (e.g., in navigation applications), we apply the inverse projective transformation to the samples. The scheduler then groups samples that fall into the same cell so they can be processed in parallel. The results are precisely the same and the sample scheduler allows more general used-defined warps. Native support for projective path transformations would be preferable for modeling reasons, and we leave it as future work. Although the work of Loop and Blinn [2005] and its derivatives support such transformations, most other renderers do not. Note that our pipeline does offer native support for *rational quadratic* Béziers, which we use to represent elliptical arcs.

We have carefully considered adopting an analytic solution to the antialiasing problem [Catmull 1978, 1984; Duff 1989; McCool 1995; Guenter and Tumblin 1996; Lin et al. 2005; Manson and Schaefer 2011, 2013]. Unfortunately, under the general setting of a full-fledged real-time vector graphics rendering pipeline, this is not possible from both practical and theoretical standpoints. The connection between area integrals and their closed boundaries, given by Green's

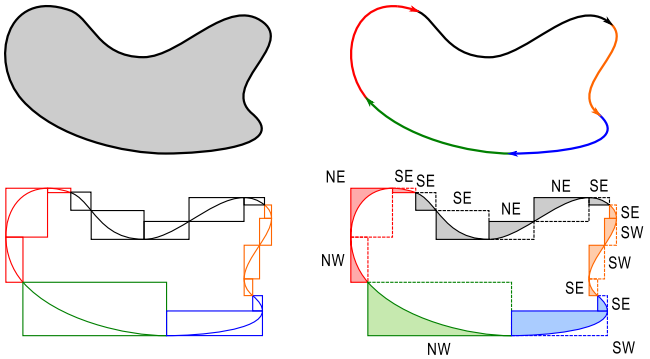


Figure 2: *The segments in each path are classified and decomposed into monotonic* abstract segments. *Abstract segments can be queried for their bounding box, their orientation* (NE,NW,SE,SW)*, and for the side on which a sample lies.*

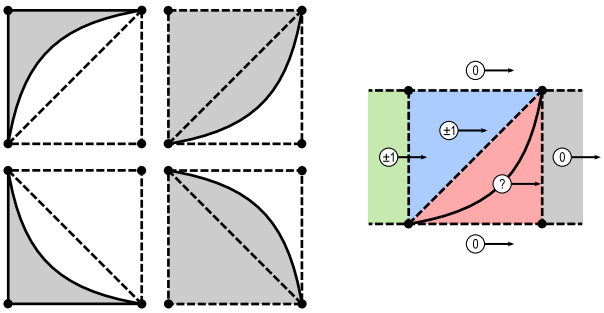


Figure 3: *Monotonic segments can appear in one of four configurations. Intersections with horizontal rays can be ruled out or confirmed trivially unless the sample and the segment are in the same side of the dashed bounding box diagonal. In that case, an implicit test is used. The intersection itself need not be computed.*

theorem, is only applicable to vector graphics paths when they are composed exclusively of non-overlapping simple shapes. Since it is not acceptable to place such restrictions on the input, the decomposition would have to be performed on the fly, by means of a general polygon clipper [Vatti 1992], extended to support polynomial segments. Even if this were made practical, we would still be left with an incomplete solution that does not support paths warped by non-trivial functions, layered semi-transparent gradient-filled paths (or even single-layer gradients with multiple stops in the color ramp), or textured paths. Monte Carlo integration (i.e., supersampling) is therefore the only viable alternative. In our approach, this is a nonissue, since we use enough samples to make exact and numerical integration visually indistinguishable.

Many real-time renderers rasterize shapes independently and blend results into the output image. This policy of *immediate-mode* rendering is analogous to the z-buffering algorithm for 3D rendering. As an optimization, many such systems approximate antialiasing by transforming per-pixel coverage into transparency prior to blending. This *conflation* leads to incorrect rendering of correlated layers [Porter and Duff 1984]. Solving this problem within the immediate-mode paradigm requires allocating memory for multiple samples per output pixel, or perhaps going over the input multiple times [Kilgard and Bolz 2012]. We keep the entire illustration in GPU memory, in *retained mode*. All layers in the illustration are sampled in a single pass without conflation, much like 3D rendering by ray tracing accelerated with a space-partitioning data structure.

Unlike real-time multi-sampling implementations (which tend to antialias with the box filter), our pipeline supports the wide antialiasing filters that are popular in production settings, such as the cubics by Catmull and Rom [1974] and by Mitchell and Netravali [1988]. We were particularly interested in antialiasing with the exceptional *cardinal* cubic B-spline filter, as proposed by McCool [1995]. The process reduces to applying a digital recursive filter [Unser et al. 1991] as a post-process to an image that has been antialiased with the standard cubic B-spline. The parallelization of the required recursive filters has recently enabled their use in real-time applications [Nehab et al. 2011]. Unfortunately, the noise due to supersampling is greatly magnified by this post-processing *unless* samples are shared between neighboring pixels with overlapping antialiasing filters [Nehab and Hoppe 2014]. McCool [1995] did not face this problem because analytic prefiltering does not generate noise. We use our sample scheduler to compute the color of each sample only once, even when the sample lies under the filters of multiple neighboring pixels, and even in the presence of user-defined warps. This sample sharing not only solves the noise issue, but also enables the efficient use of wide antialiasing filters. Since sample accumulation happens in fast local memory, increasing the number of samples to improve image quality has little effect on bandwidth and no effect on memory requirements.

Our choice of which features to include was guided by an attempt to provide support for as much of SVG [2011] and PDF [2006] as possible. Notable omissions are filter effects (e.g., Gaussian blur) and mesh-based gradient fills, which we leave as future work. Our APIs should be intuitive to those familiar with OpenVG [2008] or NV_path_rendering [2011]. Our API in the Lua programming language allows non-specialist users to write programs that render vector graphics using our pipeline and provides a portable format for illustrations and animations (see the supplemental materials). We also provide a C++11 API geared towards the expert programmer.

3 Abstraction

The scene is stored as a stream that contains the path geometry, as well as auxiliary information such as paint data and delimiters for clipping operations. The structure of the stream is best described by a context-free grammar, whose production rules are:

$$\textit{scene} \rightarrow \texttt{(}\ \textit{fill}^*\ \texttt{)} \tag{1}$$

$$\textit{fill} \rightarrow \texttt{F} \tag{2}$$

$$\textit{fill} \rightarrow \texttt{(}\ \textit{clip-path}^*\ \texttt{|}\ \textit{fill}^*\ \texttt{)} \tag{3}$$

$$\textit{clip-path} \rightarrow \texttt{C} \tag{4}$$

$$\textit{clip-path} \rightarrow \texttt{(}\ \textit{clip-path}^*\ \texttt{|}\ \textit{clip-path}^*\ \texttt{)} \tag{5}$$

Here, terminal production (2) stands for a filled path. Terminal production (4) stands for a clip test. The geometry of filled paths and clip tests is given by a list of segments that form the outlines of the shape. Filled paths contain additional paint information enabling the computation of sample colors, which can potentially vary based on sample position (e.g., gradient and texture paints).

The remaining terminals '`(`', '`|`', and '`)`' are short for *push*, *activate*, and *pop*, respectively, and are used to delimit clipping operations. The clip-path area starts empty with a *push*, and is given by the union of an arbitrary number of clip tests appearing before its matching *activate*. These clip-paths can themselves be clipped by other clip-paths, so that nesting is equivalent to intersection. The clip-path is active between the *activate* and its matching *pop*. The entire scene is delimited by a dummy *push–pop* pair.

Contrary to the back-to-front way in which the API receives paths from the user, the scene is represented internally so that the *topmost* paths appear *first*. This allows us to blend front-to-back and abort the computation as soon as the sample color becomes opaque. (Recall blending is associative [Wallace 1981; Porter and Duff 1984].)

The color of a sample is computed by selectively blending the paints of all paths for which the sample passes the inside-outside test, in addition to the inside-outside test of all active clip-paths. The fundamental inside-outside test consists of applying the path's fill rule to

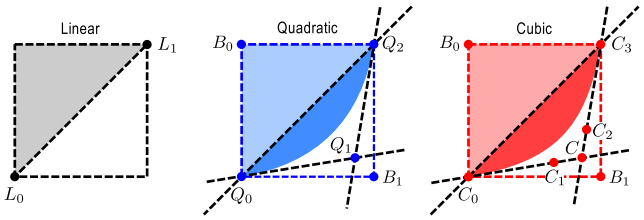


Figure 4: *Linear segments pose no problems. Implicit tests must be restricted to triangle $Q_0B_1Q_2$ in the case of quadratics, and to triangle C_0CC_3 in the case of cubics.*

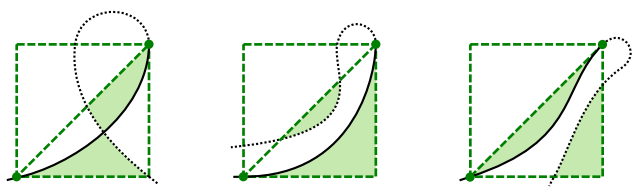


Figure 5: *Within the implicit test region, the implicit function must only change sign once along any horizontal rays, or the algorithm would report incorrect intersection counts as in the examples.*

the winding number of the path about the sample. The winding number is computed by counting the number of intersections between a horizontal ray, shot from the sample to infinity in the $+x$ direction, and all segments in the path, incrementing or decrementing depending on whether segments are going up or down at each intersection.

Nehab and Hoppe [2008] compute winding numbers by solving for the parameter values corresponding to the intersections between each segment and the ray (by solving linear, quadratic, or cubic equations), keeping those in the interval $[0, 1]$, substituting into the parametric equation of the segment to find the intersection point, and accepting only the intersections to the right of the sample point.

We use *abstract segments* (using monotonization and implicitization) to greatly simplify this process. Abstract segments can be queried for a bounding box, for an orientation (NE, NW, SE, SW), and for the side of the segment on which a given sample lies. Figure 2 illustrates the decomposition of a contour into abstract segments.

