Random-Access Rendering of General Vector Graphics

Diego Nehab Hugues Hoppe
Microsoft Research

Figure 1: Given a vector graphics drawing, we locally specialize its description to each cell of a lattice, and apply both prefiltering and spatially adaptive supersampling to produce high-quality antialiased renderings over arbitrary surfaces on the GPU.

Abstract

We introduce a novel representation for random-access rendering of antialiased vector graphics on the GPU, along with efficient encoding and rendering algorithms. The representation supports a broad class of vector primitives, including multiple layers of semitransparent filled and stroked shapes, with quadratic outlines and color gradients. Our approach is to create a coarse lattice in which each cell contains a variable-length encoding of the graphics primitives it overlaps. These cell-specialized encodings are interpreted at runtime within a pixel shader. Advantages include localized memory access and the ability to map vector graphics onto arbitrary surfaces, or under arbitrary deformations. Most importantly, we perform both prefiltering and supersampling within a single pixel shader invocation, achieving inter-primitive antialiasing at no added memory bandwidth cost. We present an efficient encoding algorithm, and demonstrate high-quality real-time rendering of complex, real-world examples.

1. Introduction

Vector graphics are commonly used to represent symbolic information such as text, maps, diagrams, and illustrations. Because finding the color of an individual pixel within a vector graphics object requires traversing all its primitives, the object is usually rendered atomically into a framebuffer.

In contrast, raster images offer efficient random-access evaluation at any point by filtering of a local pixel neighborhood. Such random access allows images to be texture-mapped onto arbitrary surfaces, and permits efficient magnification and minification. However, images do not accurately represent sharp color discontinuities: as one zooms in on a discontinuity, image magnification reveals a blurred or jagged boundary.

As reviewed in Section 2, recent *vector texture* schemes explore encoding vector primitives within a raster structure, for instance to support discontinuities in images, or to encode glyph regions.

Our aim is to directly model general vector graphics, composed of multiple semitransparent layers of overlapping gradient-filled and outlined primitives, including thin strokes that would be difficult to antialias properly when represented as filled primitives.

Approach: Like [Ramanarayanan et al. 2004], we construct a lattice in which each cell contains a local graphics description specialized to that cell region. We store this local description as a variable-length stream that is parsed within a programmable pixel shader at rendering time. The complexity of each cell stream is directly related to the number of vector primitives overlapping the cell, and thus complexity can be arbitrary and is only introduced where needed. Moreover, processing time in the pixel shader also adapts to this complexity (subject to local SIMD parallelism), so that large areas of continuous color are rendered quickly.

We show that traditional vector graphics can be quickly converted into cell-specialized streams using a novel lattice-clipping algorithm that is simpler and asymptotically faster than hierarchical clipping schemes. A unique aspect of the algorithm is that it requires a single traversal of the input graphics primitives.

Benefits: Cell-specialized streams nicely encapsulate the data involved in rendering a region of the domain. Conventional vector graphics rasterization would traverse all input primitives and perform row-by-row updates to the output raster. Instead, we read a small subset of specialized primitives (which gets cached across nearby pixels), combine the primitives within the shader (albeit with significant computation), and directly obtain the color of any individual pixel. Such low-bandwidth localized memory access on both input and output should become increasingly advantageous in many-core architectures.

Another benefit is antialiasing. Because cell streams provide a (conservative) list of all primitives overlapping a pixel, we can evaluate an antialiased pixel color in a single rendering pass, without resorting to A-buffer fragment lists [Carpenter 1984]. Our scheme combines the power of two antialiasing techniques: (1) approximate prefiltering within each vector primitive, and (2) supersampling across primitives. Although supersampling adds computational cost, it involves no extra memory access. Moreover, we show that it is easy to spatially adapt the supersampling density to local graphics complexity, e.g. falling back to a single sample per pixel in regions of continuous color. Such sampling adaptivity has only recently been explored in real-time rendering applications [Persson 2007].

Our cell-specialized graphics inherit advantages already shown in vector texture schemes. Shape primitives can be encoded using lower-precision (e.g. 8- or 16-bit) cell-local coordinates. Also, since the vector graphics can be evaluated entirely in a shader, they can be mapped onto general surfaces just like texture images.

Contributions:

- Variable-length cell streams for locally specialized encoding of general vector graphics including multiple layers and strokes;
- Efficient construction from a vector graphics input, using a novel and efficient lattice-clipping algorithm;
- Concept of overlapping extended cells for correct prefiltering;
- Fast computation of approximate distance to a quadratic curve;
- Fast anisotropic prefiltering approximation of thin strokes;
- Low-bandwidth high-quality antialiasing by combining prefiltering and supersampling in a single pass;
- Random-access vector graphics with linear and radial gradients.

Limitations:

- Rendering from cell-specialized representations assumes a static layout of graphics primitives in the domain, so animations would require re-encoding dynamic shapes at each frame (which fortunately is very fast);
- The description of each vector path segment is replicated in all cells over which it overlaps, but there is little effective storage overhead because segments typically have small footprint;
- All cells in the interior of a filled shape must include the shape color, just as in an ordinary image; on the other hand, there is no need to store a tessellation of the shape;
- The current implementation does not support all vector graphics attributes, such as stylized strokes or image-space blur filters, but these can be implemented as pre- or post-processing;
- Filtered minification requires fallback to a mipmap image pyramid, but this is true of all other approaches;
- Because the cell descriptions have variable lengths, we must use an indirection scheme to compact the data.

2. Related work

Several schemes incorporate sharp outlines in a raster texture by encoding extra information within its pixels. Prior GPU schemes limit the complexity of the outline within each image cell, such as a few line segments [Sen et al. 2003, 2004; Tumblin and Choudhury 2004; Lefebvre and Hoppe 2006], an implicit bilinear curve [Tarini and Cignoni 2005; Loviscach 2005], a parametric cubic curve [Ray et al. 2005], two quadratic segments [Parilov and Zorin 2008], or a fixed number of corner features [Qin et al. 2006]. A drawback of fixed-complexity cells is that small areas of high detail (e.g. cities on maps, or font serifs) require fine lattices, which globally increases storage cost. A quadtree structure provides adaptivity [Frisken et al. 2000], but still limits the number of primitives at the leaf nodes. Our variable-length cell representation allows for graphics of arbitrary complexity.

Most prior schemes consider a single layer of non-overlapping vector graphics. An extension explored by Ray et al. [2005] is to create a specialized shader that implements a fixed compositing hierarchy of several vector textures. However, such hierarchies may become impractical as the input complexity increases. Moreover, the high evaluation cost is uniform over all pixels. In essence, our scheme adaptively simplifies this hierarchy per cell.

The feature-based textures of Ramanarayanan et al. [2004] allow each image texel to contain an arbitrary set of regions used to override smooth interpolation. Their approach should be implementable on present GPUs, given a scheme to pack the variable-length texel descriptions. Whereas their strategy is to add discontinuities to raster images, ours is to directly render general vector graphics. This requires high-quality prefiltering (especially of thin strokes) and correct blending of transparent gradients, neither of which seems feasible with a region-based decomposition. Our strategy of maintaining a layered description and representing paths explicitly was guided by these requirements.