Since abstract segments are monotonic, they can be intersected only once by horizontal rays. As shown in figure 3, an intersection with a horizontal ray can be ruled out if the sample is above or below the segment's bounding box, or if it is to its right. Otherwise, if it is to the left of the bounding box, there is an intersection. Else, if the sample and segment are on opposite sides of the diagonal defined by the segment's endpoints, there is an intersection if and only if the sample is to the left of the diagonal. These cases are very easy to identify and treat. When the sample and the segment are on the same side of the diagonal, we use the implicitization. By testing the sign of the implicit form of the parametric segment at the sample position, we can determine the side of the segment on which the sample lies. If it is on the left, there is an intersection, otherwise there isn't. The intersection itself need not be computed. Similar reasoning applies to vertical rays, which are used during the construction of the shortcut tree.

3.1 Monotonization

Each input segment is specified as a parametric curve $\gamma : \mathbb{R} \to \mathbb{R}^2$

$$\gamma(t) = \big(x(t), y(t)\big), \quad t \in [0, 1]. \tag{6}$$

We first compute parameter values t_j

$$0 = t_0 < t_1 < \cdots < t_{k-1} < t_k = 1 \tag{7}$$

that satisfy either of the equations

$$x'(t_j) = 0 \quad \text{or} \quad y'(t_j) = 0. \tag{8}$$

These t_j values break γ into k monotonic segments corresponding to the intervals $[t_{j-1}, t_j]$, for $j \in \{1, \ldots, k\}$. Where needed, we use the multiaffine representation of Ramshaw [1988] to generate the control points corresponding to each parameter interval.

Linear segments are monotonic on their own. Otherwise, finding the t_j leads to linear equations for integral quadratic segments, and to quadratic equations for rational quadratic and integral cubic segments. We use an algorithm by Blinn [2005] to solve the quadratics in a numerically robust way.

3.2 Implicitization

Linear segments Let vector $\boldsymbol{s} = \begin{bmatrix} s_x & s_y & 1 \end{bmatrix}^T$ hold the homogeneous coordinates of the sample. The equation for line $L_0\,L_1$ in figure 4 can be written in the form $\boldsymbol{k}\,\boldsymbol{s} = 0$, where $\boldsymbol{k} = \begin{bmatrix} a & b & c \end{bmatrix}$ is an affine function (i.e., a row vector). It is easy to select $\boldsymbol{k}$ such that $\boldsymbol{k}\,\boldsymbol{s} > 0$ for samples to the left of L_0L_1, so this implicit line test can decide on which side of a linear segment each sample lies.

Loop and Blinn [2005] gave an elegant procedure for generalizing this implicit test for quadratic and cubic Bézier segments. They use a result by Salmon [1852] that ensures it is always possible to find affine functions $\boldsymbol{k}$, $\boldsymbol{\ell}$, and $\boldsymbol{m}$ such that the tests become

$$\text{integral quadratic:} \quad (\boldsymbol{k}\,\boldsymbol{s})^2 - \boldsymbol{l}\,\boldsymbol{s} > 0, \tag{9}$$

$$\text{rational quadratic:} \quad (\boldsymbol{k}\,\boldsymbol{s})^2 - (\boldsymbol{l}\,\boldsymbol{s})(\boldsymbol{m}\,\boldsymbol{s}) > 0, \tag{10}$$

$$\text{integral cubic:} \quad (\boldsymbol{k}\,\boldsymbol{s})^3 - (\boldsymbol{l}\,\boldsymbol{s})(\boldsymbol{m}\,\boldsymbol{s}) > 0. \tag{11}$$

Each abstract segment stores the row vectors corresponding to the required affine functions, and we use them to quickly perform implicit tests on the required samples.

There is one important caveat. Within the region where implicit tests are used, we must ensure that the implicit function changes sign only once along axis-aligned rays. This is to prevent the situations depicted in figure 5, which would cause an incorrect number of detected intersections and ultimately to incorrect rendering. Since segments have been monotonized, it suffices to prove that the parametric curve is outside the test region for all parameters outside of $[0, 1]$. The diagrams in figure 4 illustrate the proofs that follow.

Quadratic segments It is sufficient to restrict the test to triangle $Q_0B_1Q_2$. This requires one implicit test against segment Q_0Q_2 and two comparisons against bounding box coordinates.

Proof: The quadratic curve cannot cross segment Q_0Q_2 outside of points Q_0 and Q_1 since it can intersect a straight line at most twice. Similarly, it cannot cross segments Q_0Q_1 and Q_1Q_2. Indeed, since the curve is tangent at both Q_0 and Q_2, these points count as double intersections. Finally, note the quadratic cannot intersect segments Q_0B_1 and B_1Q_2 without first incurring forbidden additional intersections with segment Q_0Q_1 or Q_1Q_2, respectively. □

Loop and Blinn [2005] use the GPU rasterizer to generate fragments only inside triangle $Q_0Q_1Q_2$. Using their solution would require us to perform three implicit line tests.

Cubic segments Cubics are more demanding. To prevent the curve from looping back and intersecting segment C_0C_3 (and the curve itself), Loop and Blinn [2005] split cubics at a double point whenever one is found for a parameter t_d with $0 < t_d < 1$. This requires solving a quadratic equation, and we do the same. We go one step further and split the cubics at an inflection point whenever one is found for t_i with $0 < t_i < 1$. This requires solving another quadratic, but ensures the intersection of lines C_0C_1 and C_2C_3 happens at a point C inside the bounding box. Then, it is sufficient to restrict the implicit test to the triangle C_0CC_3. Note that these quadratics must be solved during the implicitization process anyway.

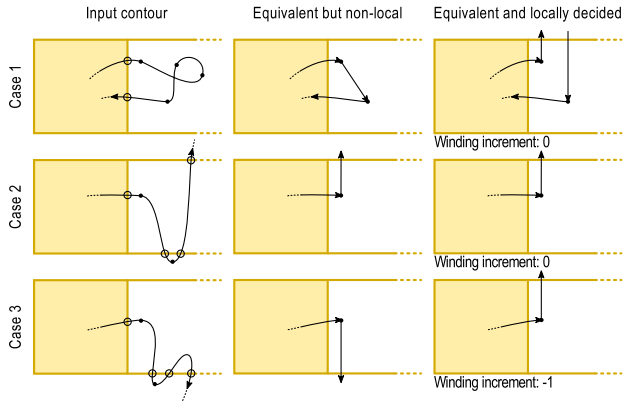


Figure 6: *Contours on the left column are equivalent to contours in the central and right columns. The intuitive representation in the center requires knowledge about segments that can be far apart in the input. The equivalent representation on the right can be generated locally by inspecting segments independently.*

Proof: We again use root-counting arguments. First, we show that the curve cannot intersect segment C_0C_3. If it did, it would either have to exit triangle C_0CC_3 again through segment C_0C_3 (but it cannot have four intersections with line C_0C_3), or it would have to self-intersect (but by assumption it has no double point for $t \in (0, 1)$). The arguments for why the curve cannot intersect segment C_0C and CC_3 are analogous, so consider segment C_0C. We start from the part of the curve that exits triangle C_0CC_3 at C_0. If it is below C_0C, then C_0 is an inflection and exhausts all three possible intersections with line C_0C. If it is above C_0C, then C_0 is only a tangent. Now recall the curve cannot intersect C_0C_3. Therefore, in order to intersect C_0C a third time, it would have to either go up around triangle C_0CC_3, thereby intersecting CC_3 four times (twice at the tangent C_3 and twice before it can reach C_0C), or go down back into C_0C (wasting the third and last intersection with line C_0C at a point outside of segment C_0C). Now consider the part of the curve exiting at C_3. If it exits to the right of CC_3, then C_3 is an inflection and precludes the fourth intersection with line CC_3, needed to reach segment C_0C. If it exits to the left, it would intersect line C_0C the third time outside of segment C_0C, since it cannot intersect segment C_0C_3. □

Loop and Blinn [2005] use the GPU rasterizer to generate fragments only inside the two triangles that form the convex hull of the cubic control polygon. Adopting this approach would require us to perform at least four implicit line tests and maintain some bookkeeping.

4 The shortcut tree

Assume that we have partitioned the illustration area into a union of small cells. The key strategy for speeding up the inside-outside test within each cell is to reduce the number of segments that must be tested for intersection during the computation of winding numbers. We will specialize the representation of the illustration within each cell while maintaining the following invariant: the winding number of any path about any sample in a cell, computed by shooting a horizontal ray from the sample to infinity in the $+x$ direction, is the same as in the original illustration. It is clear that we can eliminate a segment whenever its bounding box is completely above, below, or to the left of the cell. The difficulty is what to do otherwise. The breakthrough in the *lattice clipping* algorithm of Nehab and Hoppe [2008] is to include in each cell only the parts of segments that overlap with them, with the addition of *winding increments* and *shortcut segments* that restore the invariant. We describe an improved version of the idea that does not clip segments to the interior of cells and thus eliminates all intersection computations.

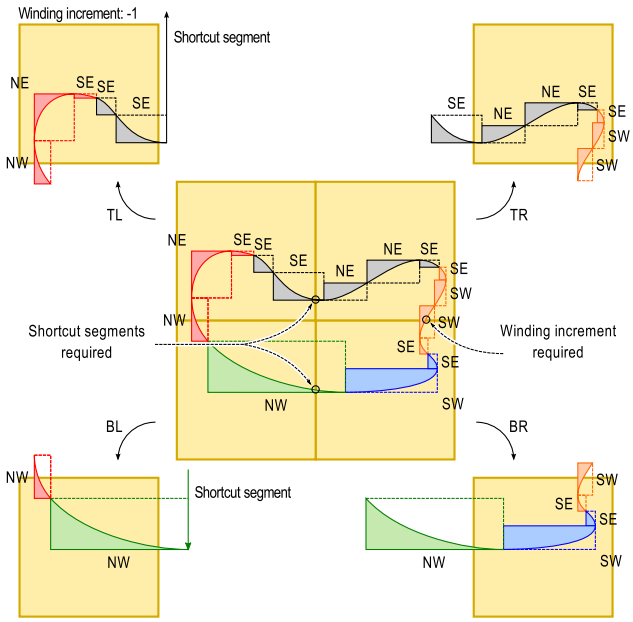


Figure 7: *Example subdivision of a shortcut tree cell. Segments are included in a child cell if and only if they intersect its area. Marked intersections generate shortcut segments and winding increments that restore the invariant within each child cell.*

The examples in figure 6 show how a contour's behavior to the right of a cell boundary can be summarized with shortcut segments. Since all input contours are closed, any contour that leaves a cell by crossing its right boundary must later return to the cell. If the contour does not return to the cell via the right boundary, it must return from a different side. In order to do that, the contour must first leave the row of cells, and this must happen in the region to the right of the cell. Therefore, there are only three possibilities: (1) the contour comes back inside by crossing the right boundary again; (2) it exits to the row above; or (3) it exits to the row below. In case 1, we can represent all omitted segments by a shortcut connecting the end of the exiting segment and the beginning of the entering segment. In case 2, we can add a shortcut going up from the end of the segment that intersects the right boundary. Finally, in case 3, we can add a shortcut going down from the end of the segment that intersects the right boundary. The reasoning is similar for contours entering the cell from the right boundary. In figure 6, note how the winding numbers obtained from the input contours in the left column are the same as those obtained with the compressed representation in the central column, no matter where the sample lies inside the cell area.

Unfortunately, no local procedure can distinguish between these cases by inspecting one segment at a time. Nehab and Hoppe solve this problem by always assuming case 2 and including winding increments when needed. A classification error can be detected locally for each input segment that crosses a bottom cell boundary. The invariant can be restored by modifying the initial winding number of all cells to the left of the violating intersection. In the right column of figure 6, note how case 1 generates two shortcuts, one for each intersection with the right cell boundary. These effectively cancel each other when vertically overlapping. Also note how in case 3 the winding increment effectively flips the orientation of the incorrect shortcut segment, and that these winding increments accumulate (independently per path). Naturally, case 2 requires no modification.

The result of the original lattice clipping algorithm is a regular grid of cells. We now proceed to the description of the shortcut tree, our hierarchical data structure based on the same ideas and, more importantly, how to build it efficiently and in parallel.

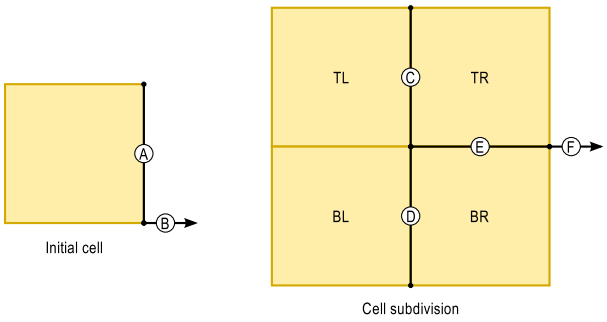


Figure 8: *During subdivision, segments are classified based on their endpoints, and on intersections with the marked boundaries.*

4.1 Subdivision

The key operation when building the shortcut tree is *cell subdivision*. Assuming that a parent cell respects the invariant, our task is to find subdivision rules that produce child cells that also respect it. Then, by induction, the resulting tree will satisfy the invariant everywhere.

Figure 7 shows an example of cell subdivision. A child cell includes a segment (in its original form, *including the parts outside the cell*) if and only if the segment intersects its area. The intersections with the TL / TR and the BL / BR boundaries respectively generate shortcut segments in child cells TL and BL. The intersection with the TR / BR boundary generates a winding increment in child cell TL that corrects the misclassification of child cell TL from case 2 to case 3.

In general, for each segment in the parent cell, we must decide in which child cells to include it, and whether it generates shortcut segments and winding increments. The inclusion test is particularly simple: a segment is included in a cell if one of its endpoints is inside the cell or if it intersects one of the cell's boundaries. Figure 8 illustrates the procedure used to identify shortcut segments and winding increments. During tree creation, the root of the shortcut tree is generated first. Shortcut segments are generated for all segments crossing boundary A. Winding increments are generated for all segments crossing the half-line boundary B (which extends to infinity). Cell subdivision is performed in a similar way. Shortcut segments are generated in child cells TL and BL for all segments intersecting boundaries C and D, respectively. Segments intersecting boundary E generate winding increments in child cell TL. Segments intersecting the half-line boundary F generate winding increments in both child cells TL and TR. We can detect intersections by testing whether cell boundary vertices lie on different sides of the abstract segment. To distinguish between intersections with C and D, or with E and F, we test the side of their shared vertex.

Stopping criteria We stop cell subdivision when: (1) the amount of memory taken by the shortcut tree reaches a maximum threshold; (2) the number of segments in a cell is smaller than a minimum threshold. Criterion 1 allows users to limit memory consumption; criterion 2 prevents futile subdivisions. More sophisticated criteria will be investigated as future work.

4.2 Parallel subdivision

The shortcut tree is generated in parallel, in breadth-first order, with each step subdividing all cells at progressively deeper tree levels. The cells to be subdivided are laid out contiguously in memory. Each cell contains a specialized description of the scene that is correct within the cell's boundaries. The contents of each path in the specialized scene are also laid out contiguously. Paths are represented as sequences of lightweight *segment entries*. Each segment entry uses a few bits to distinguish between clipping control terminals, abstract segments, shortcut segments, and winding increments. (The

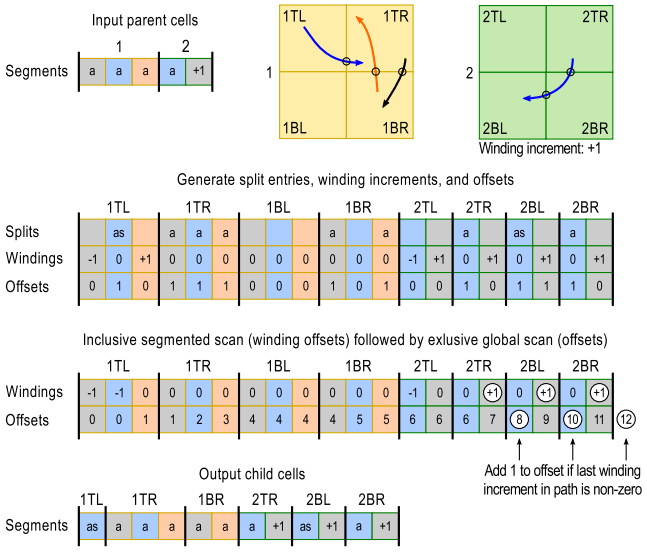


Figure 9: *Parallel shortcut tree subdivision. Entries for each segment are shaded with its hue. Abstract segments, shortcut segments, and winding increments are respectively marked with a, s, and* ± 1.

initial winding number for a path is the sum of all its winding increments.) There is also room for references to the corresponding abstract segment, the originating path, and the originating cell.

Each round of subdivision independently processes every entry, of every path, of every cell at the current subdivision level. According to the subdivision rules, each parent entry may generate, in each of the four child cells: a reference to an abstract segment, a shortcut segment, and a winding increment. In other words, the number of outputs varies by input entry. This requires us to compute an *output offset* before we can write the child cells, which in turn forces us to split the process into four computational kernels. Figure 9 shows an example in which two cells are subdivided in parallel. For simplicity, each cell contains a single path and only a few entries.

The first kernel is the most important. It computes an array of *splits*, with one slot for each parent entry in each of the four child cells. Each thread inspects a single parent entry and generates one split for that entry in each child cell. These splits specify whether to include a parent abstract segment in the child cell and whether to add a shortcut segment. The kernel also computes two additional values per entry: a winding increment, if any, and an offset that counts the number of output entries generated by the parent split entry. For layout reasons in figure 9, these are shown as two independent arrays. In reality, we represent them as an array of pairs.

The purpose of the second kernel is to consolidate the winding increments into a single entry per path, per cell. Consolidation is necessary because the first kernel can produce multiple winding increments within each path in a child cell, and we need to prevent their uncontrolled proliferation. Consolidation is achieved with an inclusive segmented scan on the winding increment array. This scan adds together the winding increments that belong to the same path, leaving the result as the last entry for that path in the windings array.

The role of the third kernel is to compute the global offset for the output of each parent entry into each child cell. This is accomplished with exclusive scan on the windings/offset array. (Recall they are stored as pairs in a single array.) Besides adding the values in the offset entry, the operator used for the scan checks the corresponding value in the windings entry. If the entry is the last in a path, and if the windings entry is non-zero (as consolidated by the second kernel), the scan operator adds an extra 1 to the value to make room for a single winding increment for the path.

The fourth and final kernel inspects all split entries in parallel. Each thread loads the corresponding offset and writes the appropriate segments to their final positions in the child cell. If an entry is the last in a path, the kernel also inspects the windings array, and generates the appropriate winding increment if needed. The offsets are such that the order in which the paths appear in the child cells is the same as their order in the parent cell. In fact, even the order of entries within each path is preserved. Note that all kernels involved in the subdivision process are parallel at the segment level.

4.3 Pruning

After every subdivision step, the total number of entries in the child cells is likely to be greater than the original number of segments in the parent cell. After all, segments that cross the subdivision boundaries are replicated and may generate additional shortcut segments. Certain optimizations can be performed locally per segment and help attenuate this growth. For example, shortcut segments that cannot be intersected by any ray emanating from a cell need not even be generated. Conversely, shortcut segments that are intersected by *all* rays can be converted to the equivalent winding increment.