Like [Frisken et al. 2000; Loop and Blinn 2005; Ray et al. 2005], we use approximate distances to primitives for screen-space prefiltering. Qin et al. [2008] present an iterative method for computing precise distances to curved segments within their radius of curvature. We instead present a formula for approximate distance to quadratic segments that is spatially continuous and faster to evaluate.

The recent work of Qin et al. [2008] builds on some of the ideas presented in a previous version of this paper [Nehab and Hoppe 2007]. In particular, Qin et al. also consider general vector graphics defined as layers of filled and stroked primitives, use extended cells for correct prefilter antialiasing, and render thin curved strokes using a distance function. Whereas they use corners to attempt to infer the sign of the distance from the nearest feature (which can lead to artifacts), our features are simply the path segments themselves. This simplicity, which comes from the fact that our inside/outside classification is separate from the distance computation, leads to an efficient representation that guarantees exact interior classification.

3. Our vector graphics representation

Our basic shape primitive is a path with linear and/or quadratic segments specified by a sequence of 2D points. Cubic segments are adaptively subdivided into quadratic ones; these form an excellent approximation for rendering purposes. Paths are defined by lists of points, each one marked by one of four possible tags: *Moveto*, *Drawto*, *Curvepoint*, or *Last*, as shown in Figure 2a.

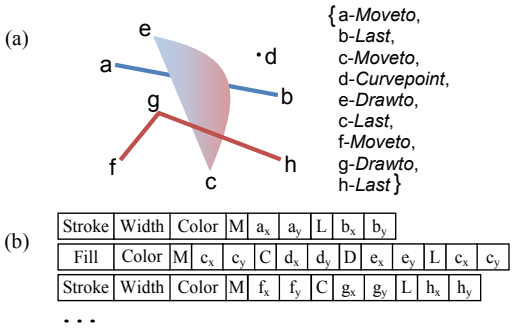


Figure 2: Example vector graphics and its encoding.

A *layer* associates a rendering state to the path, including whether it is stroked and/or filled, its color, and stroke width. A filled path must be closed; we define its interior using the even-odd fill rule. The overall vector graphics consists of a back-to-front ordered list of layers, and is encoded as a stream of instructions (Figure 2b). The color specification consists of either a constant RGBA tuple or an index into an array of gradient descriptors (Section 6).

4. Rendering

Most rendering algorithms for vector graphics objects operate by rasterizing the input into an uniformly sampled output image. Instead, to support random-access, we must design an algorithm that can efficiently evaluate the color at any given point, in any order, as requested by a pixel shader program.

(a) (b) (c) (d)

Figure 3: The importance of compositing in the correct color-space. (a) Correct, linear coverage computation. (b) Incorrect thicker results due to nonlinear coverage. (c) Nonlinear intrinsic transparency expected by the artist. (d) Result of linear compositing. Our hybrid blending operation combines (a) and (c) to always produce the correct/expected results.

The basic approach is to compute the color (including a partial coverage opacity) contributed by each graphics layer at the current pixel p, and to composite these colors in back-to-front order, all within a single invocation of the pixel shader.

4.1 Compositing

When compositing, it is vital to distinguish between the two sources of a layer's transparency: the intrinsic transparency from its color specification (alpha component), and the partial coverage opacity near primitive boundaries. Partial coverage must be dealt with in a linear color-space, lest dark primitives appear thicker than they should [Blinn 1998] (see Figure 3a-b). Unfortunately, most content creation tools represent colors in a nonlinear, gamma-adjusted space (e.g. sRGB [Stokes et al. 1996]), and perform compositing in this space. Misguided by this feedback, artists select transparency parameters that would look entirely different if linearly composited (see Figure 3c-d). Caught between the hammer and the anvil, we must resort to a hybrid blending operation that independently treats each source of transparency.

Our hybrid blending operation takes an sRGB input layer color $\tilde{f}$ (including an intrinsic alpha), a coverage opacity $o \in [0,1]$, and a background color c in linear space, and returns the composite color also in linear space:

$$\text{blend}\big(c, \tilde{f}, o\big) = \text{lerp}\left(\text{sRGB}^{-1}\left(\text{over}\big(\tilde{f}, \text{sRGB}(c)\big)\right), c, o\right).$$

For efficiency, the over compositing operation can assume premultiplied alpha, as long as the color-space conversion functions sRGB and sRGB^{-1} also operate in that space. Needless to say, the process would be much simpler if all operations could be performed in a linear color space, as they were meant to be.

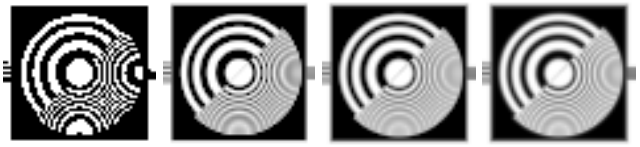
The following section describes how to compute the partial coverage opacity o by prefiltering the vector graphics.

4.2 Prefiltering

Prefiltering eliminates high frequencies from the vector graphics *before* sampling to prevent them from causing aliasing artifacts. Given a kernel $k(p)$ (i.e. a low-pass filter) and an indicator function $l(p)$ for a layer (i.e. $l(p) = 1$ inside and 0 outside), the partial coverage opacity is given by a screen-space 2D convolution $o(p) = k * l = \iint k(p' - p) \cdot l(p')\, dp'$.

Although exact prefiltering would be too costly to evaluate in real-time, we employ a variety of simplifying assumptions that make the process practical. For instance, we prefilter each layer independently, which ignores the fact that layer compositing and prefiltering do not generally commute, and thus may be incorrect when multiple layers partially overlap a pixel.

Furthermore, within each path, we consider only the closest segment to a pixel (which is incorrect when multiple path features overlap a pixel). Then, in the case of filled paths, we locally approximate l by an infinite half-space (which breaks down near sharp angles). Stroked paths can similarly be approximated by the difference between two half-spaces, offset each way by half the stroke width. Since convolution is a linear operation, stroke prefiltering also reduces to convolution between a kernel and a half-space: $k * (l_{+w} - l_{-w}) = k * l_{+w} - k * l_{-w}$.



No prefiltering Box Parabolic Gaussian

Figure 4: Prefiltering results using different kernels. The Moiré patterns are due to inter-primitive interactions, and are handled by the super-sampling strategy of Section 4.5.

Kernel choice: If the kernel is radially symmetric, or if the kernel is rotationally aligned with the half-space (which does not seem to degrade the quality of the results), the convolution can be expressed as a function $o(d)$ of the signed distance d between the pixel and the half-space [Gupta and Sproull 1981].

We have experimented with box, parabolic, and Gaussian kernels. Figure 4 shows prefiltered renderings of the inset resolution chart. Since the simple box kernel is clearly unsatisfactory, our preferred choice is the parabolic kernel $k_p(d) = \frac{4}{3}(1 - d^2),\ d \in [-1,1]$, which yields

$$o_p(d) = \tfrac{1}{2} + \tfrac{1}{4}(3d - d^3) = \text{smoothstep}(-1,\, 1,\, d).$$

This provides a good tradeoff between quality and simplicity, and is equivalent to the *transition function* intuitively proposed by [Qin et al. 2006].

Evaluation: Each **filled** layer successively updates the pixel color c, by means of the hybrid blending operation defined in Section 4.1:

$$c = \text{blend}\big(c,\ \textit{fillcolor},\ o_p(d)\big).$$

Similarly, a **stroked** layer with half-width w updates the pixel color c according to the rule:

$$c = \text{blend}\Big(c,\ \textit{strokecolor},\ o_p(|d| + w) - o_p(|d| - w)\Big).$$

For paths that are both filled and stroked, we perform two successive blend operations, first with the fill, and next with the stroke.

For simplicity, we first compute d' and w' in texture space, then map them to screen space. The sign of d', which defines the interior of filled primitives, is obtained by shooting a ray from the pixel to the right $(+x)$, tracking the number of intersections with the path segments [Foley et al. 1990]. For each segment, we determine the number h_i of ray intersections and the vector v_i from the pixel to the closest point on the segment (see Sections 4.3 and 4.4). We combine these as

$$v = \arg\min_{v_i} \|v_i\| \quad \text{and} \quad h = \left(\textstyle\sum_i h_i\right) \bmod 2,$$

to obtain

$$d' = -(-1)^h \|v\|.$$

Now consider the half-space perpendicular to vector v. Using the Jacobian J of the map from screen to texture coordinates (which we obtain directly from the built-in ddx/ddy operations), we can transform this half-space to screen coordinates. There, it lies at a distance $d = s(v)\, d'$ from the pixel, with an anisotropic scaling factor

$$s(v) = \frac{\|v\|}{\|Jv\|}.$$

Stroke widths are similarly scaled by $s(v)$, so that $w = s(v)\, w'$.

(a) Isotropic filtering for comparison.

(b) Prefiltering of Section 4.2, using a parabolic filter.

(c) Supersampling of Section 4.5, $k = 400$ (no prefiltering).

Figure 5: Anisotropic antialiasing of filled and stroked primitives.

The example in Figure 5 shows two characters, a filled 'A' and a stroked 'B', rendered at a grazing angle. The difference between (a) and (b) shows the importance of anisotropy. The similarity between (b) and (c) shows the accuracy of our approximation.

The overall rendering algorithm can be summarized as follows:

```
Initialize the pixel color, e.g. to white or transparent.
for (each layer)
    Initialize the pixel winding parity h to 0 and distance vector v to ∞.
    for (each segment in layer)
        Shoot ray from pixel through segment and update h.
        Compute distance vector to segment and update v.
    Compute texture-space signed distance d′ from h and v.
    Use v to obtain screen-space signed distance d and stroke width w.
    if (fill) Blend fill color with prefiltered opacity.
    if (stroke) Blend stroke color with prefiltered opacity.
```

We now describe the process of obtaining the winding number increment h_i and vector distance v_i for each segment type.

4.3 Linear segments

For each linear segment (b_i, b_{i+1}), the ray intersection count $h_i \in \{0,1\}$ and distance vector v_i are given by (see Figure 6a):

$$t_i = \frac{p_y - b_{i,y}}{b_{i+1,y} - b_{i,y}}, \qquad q_i = \mathrm{lerp}(b_i, b_{i+1}, t_i),$$

$$h_i = \begin{cases} 1, & \text{if } 0 \le t_i \le 1 \text{ and } q_{i,x} > p_x \\ 0, & \text{otherwise,} \end{cases}$$

$$t_i' = \mathrm{clamp}\left(\frac{(p - b_i)\cdot(b_{i+1} - b_i)}{(b_{i+1} - b_i)\cdot(b_{i+1} - b_i)}, 0, 1\right),$$

$$v_i = p - \mathrm{lerp}(b_i, b_{i+1}, t_i').$$

If the ray passes exactly through a vertex at rendering time, we perturb its vertical coordinate imperceptibly for robustness.

4.4 Quadratic segments

Each segment (b_{i-1}, b_i, b_{i+1}), with b_i tagged *Curvepoint*, defines a Bézier curve $b(t) = (1-t)^2 b_{i-1} + 2(1-t)t b_i + t^2 b_{i+1}$ on the interval $0 \le t \le 1$. The intersection count $h_i \in \{0,1,2\}$ and vector v_i are found as follows (see Figure 6b).

Any intersections of the $+x$ ray from pixel p with the (infinite) quadratic curve are found as the roots t_1 and t_2 of the quadratic equation $b_y(t_j) = p_y$. For each root t_j, we increment the ray intersection count w_i if the point $b(t_j)$ lies within the curve segment (i.e. $0 \le t_j \le 1$) and to the right of p (i.e. $b_x(t_j) > p_x$).

The quadratic equation $b_y(t) = p_y$ becomes linear if the parabola axis is horizontal, i.e. $b_{i,y} = \frac{1}{2}(b_{i-1,y} + b_{i+1,y})$. To avoid having to test this condition at runtime, we imperceptibly perturb the point b_i by one bit during encoding.

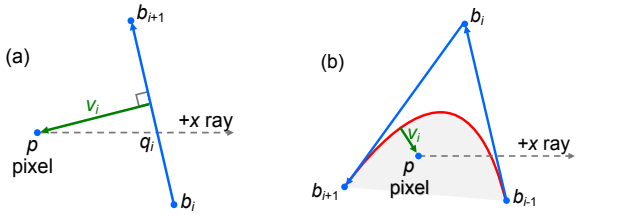


Figure 6: Ray intersection testing and distance computation, for (a) linear and (b) quadratic path segments.

Computing the distance to the quadratic Bézier curve involves finding the roots of a cubic polynomial. Analytic roots require transcendental functions and are thus expensive to evaluate [Blinn 2006]. Iterative solvers [Qin et al. 2008] can be a faster alternative within the radius of curvature of the curve.

Instead, like Loop and Blinn [2005], we develop a fast approximate technique based on *implicitization*. But whereas their triangle-bounded rasterization approach only needs distance to an infinite curve, we require distance to a curve *segment*, i.e. taking the endpoints into account.

At rendering time, we convert the Bézier curve to its implicit quadratic representation $f(p) = 0$, given by the Bezout form of its resultant [Goldman et al. 1984]:

$$f(p) = \beta(p)\delta(p) - \alpha^2(p), \text{ with}$$

$$\beta(p) = 2\,\mathrm{Det}(p, b_i, b_{i-1}),\ \delta(p) = 2\,\mathrm{Det}(p, b_{i+1}, b_i),$$

$$\alpha(p) = \mathrm{Det}(p, b_{i-1}, b_{i+1}), \text{ and } \mathrm{Det}(p,q,r) = \begin{vmatrix} p & q & r \\ 1 & 1 & 1 \end{vmatrix}.$$

The zero set of the first-order Taylor expansion of f centered at p defines a line, and the closest point p' to p on this line is given by

$$p' = p - \frac{f(p)\nabla f(p)}{\|\nabla f(p)\|^2}.$$

For p' exactly on the curve, we can find the parameter t'such that $b(t') = p'$ by *inversion*. In the quadratic Bézier case, Goldman et al. [1984] show that inversion can be obtained by

$$t' = \frac{u_j'}{1 + u_j'}, \quad \text{with either } u_1' = \frac{\beta(p')}{\alpha(p')} \text{ or } u_2' = \frac{\alpha(p')}{\delta(p')}.$$

In fact, if p' lies on the curve, the resultant vanishes, and $u_1' = u_2'$. Since this is generally not the case, the two values differ, and we must rely on an approximation. Our contribution is the choice

$$\bar{u} = \frac{\alpha(p') + \beta(p')}{\alpha(p') + \delta(p')}$$

which gives

$$\bar{t} = \frac{\alpha(p') + \beta(p')}{2\alpha(p') + \beta(p') + \delta(p')}.$$

Notice that $\bar{t}$ is *exact* whenever p' is on the curve. The biggest advantage of $\bar{t}$, however, is that it is *continuous*, even when p' coincides with one of the control points, where either u_1' or u_2' becomes undefined. The denominator of $\bar{t}$ is zero only if b_0, b_1, and b_2 are collinear, in which case we would have converted the primitive to a linear segment. The same holds for $\|\nabla f(p)\|$.