The most powerful strategies for keeping this growth in check, however, involve path interactions at the cell level. For example, when a path is opaque and covers the entire cell, all paths underneath it can be pruned. Clip-paths complicate the pruning algorithm, but also provide us with additional opportunities for optimization. For example, if a clip-path is empty when restricted to a cell, it can be pruned along with all paths under its influence.

Pruning is easiest to understand by means of *stream rewriting rules*. Each rewrite rule simplifies the stream while maintaining the following invariant: within the cell, the output stream is *valid and equivalent* to the input stream. Pruning is performed by the repeated application of rewrite rules, until no rule can be applied.

We introduce new terminals $\mathtt{F}_0$ and $\mathtt{C}_0$ to represent filled paths and clip tests, respectively, that fail all inside-outside tests for samples in the cell area. Conversely, $\mathtt{F}_1$ and $\mathtt{C}_1$ refer to paths that pass the inside-outside test for all samples in the cell area, and in addition the paint associated to $\mathtt{F}_1$ is fully opaque. Since our shortcut tree is tight, such paths are easy to identify: simply apply the inside-outside test to the initial winding number of paths that contain no segments. Finally, non-terminals A, B, and C represent well-formed streams, while ϵ represents the empty stream. Given these definitions, the stream rewrite rules are as follows:

$$\mathtt{F}_0 \rightarrow \epsilon \tag{12}$$

$$\mathtt{C}_0 \rightarrow \epsilon \tag{13}$$

$$\mathtt{(}\ A\ \mathtt{|}\ B\ \mathtt{F}_1\ C\ \mathtt{)} \rightarrow \mathtt{(}\ A\ \mathtt{|}\ B\ \mathtt{F}_1\ \mathtt{)} \tag{14}$$

$$\mathtt{(}\ \mathtt{|}\ A\ \mathtt{)} \rightarrow \epsilon \tag{15}$$

$$\mathtt{(}\ A\ \mathtt{|}\ \mathtt{)} \rightarrow \epsilon \tag{16}$$

$$\mathtt{(}\ A\ \mathtt{C}_1\ B\ \mathtt{|}\ C\ \mathtt{)} \rightarrow C \tag{17}$$

$$\mathtt{(}\ A\ \mathtt{|}\ B\ \mathtt{C}_1\ C\ \mathtt{)} \rightarrow \mathtt{(}\ A\ \mathtt{|}\ \mathtt{C}_1\ \mathtt{)} \tag{18}$$

Two key properties have guided the selection of rewrite rules: (1) there is no reordering, only elimination of elements, and (2) the rules can be applied in parallel since they do not interfere with each other.

Rules (12) and (13) state that empty paths can be summarily eliminated from the stream. Rule (14) states that a fully opaque path covering the entire cell occludes all content that comes behind it at the same clipping nesting depth. Rule (15) states that a clip-path that always fails can be eliminated along with all content under its influence. Rule (16) states that a clip-path that has no content under its influence can also be eliminated. Rule (17) short-circuits the evaluation of a clip-path that always succeeds within the cell, leaving behind only the content that was under its influence.

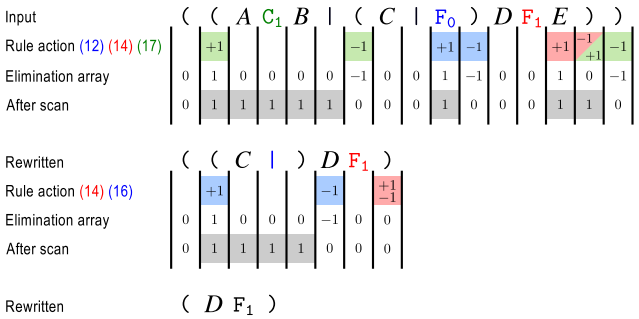


Figure 10: *Parallel pruning example. The first pruning round engages three different rules. The segment entries that trigger action in each rule are colored. The second and final round engages only two rules. Elements selected for elimination are marked in gray.*

Rule (18) is more subtle. It implements short-circuiting in the evaluation of a nested clip-path. When rules are evaluated one at a time, we could prune more aggressively:

$$\mathtt{(}\ A\ \mathtt{|}\ B\ \mathtt{C}_1\ C\ \mathtt{)} \rightarrow A \tag{19}$$

However, in the next section we will parallelize the pruning and rule (17) will be applied simultaneously in a single step. The aggressive rule (19) combines with (17) to produce incorrect results:

$$\mathtt{(}\ A\ \mathtt{C}_1\ B\ \mathtt{|}\ C\ \mathtt{C}_1\ D\ \mathtt{)} \xrightarrow{17,\,19} \epsilon \neq \mathtt{C}_1 \tag{20}$$

Rule (18) does not interact with rule (17), and results are correct:

$$\mathtt{(}\ A\ \mathtt{C}_1\ B\ \mathtt{|}\ C\ \mathtt{C}_1\ D\ \mathtt{)} \xrightarrow{17,\,18} \mathtt{C}_1 \tag{21}$$

4.4 Parallel pruning

Pruning is a challenging operation to perform efficiently and in parallel. The key is to split the computation into simple massively parallel tasks, and to ensure our invariant is preserved at each step. We proceed with multiple iterations of *mark-and-sweep*. During each iteration, we mark the elements in the stream that each rewrite rule wants to eliminate. We then sweep the marked elements away by compacting the stream. The mark-and-sweep process is repeated until no element can be eliminated. Naturally, the difficult part is marking the correct elements for elimination.

Other than rules (12) and (13), all rules require the matching of delimiters `(`, `|`, and `)`. We start by cross-linking them so we can freely move from one to the other in constant time. To do so, we first obtain the clip nesting depth of each element. This is a simple matter of initializing to 0 a linear array with one element per segment entry. Values associated to `(` are then set to $+1$, and values immediately to the right of `)` are set to -1. An inclusive-scan of this array produces the required result. For a maximum nesting depth n, we then perform n segmented scans. Each segmented scan produces the links from all elements at depth d to their matching `(`. To do so, the initialization sets all entries to 1, and marks as boundaries all `(` elements at depth d. The segmented scan results in spans that start at 1 and progressively increase, but that restart for every `(` at depth d. In other words, all elements at depth d are associated to their distance to the matching `(`, which we convert to absolute pointers. To complete the process, we use this information to cross-link the matching `(`, `|`, and `)` at depth d between them. Recall we already could reach `(` from the matching `|` and `)`. After n segmented scans, all matching delimiters are properly cross-linked.

Given this information, we can finally describe the procedure that marks segment entries for elimination. Marking starts with the allocation of an *elimination array* with one element associated to

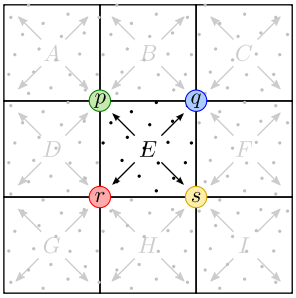


Figure 11: *Integration with sample sharing for 2×2 antialiasing filters. Samples in each unit area are evaluated only once. Unit area E sends appropriately weighted sums to pixels p, q, r, and s. Pixel p receives contributions from unit areas A, B, D, and E.*

each segment entry, all initially set to 0. Then, all segment entries are inspected in parallel, and each rule is given a chance to conditionally modify the elimination array by atomic increments or decrements to appropriates elements. When all rules have been executed, the elimination array is subjected to an inclusive scan. The segment entries to be eliminated are the ones associated to a *positive* element in the elimination array. Figure 10 shows an example that includes two rounds of pruning with the execution of multiple simultaneous rules. We describe rules (16) and (17). Other rules are analogous.

Rule (16) only requires action when inspecting a `|` segment entry. In that case, it checks the segment entry to its right, in search of a matching `)`. If it finds one, it atomically adds $+1$ to the element associated to its matching `(` in the elimination array, and atomically adds -1 to the element to the right of that associated to the matching `)`. After the scan, the effect is to add $+1$ to all elements between `(` and `)` (including the terminals themselves), thereby marking them for elimination as the rule dictates.

Rule (17) is a bit more involved. It acts unconditionally when inspecting a $\texttt{C}_1$ segment entry. It atomically adds $+1$ to the elimination array element associated to the `(` that is reachable from $\texttt{C}_1$. From the `(`, the matching `|` is also accessible. The rule atomically adds -1 to the elimination array element immediately past it. The rule can also reach `)`. There, it atomically adds $+1$, and adds -1 to element immediately past it. After the scan, only the region matched by the C is preserved. Note that this works even in the presence of multiple elements $\texttt{C}_1$ between the `|` and `)`. The only side effect is that certain elements that are marked for elimination may end up associated to numbers larger than 1 after the scan.

5 Rendering

In antialiased rendering, the color $c(p)$ of each pixel p is given by the convolution between the illustration and an antialiasing filter:

$$c(p) = \int_\Omega f(p-u)\,\psi(u)\,\mathrm{d}u. \tag{22}$$

Here, f represents the vector graphics illustration, so that $f(v)$ is the color of the sample at v. The antialiasing filter ψ vanishes outside of a compact support Ω containing the origin. As discussed before, the analytic evaluation of (22) is impossible for the general illustrations we want to render. The integral is thus expressed as an expectation and estimated by the Monte Carlo method (i.e., by *supersampling*):

$$c(p) = A_\Omega\,\mathrm{E}_{U_\Omega}\left[f(p-U_\Omega)\,\psi(U_\Omega)\right] \tag{23}$$

$$\approx \frac{A_{\Omega_\psi}}{m}\sum_{i=1}^{m} f(p-u_i)\,\psi(u_i). \tag{24}$$

Here, A_Ω is the area of support Ω and U_Ω is a random variable uniformly distributed over Ω; the estimator simply computes the average value of the integrand over m variates u_i drawn from U_Ω.