From $\bar{t}$, we then obtain the distance vector as

$$v_i = p - b\big(\mathrm{clamp}(\bar{t}, 0, 1)\big).$$

Note that implementation of these formulas is simpler if the Bézier points b are translated so as to place p (and later p') at the origin. The HLSL source code shown below compiles to just 32 assembly instructions, contains only two division instructions and no transcendental functions, square roots, loops, or branching.

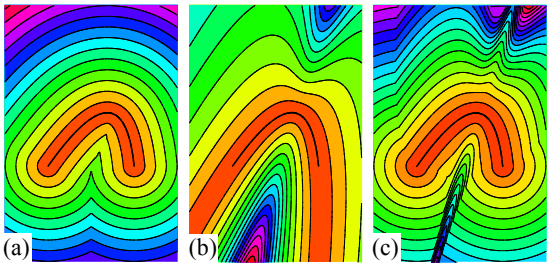


Figure 7: (a) Exact distance to a quadratic curve involves cubic polynomial roots. (b) A first-order implicit approximation [Loop and Blinn 2005] does not capture the segment endpoints. (c) Our inversion-based approximation is fast, and visual analysis reveals that it is sufficiently accurate in the vicinity of the curve.

```
inline float det(float2 a, float2 b) { return a.x*b.y-b.x*a.y; }
// Find vector v_i given pixel p=(0,0) and Bézier points b_0,b_1,b_2.
float2 get_distance_vector(float2 b0, float2 b1, float2 b2) {
    float a=det(b0,b2), b=2*det(b1,b0), d=2*det(b2,b1);     // α,β,δ(p)
    float f=b*d-a*a;                                        // f(p)
    float d21=b2-b1, d10=b1-b0, d20=b2-b0;
    float2 gf=2*(b*d21+d*d10+a*d20);
    gf=float2(gf.y,-gf.x);                                  // ∇f(p)
    float2 pp=-f*gf/dot(gf,gf);                             // p'
    float2 d0p=b0-pp;                                       // p' to origin
    float ap=det(d0p,d20), bp=2*det(d10,d0p);               // α,β(p')
    // (note that 2*ap+bp+dp=2*a+b+d=4*area(b0,b1,b2))
    float t=clamp((ap+bp)/(2*a+b+d), 0,1);                  // t̄
    return lerp(lerp(b0,b1,t),lerp(b1,b2,t),t);             // v_i = b(t̄)
}
```

We have found that this approximate distance is sufficiently accurate for the purpose of prefiltering and thin-stroke rendering, as visually demonstrated in Figure 7. Note the correct behavior at the endpoints, and the high precision in the neighborhood of the curve, where it is most relevant for prefiltering. Wide stroke paths whose outlines are poorly approximated are pre-converted to filled primitives. It would be desirable to characterize the maximum error analytically, but unfortunately this seems hard.

4.5 Supersampling

The prefiltering strategy described in Section 4.2 relies on simplifying assumptions that break down when many primitives overlap a pixel. One possible high-quality approximation in traditional rasterization is the A-buffer [Carpenter 1984], which maintains per-pixel lists of fragments, with each fragment containing a subpixel bitmask. However, this approach is challenging to implement efficiently in hardware [Winner et al. 1997].

Because cells store the full list of relevant vector primitives (on the current surface), we can evaluate and combine colors at multiple subpixel samples, without any added bandwidth.

The first step is to determine the footprint of the pixel in texture space, just as in anisotropic texture filtering [Heckbert 1989]. This footprint could overlap several cells, which would require parsing of multiple cell streams. Fortunately, use of extended cells (Section 5.1) provides a margin so we need only consider the current cell. When a pixel footprint grows larger than the overlap region, we transition to conventional anisotropic mipmapping.

Our system evaluates a set of k weighted samples within a parallelogram footprint as shown in Figure 8. More precisely, given a sampling pattern defined by local displacements u_j and weights a_j, the final pixel color is computed as the weighted sum $\sum_j a_j c_j$ where c_j is the sample color evaluated at displaced position $p_j = p + Ju_j$. We use sampling patterns P(1,4,2) or P(1,4,4) from [Laine and Aila 2006], corresponding to k=4 or 8. Because the footprints overlap in screen space, it would be ideal if samples could be shared among adjacent pixels, as suggested by Laine and

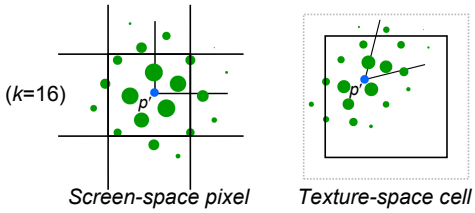


Figure 8: Supersample antialiasing evaluates color at multiple samples (shown in green) during a single shader evaluation.

Aila [2006]. Unfortunately current hardware does not permit this, but perhaps this will be possible in the future.

We traverse the cell stream just once, updating all samples c_j as each primitive is decoded. This requires allocating a few temporary registers per sample (accumulated color c, accumulated winding parity h, and shortest distance vector v). For each subpixel sample, we still evaluate the prefilter antialiasing of Section 4.2, but with a modified Jacobian matrix $J' = \sqrt{s/k}\,J$ where s is the kernel support area in pixels, to account for the changed inter-sample spacing.

The number k of samples is adaptively selected based on the cell complexity. In our current system, we use the simple approach of overriding k to equal 1 in cells containing a single graphics layer with uniform color. An alternative would be to let the cell encoding explicitly specify the desired supersampling complexity.

Figure 9 compares the rendering quality with different antialiasing settings. Although the computation has a cost that scales as $O(k)$, it is performed entirely on local data, and is therefore amenable to additional parallelism in future GPUs. For ground-truth, we use k=400 samples, weighted according to the $(\frac{1}{3},\frac{1}{3})$ kernel of Mitchell and Netravali [1988]. Notice how results are close to ground-truth with just 8 samples/pixel, but *only* if prefiltering is enabled.

The overall rendering algorithm with supersample antialiasing can be summarized as follows:

```
for (each sample)
    Initialize the sample color, e.g. to white or transparent.
for (each layer)
    for (each sample i)
        Initialize the sample winding parity h_i to 0.
        Initialize the sample distance vector v_i to ∞.
    for (each segment in layer)
        for (each sample i)
            Shoot ray from sample through segment, and update h_i.
            Compute absolute distance to segment, and update v_i.
    for (each sample i)
        Use v_i to obtain screen space distance d_i and stroke width w_i.
        if (fill) Blend fill color over sample with prefilter opacity.
        if (stroke) Blend stroke color over sample with prefilter opacity.
Compute the weighted sum of sample colors to obtain the pixel color.
```

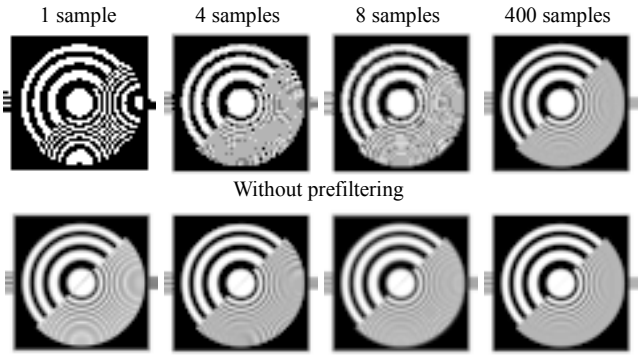


Figure 9: Use of supersampling to resolve interactions between multiple primitives per pixel, with and without the prefiltering of Section 4.2.

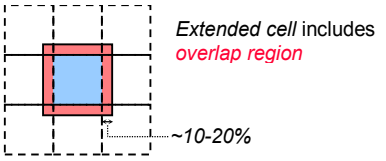


Figure 10: For correct antialiasing, each cell stores all primitives that affect an extended cell region.

5. Conversion to specialized cells

We now describe our method for converting vector graphics to a cell-specialized description, to enable compact storage and efficient runtime evaluation. Because the conversion algorithm is very fast, it may also prove useful for software rasterization, especially on many-core processors.

5.1 Extended cells

First, we must identify the set of primitives that could contribute to the rendering of any pixel falling within a given cell. Obviously, any filled path must be included if its interior overlaps the cell. Stroked paths must be included if the path outline (offset either way by half the stroke width) overlaps the cell.

Furthermore, when either prefiltering or supersampling is enabled, a primitive must also be represented in a cell if its convolution with the associated kernel overlaps the cell, i.e., the minimum screen-space distance between the primitive and the cell is less than the finite kernel support (see Section 4.2 and 4.5). This screen-space distance may vary at runtime. As the cells shrink to the size of a screen pixel, the width of the kernel may become as large as an entire cell. At extreme minification, several cells map into individual pixels, so antialiasing breaks down, and one must instead transition to a conventional raster mipmap pyramid.

To allow satisfactory antialiasing up to a reasonable minification level (where we transition to a mipmap), we find the primitives that overlap an *extended cell* as illustrated in Figure 10. Growing this extended cell allows a coarser mipmap pyramid, but increases the length of the cell streams since more primitives lie in the extended cell. We have found that a good tradeoff is to set the overlap band to be 10-20% of the cell size.

5.2 Cell-based specialization

Although specializing stroked paths to an extended cell is trivial, dealing with filled primitives requires some form of polygon clipping. Of course, independently clipping each filled input path against each extended cell in an entire lattice would be prohibitively inefficient. The time required (say, by using an extension of [Sutherland-Hodgman 1974] to piecewise quadratic segments) would be proportional to $O(nr^2)$, where n is the number of input vertices and r^2 is the number of grid cells. To accelerate the process, we could perhaps use recursive subdivision over a quadtree of cells, in line with what is described by [Warnock 1969]. This strategy could reduce the time complexity to $O(n \log r + a)$, where a is the output size.

Fortunately, we can do even better than that. Our *lattice-clipping* algorithm streams over the primitives just once, and directly clips each filled path against every cell that overlaps it. The resulting time complexity is only $O(n + r + a)$, and the simplicity of the algorithm contributes to its speed in practice.

To achieve a more compact representation, we exploit knowledge of our particular rendering algorithm. Specifically, since rendering is based on shooting rays in the $+x$ direction, we can safely omit segments that lie above, to the left, or below each extended cell, as they never take part in the winding number computation.

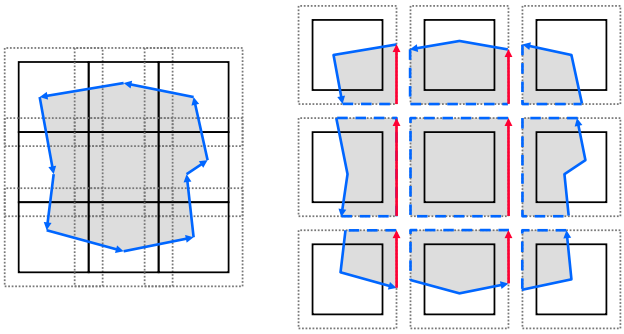


Figure 11: For a filled shape, polygon clipping to each extended cell gives the paths on the right; the dashed segments have no effect on our winding rule computation and are therefore ignored, whereas the red segments must be included.

5.3 Fast lattice-clipping algorithm

Figure 11 shows an example of the desired output. The main difficulty is the introduction of auxiliary segments on the right boundary of each extended cell (shown in red), so that rays cast in the $+x$ direction encounter the correct number of intersections. In the top row of the example, these auxiliary segments are connected to the bottom of the cell. However, in the bottom row they are connected to the top. Unfortunately, there is no local decision procedure for determining which is the case at hand. Notice that, compared to a downward auxiliary segment, an upward auxiliary segment uniformly increments the winding number within the cell. The trick is therefore to make a consistent, arbitrary decision, and correct it afterwards.

To that end, we always extend the path segments crossing the right boundary of a cell to the top right corner of the cell. Figure 12 shows this intermediary representation (notice the green arrows). Whenever a path segment crosses the bottom of a cell, we know that every cell to its left will make a mistake. We therefore record into the cell a change Δh in winding number, to affect all cells to the left in that row ($+1$ for upward segment and -1 for downward segment). Once all segments have been processed, we efficiently traverse each row right-to-left, integrating the winding number changes. For each cell with a resulting nonzero winding increment, we add an appropriate number of upward or downward auxiliary segments, spanning the entire row (see the red arrows in the figure). The resulting green and red edges are finally merged together to produce a result equivalent to that of Figure 11.