One of the reasons for the popularity of the box filter is that its support has *unit area* in terms of the inter-pixel spacing. In that case, the integral can be computed independently for each pixel. In contrast, higher-quality filters can have support larger than 4×4 unit areas, where at least 16 filters overlap each sample in the illustration. Since computing sample colors dominates the cost of rendering, we cannot afford to recompute them so many times.

5.1 Integration

We break integral (22) into a sum of n integrals over the unit areas A_j that tile the filter domain Ω:

$$c(p) = \sum_{j=1}^{n}\int_{A_j} f(p-u)\,\psi(u)\,\mathrm{d}u \tag{25}$$

$$= \sum_{j=1}^{n}\mathrm{E}_{V_j}\left[f(p-V_j+a_j)\,\psi(V_j-a_j)\right] \tag{26}$$

$$\approx \frac{1}{m}\sum_{j=1}^{n}\sum_{i=1}^{m} f(p-v_{ji}+a_j)\,\psi(v_{ji}-a_j). \tag{27}$$

Here, V_j is a random variable distributed over the unit area centered at the origin, and a_j is the center of unit area A_j in the tiling of Ω. From the expression, we see that if $p + a_j = p' + a'_k$, then

$$f(p-v_{ji}+a_j) = f(p'-v_{ji}+a'_k), \tag{28}$$

and therefore unit area A'_k of pixel p' can reuse sample colors of unit area A_j of pixel p with appropriately changed filter weights.

Figure 11 illustrates the method. Without loss of generality, assume a 2×2 filter (e.g., the hat filter). In this case, every unit area is covered by exactly 2×2 neighboring filters: unit area E is covered by filters centered at pixels p, q, r, and s. Independent computation of these pixels would unnecessarily evaluate four times each sample in unit area E. Instead, we structure our computation around the unit areas themselves. Samples in unit area E are evaluated and accumulated into four appropriately weighted sums, which are in turn atomically added to pixels p, q, r, and s. Pixel p is complete after receiving independent contributions from unit areas A, B, D, E. The process is parallel at the sample and pixel levels.

A key property of this method is that it uses a fixed amount of memory per pixel, regardless of the number of samples per unit area. Our pipeline supports a variety of different filters and sample distributions. The default high-quality setting uses a blue-noise pattern generated with the method of Balzer et al. [2009] with 32 samples per unit area, weighted by the 4×4 cubic B-spline. The combination is equivalent to $32 \times 16 = 512$ samples per pixel. The B-spline filters are then reshaped to cardinal cubic B-splines with a post-processing parallel recursive-filter [Nehab et al. 2011]. This explains the high quality of our results.

5.2 Sampling

Absent clip-paths, the sampling algorithm would be straightforward. Preprocessing the input to eliminate clip-paths, however, requires a fully general polygon clipping algorithm such as that proposed by Vatti [1992]. To the best of our knowledge, Vatti's algorithm has not been mapped to GPUs. Its robust implementation is notoriously difficult and its extension to curved segments requires the numerical computation of intersections between them. Understandably, most implementations rasterize clip-paths to a stencil buffer, and use it to mask out the samples that fail the clip test. Instead, we add

clip-paths to the shortcut tree like any other path geometry, and maintain in each shortcut tree cell a stream that matches the scene grammar described in section 3. Clipping operations are performed per sample and with object precision.

When evaluating the color of each sample, the decision of whether or not to blend the paint of a filled path is based on a Boolean expression that involves the results of the inside-outside tests for the path and all currently active clip-paths. Since this expression can be arbitrarily nested, its evaluation seems to require one independent stack per sample (or recursion). This is undesirable in code that runs on GPUs. Fortunately, as discussed in section 4.3, certain conditions (see the pruning rules) allow us to skip the evaluation of large parts of the scene. These conditions are closely related to the short-circuit evaluation of Boolean expressions. Once we include these optimizations, it becomes apparent that the value at the top of the stack is never referenced. The successive simplifications that come from this key observation lead to the *flat clipping* algorithm, which does not require a stack (or recursion).

Flat clipping The intuition is that, during a union operation, the first inside-outside test that succeeds allows the algorithm to skip all remaining tests at that nesting level. The same happens during an intersection when the first failed inside-outside test is found. Values on the stack can therefore be replaced by knowledge of whether or not we are currently skipping the tests, and where to stop skipping. The required context can be maintained with a finite-state machine.

The machine has three states: *processing* (P), *skipping* (S), and *skipping by activate* (SA). Inside-outside tests and color computations are only performed when the machine is in state P. The S and SA states are used to skip over entire swaths of elements in the stream.

In addition to the machine state, the algorithm maintains the sample color currently under computation and three state variables that control the short-circuit evaluation. The first two state variables keep track of the current clipping nesting depth d and the number u of nested clip-paths that have not yet been activated. These variables are updated when the machine comes across terminals `(` , `|` , and `)`:

$$\texttt{(} \;\Rightarrow\; d \leftarrow d+1,\; u \leftarrow u+1 \tag{29}$$

$$\texttt{|} \;\Rightarrow\; u \leftarrow u-1 \tag{30}$$

$$\texttt{)} \;\Rightarrow\; d \leftarrow d-1 \tag{31}$$

Skipping is interrupted when one of terminals `|` or `)` is found at a depth at least as shallow as the current *stopping depth* s. The stopping depth is set right before any transition to a skipping state, and is the third and last state variable needed by the algorithm.

Figure 12 shows the state transition diagram. Each transition is marked by an annotated arrow. Arrow annotations can have one or two rows. The first row specifies the conditions that trigger the transition. The first condition is the *triggering terminal*. Besides the clipping operators `|` and `)`, terminals $\texttt{f}_1$ and $\texttt{c}_1$ can also trigger transitions. These terminals denote, respectively, a filled path and a clip test for which the *current sample* has passed the inside-outside test. (This is in contrast to terminals $\texttt{F}_1$ and $\texttt{C}_1$ from section 4.3, which denoted paths that pass inside-outside tests for *all samples* in the cell area.) After the triggering terminal, additional required conditions can be specified. The second row in arrow annotations is optionally used to update the stopping depth d.

The machine starts in P with $d \leftarrow 0, u \leftarrow 0$. Consider the transitions between P and S. In the transition triggered by $\texttt{f}_1$, the additional condition $\alpha = 1$ tests if the sample color is now opaque. In that case, since we render primitives front-to-back, there is no point in continuing. The machine transitions to S, and sets s to 0. Condition $\texttt{c}_1$ means a clip test has succeeded. The remaining clip tests in the clip-path can therefore be skipped by short-circuit. The machine transitions to S and sets the stop depth to d. There are

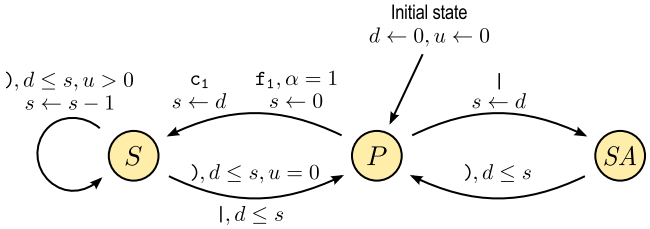


Figure 12: *State transition diagram for the finite-state machine of the flat-clipping algorithm.*

two transitions away from S. The first transition happens when an *activate* operation is found. Looking at the scene grammar, we see that this can only happen if the machine arrived at S due to a $\texttt{c}_1$ transition from P. In other words, an entire clip-path test has succeeded, and therefore we transition unconditionally back to P. The second transition happens when a matching `)` is found. The condition $u = 0$ means the machine is not inside a nested clip-path test, so it simply transitions back to P. If the machine is skipping *inside* a nested clip-path test, one of the inner clip tests must have passed, and therefore the outer test can be short-circuited as well. The machine simply resets the stop depth to the outer level and continues in state S.

The remaining transitions are between P and SA. If the machine finds a `|` while in state P, it must have been performing a clip-path test that failed. Otherwise, it would have been in state S. Since the test failed, it can skip until the matching `)`. This is what motivates the name *skipping by activate*.

5.3 Scheduling

The pipeline allows a user to specify a 3×3 projective transformation to be applied to the sample coordinates. Experienced users can design arbitrary warping functions in CUDA.[1] Since the pipeline remaps individual samples, and not the rendered image, results are exactly the same as if the illustration had been warped in object space by the inverse of the warp function, and only then rendered.

With the integration and sampling algorithms in place, we can complete the rendering algorithm. The sample positions from each unit area must be warped by the user-supplied function and pushed down the shortcut tree until the appropriate leaf cell is found. With the cell contents and warped sample positions, the sampling algorithm computes their colors. These colors are weighted, added together, and routed to all the appropriate pixels by the integration algorithm.

The scheduler plays two key roles: it minimizes the global memory bandwidth requirements by allowing cell contents to be loaded once and reused by multiple unit areas, and it minimizes control-flow divergence by grouping together samples that fall in the same cell. To do so, we use three computational kernels.

The first kernel goes over each unit area and identifies the set of leaf cells that contain at least one of the warped sample positions. This information is obtained by descending with each warped sample position down the shortcut tree until a leaf cell is found. The resulting list of cell ids is compressed within shared memory to eliminate repetitions. A list with pairs containing the originating unit area ID and the required cell ID is stored into global memory.

The role of the second kernel is to transpose the results of the first kernel, which come naturally sorted by unit area ID, so that they are instead sorted by cell ID. This is accomplished with a simple parallel sort.

The third and last kernel performs the actual rendering. Each computational block is responsible for a batch of U unit areas from the

[1] This feature currently requires recompiling the scheduler. Changing the API to support warps defined at runtime is a simple if tedious task.