For completeness we provide here a more detailed algorithm:

```
for (each segment in a layer)
    Enter the segment into the lattice cell(s) that it overlaps,
      clipping it to the cell boundaries.
    if the segment leaves a cell through its right boundary,
      add a segment from the intersection to the top right corner.
    if the segment enters a cell through its right boundary,
      add a segment from the top right corner to the intersection.
    if the segment enters a cell through its lower boundary,
      increment Δh_c on the cell.
    if the segment leaves a through its lower boundary,
      decrement Δh_c on the cell.
Sort all modified Δh_c (merge rows, bucket sort columns, split rows)
for (each row of modified cells)
    Initialize the winding number h = 0 (associated with the row).
    for (each span of cells in right-to-left order) {
      Add |h| vertical segments on the right boundary of the cell,
        pointing up if h > 0, or down otherwise.
      Merge the cell segments if possible.
      Update the winding number as the running sum h = h + Δh_c.
Clear only the modified Δh_c (i.e., using linked lists)
```

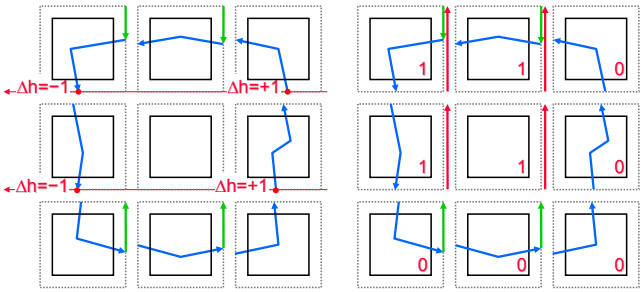


Figure 12: To recover correct winding numbers within the cells, our fast algorithm inserts auxiliary segments (green) on right boundaries, and makes a right-to-left sum of bottom-boundary intersections to appropriately insert right-boundary edges (red). After some local simplification, the result is equivalent to that in Figure 11.

Overall the algorithm is extremely fast, processing even our most complicated example (~100K segments) in less than a second (see Table 3). For the case of the even-odd fill rule, the algorithm can be simplified further to preserve just the parity of the winding number. Figure 13 shows an example with an intricate self-intersecting shape. On the right, the figure shows the resulting intermediary and final specialized encodings for selected cells.

As a final step, we convert all points to a $[0 \ldots 1]^2$ coordinate system over the extended cell. In the rare case that a middle *Curvepoint* Bézier control point lies outside the extended cell, we recursively subdivide the curve into smaller Bézier segments.

5.4 Occlusion optimization

When the vector graphics is specialized to a cell, it is possible for the shape within one layer to become completely occluded by one or more layers in front of it. In traditional rendering this would cause overdraw. Now we have the opportunity to locally remove the occluded layer. One could use general polygon-polygon clipping [Greiner and Hormann 1998] to check if any layer is fully occluded by the union of the layers in front of it. In our current system, we simply check if any opaque, filled layer fully occludes the cell, and if so remove all layers behind it. As an example, for the Tiger model in Figure 19, the average cell stream length is reduced from 9.5 to 7.0 words.

6. Implementation details

Our system uses the Microsoft DirectX 10 API to benefit from linear memory buffers. (An earlier implementation using DirectX 9 was slightly more complicated [Nehab and Hoppe 2007].)

Storage of nonuniform cells: Cell streams are variable-length, so we need a data structure to pack them in memory. Note that the cell streams are self-terminating, so it is unnecessary to store their lengths explicitly. We simply concatenate all streams in raster-scan order into a linear memory buffer (Figure 14), letting a 2D indirection table contain pointers to the start of each cell stream. Naturally, we coalesce identical streams to allow for sharing between cells. Although our implementation employs a flat indirection structure, it would be easy to replace it with a variable-rate perfect spatial hash [Nehab and Hoppe 2007] or other space-efficient data structure [Lefohn et al. 2006].

Cell stream encoding: Our streams follow a simple grammar that closely resembles a sequence of instructions (Figure 2b). Each layer starts with a 32-bit RGBA layer header. We reserve one bit of each channel to define 4 flags: *Last*, *Stroke*, *Fill*, and *Gradient*. For stroked paths, the stroke width is stored in the alpha channel. The same layer path can be both stroked and filled in the common case that its stroke color is black and fill color is opaque.

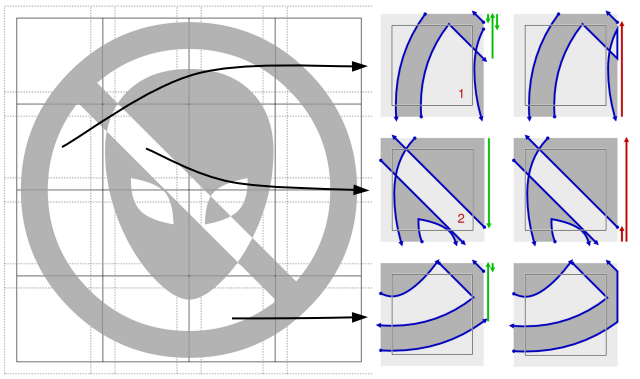


Figure 13: Demonstration of our fast cell-specialization algorithm on a complex shape with self-intersections.

Gradient fills can be used to map 2D global coordinates to colors. This mapping consists of four stages: (1) an affine mapping M from 2D lattice coordinates to 2D gradient space, (2) a linear or radial mapping from 2D gradient space to a parameter $t' \in \mathbb{R}$, (3) a wrapping function from t' to $t \in [0,1]$, (4) and a piecewise-linear color ramp function C sampled at t.

We encode these gradient mappings as follows. We store an approximation of each color ramp function C as a rasterized 1D texture, and concatenate these into a single 1D texture.

To evaluate the gradient color for a layer, we first obtain the point $p_g = Mp$ in gradient space. Then, for linear gradients, t' is simply given by the x-coordinate of p_g (Figure 15 left). Radial gradients are parameterized by a scalar c_x that defines the center $(c_x, 0)$ of a unit circle in gradient space. The value t' is then given by the ratio $\|p_g\|/\|q\|$, where point q is the intersection of the ray $(0, p_g)$ with the unit circle (Figure 15 right). Thus, a radial gradient descriptor requires the full 2×3-matrix M plus the value c_x (7 floats); a linear gradient requires only the first row of M (3 floats). We pack these coefficients into 4-vectors, using the leftover element to store an offset into the concatenated ramp texture. We store these in a linear buffer of gradient descriptors.

If the *Gradient* flag of a layer header is set, we use the color channels to encode an index into the gradient descriptor buffer, and a scaling factor used in accessing the gradient ramp texture. The type of gradient (linear or radial) and the wrapping function for $t' \to t$ (clamp, reflect, or repeat) are encoded by three bits in the layer header.

Each layer header is followed by a stream of cell-local coordinates (x, y), quantized to 16 bits. We reserve one bit of each coordinate to encode the 4 tags: *Moveto*, *Drawto*, *Curvepoint*, and *Last*. As an optimization, we assume that each layer is implicitly prefixed by a *Moveto* instruction to the lower-right of the cell, as this helps remove a point in many cases (e.g. all 6 cells with red segments in Figure 11). Table 1 shows the storage size of different path configurations per layer.

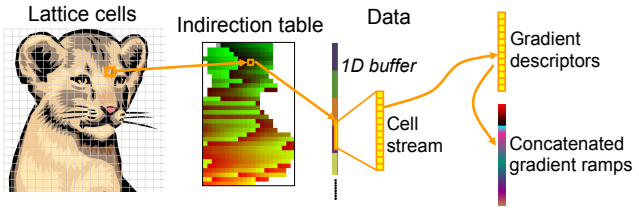


Figure 14: We pack the variable-length cell streams into a 1D memory buffer using a 2D indirection table.