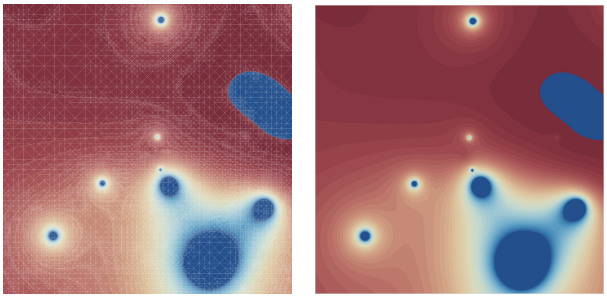


Figure 13: *(Left) Artifacts appear when polygons that share an edge are independently resolved to pixels before blending. (Right) Our renderer blends colors independently per sample and resolve later.*

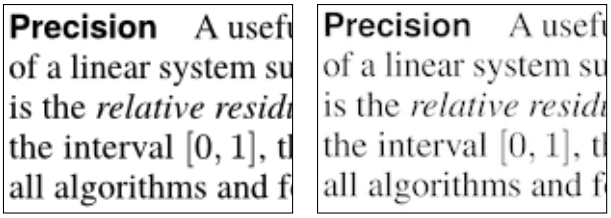


Figure 14: *(Left) Integration in gamma space incorrectly widens dark regions and produces heavier text. (Right) Our renderer integrates in linear space to produce text with the intended weight.*

list produced by the second kernel. The different unit areas in each batch send at least one warped sample position to the same given cell. There is enough shared memory to load I input segments. The context for the S samples that will be evaluated simultaneously is stored in the registers of independent threads. While there are unit areas to be processed, the algorithm warps their samples and eliminates those that fall outside the cell. This process is repeated until S samples are found (potentially originating from distinct unit areas). Then, it loops over the cell stream, loading chunks of I segments to shared memory. For each chunk, it advances the sampling algorithm in parallel for the S samples over these I input segments. When the entire cell contents have been processed, the algorithm computes independent weighted sums for the samples originating from each unit area, and atomically adds them to the appropriate pixels. It then goes back for more unit areas in the batch until they have all been processed. We use $S = 128$, $I = 32$, and $U = 32$ in all our tests. With this setup, we are able to process 4 unit areas in the same cell, with 32 samples each, without reloading the input.

A specialized version of the scheduler handles the common case when there is no user-defined warp. We align the shortcut tree cell boundaries with the unit areas, so that all samples originating from a given unit area fall within the same cell. This greatly simplifies the generation of the list of unit areas per cell: it suffices to descend on the shortcut tree with the unit area center. It also simplifies the integration step: there is no need to keep track of which samples belong to each unit area since no sample is ever eliminated.

6 Results and discussion

We ran a variety of different tests to evaluate the performance and quality of our rendering pipeline. All tests were run on an NVIDIA GeForce GTX Titan (2688 CUDA cores, 6GB of global memory) hosted by an Intel Core i7 980 at 3.33GHz with 24GB of system memory. For fairness when comparing against competing algorithms, we used the original published implementation and demo programs, running on the same hardware.

Conflation Figure 13 shows renderings of a contour plot (exported to SVG by the Mathematica software), in which areas of constant color are unions of precisely abutting triangles. When independently rendered triangles are blended together, as many renderers do (e.g., Cairo, Adobe Reader, Apple's Quartz, etc.), the correlated mattes lead to incorrect results and the underlying mesh appears. Our pipeline renders these areas as intended.

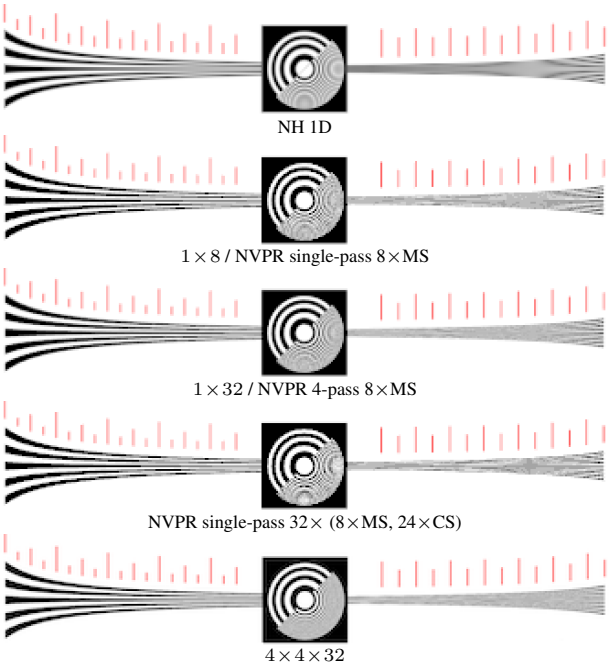


Figure 15: *An aliasing-prone resolution chart rendered by NH's 1D mode is free of noise but shows aliasing and conflation. Mode* 1×8 *shows both noise and aliasing. As expected, mode* 1×32 *reduces noise, but aliasing persists. NVPR's single pass mode* $32\times$ *(*$8\times$*MS,* $24\times$*CS) is too crude an approximation. The sharper antialiasing filter we use in mode* $4\times4\times32$ *is made possible by sample sharing.*

Integration in linear RGB Another common problem is with renderers that evaluate the antialiasing integral (22) in gamma space. This leads to dark regions that look wider than intended. Our renderer can transform colors to linear space before integration and reapply gamma in the end. The difference is obvious on the right of figure 14, which renders text with the correct weight. Unfortunately, many users have grown accustomed to incorrect rendering. As a compromise, we support both alternatives.

Antialiasing quality Figure 15 shows an aliasing-prone resolution chart rendered with different antialiasing strategies. Nehab and Hoppe [2008] (NH) employ a very efficient 1D prefiltering approximation. Although results are very good in certain areas, aliasing and conflation artifacts are clearly visible in others. Modes 1×8 and 1×32 are box-filtered with respectively 8 and 32 samples per unit area. As shown in figure 15, mode 1×8 shows significant amounts of noise. It is included in our tests simply because it is the limit of what Kilgard and Bolz [2012] (NVPR) can accomplish in a single pass using multisampling in current hardware ($8\times$MS). Although the hardware supports a hybrid single-pass mode $32\times$ ($8\times$MS, $24\times$CS), it is too crude an approximation for 1×32. NVPR's demo offers a much better approximation by accumulating 4 passes with $8\times$MS. The amount of aliasing is a property of the box filter and remains the same regardless of the number of samples. Our $4\times4\times32$ mode uses a cardinal cubic B-spline with 512 samples under the 4×4 support of each pixel's filter, sharing samples across overlapping filters. The results are visibly reduced aliasing in challenging areas, and renderings that are virtually free of noise.

Figure 16: *Different levels of detail can be obtained for the same region using OpenStreetMaps. (Left) Paris 50k. (Right) Paris 70k.*

Test cases Table 1 shows statistics for some of the illustrations we used in our tests. These illustrations are available in the supplemental materials. Drops, Car, and Embrace make heavy use of semitransparent gradient fills, but are otherwise simple illustrations. Reschart is the alias-prone resolution chart that contains the pattern of figure 15. Contour appears in figure 13, and is a dense triangulation with flat-colored triangles. Tiger is the standard PostScript sample and contains many opaque overlapping paths that simulate gradients. Paper 1 and Paper 2 are SIGGRAPH paper pages using Type 1 and TrueType fonts, respectively (i.e., cubics vs. quadratics). Boston, Paris, and Hawaii are maps. Hawaii appears in figure 17 and includes many overlapping semi-transparent layers. We tested maps of Paris at different scales (from OpenStreetMaps), spanning a large variation in complexity. These maps include finely dashed strokes with decorations that significantly increase the rendering complexity beyond the number of input segments. See figure 16.

Performance Table 1 also shows a performance comparison between our work and those of Kilgard and Bolz [2012] (NVPR) and of Nehab and Hoppe [2008] (NH)[2]. Rendering times do not include preprocessing time, although we also provide preprocessing times and memory consumption for our method.

Let us focus on the 1×32 rendering mode. For each input, the times for the fastest method are shown in blue, and the others in red. The key comparison is between our method and NVPR. This is because NH was optimized for the 1D mode, where it excels. In 1×32 mode, it performs its own supersampling instead of taking advantage of hardware-accelerated multisampling.

Results show that, once input complexities are sufficiently high, our pipeline has the advantage. We believe that the main reason for this behavior is that NVPR renders each individual path one after the other. Even though the hardware rasterizer can process paths much faster than our pipeline, as the number of paths increases, this sequential processing becomes a bottleneck. Our performance advantage can already be noticed while rendering a typical page of text, with subpixel positioning of characters, at 32 samples per pixel and 100 pixels per inch. Grouping shapes with the same paint into a single path would significantly improve NVPR's performance. Unfortunately, this could result in incorrect rendering where such shapes overlap spatially. An optimization along these lines could be implemented at the application level, at least for simple and common cases such as pages of text, where it would be very effective.

Our improvements are even more pronounced for inputs of higher complexity. In fact, due to sample sharing, we can render both faster *and* at a higher quality level. Such results are marked in bold in table 1. We would like to stress that this is not the result of extensive optimization. It is the result of new algorithms that map better to massively-parallel hardware.

[2]Nehab and Hoppe [2008] only include a subset of the inputs we tested.

Figure 17: *Examples of user-defined object-space warps.*

User-defined warps Figure 17 shows three user-defined warps: a twisting warp on the Tiger, a zoom-lens effect on Paper 1, and a projective transformation on a map of Hawaii. The pipeline renders these effects as if the illustration had been warped in object space. The scheduler ensures samples are shared between all pixels with overlapping antialiasing filters while minimizing control-flow divergence as well as memory and bandwidth requirements.

Relative costs of main algorithmic steps Figure 18 shows the relative cost of the main steps in our rendering pipeline, using the output resolutions of table 1. The first two plots show the steps involved in subdivision and pruning (The abstraction process is very fast.) The only detail of note is the pruning of Paper 2, which includes clipping and is therefore more demanding. The third plot shows the steps used in rendering. As expected, most of the time is spent in sampling and integration. The last three examples (those marked with 'w') include a user-defined warp as in figure 17, which stresses the scheduler. Otherwise, scheduling time is negligible.

User-defined warps, scheduling, and integration In the presence of user-defined warps, we must use a more general scheduler and integrator. The scheduler must find the cells where each sample falls, and the integrator must keep track of the unit area they belong to. To measure the overhead of this process, we compare it with the specialized scheduler that we use when no warp is supplied. For the inputs in table 1, this overhead ranges from 35% to 310%, and is more marked for shortcut trees that are densely subdivided.

Scalability to output resolution Figure 19 shows the behavior of our rendering stage as the number of output pixels is increased progressively from 256×256 to 2048×2048. For each sample, image dimensions were selected to maintain the original aspect ratio while matching the specified number of output pixels. Results show that the rendering algorithm scales close to linearly with image resolution. Small deviations are due to the different shortcut tree structures that result for different target resolutions.

Pruning and clipping Although typical illustrations do not include deeply nested clip-paths, the pipeline supports them as specified by the standards. Figure 20 shows one of our test cases. Clipping (or equivalently, occlusion) is the main justification for the pruning

Table 1: *Description of tests and performance comparison. Except for the Paris dataset, images were rendered with 1024 width and proportional height. Number of filled and stroked paths are given, with segments broken into each type. All times are expressed in milliseconds (smaller are better). "Pre." denotes the preprocessing time, "Mem." is memory usage in MiB (i.e.,* mebibyte*, or 2^{20} bytes). All rendering modes (*1×8*, *1×32 *etc.) are explained in the "antialiasing quality" discussion.*

Input	Resolution	Filled paths	Fill segments			Stroked paths	Stroke segments			Total segments	Our method					NVPR		NH	
			L	Q	C		L	Q	C		Pre.	Mem.	1×8	1×32	4×4×32	1×8	1×32	1D	1×32
Car	1024×682	361	701	165	3187	59	32	12	183	4280	28.45	8.68	12.86	14.73	28.85	3.42	10.56		
Drops	1024×1143	204	45		1359					1404	21.77	2.83	14.28	18.59	46.03	2.63	5.11	0.91	33.61
Embrace	1024×1096	225	25		4621					4646	24.06	4.14	15.50	19.38	48.07	2.78	5.08	0.88	31.18
Reschart	1024×625	723	7823		96	24	140			8059	24.96	3.10	8.51	11.14	32.34	2.88	10.84	0.58	19.33
Tiger	1024×1055	236	177		1988	66	16		346	2527	31.04	4.12	12.89	17.24	52.70	2.66	5.50	0.82	34.15
Boston	1024×917	122	1818		13669	1800	137		12470	28094	128.02	46.92	37.22	41.81	71.14	8.28	31.02	2.45	66.77
Hawaii	1024×844	1008	6312	6	43208	131	8	6	2129	51669	115.29	42.15	26.16	29.48	50.50	3.68	14.70		
Paper 1	1024×1325	5099	39573		59708	9	23			99304	53.17	13.22	19.28	23.71	**67.65**	20.80	78.64		
Paper 2	1024×1325	5621	42620	85216	26096	68	111		39	154082	72.51	13.86	10.80	17.50	**35.09**	24.95	95.72		
Contour	1024×1024	53241	188340							188340	77.58	41.29	30.07	30.36	**63.21**	203.93	1025.11		
Paris 70k	470×453	32454	2983		1999	13136	987		1487	7456	101.85	45.96	22.39	21.00	**28.45**	151.67	796.34		
Paris 50k	657×635	32639	2443		4149	13157	759		1679	9030	110.12	51.03	26.82	25.22	**34.96**	155.88	737.20		
Paris 30k	1096×1060	34751	7186		22155	15939	5437		18109	52887	192.94	94.72	49.51	48.81	**78.59**	176.54	904.17		

of the shortcut tree. Enabling pruning in the scene shown in the figure leads to a 25% reduction in memory consumption and 45% improvement in rasterization time. The total time reduction for preprocessing and rasterization is 10%.

Front-to-back rendering Many vector graphics renderers draw shapes back-to-front. This is inefficient when there is substantial overdraw. We address this problem in two ways. First, we proceed front-to-back when rendering. As soon as a sample becomes opaque, the remaining scene content can be safely ignored. In scenes with high depth complexity, this optimization can significantly improve rendering performance (e.g., by 33% in the Paris 30k input). Second, the shortcut tree pruning algorithm eliminates from the stream all paths that would have been completely occluded by an opaque path within a cell. Pruning does not take into account the possibility of multiple semi-transparent paths combining into an opaque layer and obscuring the paths underneath. This would be difficult to accomplish, especially in the presence of gradient fills and textures.

Subpixel positioning of text Our renderer treats character glyphs as regular paths, in object precision. Many renderers pre-render glyphs in image precision. This is especially noticeable when scrolling or resizing text, which causes pre-rendered glyphs to move horizontally or vertically relative to one another. See the animations of Paper 1 available in the supplemental materials.

Shortcut tree behavior Table 2 shows the behavior of shortcut trees for increasing levels of subdivision. The examples were selected to span the range of behaviors observed in practice. Most content in Paper 1 consists of relatively small characters. Once subdivision is deep enough to isolate them, it stops. The high density of detail in Paris 30k, which also includes significant overdraw, forces tree subdivision to proceed further. In general, we do not observe an explosion in the number of segments shared by different cells.

7 Conclusions and future work

The task of rendering vector graphics has traditionally been performed by CPUs. The large increase in computational power of GPUs over the past decade has attracted a significant amount of interest in parallel algorithms for vector graphics rendering, with each innovation requiring less CPU involvement. Although there are certain benefits to a tight integration with the standard 3D rendering pipeline, our work shows the advantages of breaking with legacy in favor of a complete massively parallel redesign of vector graphics rendering. Our work opens the door for a variety of interesting follow-up research and engineering problems:

- Find an auto-tuning method to decide when to stop subdividing shortcut tree cells for best rendering performance;
- Use a separate algorithm to bootstrap the shortcut tree subdivision with a coarse regular grid. This would cut down the number of subdivision passes and could speed up preprocessing;
- Find an efficient and robust solution for the monotonization of rational cubic Béziers, making the entire pipeline closed under projective transformations;
- Parallelize the conversion of stroked paths to filled primitives, including dashes, caps, and joins, and move it to the GPU;
- Add support for filter effects on groups of paths. In particular, perform these operations in parallel whenever multiple groups can be filtered independently;
- Add support for subpixel rendering (e.g., [Betrisey et al. 2000]);
- Add support for transparency groups;
- Add support for alternative color models (e.g., CMYK);
- Add support for mesh-based gradient paints;
- Dynamically compile and load user-defined warping functions;
- Invest significantly more effort optimizing the GPU code;
- Implement the pipeline on the CPU.

We are particularly interested in investigating the implementation of our pipeline on the CPU. On the one hand, it would seem overkill to maintain the preprocessing stage parallel at the segment level and the rendering parallel at the sample and pixel levels. A more coarse division of work between fewer threads should be more appropriate. On the other hand, most of the effort we have invested in minimizing control-flow divergence on the GPU should also be effective when used with the vectorized instructions available in modern CPUs.

8 Acknowledgments

This work has been funded in part by a doctoral scholarship and grants from CNPq, and by an INST grant from FAPERJ. NVIDIA Corporation has generously donated the GPUs used in this project.

References

BALZER, M., SCHLOMER, T., and DEUSSEN, O. 2009. Capacity-constrained point distributions: A variant of Lloyd's method. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)*, 28(3):86.

BETRISEY, C., BLINN, J. F., DRESEVIC, B., HILL, B., HITCHCOCK, G., KEELY, B., MITCHELL, D. P., PLATT, J. C., and WHITTED, T. 2000. 20.4: Displaced filtering for patterned dis-

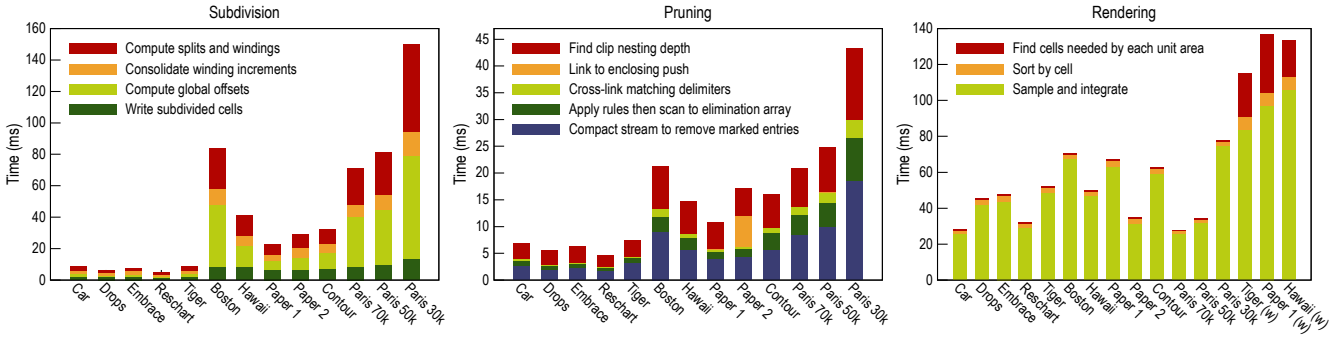


Figure 18: *Relative costs of major steps in the rendering pipeline.*

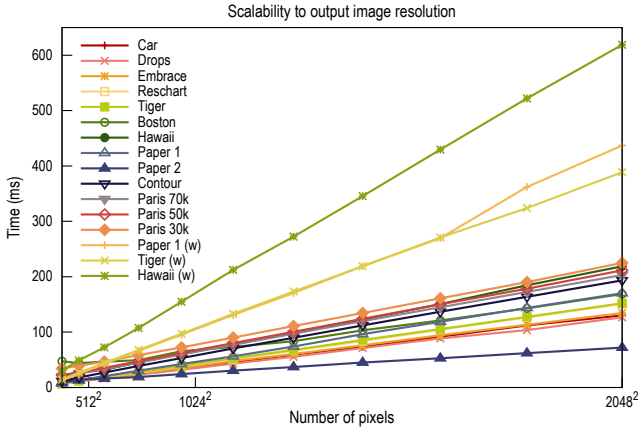


Figure 19: *Rendering times are linear on the number of output pixels. Small variations are due to the different shortcut trees used at different target resolutions.*

plays. *Society for Information Display Symposium Digest of Technical Papers*, 31(1):296–299.