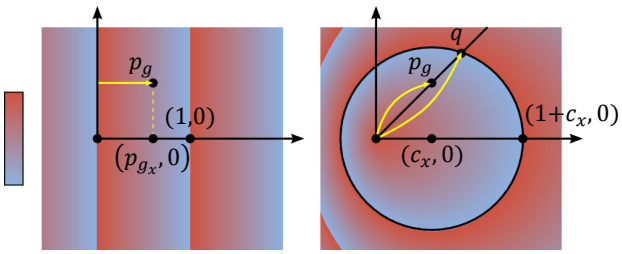


Figure 15: Evaluation of linear and radial color gradients.

Cell contents	Length
Constant color	1*
1 linear segment	3
2 linear segments	4-5
3 linear segments	5-7
1 quadratic segment	4
2 quadratic segments	6-7
3 quadratic segments	8-10

Table 1: Number of 32-bit words required per cell layer as a function of path complexity. Ranges depend on whether *Moveto* points are needed. *Likely shared by many cells.

Cell stream parsing: Within the pixel shader, we interpret the cell stream and evaluate the rendering algorithm of Section 5. This interpretation involves a set of nested loops: over layers, control points, and supersample locations. The resulting pixel shader program complexities for various graphics types are summarized in Table 2.

7. Results and discussion

All results are obtained using an NVIDIA GeForce 8800 GTX, in a 720^2 window. The examples in this section use an overlap region of size 10-20% (depending on the maximum stroke width).

Timing analysis: As far as preprocessing is concerned, the construction method of Section 5.3 takes less than a second to encode even our most complex examples (see Table 3). Although this was not our goal, simple animated textures (i.e., few thousand animated vertices, or restricted to particular layers) should be possible. By spreading the encoding across multiple CPU cores, it should be possible to support even complex animated textures.

Figure 16 plots the rendering rate from cell streams as a function of lattice resolution, for the Tiger in Figure 19. At coarse lattice resolutions, the larger cells increase the number of primitives per cell, which in turn increases the per-pixel rendering cost and therefore lead to slower rendering rates. Conversely, at very fine lattice resolutions, the majority of cells contain a uniform (solid or gradient) color, so rendering speed reaches a plateau.

Our evaluation algorithm should not be memory-bound because the same cell data is reused by many nearby pixels. Indeed, we have run some tests where we let each pixel parse its cell stream without performing rendering computations, and the frame rates increased by a factor of 3–4. This indicates that we are presently compute-bound, and performance will benefit greatly from additional computational cores in future hardware.

The pixel shader makes several coarse-grain branching decisions, based on the number and types of primitives in the cell stream. Fortunately, these decisions are identical for nearby pixels accessing the same stream, which occurs if pixels lie in the same lattice cell, or if adjacent cells share the same cell stream (e.g., in areas of constant color), so the SIMD branching penalty is reduced.

Space analysis: Table 3 shows the size of the stream buffers for each of our examples. Under reasonable lattice resolution settings, the cell specialized representation is comparable in size to the input textual vector graphics description. Naturally, as shown in Figure 16, the stream buffer size grows *linearly* with the lattice resolution. This was expected due to increased sharing of constant color cells at higher resolutions (i.e., only boundaries need be encoded individually). In contrast, the memory requirements of the indirection table grow *quadratically* with the lattice resolution (just like a regular image). Fortunately, if memory is at a premium and the lattice resolution is very high, this problem could be eliminated by replacing the currently flat indirection structure with a hierarchical data structure, such as a quadtree.

Vector primitive types	No prefilter	Prefilter	Prefilter + supersampling
Linear paths, no strokes	117	145	205
Linear paths, with strokes	152	167	230
Quadratic paths, no strokes	160	221	301
Quadratic, with strokes	228	243	327
+ Gradients	310	325	418

Table 2: Pixel shader instruction counts (including loops).

We manually selected lattice resolutions. Table 3 shows the chosen values for each dataset. Note that the memory sizes are on the same order as a typical image representation (but of course cell-specialized graphics allow resolution-independent rendering).

Figure 17 shows a histogram of cell stream lengths for the Lion example. The most common type of cell is one containing a single word indicating a constant color. Figure 18 visualizes the varying length of the cell streams for another encoded illustration.

Examples: Figure 19 presents a collection of vector graphics examples of various types, and Table 3 reports on their complexities, sizes, and rendering rates. Our representation is trivial to map onto surfaces as demonstrated in Figure 1, or to deform by arbitrary parametric warps, as shown in Figure 20.

Dataset	Input vertices	Encoding time (s)	Cell-specialized representation				Rendering (fps)	
			Size (KB)	Lattice resolution	Stream length			
					Avg.	Max.	k=1	k=8
Glass	390	0.02	20	24×32	6.3	69	238.2	44.3
Logo	498	0.01	16	32×19	5.1	39	381.4	70.5
Dice	942	0.02	20	32×20	7.3	85	271.2	42.4
Eyes	1061	0.05	44	64×41	4.3	58	285.9	54.9
Penguin	1760	0.05	36	64×58	2.6	92	294.9	53.7
Lion	2078	0.06	48	21×32	17.4	80	212.4	30.8
Scorpion	2129	0.06	80	63×64	5.0	92	193.1	31.1
Dragonfly	2696	0.14	128	128×128	2.9	96	214.9	43.0
Dancer	2935	0.10	80	35×64	9.1	71	217.4	35.4
Skater	4122	0.07	64	30×48	11.1	132	176.6	26.8
Drops	5716	0.27	344	115×128	6.6	110	141.8	23.7
Butterfly	5836	0.08	92	64×41	8.8	91	194.0	31.0
Desktop	8687	0.11	132	64×41	13.7	280	113.4	16.8
Reschart	9445	0.23	212	128×79	6.3	132	232.7	37.1
Hygieia	9922	0.10	128	28×64	18.0	107	162.4	23.2
Tiger	16315	0.44	428	125×128	7.0	134	123.5	17.5
Embrace	18761	1.02	792	240×256	4.5	165	129.7	22.8
CAD	22393	0.30	360	128×100	7.5	172	140.0	24.0
Denmark	101386	0.59	912	128×99	19.2	286	58.2	7.9
Boston	108719	0.91	1132	200×180	8.8	351	63.2	8.4

Table 3: Quantitative results, including input vertices, construction times, cell stream statistics, and rendering rates without/with supersampling. Stream lengths are in 32-bit words.

8. Summary and future work

Cell streams are constructed by spatially specializing a vector graphics description to the cells of a lattice using a fast algorithm. These streams allow efficient random-access evaluation of composited layers of filled and stroked shapes, complete with transparency, color gradients, and high-quality antialiasing.

Avenues for future work include:

- Extension of cell streams to allow more rendering attributes (e.g. blurs or vector graphics instancing). Streams could generalize to full programs, including subroutines and recursion;
- Improvements to the adaptivity of the supersampling algorithm;
- Parallelized implementation of the encoding stage, enabling real-time animations, perhaps even a GPU implementation;
- Generalization of the concept of cell-based specialization to other applications besides vector graphics.

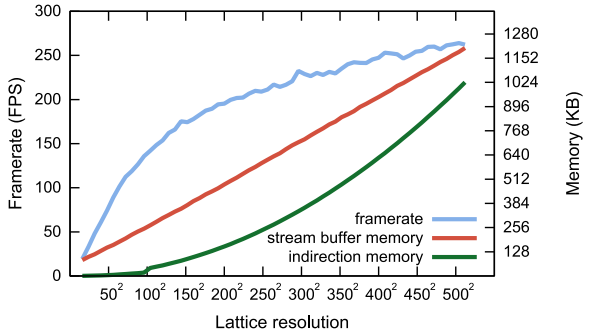


Figure 16: Memory usage and rendering rate as a function of lattice resolution for the Tiger in Figure 19.

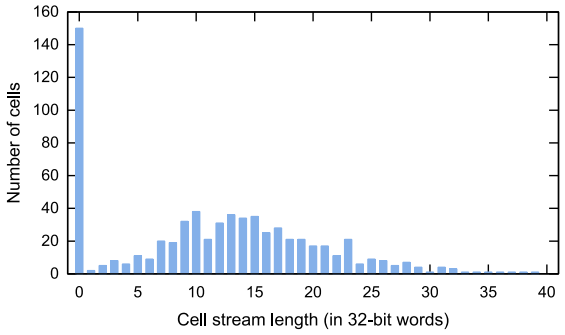


Figure 17: Histogram of cell stream lengths for the Lion in Figure 1.

Acknowledgments

The authors would like to thank Renato Werneck and Pedro V. Sander for fruitful discussions about the lattice-clipping algorithm. Daniel Szecket created the 3D model for the foldable map in Figure 1. The vector art on the cup model in the same figure is based on work by Aurelio A. Heckert.

References

BLINN J. 1998. A ghost in a snowstorm, *IEEE CG&A*, 18(1), 79-84.

BLINN J. 2006. How to solve a cubic equation, Part 2: The 11 Case. *IEEE CG&A*, 26(4), 90-100.

CARPENTER L. 1984. The A-buffer, an antialiased hidden surface method. *ACM SIGGRAPH*, 103-108.

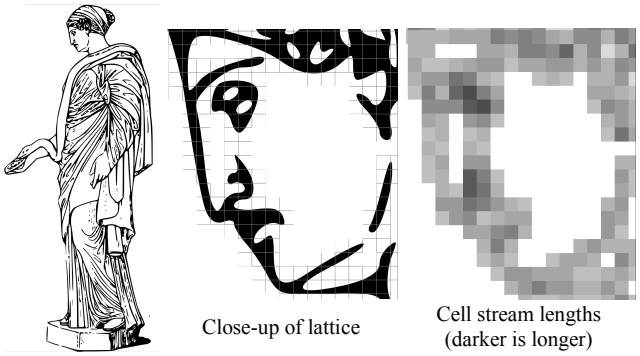


Figure 18: Visualization of the varying length of cell streams.

FOLEY J., VAN DAM A., FEINER S., AND HUGHES J. 1990. *Computer Graphics: Principles and Practice*. Addison Wesley.

FRISKEN S., PERRY R., ROCKWOOD A., AND JONES T. 2000. Adaptively sampled distance fields: A general representation of shape for computer graphics. *ACM SIGGRAPH*, 249-254.

GOLDMAN R., SEDERBERG T., AND ANDERSON D. 1984. Vector elimination: A technique for the implicitization, inversion, and intersection of planar parametric rational polynomial curves. *CAGD* 1, 327-356.

GREINER G., AND HORMANN K. 1998. Efficient clipping of arbitrary polygons. *ACM TOG* 17(2), 71-83.

GUPTA S., AND SPROULL R. 1981. Filtering edges for gray-scale displays. *ACM SIGGRAPH*.

HECKBERT P. 1989. Fundamentals of texture mapping and image warping. M.S. Thesis, UC Berkeley, Dept of EECS.

LAINE S., AND AILA T. 2006. A weighted error metric and optimization method for antialiasing patterns. *Eurographics*, 83-94.

MITCHELL D., AND NETRAVALI A. 1988. Reconstruction filters in computer graphics. ACM SIGGRAPH, 221-228.

LEFEBVRE S., AND HOPPE H. 2006. Perfect spatial hashing. *ACM SIGGRAPH*, 579-588.

LEFOHN A., KNISS J., STRZODKA R., SENGUPTA S., AND OWENS J. 2006. Glift: Generic efficient random-access GPU data structures, *ACM TOG* 25(1), 1-37.

LOOP C., AND BLINN J. 2005. Resolution-independent curve rendering using programmable graphics hardware. *ACM SIGGRAPH*, 1000-1009.

LOVISCACH J. 2005. Efficient magnification of bi-level textures. *ACM SIGGRAPH Sketches*.

NEHAB D., AND HOPPE H. 2007. Texel programs for random-access antialiased vector graphics. Microsoft Research Technical Report MSR-TR-2007-95, July 2007.

PARILOV E., AND ZORIN D. 2008. Real-time rendering of textures with feature curves. ACM TOG, 27(1).

QIN Z., MCCOOL M., AND KAPLAN C. 2006. Real-time texture-mapped vector glyphs. *Symposium on Interactive 3D Graphics and Games*, 125-132.

QIN Z., MCCOOL M., AND KAPLAN C. 2008. Precise vector textures for real-time 3D rendering. *Symposium on Interactive 3D Graphics and Games*.

RAMANARAYANAN G., BALA K., AND WALTER B. 2004. Feature-based textures. *Symposium on Rendering*, 65-73.

RAY N., CAVIN X., AND LÉVY B. 2005. Vector texture maps on the GPU. Technical Report ALICE-TR-05-003.

PERSSON E. 2007. Selective supersampling. Shader X^5, 177-183.

Figure 19: Example rendering results. On the left, prefilter antialiased results. The close-ups on the right show (a) high-resolution renderings overlaid with the lattice grid, (b) non-antialiased, (c) prefiltering only, and (d) prefiltering with k=8 supersampling.

SEN P., CAMMARANO M., AND HANRAHAN P. 2003. Shadow silhouette maps. *ACM SIGGRAPH*, 521-526.

SEN P. 2004. Silhouette maps for improved texture magnification. *Symposium on Graphics Hardware*, 65-73.

STOKES M., ANDERSON M., CHANDRASEKAR S. AND MOTTA R. 1996. A standard default color space for the Internet – sRGB http://www.w3.org/Graphics/Color/sRGB.html

SUTHERLAND I., AND HODGMAN G. 1974. Reentrant polygon clipping. *Communications of the ACM* 17(1), 32-42.

TARINI M., AND CIGNONI P. 2005. Pinchmaps: Textures with customizable discontinuities. *Eurographics*, 557-568.

TUMBLIN J., AND CHOUDHURY P. 2004. Bixels: Picture samples with sharp embedded boundaries. *Symposium on Rendering*, 186-194.

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

WINNER S., KELLEY M., PEASE B., RIVARD B., AND YEN A. 1997. Hardware accelerated rendering of antialiasing using a modified A-buffer algorithm. *ACM SIGGRAPH*, 307-316.

Figure 20: Parametric warps. By perturbing the texture coordinates prior to rendering, we can produce distortion effects that are gaining popularity in modern user interfaces, while retaining the crispness and resolution-independence of the vector graphics description.