BLINN, J. F. 2005. How to solve a quadratic equation. *IEEE Computer Graphics and Applications*, 25(6):76–79.

CATMULL, E. 1978. A hidden-surface algorithm with anti-aliasing. *Computer Graphics (Proceedings of ACM SIGGRAPH 1978)*, 12 (3):6–11.

CATMULL, E. 1984. An analytic visible surface algorithm for independent pixel processing. *Computer Graphics (Proceedings of ACM SIGGRAPH 1984)*, 18(3):109–115.

CATMULL, E. and ROM, R. 1974. A class of local interpolating splines. In R. Barnhill and R. Riesenfeld, editors, *Computer Aided Geometric Design*, Academic Press, 317–326.

DUFF, T. 1989. Polygon scan conversion by exact convolution. In J. André and R. D. Hersch, editors, *Raster Imaging and Digital Typography*, Cambridge University Press, 154–168.

FINCH, M., SNYDER, J., and HOPPE, H. 2011. Freeform vector graphics with controlled thin-plate splines. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2011)*, 30(6): 166.

FRISKEN, S. F., PERRY, R. N., ROCKWOOD, A. P., and JONES, T. R. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. In *Proceedings of ACM SIGGRAPH 2000*, 249–254.

GUENTER, B. and TUMBLIN, J. 1996. Quadrature prefiltering for high quality antialiasing. *ACM Transactions on Graphics*, 15(4): 332–353.

HAINES, E. 1994. Point in polygon strategies. In P. S. Heckbert, editor, *Graphics Gems IV*. Morgan Kaufmann.

KERR, K. 2009. Introducing Direct2D. MSDN Magazine.

KILGARD, M. 1997. A simple OpenGL-based API for texture mapped text. Silicon Graphics. `ftp://ftp.sgi.com/opengl/contrib/mjk/tips/TexFont/TexFont.html`.

KILGARD, M. J. and BOLZ, J. 2012. GPU-accelerated path rendering. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2012)*, 31(6):172.

KOKOJIMA, Y., SUGITA, K., SAITO, T., and TAKEMOTO, T. 2006. Resolution independent rendering of deformable vector objects using graphics hardware. ACM SIGGRAPH Sketches.

LEFEBVRE, S. and HOPPE, H. 2006. Perfect spatial hashing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2006)*, 25(3):579–588.

LIN, Z., CHEN, H.-T., SHUM, H.-Y., and WANG, J. 2005. Prefiltering two-dimensional polygons without clipping. *Journal of graphics, GPU, and game tools*, 10(1):17–26.

LOOP, C. and BLINN, J. F. 2005. Resolution independent curve rendering using programmable graphics hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2005)*, 24 (3):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(2pt4):499–507.

MCCOOL, M. D. 1995. Analytic antialiasing with prism splines. In *Proceedings of ACM SIGGRAPH 1995*, 429–436.

MITCHELL, D. P. and NETRAVALI, A. N. 1988. Reconstruction filters in computer graphics. *Computer Graphics (Proceedings of ACM SIGGRAPH 1988)*, 22(4):221–228.

NEHAB, D. and HOPPE, H. 2008. Random-access rendering of general vector graphics. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27(5):135.

NEHAB, D. and HOPPE, H. 2014. A fresh look at generalized sampling. *Foundations and Trends in Computer Graphics and Vision*, 8(1):1–84.

NEHAB, D., MAXIMO, A., LIMA, R. S., and HOPPE, H. 2011. GPU-efficient recursive filtering and summed-area tables. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2011)*, 30(6):176.

NEIDER, J., DAVIS, T., and WOO, M. 1993. *OpenGL Programming Guide, Release 1*. Addison Wesley. "Drawing Filled, Concave Polygons Using the Stencil Buffer".

NV_PATH_RENDERING. 2011. *OpenGL extension specification for NV_path_rendering*. NVIDIA Corporation.

OPENVG. 2008. *OpenVG Specification, v. 1.1*. Khronos Group.

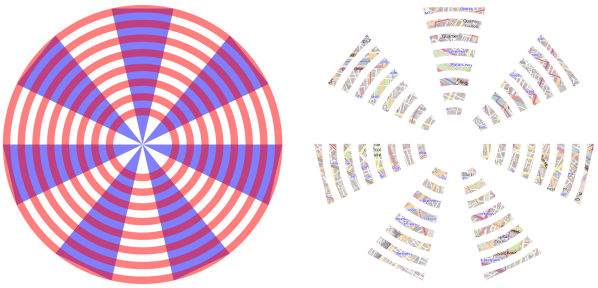


Figure 20: *Nested clip-paths. A shape defined as the intersection a set of concentric rings (in red) and a set of triangles (in blue) is used to clip the map of Paris. Pruning speeds up the process.*

OpenXPS. 2009. *Open XML Paper Specification*. Ecma International, first edition. ECMA-388.

Orzan, A., Bousseau, A., Winnemöller, H., Barla, P., Thollot, J., and Salesin, D. 2008. Diffusion curves: A vector representation for smooth-shaded images. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2008)*, 27(3):92.

Parilov, E. and Zorin, D. 2008. Real-time rendering of textures with feature curves. *ACM Transactions on Graphics*, 27(1):3.

PDF. 2006. *Adobe Portable Document Format, v. 1.7*. Adobe Systems Incorporated, sixth edition.

Porter, T. and Duff, T. 1984. Compositing digital images. *Computer Graphics (Proceedings of ACM SIGGRAPH 1984)*, 18(3): 253–259.

PostScript. 1999. *PostScript Language Reference*. Adobe Systems Incorporated, third edition.

Qin, Z., McCool, M., and Kaplan, C. 2006. Real-time texture-mapped vector glyphs. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 125–132.

Qin, Z., McCool, M., and Kaplan, C. 2008. Precise vector textures for real-time 3D rendering. In *Proceedings of Symposium on Interactive 3D Graphics and Games (I3D)*, 199–206.

Ramanarayanan, G., Bala, K., and Walter, B. 2004. Feature-based textures. In *15th Eurographics Symposium on Rendering*, 265–274.

Ramshaw, L. 1988. Béziers and B-splines as multiaffine maps. In *Theoretical Foundations of Computer Graphics and CAD*, volume 40 of *NATO ASI Series*, 757–776. Springer Berlin Heidelberg.

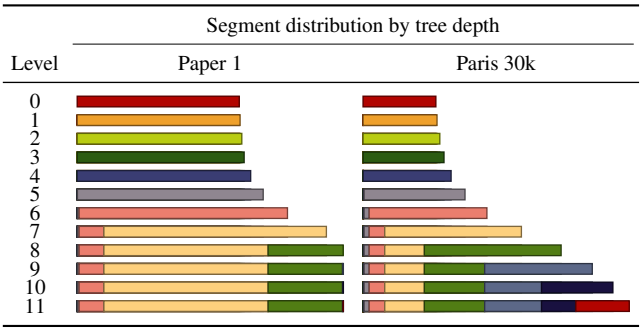
Ray, N., Cavin, X., and Lévy, B. 2005. Vector texture maps on the GPU. Technical Report ALICE-TR-05-003, INRIA.

Rougier, N. P. 2013. Higher quality 2D text rendering. *Journal of Computer Graphics Techniques*, 2(1):50–64.

Salmon, G. 1852. *A Treatise on the Higher Order Plane Curves*. Hodges & Smith.

Sen, P. 2004. Silhouette maps for improved texture magnification. In *Graphics Hardware*, 65–73.

Table 2: *Behavior of the shortcut tree subdivision. Each horizontal bar represents the number of segments after each subdivision level. The visualization shows there is no explosion in the number of segments. Each bar is further divided to represent the number of segments at each tree depth. Bar sizes are normalized by the highest subdivision level in each input and are not comparable across inputs.*



Sen, P., Cammarano, M., and Hanrahan, P. 2003. Shadow silhouette maps. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3):521–526.

Sun, T., Thamjaroenporn, P., and Zheng, C. 2014. Fast multipole representation of diffusion curves and points. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2014)*, 33 (4):53.

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 Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012)*, 31(4):74.

SVG. 2011. *Scalable Vector Graphics, v. 1.1*. W3C, second edition.

SWF. 2012. *SWF File Format Specification, v. 19*. Adobe Systems Incorporated.

Unser, M., Aldroubi, A., and Eden, M. 1991. Fast B-spline transforms for continuous image representation and interpolation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(3):277–285.

Vatti, B. R. 1992. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63.

Wallace, B. A. 1981. Merging and transformation of raster images for cartoon animation. *Computer Graphics (Proceedings of ACM SIGGRAPH 1981)*, 15(3):253–262.

Warnock, J. 1969. *A hidden surface algorithm for computer generated halftone pictures*. PhD thesis, University of Utah.

Warnock, J. and Wyatt, D. K. 1982. A device independent graphics imaging model for use with raster devices. *Computer Graphics (Proceedings of ACM SIGGRAPH 1982)*, 16(3):313–319.

Wylie, C., Romney, G. Evans, D., and Erdahl, A. 1967. Halftone perspective drawings by computer. In *Proceedings Fall Joint Computer Conference*, 49–58.